Interaction-Oriented Rule Management for Mobile Agent Applications

M.T. Tu, F. Griffel, M. Merz, W. Lamersdorf *
Distributed Systems Group, Computer Science Department, University of Hamburg
Vogt–Kölln–Str. 30, D–22527 Hamburg
[tu|griffel|merz|lamersd]@informatik.uni-hamburg.de

Abstract

The characteristic features of delegation and interaction make the mobile agent programming paradigm on the one hand attractive for a wide range of distributed applications – in particular those related to E-Commerce – but on the other hand also pose a considerable semantical risk potential. In this paper, we present a generic approach of imposing rules on the behavior of mobile agents aiming primarily at reducing this kind of risk without impairing the agents' basic decision logic and interaction ability. On the contrary, strong emphasis will be put on the algorithms, design and implementation of rules used to support the interaction between agents.

Keywords: dynamic rule management, mobile agent, interaction support, E-Commerce.

1 Introduction

1.1 Motivation

Mobile agent technology is increasingly considered as the right mean to build innovative, user-friendly and more "intelligent" E-Commerce applications (s. [KJ98] for a broad overview). However, the reason of its popularity is not mainly technological, since there are no application problems known that would necessarily require agent technology to be solved[1], but rather conceptual, that means mobile agent technology seen as a programming paradigm seems to provide exactly the right abstraction level and the right model to build a lot of distributed applications, especially those in the area of electronic commerce. This is due to the fact that agents are commonly regarded as autonomous components to which some given tasks can be completely *delegated*. In case of mobile agents, the delegation capability is even more distinctive because due to their location independence, the delegating instance, which is often a human principal, does not have to care of providing a place for the agents to act and therefore can go offline while his software delegate is carrying out the tasks. Moreover, agents are generally conceived to carry out tasks that require an essential amount of

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

[1]Even arguments regarding performance have to be verified carefully, since there is always a trade-off between migration and communication costs.

interaction or communication with the environment such as making inquiries, taking orders, negotiating deals etc. so that they represent the adequate abstraction of instances that can fill out corresponding roles such as buyers, sellers or mediators in commercial transactions.

However, it is quite obvious that exactly these characteristic abilities of delegation and interaction also pose a great risk potential because the decisions delegated to an agent could bring about results that are not desired by its principal. Of course, this risk potential and the corresponding "surprise effect" could be reduced to a minimum by restricting the decision logic to a finite set of concrete, i.e. fully specified, situations which can be interpreted and handled by the agent, but such a "hardwiring" solution would in turn substantially reduce the interaction ability of the agents – especially in open enviroments, where potential business partners are generally unknown – and the agents' overall functionality, consequently. In other words, the more decision autonomy is delegated to the agents, the greater the risk of undesired or unforseen decisions that has to be taken into account is. In this paper, we present a generic approach of imposing rules on the behavior of mobile agents aiming primarily at reducing this kind of "semantical" risk without impairing the agents' basic decision logic and interaction ability. On the contrary, strong emphasis will be put on the algorithms, design and implementation of rules used to support the interaction between agents. Moreover, it will become clear that by adding rules of appropriate types to the agents at run-time, their functionality can not only be modified but also extended in a flexible and dynamic way.

1.2 Related Works

One of the probably most important and interesting trends which the field of Distributed Applications has witnessed during the last few years is the emergence of so-called *Business Objects* (BOs) [Sim94, AF98]. Currently, both the specification and development of BOs are mainly being driven by the efforts of a few, but very influential organisations and companies, most typically those of the *Object Management Group* (OMG) – through the standardization of the *Business Object Facility* [OMG98a] – and those of IBM – through its product *SanFrancisco* [IBM98].

In general, the main idea behind BOs is that the whole business of an enterprise should be modeled and supported by *high-level* constructs that directly represent the semantics of real-world business entities and processes. Taken as a goal, this might not sound like a real novelty since at present, a lot of enterprises are already applying quite sophisticated software and information systems to support and/or automate their business processes. The real progress which is expected from using BOs is rather to be found in the way of modeling and handling the business by means of software components which can be described by the following "vision"[2]: The business is processed by freely *inter-*

[2]The fact that this seemingly visionary goal is concretely being translated into action can be positively interpreted as that the state-of-the-art of distributed systems has now reached a stage at which it really seems promising to develop complete support not only on different

operating, *reusable* and *portable* software objects spread across the distributed enterprise. Especially, it should be possible to specify several *semantical* aspects of the reusable BOs such as state declarations, attributes, relationships, rules etc. in a dynamic and simple manner so that an average developer or even non-technical personnel could set up and maintain the business applications by taking prefabricated BOs out of a building kit and adding the application-specific semantics to these objects. Indeed, the possibility of dynamically incorporating some types of semantics into BOs can be considered the most important characteristic that differs them from more "conventional" objects, apart from the trivial fact that a BO always represents some *business* entity. However, presently available implementations of platforms for BOs such as SanFrancisco only provide very limited support for such add-on semantics and there are no products compliant to the BO standard of the OMG available yet.

The mechanisms presented in this paper can be considered as an approach of realizing one type of add-on semantics, which corresponds to *Event Condition Rules (ECRule)* in the BOCA terminology. In BOCA [OMG98a, OMG98b], ECRules are specified as "an abstraction for any behavior specification that is enforced based on one or more events" ([OMG98b], p.64) and are classified into 3 subtypes:

- Invariant: "A requirement on the state of the object as an expression that must evaluate to true."
- StateTransitionRule: "specifies the conditions under which the type will go from one state to another."
- ECARule (Action Rule): "augments ECRule with a generic action specification."

Since BOCA aims at a standard, it is not concerned with implementation aspects. Also, at the current state of specification, semantical considerations still stay behind syntactical ones, i.e. those related to data structures. As presented in the next section, a substantial part of our approach is dedicated to designing and implementing different activation semantics. Especially, our rule system is conceived to support (semantical) interaction between distributed applications which is something beyond the current functionality of the BOCA rules.

The rest of the paper is organized as follows: In Section 2, the concept of interaction-oriented rule management – including rule types, rule expressiveness, generic processing functionality and different modes of activating rules locally or remotely – is outlined. Thereafter, a concrete rule management architecture satisfying the requirements of decentralization, event-orientation, mobility and transactional activation is presented in Section 3. Relevant aspects of the prototype implementations are then described in Section 4 and finally, the paper is concluded by Section 5.

system levels but also and rather directly on the application level.

2 An Interaction-Oriented Rule Management Approach

As mentioned above, the ability of correct interaction is essential for mobile agent applications in open and dynamic environments (such as E-Commerce over the Internet) and therefore should be supported by generic rule management mechanisms. In this section, the basic rule concept for such an interaction-oriented rule management approach including the logical expressiveness, the corresponding processing functionality and the activation semantics of different rule types is described.

2.1 Rule Types

In order to be able to process rules of different types in a uniform way, a generic rule concept, which can be considered the abstract supertype of all concrete rules, is needed. In our approach, this concept consists of:

- A *condition*: Every rule contains a logic, represented by a well-formed logical expression.
- A *trigger list*: Every rule is triggered by occurences of certain event types which are registered in the trigger list.
- An *activation*: Every concrete rule type must have an activation semantics specifying what to be performed when a triggering event has occurred.

Based on this generic rule concept, different concrete rule types – including those suggested by BOCA – can be defined which only differ in their activation semantics:

1. Requirement rule (R-rule)[3]: A R-rule represents a requirement that has to be satisfied when a triggering event occurs. Thus, its activation causes the condition part to be evaluated and an exception will be thrown if it evaluates to wrong.
2. State transition rule (S-rule): S-rules implement the concept of "switching from one (discrete) state to another". Therefore, every S-rule is associated with two state specifications and the activation semantics: If the first state is the current one and the condition part evaluates to true, then the second one becomes the current state.
3. Action rule (A-rule): An A-rule is associated with the specification of an action to be performed when the condition part evaluates to true.
4. Policy (P-rule): A policy represents a goal that should be fulfilled by performing some (generic) action (s. [TGML97]). Thus, a P-rule is associated with the specification of an action and the activation semantics:

[3]This type corresponds to an *invariant* in the BOCA terminology.

when the condition part evaluates to false, the action is performed. After that, the condition part is re-evaluated, and if it still evaluates to false, an exception is thrown.

2.2 Expressiveness of Rule Conditions

The condition of a rule can be considered its most important part, since it essentially determines the rule activation. As will be elaborated in the following, the basic idea of realizing support for interaction between mobile agents or distributed applications in general is providing a matching function that computes the common basis of two or more rules (belonging to different cooperation partners). Such a function can basically be reduced to matching or unifying the rule conditions, and it depends on the expressiveness of the condition part whether corresponding algorithms can be found. In [TGML97], an algorithm is presented that can process expressions which contain only one variable or *property* in each atomic constraint. In the next subsection, we present processing functions which are not restricted with respect to the number of properties in an atomic constraint, i.e. which can handle rule conditions of the following expressiveness:

Condition: A condition is either a (atomic) constraint or a logical combination of constraints using the operators AND, OR and NOT.

Constraint: A (atomic) constraint is a full-ordered relation between the linear combination of n properties and a literal.

Property: Properties are used to express and/or modify the (partial) behavior of an object explicitly. A property consists of a name, a type and a value.

Restrictions:

Due to the need of finding generic and efficient algorithms, a few restrictions are currently imposed on these definitions:

- The NOT operator is only allowed on constraints.[4]
- Although constraints are in the first place supposed to represent a (linear) relationship between numerical properties, e.g. `(2*speed - cost` $\geq$ `0)`, also string properties can be processed by our functions. In case of strings however, the left-hand side of the constraint can only contain one property, since in general, a linear combination of strings does not make sense.

2.3 Generic Processing Functions

In order to provide a dynamic rule management system to control and influence the behavior – especially the interaction and cooperation behavior – of

[4]Even though breaking down general negation to the constraint level can always be achieved by applying DeMorgan transformations on any negative expression, it conflicts with transforming conditions into DNF which is required for the unify-algorithm presented below.

distributed applications in a flexible way, *generic* functions to process rules are needed, i.e. functions that are independent of a specific application semantics. In [TGML97], we presented generic functions to *evaluate, compare, unify, arbitrate* and *activate* policies which are restricted to one property in the condition part. Meanwhile, new algorithms which are based on the simplex method ([Sch87]) have been developed to overcome this restriction for all numerical types so that these functions can now process rules of the expressiveness specified above. The simplex method also enables the development of a new function to determine optimal rules. In the rest of this section, we will outline the new algorithm for the most important function to provide support for the interaction behavior of distributed applications, i.e. the *unify* function. Extensions of the other functions are based on the same principle.

The unification of two rules of the same type is based on the unification of the respective condition parts which is defined as the weakest condition that logically implies both of these conditions. Formally:

Definition 2.1 *Unify:*
For all well-formed conditions $C_1, C_2, R : R = unify(C_1, C_2)$ *iff*
$R \rightarrow C_1$ *and* $R \rightarrow C_2$ *and* $\neg(\exists R') : (R' \not\equiv R) \wedge (R' \rightarrow C_1) \wedge (R' \rightarrow C_2) \wedge (R \rightarrow R')$

Example 2.2 *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)*

The algorithm to compute this function is mainly based on transforming the input conditions into a special normal form called *LDNF* and determining implications between the disjuncts which can be solved with the aid of the simplex method. In the following algorithm, places where this method comes into effect are marked by comments beginning with "SM".

Definition 2.3 *LDNF (Lean Disjunctive Normal Form):*
A condition $P = (d_1 \vee d_2 \vee d_3 \vee \ldots \vee d_n)$ *in Disjunctive Normal Form is in LDNF if* $\forall(1 \leq i, j \leq n; i \neq j) : \neg(d_i \rightarrow d_j)$

Algorithm 2.4 *Unify:*

P_1, P_2 `in LDNF`
`// SM: LDNF reduction`
D_1 `:= set of all disjuncts in` P_1
D_2 `:= set of all disjuncts in` P_2
`TMP :=` $\emptyset$
`For every` d_i `in` D_1

```
    For every d_j in D_2
        If (di → dj) then TMP := TMP ∪{d_i}
        If (dj → di) then TMP := TMP ∪{d_j}
        // SM: check implications
D_1 := D_1\ TMP
D_2 := D_2\ TMP
For every d_i in D_1
    For every d_j in D_2
        If (d_i ∧ d_j) ≡ FALSE then TMP := TMP ∪{(d_i ∧ d_j)}
        // SM: check conjunction
unify(P_1, P_2) := disjunction of all elements of TMP
```

Checking Implications with the Simplex Method:

Since the reduction of well-formed conditions to LDNF is also based on testing implications between disjuncts, the test of $(di \rightarrow dj)$ is the basic step in this algorithm which can be done as follows:

$$
\begin{aligned}
& d_i \rightarrow d_j \equiv \neg(d_i \wedge (\neg d_j)) \\
& \text{with:} \\
& d_i := a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \\
& d_j := a_{j_1} \wedge a_{j_2} \wedge \ldots \wedge a_{j_m} \\
& \text{where a's represent atomic constraints, we can derive:} \\
& d_i \rightarrow d_j \equiv \\
& \neg((a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_1})) \\
& \quad \vee \\
& \quad (a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_2})) \\
& \quad \vee \ldots \vee \\
& \quad (a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_m})))
\end{aligned}
\tag{2.1}
$$

Each of the conjunctions on the right-hand side corresponds to an unequation system (since it contains only atomic constraints), the solvability of which can be determined be the simplex method[5]. If there does not exist a solution for every conjunction, then $(di \rightarrow dj) \equiv TRUE$ holds.

2.4 Rule Activation Modes

In order to provide adequate support for interactive mobile agents – or application components in a distributed environment in general – a rule which belongs to a certain component should not always be activated locally, i.e. with respect

[5]This can be done basically by determining an objective function and setting up a simplex tableau in such a way that all atomic constraints connected by '∧' in each line are seen as restrictions. A detailed elaboration of this method would clearly exceed the size of this paper and is therefore omitted here.

to and effecting the behavior of the local component, but it should also be able to activate the same rule *remotely*, i.e. effecting the behavior of a component other than that containing the rule. 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 first 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 ONEWAY, CALLBACK and FILTER (s. Table 2.1).

Mode	Description
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.[6]
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[7], if not empty, is then activated.

Table 2.1: External activation modes

3 The DynamiCS Rule System Architecture

In the following, we describe the concrete system design for the implementation of rules that can be dynamically imposed on mobile agent applications, as developed in the DynamiCS[8]project at University of Hamburg. The requirements to be fulfilled by this design are:

[6]This mode is used, for example, when it is necessary that both customer and provider utilize the same support services with exactly the same parameters.

[7]This represents the common basis for the pending transaction between the components.

[8]*Dynami*cally *C*onfigurable *S*oftware

- **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[9], 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, have 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.

3.1 Rule Events

In DynamiCS, rule interactions take place exclusively through so-called *rule events* which are typed objects that are sent to and received from an *event channel*. Rule events are subdivided into *trigger events* and *reply events*. In general, rule events can be generated from application objects or from rule objects. The most typical trigger events are those associated with method calls. If these are directly caused by application objects, they are called *method rule events*, otherwise, i.e. if such triggering events are caused by rule objects, they are denoted *filter rule events*, since those events are destined for filter rules (s. Table 2.1).

The corresponding event classes are illustrated in Fig. 3.1. Every rule event inherits from the abstract base class `RuleEvent` the attribute `contextID` which is the identifier of the context, in which the event is generated[10]. This ID is used to map reply events to the correct source. Every `MethodRuleEvent` carries the `signature` of the method being called and the `state` indicating whether the event stands for, e.g., "before execution" or "after execution". `FilterRuleEvents` are generated by rule objects in EXTERNAL mode to transfer the rule semantics which is coded in `unifyParam` to the corresponding filter rule of the remote `target` object (s. Section 3.4). `ReplyRuleEvents` are used

[9] or *appliances* in the current BOCA terminology

[10] This ID can, but must not be unique for each event. For example, it can be the ID of the application object that generated the event.

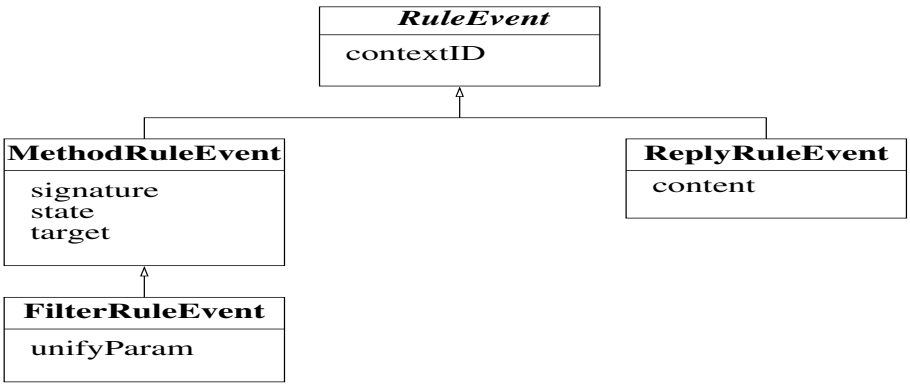

Figure 3.1: Rule event class diagram

to indicate (by the attribute `content`) whether the activation of the rule was successful and also to return the setting of corresponding properties in case of the CALLBACK mode (s. Table 2.1).

3.2 Rule Objects

As mentioned, rules in DynamiCS are active objects acting as *event listeners* and activate themselves upon occurences of corresponding *trigger events*. Rule objects can be added to every application object providing the interface of a *rule container* which determines the *context* for activation.

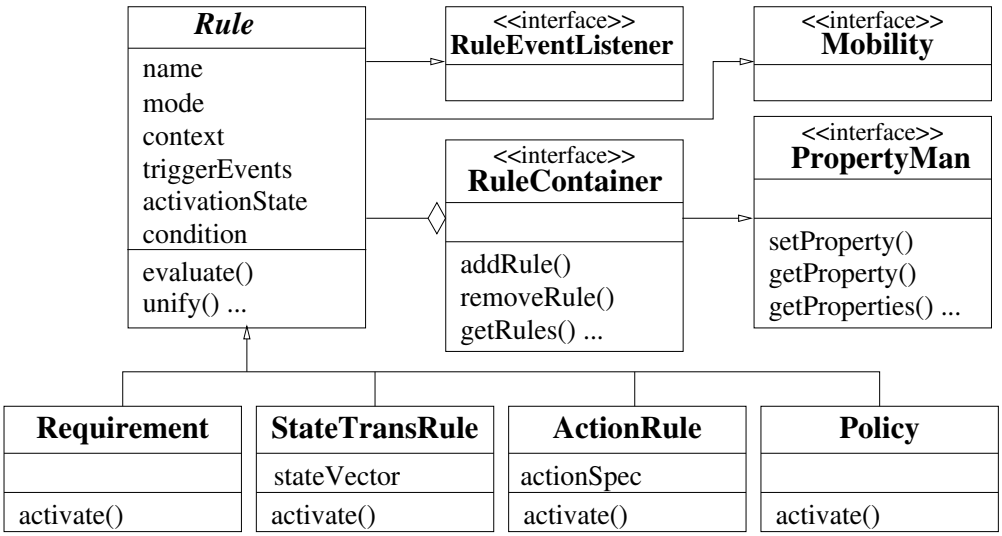

Figure 3.2: Rule object diagram

As depicted by Fig. 3.2, every rule inherits from the abstract base `Rule` class which consists of a `name`, an activation `mode` (s. Section 2.4), a `context` which is a pointer to the current container, a list of `triggerEvents`, an `activationState`

indicating whether the rule is being activated (and therefore not ready to react to further trigger events kept in the event channel), the `condition` part and provides the generic methods implementing the logic to process rule conditions. Furthermore, every rule implements the `RuleEventListener` and the `Mobility` interfaces. Every application object to be supported by rules must provide besides the `RuleContainer` also the `PropertyMan` interface representing the external access to relevant application or interaction properties. Every concrete rule type implements the specific rule semantics by its own `activate` method and additional attributes.

3.3 Rule-Sensitive Dynamic Invocation Interface

DynamiCS provides a rule-sensitive dynamic invocation interface (RS-DII) 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[11] thrown by rule objects are caught and delivered further to the application object (making the call) and the call is interrupted (s. also 3.4).
- 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.

3.4 Rule Activation

In order to get a complete view of the interaction of the described rule-related objects, let's look at a typical scenario of activating rules in EXTERNAL mode when a method is called, as illustrated by Fig. 3.3.

`agent 1` and `agent 2` represent two rule-sensitive mobile agents supporting the `RuleContainer` interface. Now, when `agent 1` calls a method offered by `agent 2` 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 `agent 1` in CALLBACK 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`

[11] Note that rule (violation) exceptions are thrown by means of events of type `ReplyRuleEvent` (s. 3.1).

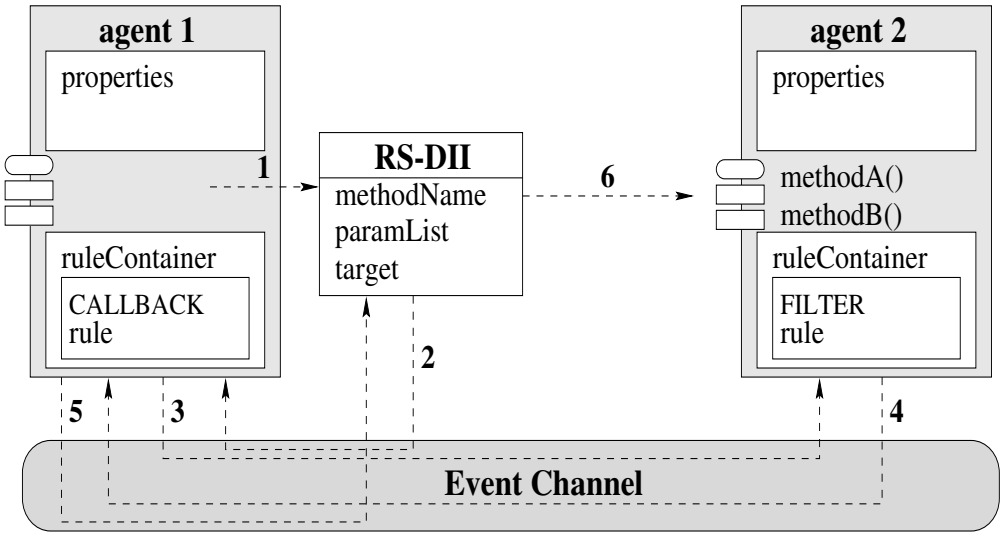

Figure 3.3: Activation of rules in EXTERNAL mode

and sends this event to the event channel (3), thereupon the corresponding FILTER rule of `agent 2` is triggered. There, both rules are matched using the `unify` method, and if the result is non-empty, it is activated on `agent 2` side. Thereafter, the allocation of the corresponding properties of `agent 2` (as the common setting for the pending cooperation) is sent back via a `ReplyRuleEvent` to the the CALLBACK rule of `agent 1` (4). Using this common setting as the actual condition part, the activation of the rule is now completed as if it was in INTERNAL mode. Then, a `ReplyRuleEvent` is sent back to the RS-DII component by `agent 1` 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 `agent 2`, 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.

4 Implementation Issues

The current prototype of the presented rule management architecture is completely implemented in Java which is presently the most relevant language to develop innovative distributed applications in practice.

As illustrated by Fig. 4.1, the DynamiCS rule system implementation consists of:

- the logic library providing generic rule processing functions including those to evaluate, compare, unify and optimize rule conditions. This library is implemented in plain Java.
- the rule object layer providing implementations of the rule types, default rule container and rule-sensitive agent classes which can be used

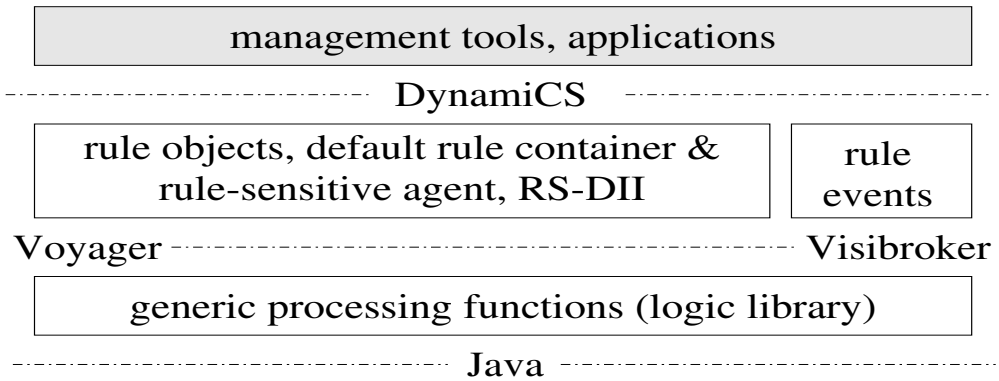

Figure 4.1: DynamiCS rule system implementation layers

as skeletons to build rule-sensitive applications. These classes as well as the RS-DII are implemented on top of Voyager ([Obj98]) which provides an efficient toolkit to develop Java-based mobile application objects. The event channel and rule event classes are based on the Event Service of Visibroker ([Inp98]) which is one of the most wide-spread CORBA products at present.

- the application layer providing management tools and sample E-Commerce related applications.

Currently, the distributed transaction service employed by the RS-DII is a self-developed prototype which could be soon replaced by the announced, but not yet available Voyager Transaction Service ([Obj98]) which promises better transparency and performance since it is directly integrated with the Voyager-DII.

Fig. 4.2 shows the graphical user interface of the rule management tool which is an applet that can be used to contact a rule-sensitive mobile agent, list the rules and corresponding properties carried with him, add new, delete or clone and move rules to other agents or applications.

5 Summary and Outlook

In this paper, we have presented a dynamic approach to impose rules on mobile agent applications aiming both at making their behavior more reliable in the sense of not violating certain semantical conditions as well as improving their interaction behavior. First, after describing the motivation and overall context of the work, we outlined the concept of interaction-oriented rule management including rule types, rule expressiveness, the corresponding unify algorithm to find the common basis of two rules and different modes of activating rules locally or remotely. Then, a concrete rule management system architecture which satisfies the requirement of decentralization, object- and event-orientation, mobility and transactional activation was presented. Finally, some relevant aspects

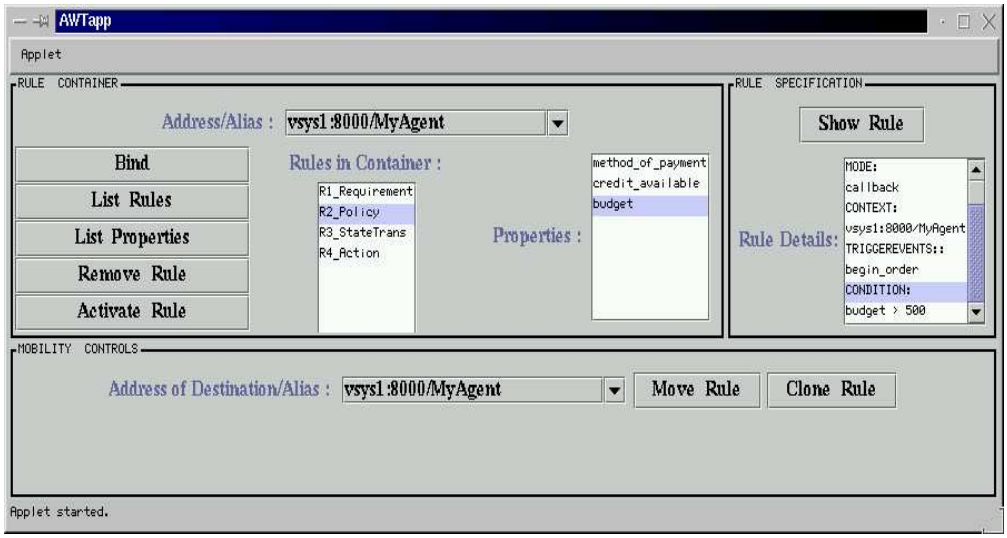

Figure 4.2: Rule management tool UI

of the current prototype implementation of the DynamiCS rule management approach were given.

Since the rule management mechanisms described in this paper are not only appropriate to restrict or modify the interaction behavior of mobile agents, but might obviously extend their application semantics in a very dynamic manner – especially through the use of action and state transition rules – more practical research seems to be necessary to explore the potential as well as risk of such rule-sensitive applications. Moreover, the presented mechanisms are not only restricted to mobile agents, but also applicable to software components in general, particularly those that can be dynamically assembled from building blocks which are themselves active, autonomous components. We will discuss the specific requirements for rule management mechanisms resulting from such a *componentware* view ([Gri98]) in another paper.

References

[AF98] P. Allen and S. Frost. *Component-Based Development for Enterprise Systems.* Cambrige University Press, 1998.

[Gri98] Frank Griffel. *Componentware.* dpunkt–Verlag, 1998. (In German).

[IBM98] SanFrancisco, 1998. www.ibm.com/Java/Sanfrancisco/.

[Inp98] Inprise. Visibroker product documentation, 1998. `www.inprise.com/techpubs/visibroker/visibroker33/`.

[KJ98] M. Knapik and J. Johnson. *Developing Intelligent Agents for Distributed Systems.* McGraw-Hill, 1998.

[Obj98] ObjectSpace. Voyager — user guide & white papers, 1998. `www.objectspace.com/developers/voyager/white/index.html`.

[OMG98a] Business Object Component Architecture Revision 1.1. OMG Document 98-01-07, 1998.

[OMG98b] Business Object Component Architecture Revision 1.2. OMG Document 98-07-01, 1998.

[Sch87] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1987.

[Sim94] O. Sims. *Business Objects.* McGraw-Hill, 1994.

[TGML97] M.T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. Generic Policy Management for Open Service Markets. In H. König and K. Geihs, editors, *Proc. of the Int. Working Conference on Distributed Applications and Interoperable Systems (DAIS'97), Cottbus, Germany.* Chapman & Hall, September 1997.