A Rule Management Framework for Negotiating Mobile Agents

M.T. Tu, C. Kunze, W. Lamersdorf *
Distributed Systems Group, Computer Science Department
University of Hamburg, Germany
Vogt–Kölln–Str. 30, 22527 Hamburg, Germany
[tu,3kunze,lamersd]@informatik.uni-hamburg.de

Abstract

This paper proposes an application framework for mobile agents which provides rule-based generic services to support the three phases of a market transaction (information phase, contracting phase and settlement phase). It is described within the four views *of an electronic market which describe organizational as well as technological aspects. The focus of this paper is on the technological aspects of the contracting phase in which the participating agents carry out an automated negotiation process to determine the terms of contract.*

Keywords: rule management, negotiation, mobile agents, middleware, E-Commerce.

1. Introduction

Using software agents to automate commercial transactions in electronic markets is a very appealing idea, since automation could reduce the transaction costs considerably. Furthermore, due to the computational power available today, even better results could be achieved if the agents implement appropriate “intelligent” trading capabilities, e.g. to carry out a negotiation about complex contract terms that might yield too many options for a human user to survey completely. This potential has been recognized by many authors like [4, 3]. Some technical frameworks to embed negotiation capabilities into agents have also been proposed, e.g. in [7]. However, an important question regarding this kind of automation is how to control the behavior of the agents, especially how to impose user-defined rules on the logic – especially the negotiation logic – implemented by the agent. This paper suggests a concrete answer to this question by introducing an application framework that provides rule-based support for agents to carry out a commercial transaction which is considered under technological *as well as* organizational aspects, although the focus is put on the technological ones. Moreover, *mobility* as an important technical feature is also supported by the framework, meaning that the agents can flexibly roam through the network while performing their tasks, thus possibly saving bandwidth and freeing (resources on) the user’s machine.

The basic idea of the framework described in this paper is to provide rule-based generic services to support the three phases of a market transaction:

information phase Market participants look for potential negotiation partners. They do this usually by using a *broker* which matches potential participants.

contracting phase After the market participants have been matched using the broker, *electronic contracting services* can be used to support an automated negotiation. The contracting phase usually results in a contract which the participating parties have to sign to show their agreement.

settlement phase To automate the exchange of services and goods which are part of the contract, *workflow systems* can be used to automate and control the settlement.

This description of the phases of a market transaction also shows that *well-defined interfaces* between the services involved in different phases are needed to support a complete, integrated electronic market. With these services as atomic building blocks, different business scenarios can be supported. The rules can be used to influence the outcoming results of the different phases of a market transaction. In the *information phase*, rules to describe the requirements a potential partner must fit can be used to influence the broker’s behavior. These requirements can change from time to time, so that a reconfiguration of the agent by dynamically changing the rules should be possible. In the *contracting phase*, rules can be used to restrict the order of the possible actions of an agent, e.g. how offers are exchanged. This can

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

be regarded as a rule-based support for the enforcement of a *negotiation protocol*. Agents as well can follow different *negotiation strategies* which determine how to achieve their goal. Rules can be used as a *meta-strategy* to control the strategy's logic. By changing rules on a strategy, the same strategy can be used to achieve the goal in a different way or to achieve a different goal. This paper will mainly focus on the contracting phase. A contract resulting from this phase can then be used as the basis to fulfill the transaction in the *settlement phase* in which rules can be used to control the settlement and to detect violations caused by the contract parties.

The remainder of the paper is organized as follows: Section 2 introduces the basic rule concepts and the four views of the *ec-framework* upon which the framework presented here is based. Then, the rule management framework and its functionality is presented in detail in Section 3. The implementation is briefly described in Section 4 and a short outlook on subsequent work is given in Section 5.

2. Rule Management for Negotiating Agents

The rule system used by this framework is an extension and application of the rule system described in [8]. Basically, a rule consists of three parts: a *condition*, represented by a well-formed logical formula, a *trigger list* which contains the types of events which trigger the rule and an *activation*, which describes what to do if the rule has been triggered. The condition part is based on *interaction policies* (s. [6, 8]) which can be matched with each other (using a logical function called *unification*) to determine the greatest common denominator as a common interaction basis for different communication partners.

A rule needs to be hosted by a *rule container* which defines its application context. In this container, also *properties* are defined, on which the condition is formed and which can be modified by the activation of rules. Four rule types have been classified by their activation semantics. The *requirement rule* describes a condition which must evaluate to true when the rule is activated. The *state transition rule* describes two additional states as pre- and postcondition. If the condition is true and the precondition is the current state, it is switched to the postcondition. The *action rule* describes an action in form of a method call which takes place if the condition evaluates to true. The *policy rule* describes a generic action that will be performed, if the condition (meant as a goal) evaluates to false, to let the condition become true.

In addition to the rule types, four different *modes* are defined which describe the location or context of activation. The *internal mode* describes that the activation takes place where the rule is located. A copy of a rule in *oneway mode* is sent to the communication partner and is unified with a rule in *filer mode*, before it is activated at the remote location. If the conditions cannot be unified, a rule exception is thrown, which stops the program flow. The *callback mode* is an enhancement of the oneway mode. The values of the properties which are part of the unified condition are sent back and are activated locally too. This ensures exactly the same setting on both sides.

To generate *rule events* which trigger the rules, the RS-DII (Rule-Sensitive Dynamic Invocation Interface) is used. This can be thought of as a dynamic method wrapper which produces rule events before and after a method invocation. A rule object can register itself for an event type by putting it as an entry into the trigger list.

The framework of electronic markets described in [5] and [2] is chosen to describe the electronic market with rule-enabled services. In the following, this framework is referred to as *ec-framework*. The ec-framework is based on the three phases of a market transaction which are regarded from four views. These four views cover organizational as well as technological aspects of an electronic market. The ec-framework is chosen because of its comprehensive top-down description and because of its applicability as a meta-model. The focus of this paper lies on the technological aspects of the views on an electronic market. Nevertheless, a generic description of the organizational aspects is also given to define the scope of the framework. The four views of the ec-framework and their intended use for our rule management framework are:

business view This view consists of business models which are the basis of an electronic market. A business model describes aspects such as if the market is open (to every participant) or closed (only specific categories can participate). It describes the added value to the mere transaction service. The potential market participants as well as their intentions, possible actions and their general policies are described. The *transaction view* described next is used to substantiate the rules and policies of the business view.

The framework described in this paper comprises a generic business model which can be used to adapt business models of different organizations (Section 3.1).

transaction view The identification of business *processes* takes place in the *transaction view*. The description of the business processes is done by business scenarios. A business scenario is a formal description of a business process. The participants as well as the exchanged information objects are described. An information object is a formal description of the semantic content of the information exchanged. The flow control which is required for the *business view* is modeled by the business scenarios. Atomic transactions are identified which can be supported by the *services view* described next.

The goal of the framework presented here is to provide rule-sensitive generic services. Therefore, instead of prescribing specific business scenarios, the framework describes scenarios which can directly be mapped to generic services of the *services view*. The scenarios are used to describe the applicability of different rules for the generic services of each transaction phase. In this paper, rule-based scenarios are mainly limited to the contracting phase (Section 3.2).

services view Each phase of a market transaction is supported by specific services. These services are generic with respect to the organizational aspects. The generic services for each phase make up the content of the *services view*.

The main technological aspects of supporting the generic services by rules are described in the application of this view to the rule management framework (Section 3.3).

infrastructure view The *infrastructure view* describes the communication, transaction and transport infrastructure. The services of the *services view* are implemented with the help of this view.

The framework presented in this paper describes in the *infrastructure view* a rule-sensitive middleware which is an enhanced middleware providing a transparent mechanism to trigger, activate and manage rules in applications (Section 3.4).

3. Architecture and Functionality of the Framework

In this section, the architecture and functionality of the rule management framework is described within the four views of the ec-framework. The *business view* (3.1) and the *transaction view* (3.2) describe the organizational aspects. The *services view* (3.3) and the *infrastructure view* (3.4) describe the technological aspects.

3.1. Business View

The market participants are modeled as agents which can play the role of a *buyer* or the role of a *seller*. There is no restriction on the amount of agents participating in the market. A market consists of multiple *marketplaces* each of which defines a specific negotiation protocol. Deploying this *negotiation-centric* view of marketplaces means that an agent can always find out which protocol to use on a specific marketplace, so that he can switch to the right one for negotiation. The mediation or matching of participants is supported by a *broker* which always has knowledge of all buyers and sellers in the marketplace. If the contracting phase leads to a valid contract, the settlement is controlled by a *notary*.

In this paper, a marketplace which is called *bazaar* is discussed which allows negotiations in the style of a real-life bazaar: A buyer as well as a seller can look for a negotiation party. Content of the negotiation is the exchange of offers for the specific product searched by or offered by one of the market participants. Every market participant searches or offers only one product at a time (like in [1]).

Figure 1 depicts the overall agent architecture of this framework. Agents are *mobile*, so that they can roam through different marketplaces. Like it is described in the DynamiCS framework [7], agents have a communication part (the language), a strategy part (what to do with the language to achieve a given goal) and a protocol part (which elements of the language are allowed at a time). The agents of this framework have an additional feature: They are *rule-sensitive* which means that their behavior can be influenced and controlled by rules. Each agent has its own *rule container* which allows for adding and removing rules at runtime through a well-defined interface. The rules in such a container can be divided into different categories in a hierarchical form, so that rules which semantically belong together can form a *rule set*. To provide a mechanism which transparently triggers and activates rules, all communication between the agent must use the *rule-sensitive middleware*.

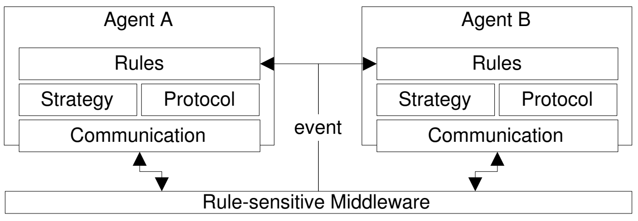

Figure 1. Agent architecture

A marketplace is represented by a *marketplace instance*. Every agent who wants to take part in transactions on this marketplace has to register with this component. The marketplace defines the market language as well as the negotiation protocol. At registration time, the agents can be checked if they speak the right market language and use the right protocol. Because the marketplace instance has knowledge about all registered agents, it can serve as the broker. If some market participants decide to negotiate, it should not be allowed that their negotiation be disturbed by another agent. The marketplace instance therefore manages *negotiation rooms* which must be entered by the negotiation parties. During the negotiation process, the participating agents are not visible to the rest of the market community until they have left their negotiation room.

In this paper, the marketplace language, which all agents must be able to understand, consists of five signatures which are described in table 1. The content of an offer is modeled by a condition object. The condition object can be used to express different requirements as a formula over properties which is expressed in first-order logic. Properties can be for example price, amount, etc. All market participants must share a common understanding of the semantics and syntax of these properties.

Table 1. Interface of the market language

Signatures
String item()
boolean isCustomer()
void offer(Condition offer, String agentAlias)
void accept(String agentAlias)
void abort(String agentAlias)

3.2. Transaction View

The market participants, the roles they can play and the rules they must follow to participate in a marketplace of this framework have been described in the generic business model of the *business view* in subsection 3.1. In the *transaction view*, a selection of simple scenarios and their support with rules is given. The focus of the rule support mechanisms in this paper lies on the contracting phase and covers the protocol bazaar. Therefore, only scenarios concerning the bazaar are described here.

The Bazaar Scenario *A seller as well as a buyer have a certain price range in which they can negotiate. If an offer is made, the respective market participant checks if it lies within his range. If it does, he accepts, if it does not, he reacts with a counter offer. This can occur as often as the participants want to exchange offers until either both accept or one aborts the negotiation.*

On the one hand, rules can be used to control the order in which offers, counter offers or aborts can be exchanged between the participants (which can be considered a simple negotiation protocol), and on the other hand, the strategy of the buyer as well as the seller can be modified by rules. The rules for keeping the negotiation conforming to a protocol are a combination of *state transition rules* with *requirement rules* as shown in detail in subsection 3.3. A buyer and a seller mostly have different strategies and different ranges of the price they would accept. A requirement rule can be used to prohibit prices which are out of range. An example is given for the buyer in table 2 and for the seller in table 3.

Table 2. Simple requirement rule for buyer

Rule-Type:	Requirement
Condition:	price $\leq$ 500.

Table 3. Simple requirement rule for seller

Rule-Type:	Requirement
Condition:	price $\geq$ 300.

If, for example, a strategy increases or decreases the offer for a specific amount in each negotiation round, a rule can be used to modify the amount (of change). E.g., after five negotiation rounds, the amount to increase the offer is reduced from 4 to 2 units. Table 4 shows a state transition rule which effects such a strategy modification.

Table 4. State transition rule to modify offer strategy

Rule-Type:	StateTransition
Condition:	rounds = 5
Precondition:	step = 4
Postcondition:	step = 2

Regarding the use of different rule types, *requirement rules* are best suited for enforcing invariants, because if the condition does not hold, an exception is thrown. In the exception handling, certain actions can be performed, such as generating a counter offer, etc. *State transition rules* as well as *policy rules* (policy rules are not mentioned in the examples) are best suited to effect property changes, because they are the only rules which are able to modify properties. *Action rules* are best suited to perform actions in the settlement phase (like paying or delivering) which is not discussed in this paper.

3.3. Services View

Within the *transaction view*, scenarios for the contracting phase have been introduced, which should be supported by a generic service. A rule-enabled contracting tool providing such a service is described in this subsection. It consists of two parts. The first one controls the allowed order of method invocations on the agent according to a property the values of which denote different states. Before the invocation of a certain method takes place, a requirement rule checks the value of the state property to determine if this invocation is allowed. After the method invocation is finished, a state transition rule switches the state property to the follow state. Using this concept, different protocols (i.e. different allowed orders of method invocations) can be realized by different rule configurations. The second part controls the strategy by rules which are used as meta-strategies. These two main functions are now elaborated by means of two agents using the protocol bazaar. It is assumed that the information phase is finished, so that the agents just start to negotiate. In figure 2, the state diagram (in UML) of the active agent (the one who makes the first offer) and in figure 3, the state diagram of the passive agent is shown.

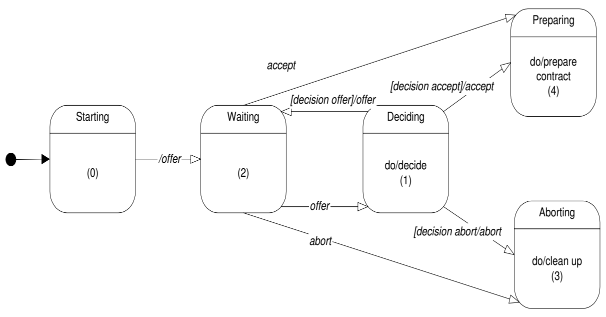

Figure 2. State diagram (UML) of active agent

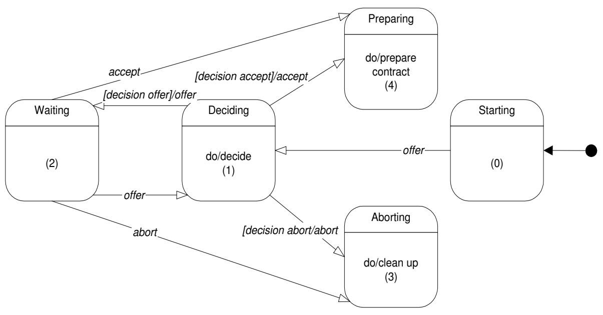

Figure 3. State diagram (UML) of passive agent

The activities are marked with numbers in brackets. These numbers describe the value of the state property and refer to a certain activity of the strategy. A notification mechanism informs the agent if the state property has changed, so that he can invoke the corresponding activity of the strategy. The mapping of the numbers to the activities of the strategy build the hard-coded part of the agent. The strategy, which calculates counter offers and evaluates offers, must know which language elements (methods) it is allowed to use. This is done by defining an interface which has a different signature for every possible combination of language elements. This interface is shown in table 5.

Table 5. Strategy interface

Signatures
void decide(Condition offer, String agentAlias)
void firstOffer(String agentAlias)
void counterOffer(Condition offer, String agentAlias)
void acceptOffer(Condition offer, String agentAlias)
void abortOffer(Condition offer, String agentAlias)
void counterOfferOrAccept(Condition offer, String agentAlias)
void counterOfferOrAbort(Condition offer, String agentAlias)
void acceptOrAbortOffer(Condition offer, String agentAlias)

The mode of each rule which controls the order of method invocation is the *internal mode*. The reason for this is that the agent should be autonomous in its actions, so he needs no rule interaction with other agents. Each agent has the number of requirement rules and state transition rules that are needed to describe the behavior shown in *both* state diagrams. This amount of rules can be titled as the *protocol set* which is loaded into the corresponding rule set of the agent.

A rule can be configured to be triggered before and/or after a certain method is called. To illustrate a rule configuration, a petri-net, which shows an invocation of a rule-sensitive method, is used. The petri-net is enhanced with roles which are shown as added ovals on the transitions. All attributes of the rule events can be seen from the illustration. For example, in Figure 4, Agent B sends a message to Agent A. So the *context id* of the sender of the message is Agent B and the *target id* of the recipient is Agent A. Agent A has two rules configured, one *before* and one *after* the *method name offer()*. The rule type and its settings are described in the transitions for the rule. (The parts which are not necessary for the rule configuration could be ignored, such as committing or rolling back the transaction as result of a rule exception). In the following, the rule configuration from the perspective of Agent A is described.

Figure 4 shows the rules which are configured before and after the call of method *offer()* on Agent A. If Agent B calls the method *offer()* rule-sensitively on Agent A, the requirement rule checks if the agent is in the starting state (0) or in the waiting state (2). After the method call, a state transition rule changes the state in accordance with the precondition (0 or 2) to the deciding state (1). This triggers the activity *decide()* of the strategy.

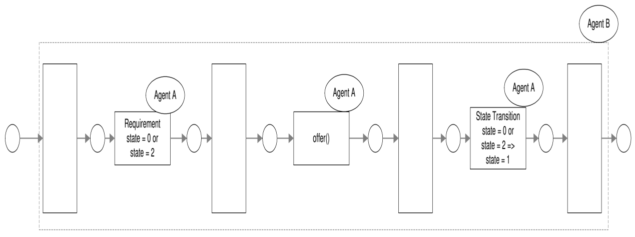

Figure 4. Rule configuration of bazaar (1)

If the strategy decides to make a counter offer, then the method *offer()* is called rule-sensitively at Agent B. Figure 5 shows the state transition rule which effects that Agent A changes from the deciding state (1) to the waiting state (2). If Agent A makes the first offer, the same state transition rule changes from the starting state (0) into the waiting state.

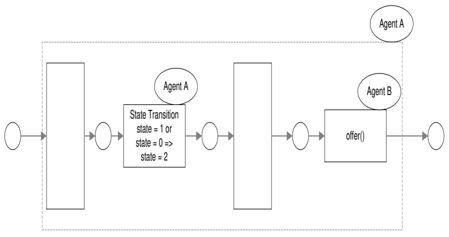

Figure 5. Rule configuration of bazaar (2)

Figure 6 describes the rules which are called before and after the invocation of method *accept()*. Before the method call, a requirement rule checks if the agent is in the waiting state (2) and after the method call, a state transition rule changes from the waiting state (2) into the preparing state (4). If the Agent A calls the method *accept()* at Agent B rule-sensitively, then the state of Agent A is changed from the deciding state (1) into the preparing state (4). This situation can occur as the consequence of an action which is performed within the activity *decide()*. Figure 7 shows this rule configuration.

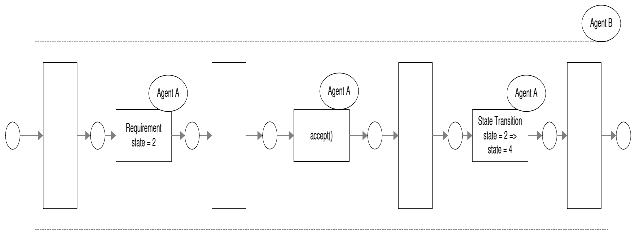

Figure 6. Rule configuration of bazaar (3)

Figure 8 shows the rules that must be configured before and after the invocation of method *abort()* on Agent A. Before the method call, a requirement rule checks if the agent

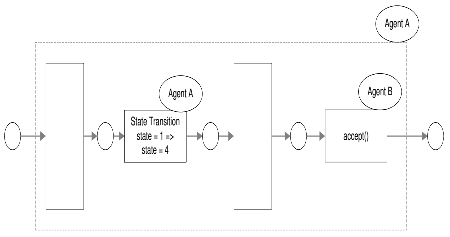

Figure 7. Rule configuration of bazaar (4)

is in the waiting state (2) and after the method call, a state transition rule changes from the waiting state (2) into the aborting state (3). If Agent A calls the method *abort()* on Agent B, the deciding state (1) is changed into the aborting state (3). This configuration is shown in Figure 9.

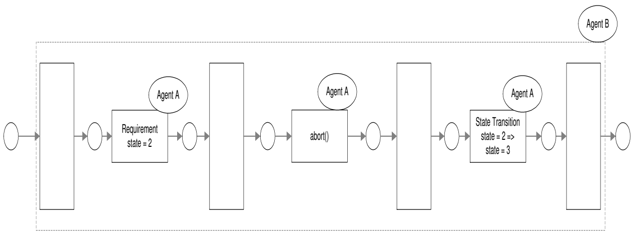

Figure 8. Rule configuration of bazaar (5)

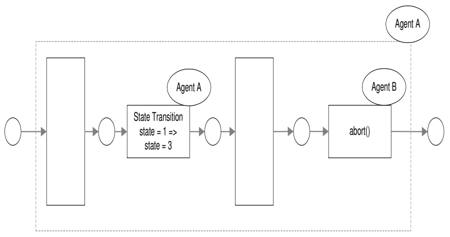

Figure 9. Rule configuration of bazaar (6)

Since the bazaar protocol is symmetric, the rule configuration for Agent B can be designed by only exchanging the roles of the agents (passive and active agent), denoted here by A and B. The rules can be configured with the help of *wild-cards*, so that the id of the counterpart must not be known. Instead, each agent must only know its own context id for the rule configuration and can use the same rule configuration for negotiation with different agents.

As mentioned, the properties of the rule set must be modified to control a strategy with rules. For this purpose, another rule set is defined, the *strategy set*. These properties define the count of negotiation rounds, the amount used to increase or decrease the next counter offer, etc. The agent must implement an interface specifying the *configuration points* of the strategy which are methods that are used to cause rule triggering events. This interface should be implemented with empty method bodies since the only usage of these methods is to configure the agents by rules before and after their invocations (this is also why this name was chosen). A rule-sensitive strategy is then implemented by calling these configuration points rule-sensitively at the same agent. This can be regarded as a hook where rules can modify or control the behavior of the strategy. It is distinguished between configuration points which check the consistency of some properties and configuration points which modify properties. The difference is that the strategy has to catch rule exceptions with the first type while ignoring them with the second one.

The result of a successful negotiation is saved in the *contract set* which provides the interface to the settlement phase. The following example should illustrate the concept described above. A strategy interface contains two configuration points: *checkConsistency()* and *modifyStrategy()*. The method *checkConsistency()* is defined to configure requirement rules to check an offer. The method *modifyStrategy()* is defined to configure rules with property writing behavior, such as state transition rule or policy rule. This configuration point is called by the strategy before making a new offer, so that it directly influences the next offer.

Figure 10 shows the complete rule support for the contracting phase. Two agents A and B are illustrated, who communicate over the rule-sensitive middleware. A *private* event mechanism is chosen for the rule support of the strategy because of the autonomy of the agents. For the rule support of the protocol, a shared communication mechanism is needed, which is illustrated as a shared event channel. Agent A starts the communication by sending the message *offer()* to Agent B **(1)**. The parameter of the message is a condition which can contain a fixed number of properties which should be negotiated. The RS-DII (used to perform rule-sensivitive method invocations) generates a rule event before the method call, which is distributed over the shared event channel **(2,3a,3b)**. If it is not allowed to send the message *offer()* in the current state, a *rule exception* is thrown, which stops the communication. If it is allowed to send the message *offer()*, then the message is sent to Agent B with the help of the ORB (Object Request Broker) **(4)**. Agent B saves the offer into a rule set. After the method call, the RS-DII distributes a rule event again over the shared event channel **(5,3a,3b)**. This event triggers the state transition rule of Agent B and eventually of Agent A to change the state property. Because a call over the RS-DII is transactional, the change of the state property will be visible only after finishing the call. Now the invocation of the method *offer()* through the RS-DII is finished. The rule set of Agent B notifies the change of the state property **(6)**, the value of which indicates that a new offer has been received. The agent logic invokes the corresponding activity of the agent **(7)**. The task

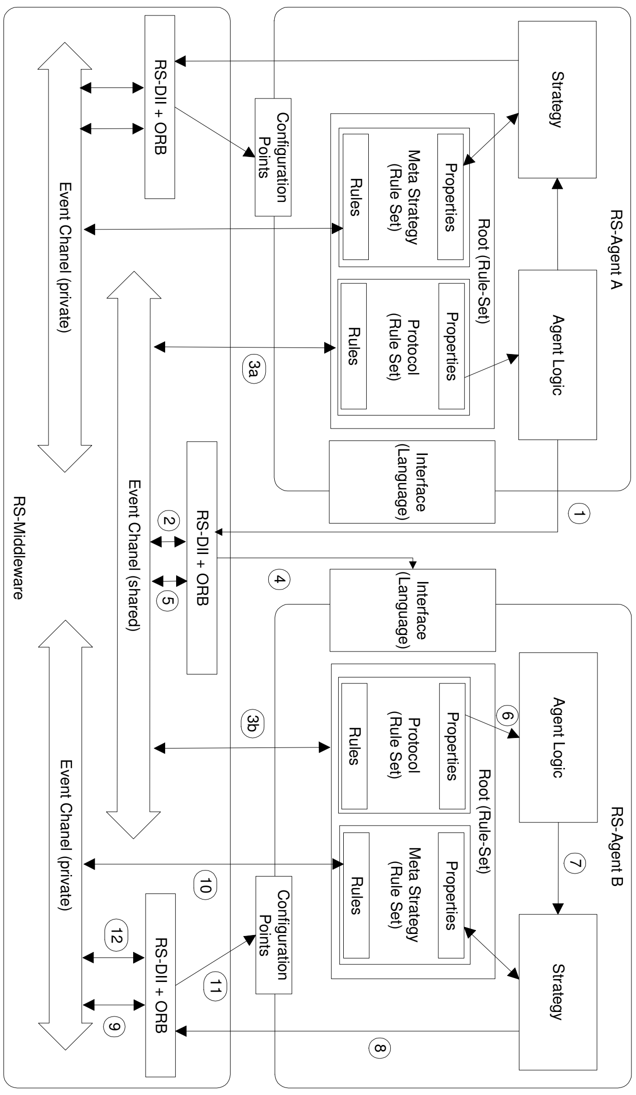

Figure 10. Negotiation of rule-sensitive agents (RS-Agents)

of the strategy is now to check the offer, to make a counter offer or to abort the negotiation. The strategy checks the offer by calling a configuration point (*checkConsistency()*) at Agent B **(8)**. The RS-DII distributes before the method call a rule event to Agent B over the private event channel **(9,10)**. This triggers a requirement rule at Agent B, which throws a rule exception if its condition does not hold. If no rule exception is thrown, the RS-DII invokes the configuration point **(11)** and distributes a rule event after the method call. The call of the first configuration point is now finished. The second configuration point (*modifyStrategy()*) is called before Agent B makes the next counter offer. The flow of execution is similar to the one described before. If Agent B does not accept the offer of Agent A, he now sends the new message *offer()* to Agent A and a similar flow of execution continues at Agent A. This process can repeat until either an offer is accepted or the negotiation is aborted.

3.4. Infrastructure View

The *services view* described above requires an infrastructure to implement the generic services. The rule-sensitive middleware described in this subsection serves as the infrastructure for this framework. Figure 11 shows its components.

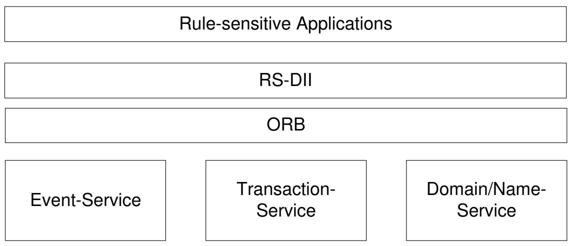

Figure 11. Rule-sensitive middleware

A common interface, the RS-DII (Rule-Sensitive Dynamic Invocation Interface), is used to make the rule execution model transparent to the application developer. A *transaction service* is used to make each call transactional which means to apply the ACID properties to the properties in the rule sets. To distribute the rule events which can trigger the rules, an *event service* is used. Each rule-sensitive agent must register with the *domain/name service*. This is because the rule event protocol depends on a static amount of rules in the system for each method call. So the domain/name service can be used to retrieve the current number of rules in the system and can wait for the right time to register or deregister an agent. Additionally, the uniqueness of context id's of the agents is checked by the domain/name service at registration time.

4. Implementation

For portability reasons, Java was chosen to implement the rule-sensitive middleware and, on top of it, a marketplace with agents. The Voyager framework, a powerful Java development tool which supports object migration, is chosen as the ORB (object request broker). As event and transaction service, the CORBA services of the VisiBroker product are used. The domain/name service uses the alias names of Voyager as context id's and is itself implemented as a CORBA object. Figure 12 shows the package dependencies of the implementation.

The packages are divided into packages which belong to the rule-sensitive middleware (*infrastructure view*) and

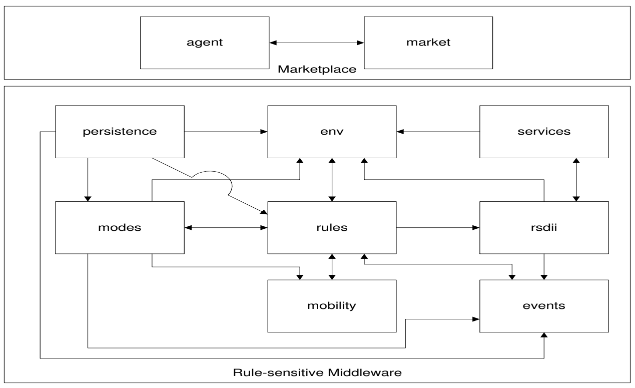

Figure 12. Package dependencies

packages which belong to the organizational aspects of the framework (*business view* and *transaction view*). The rule configurations described in the *services view* and applied to the agents are implemented in form of a sample marketplace with the bazaar protocol. To provide a short description of the content of the packages of the rule-sensitive middleware followed by the packages *market* and *agent*:

- Package **service**: Implements a special domain/name service.
- Package **env**: Everything necessary for the usage of rules on the application level, like rule sets (rule container) and the interface *RuleSensitive* which all rule-sensitive applications have to implement, is provided in this package.
- Package **rules**: An implementation of the different rule types including action rule, policy rule, state transition rule and requirement rule is provided by this package.
- Package **modes**: The different rule activation modes – internal, filter, oneway and callback – are implemented in this package.
- Package **mobility**: The rules themselves are mobile objects implemented in this package.
- Package **events**: This package provides the implementation of the rule events which are communicated through the event channel.
- Package **rsdii**: This package provides an implementation of the RS-DII.
- Package **persistence**: An implementation to persist rule configurations in an XML format is provided by this package.
- Package **market**: An implementation of a marketplace for the *business view* is given in this package.

- Package **agent**: An implementation of rule-sensitive agents which act as market participants is given in this package. Also a simple strategy is provided which implements the behavior described in the examples of the *services view*.

The rules of a rule-sensitive application can be modified by an editor with a graphical user interface has also been implemented. The editor makes the specification of rules and rule configurations more comfortable.

5. Summary and Outlook

The focus of this paper is how to support software agents, especially mobile agents, to carry out a negotiation process by rule-based mechanisms. Therefore, the technological aspects, particularly generic services supporting rule capabilities, were primarily described within the four views of the ec-framework. The *business view* introduced basic concepts of a negotiation-centric agent architecture. Within the *transaction* and *services view*, mechanisms of a rule-enabled marketplace, which allow for the control of negotiation protocols and strategies by specific rule objects, were explained. To illustrate the functionality of these mechanisms, concrete rule configurations were defined for the negotiation protocol of a bazaar and configuration points were introduced as hooks to modify a strategy. Within the *infrastructure view* and in the implementation section, it was briefly described how components of a *rule-sensitive middleware* providing the functionality of the services view were implemented.

Early experiments with the prototype of this rule-sensitive middleware have revealed some weaknesses w.r.t. the system runtime behavior which are mainly due to the overhead of triggering rules on every rule-sensitive method invocation. This might be improved by using more efficient basic middleware services, especially a better event management, in the implementation and is subject of current research. The application of rules to agents has the main benefit that the agents' behavior can be flexibly controlled and modified at runtime. On the other hand, the system becomes more complex and above all more dynamic since every additional rule can influence the whole rule system. This makes the availability of tools necessary which support the (semantical) development of rule configurations. One of the first tools in this direction is the *rule editor* implemented within the context of this framework which simplifies the creation, modification and storage of rules. This rule editor can still be seen as a “low-level” tool because it does not provide support to prevent the developer of applying rules which do not provide the desired behavior. Thus, for the different phases of a market transaction including the very important settlement phase, which could not be treated within the limited scope of this paper at all, different supporting tools and more efficient generic services still need to be developed and/or integrated into the rule-sensitive middleware.

References

[1] A. Chavez and P. Maes. Kasbah: An Agent Marketplace for Buying and Selling Goods. In *Proceedings of the 1. Intl. Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96), London, UK*, 1996.

[2] M. Lindemann and A. Runge. Electronic Contracting within the Reference Model for Electronic Markets. In *Proceedings of the 6th European Conference on Information Systems (ECIS'98), Aix-en-Provence, France*, June 1998.

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

[4] J. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiations among Computers*. MIT Press, 1994.

[5] B. Schmid and M. Lindemann. Elements of a Reference Model for Electronic Markets. In *Proceedings of the 31. Annual Hawaii International Conference on System Sciences (HICSS), IV*, Jan. 1998.

[6] M. Tu, F. Griffel, M. Merz, and W. Lamersdorf. Generic Policy Management for Open Service Markets. In H. König and K. Geihs, editors, *Proc. of the Int. Working Conference on Distributed Applications and Interoperable Systems (DAIS'97), Cottbus, Germany*. Chapman & Hall, Sept. 1997.

[7] M. 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*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[8] M. Tu, F. Griffel, M. Merz, and W. Lamersdorf. Interaction-Oriented Rule Management for Mobile Agent Applications. In L. Kutvonen, H. König, and M. Tienari, editors, *Proc. of the Second Int. Working Conference on Distributed Applications and Interoperable Systems (DAIS'99), Helsinky, Finland*. Kluwer Academic Publisher, June 1999.