From Rules To Rule Patterns

G. Kappel, S. Rausch-Schott, W. Retschitzegger, M. Sakkinen[1]

Institute of Computer Science, Department of Information Systems
University of Linz, AUSTRIA
email: {gerti, stefan, werner, markku}@ifs.uni-linz.ac.at

Abstract. Rule-based systems are a commonly accepted solution for smoothly capturing the context-dependent and time-dependent organizational knowledge of large enterprises, also known as business policies. At the same time, however, the design of rule-based applications is one of the most pressing open research problems. This is largely because of the expressive power and flexibility of existing rule-based models together with a lack of design guidelines on how to apply these models. Learning from analogous problems in object-oriented system development and borrowing their solution metaphor we introduce *rule patterns* as generic rule-based solutions for specifying business policies. The advantage of rule patterns is their predefined, reusable, and dynamically customizable nature allowing the designer to reuse existing experience for building new rule-based applications. The paper introduces the general notion of rule patterns and illustrates the approach by sample rule patterns for specifying interaction policies in workflow applications.

Keywords and Phrases. rule patterns, specification of business policies, workflow management

1 Introduction

Non-standard applications have to cope with frequently changing requirements which are, to a large extent, due to changes in the business environment [Louc91]. Those aspects of the business environment which are subject to frequent changes are often referred to as business policies. Business policies typically capture context-dependent and time-dependent organizational knowledge. They may be based on ethics, law, culture and organizational commitments by either prescribing a certain action or by constraining the set of possible actions [Herb95, Odel94, Schr95]. To cope with changing business policies, it should be possible to easily adapt the application implementing the respective policies. The introduction of the object-oriented paradigm has been one step in this direction. Object-oriented languages and development environments help to intuitively model the universe of discourse and to adapt to changing requirements [Fugi92, Tsic89]. However, a mechanism for explicitly specifying business policies in a natural and straightforward way is still missing. In most object-oriented systems business policies are implemented by some methods or parts thereof, and thus business

1 On leave from the Department of Computer Science and Information Systems, University of Jyväskylä (Finland), until July 1996.

knowledge is mixed with code realizing the basic functionality with a low modification probability. Within this respect there exist some suggestions for distinguishing methods containing policy, i.e., the "making of context-dependent decisions", from methods containing implementation, i.e., the "execution of fully specified algorithms" [Rumb91]. It has been shown, however, that a more intuitive way to describe business policies is in terms of rules, not least since domain specialists usually do so in expressing their system requirements [Petr94].

Approaches for integrating rules with object-oriented concepts have been widely accepted as a powerful means for explicitly modeling business policies [Odel94, Rubi94, Tsal91]. These approaches are also known as rule-based models, or more accurately as *active object-oriented models* whose most prominent basic mechanism is the Event/ Condition/Action rule (ECA rule)[2]. Such a rule monitors the situation represented by an event and a condition and executes the corresponding action when the event occurs and the condition holds true [Diaz91, Ditt95]. ECA rules allow for an explicit, localized, and transparent specification of business policies. With ECA rules, on the one hand, reusability of methods and thus of classes is enhanced since they no longer contain application-specific policies. On the other hand, reusability of a business policy itself is enhanced, since it is encapsulated within rules.

Unfortunately, designers are overtaxed with the power of the provided concepts, especially with the amount of possibilities for implementing a certain business policy. Thus, they would need some guidelines for defining rules realizing specific business policies. Learning from analogous design problems in object-oriented system development and borrowing their solution metaphor, we introduce rule patterns in analogy to design patterns [Coad95, Copl95, Gamm94, Pree95]. Rule patterns provide templates for an easy specification of business policies. They both categorize rules according to different types of business policies, and at the same time provide an abstraction mechanism for specifying rules in an application-independent manner. If we try to draw a rough analogy between rules and rule patterns, respectively, and standard object-oriented constructs, we can say that an ECA rule corresponds to a single method. However, there is no widely adopted concept or construct for combining several rules that would correspond to an ordinary class, even less to a parameterized (generic) class. Patterns in object-oriented programming represent a still higher level of abstraction and granularity. In the current paper, we will present only quite simple rule patterns corresponding to generic methods or procedures.

To illustrate the potential of rule patterns we fall back upon workflow applications as one prominent application area. In these applications, rules are used for specifying different kinds of business policies such as the order of processed activities, agent selection and interaction, and worklist management [Bußl94, Casa95, Eder95, Kapp95ab, Rein93]. We pick up interaction policies which, in addition to workflow applications, have been recognized to be crucial concerning the interaction between objects within object-oriented modeling as well as concurrent object-oriented programming languages.

2 Note that there are approaches which integrate the ECA paradigm into non-object-oriented environments like extended relational database systems.

The remainder of this paper is organized as follows: Section 2 introduces the general notion of rule patterns. Section 3 illustrates the approach by describing sample rule patterns for interaction policies in workflow applications. A comparison to related work follows in Section 4. Section 5 concludes with a discussion of the approach and ongoing research. Due to space limitations the paper can only discuss some rule patterns within a small fraction of a real world example. A more complete list of interaction rule patterns together with their application to a fully-fledged running example is available as a technical report [Kapp95c].

2 Rule Patterns for Specifying Business Policies

There exists a broad range of business policies which can be realized by using rules. Business policies encountered in workflow applications, such as those mentioned above, are a small fraction thereof. For a discussion of different classifications of business policies the interested reader is referred to [Gert93, Herb95, Kapp95c, Odel94]. The motivation of introducing rule patterns as abstraction of rules is at least threefold. Firstly, and as already mentioned, there exist no design guidelines for applying rules in application development. Secondly, the amount of business rules found in simple case studies reported in literature is beyond several hundreds [Knol94]. Clearly, one needs some kind of structure and/or classification in order to manage such an amount of rules. And thirdly, we have found out that rules realizing business policies can be abstracted in that they are no more restricted to a specific application domain but rather can be easily applied to other domains with little adaptation effort. Consequently, when looking at a rule realizing a specific policy, one can find components which are applicable for a number of application domains as well as components specific to a single application domain. Let us consider the following two policies. A business policy in a warehouse could be that every time the stock-keeper takes out goods, the number of goods in stock has to be checked and, if fallen below a given limit, new goods have to be ordered. A similar business policy in a bank could be that every time a customer withdraws a certain amount from his/her account, the balance is checked and if overdrawn, the bank charges are increased. It can be seen that there are a lot of similarities between these two policies. Consequently, it is possible to factor out common components valid for both application domains. An abstract formulation of these two policies could be: As soon as a certain value is changed and this value falls below a certain limit some reaction has to be undertaken.

To avoid that the application designer has to consider such fixed, i.e., common components of a rule realizing some business policy again and again for each application domain and to support the adaptation of such a rule to other application domains, rule patterns are introduced. *Rule patterns* are descriptions of rules, predefining certain event/condition/action-pairs. They are an *abstract means* to capture a certain kind of business policy in a generic and thus application-independent manner. Some rule patterns realize business policies by abstracting from a single rule only. At a higher level of abstraction, however, rule patterns are a *composition* of several rules working together to realize some specific kind of business policy. *Parameterization* makes rule patterns even more general and versatile than they could be otherwise. Parameterized rule patterns consist of components independent from particular applications, i.e.,

predefined components, as well as components specific to particular applications, i.e., *parameterized components*. The kinds of parameters as well as the degree of parameterization, i.e., the relation between parameterized components and predefined components, vary according to the different kinds of policies.

Figure 1 illustrates the process of working with rules and rule patterns. In the long run, for each kind of business policy rule patterns are provided within a pattern library. Within our research prototype, this library is organized as a set of dictionaries wherein the patterns are all stored as first-class objects. At this time, of course, we are far from claiming that our set of rule patterns is complete. Consequently, this pattern library must be extensible by both defining new patterns and specializing existing ones (① in Figure 1). At the same time, existing patterns or parts of them can be easily reused. In order to use rule patterns within an application, they have to be customized by the application designer. This is done by binding the parameters of a selected pattern on the basis of the application semantics. The system guides the application designer during the process of parameter binding by providing on-line help for each required parameter and by restricting the binding alternatives to those that do not contradict the specification of the underlying pattern (cf. the application of a rule pattern shown in Figure 4). Once a rule has been fully and correctly specified, it can be automatically generated and stored in the rule base attached to the corresponding application (② in Figure 1).

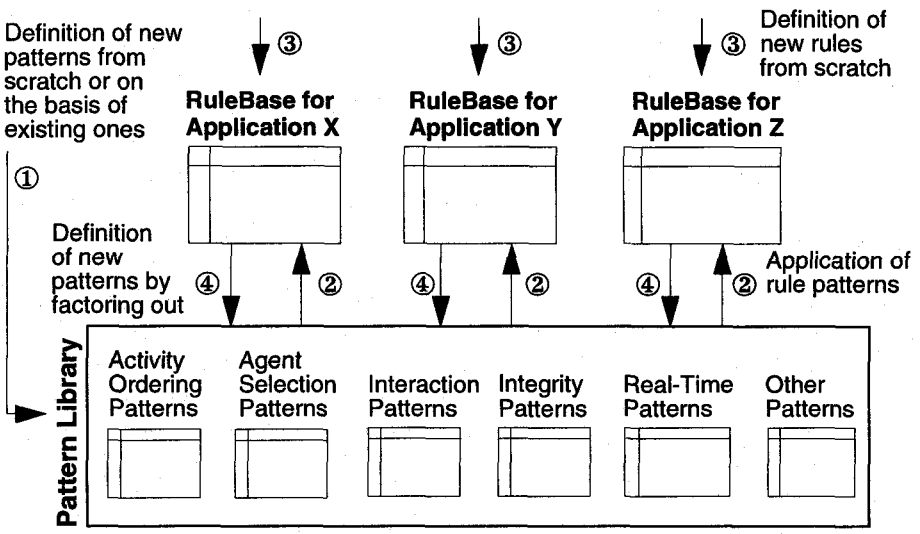


Fig. 1. Working with Rules and Rule Patterns

Furthermore, it is still possible to specify rules without using a pattern and to store them directly within the appropriate rule base (③ in Figure 1). This is normally done for rules without reusability in mind. If sometimes later an application designer recognizes that a specific design situation recurs and thus is worth to be specified by a corresponding rule pattern, existing rules can be used for this abstraction process (④ in Figure 1).

3 Rule Patterns by Example: Interaction Rule Patterns

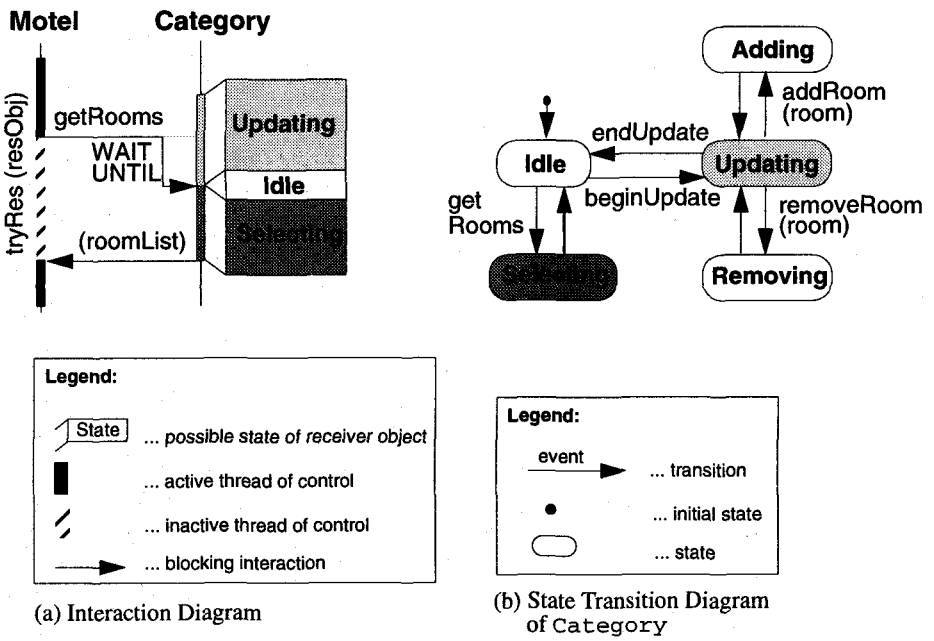
In the following, we want to illustrate our approach by sample rule patterns specifying different policies concerning interactions between a sender object and a receiver object. These rule patterns are called *interaction rule patterns*. Interaction rule patterns represent interaction structure abstractions similar to the already well known data structure abstractions (e.g., classification, association, aggregation, and generalization/ specialization) and to control structure abstractions (e.g., sequence, condition, and iteration). They enhance sequential message passing in two different ways. Firstly, in contrast to sequential message passing based on a single thread of control where any receiver object has to accept messages implicitly, i.e., unconditionally, interaction rule patterns model interactions in a concurrent object-oriented environment based on multiple threads of control. Thus, any receiver object accepts messages explicitly, i.e., depending on its actual state. Secondly, interaction rule patterns support different kinds of interaction policies such as synchronous, asynchronous and future synchronous [Nier93, Yone87]. In the following, two interaction rule patterns are described by means of an example situated in the area of active object-oriented workflow systems [Kapp95ab], where it is crucial to flexibly model the synchronization between concurrently executing tasks within activities. In a first step, each interaction policy is described by means of an application-specific rule. The syntax and semantics of these rules are based upon our underlying research vehicle TriGS (Triggersystem for GemStone) [Kapp94a] and will be described, if necessary, on the fly. In a second step, the corresponding generic rule pattern is factored out. Note that the reader should not get too much diverted with details of TriGS. The approach is intended to be applicable much more generally than just for TriGS. For a complete description of the example as well as of eight currently existing interaction rule patterns we refer to [Kapp95c].

3.1 Synchronous Interaction Policy

Problem. The first interaction policy we want to realize is synchronous interaction. Synchronous interaction means that the sender of a message blocks until the receiver has finished the requested method execution. The receiver on its part is not able to execute the requested method until it is in the right state for satisfying the request. That is, a received message is explicitly accepted on the basis of the receiver object's actual state. A state is specified in the sense of a state transition diagram [Rumb91, Hare88]. For sake of simplicity, we assume that every object has an implicit multi-valued instance variable `state` holding its actual state(s).

Example. Our running example is a typical workflow application realizing a computerized reservation management system for a nationwide consortium of motels connected via a wide area network. Each motel manages rooms grouped into different categories. In order to be able to check the availability of rooms for a certain reservation request, a motel has to retrieve a list of rooms from a `category` object corresponding to the desired room category stored within the reservation object (`resObj`). For this task, the `motel` object interacts with the `category` object by sending the message `getRooms` within the context of the method `tryRes` synchronously, i.e., blocking until the result is available.

This situation is shown in Figure 2. The notation of the interaction diagram in Figure 2a is based partly on the notation for event-trace diagrams of OMT [Rumb91], and on the interaction diagram notation of OSA [Embl92]. Sender object, i.e. the `motel`, and receiver object, i.e., the `category`, are denoted by vertical bars. Time progresses from top to bottom of the lines. Solid lines designate active threads of control and dashed lines designate inactive, i.e., blocked threads of control. Arrows denote interactions between objects. Furthermore one sequence of states of the receiver object as it might occur during a certain interaction is shown. These states are all part of the state transition diagram in Figure 2b. Note that we assume multiple threads of control in the sense that either the receiver object has its own thread of control, or another thread of control executes methods of the passive receiver object. Multiple threads are necessary since, if the receiver object is currently not in the right state for satisfying a certain request, it must be possible to execute one or more other methods in parallel, assuring that the receiver object eventually reaches the correct state. In Figure 2a, for example, `getRooms` has to wait until the `category` object has been transformed from the state `Updating` into the right prestate `Idle`, necessary for performing `getRooms`.



(a) Interaction Diagram

(b) State Transition Diagram of `Category`

Fig. 2. Synchronous Interaction Between `Motel` and `Category`

Figure 2b shows the state transition diagram based on OMT notation [Rumb91] for class `Category`. According to it, an object of class `Category` remains in state `Idle` until receiving one of the requests `getRooms` or `beginUpdate`. In the first case, the object changes its state to `Selecting` and automatically returns to the state `Idle` upon finishing the execution of the corresponding method. In the second case, the object changes its state to `Updating`. Within this state, a `category` object accepts the messages `addRoom`, `removeRoom` and `endUpdate`. Upon receiving the messages `addRoom`

and removeRoom, respectively, the object changes its state; it automatically returns to the state Updating upon finishing execution of the corresponding methods. When receiving the message endUpdate it returns to the state Idle.

Concrete Solution by using Rules. The rule blocker_getRooms depicted in Figure 3 has to be specified in order to realize the synchronous interaction between a motel object and a category object. Note, we use a simplified notation borrowed from our underlying active object-oriented model TriGS. Rules in TriGS extend the ECA paradigm to ECEA, therewith allowing a more flexible definition of rule execution points [Kapp94b]. The *condition event selector* ($Esel_C$) of the rule blocker_getRooms defines all events which are able to trigger the rule. The rule blocker_getRooms is triggered each time any object of class Motel sends the message getRooms to any object of class Category from within the method tryRes. The keyword PRE denotes that the rule is triggered *before* the requested method getRooms, further on called *triggering method*, is executed. Moreover the triggering method getRooms is blocked at least until the condition is evaluated. The *condition* part of the rule checks whether the receiver receiver is not in the appropriate prestate Idle for executing the requested message getRooms. If the condition is false, the action part is not considered at all and the triggering method getRooms proceeds immediately. If it is true, getRooms is further blocked until the correct state is reached (keyword WAIT UNTIL).

DEFINE RULE blocker_getRooms **AS**	Rule Name
ON PRE(Motel,tryRes:aResObj,Category,getRooms) **DO**	Condition Event Selector ($Esel_C$)
IF not (receiver.state includes: "Idle") **THEN**	Condition
WAIT UNTIL ON POST (receiver, getRooms) **OR** **POST** (receiver, endUpdate)	Action Event Selector ($Esel_A$)
EXECUTE NIL.	Action

Fig. 3. Synchronous Interaction Rule

The *action event selector* ($Esel_A$) of the rule blocker_getRooms is responsible for monitoring the transitions leading to the correct state Idle. Therefore the $Esel_A$ is specified as a *composite event selector* whose components are two simple messages connected by disjunction. Each of these messages, when sent to the actual receiver, leads to the state Idle (cf. also Fig. 2b). After one of these messages has finished execution (keyword POST), the *action* is executed. Since the action is defined as NIL, the triggering method getRooms is allowed to proceed immediately.

One benefit of this approach is the fact that the motel object does not have to check periodically for the right state of the category object (polling) since reaching the appropriate state is signalled by the category object itself. However, even more important is the fact that already implemented sequential message passing can be enriched with this additional synchronization semantics without any modifications of the existing implementation.

Generic Solution by using Rule Patterns: The Blocker. Although the realization

of a interaction policy by using rules provides some important advantages, there still exists one major drawback. Since the policy of synchronous interaction is not only used for the message `getRooms` but also for other parts within our reservation example and for other application domains, too, those components of the rule remaining the same for any synchronous interaction would have to be specified again and again. What we want to do is to relieve the application designer from this task. For this purpose, the application-dependent rule for synchronous interaction depicted in Figure 3 is abstracted to a generic rule pattern for synchronous interaction, called *Blocker*.

DEFINE RULE PATTERN blocker_<name> **AS**	Rule Pattern Name
ON PRE (<sender>,<senderMethod>, <receiver>,<message0>) **DO**	Condition Event Selector ($Esel_C$)
IF not (receiver.state includes: <prestate>) { **AND** not (receiver.state includes: <prestate>) }* **THEN**	Condition
WAIT UNTIL ON (**POST** (receiver, <message>) ['[' receiver.state includes: <prestate> ']']) { **OR** (**POST** (receiver, <message>) ['[' receiver.state includes: <prestate> ']']) }*	Action Event Selector ($Esel_A$)
EXECUTE NIL.	Action

Fig. 4. Synchronous Interaction (Blocker) Rule Pattern

This abstraction process is done by factoring out the recurring components of the rule and by explicitly specifying the formal parameters which have to be bound when applying the pattern. The structure of the Blocker rule pattern is depicted in Figure 4. The first parameter of each rule pattern is used for generating the specific name of each rule being deduced from the pattern at hand (<name>). The $Esel_C$ of the Blocker rule pattern consists of four different parameters denoted by angle brackets. Note that for all parameters specified within angle brackets actual parameters have to be provided. Any unbracketed occurrence of a parameter name is bound to the nearest preceding bracketed (defining) occurrence. For the parameters <sender> and <receiver>, classes as well as objects may be specified. In case of classes being specified, the corresponding rule is triggered whenever <message0> is sent from/to any instance of the <sender>/<receiver> class[3]. If the same rule shall be applied for several classes and/or methods, wildcards may be used instead of specifying actual parameters. For example, if it is immaterial from which method <message0> is sent, <senderMethod> is not specified and the rule is triggered whenever the specified <sender> sends <message0> to <receiver>. The *condition* of the rule pattern consists of a conjunction of one or more actual state checks (cf. the EBNF meta symbols "{...}*"). It checks whether the set of actual states of the receiver (`receiver.state`) includes one of the prestates of the transition denoted by the incoming message. If one of the prestates is already reached, the action part is not considered at all and the triggering method

3 Note that class methods can be specified by the keyword CLASSMETHOD within the parameters <senderMethod> and <message0>. In that case the rule is triggered whenever the <message0> is sent from/to the respective class.

(<message0>) can be executed. If the receiver is in none of these prestates, the sender blocks (keyword WAIT UNTIL).

The $Esel_A$ specifies that the action should be executed after one of the methods corresponding to a transition leading to one of the prestates has been executed. In case that one distinct message is used as event in different transitions, the resulting state set has to be checked within a *guard* (enclosed in square brackets). A guard specifies a boolean expression further restricting the set of events able to trigger the condition or action, respectively. The action of the rule pattern is defined as NIL indicating the fact that the Blocker pattern is just used for blocking the triggering method until the receiver is able to accept the sent message.

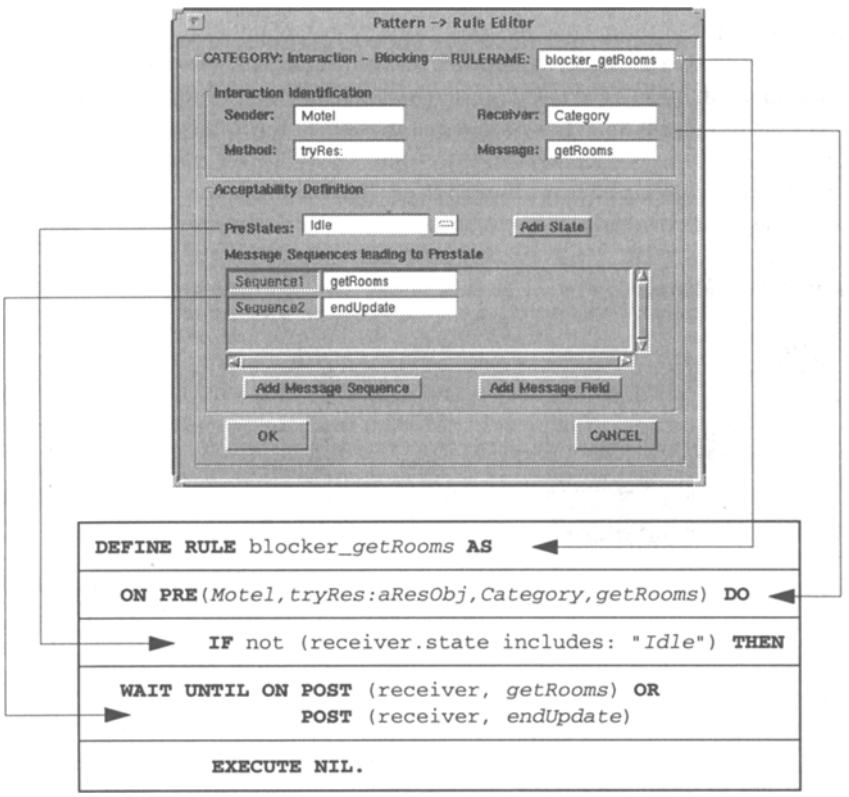


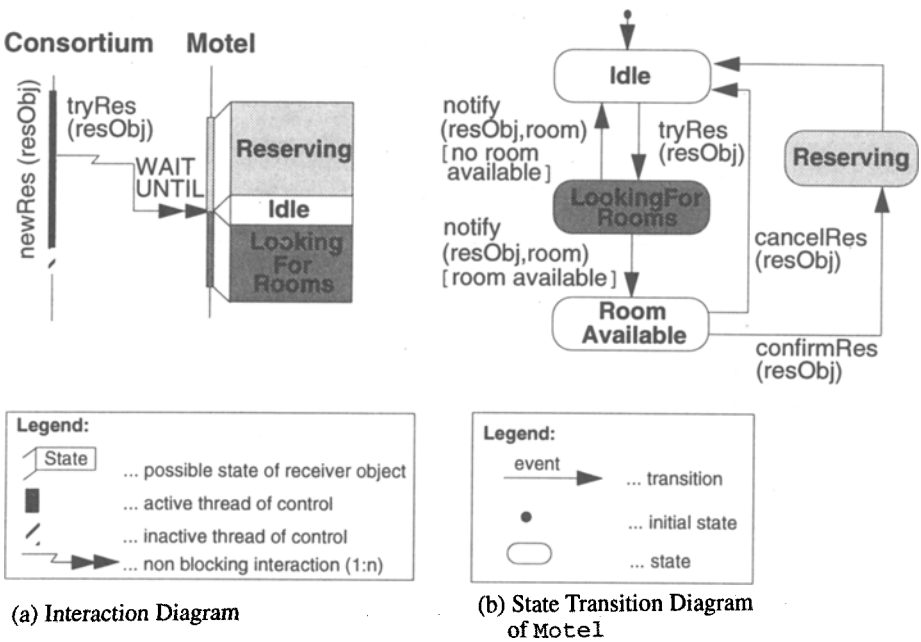
Fig. 5. Application of a Blocker Rule Pattern

Due to its generic nature, at a first glance the proposed rule pattern looks admittedly much more complicated than the application-dependent rule described above. But once a rule pattern is defined it can be easily applied just by defining its parameters using a corresponding form. On the basis of this parameter binding, the system automatically generates a rule. Anyway, the application designer is not concerned with the complexity of the rule pattern. Figure 5 illustrates the approach of generating the business rule realizing synchronous interaction as described in Figure 3 out of the Blocker rule pattern. Note that actual parameters are shown in italic. The remaining components are generic and are, thus, together with formal parameters the basic ingredients for the rule pattern.

3.2 Asynchronous Interaction Policy

Problem. The second interaction policy we want to realize on the basis of rules is asynchronous interaction. Asynchronous interaction considerably increases concurrency since the sender just initiates the interaction. This is useful for all those cases where the sender does not need any result of the task it delegated. From the sender's point of view neither a check for the receiver's actual state is necessary, nor does the sender have to wait for the receiver to finish method execution. Of course, the receiver may not satisfy the request before having entered the appropriate prestate.

Example. Continuing with our running example, reservation transactions can be concurrently submitted to the consortium from any node within the network. The consortium is responsible for delegating such requests to the appropriate motels. Consequently, the `consortium` object asynchronously propagates the message `tryRes` from within the method `newRes` to all motels satisfying the user's request stored within `resObj`, quits its current thread of control and gets ready for further requests. Figure 6 illustrates this scenario. Within the interaction diagram in Figure 6a asynchronous interaction is denoted by a flash line with two arrows, indicating the fact that interaction is done with one or more motels.



(a) Interaction Diagram

(b) State Transition Diagram of `Motel`

Fig. 6. Asynchronous Interaction Between `Consortium` and `Motel`

Figure 6b shows the state transition diagram for class `Motel`. According to it, a `motel` object remains in state `Idle` until it receives the message `tryRes`, which changes the state to `LookingForRooms`. Within that state the motel selects appropriate rooms (cf. synchronous interaction `getRooms` in Section 3.1) and asks whether one of them is available. If not, the motel returns to state `Idle`. If yes, the first available room notifies the motel and, thus, changes the motel's state to `RoomAvailable`. Finally, the

consortium object is asked to confirm or cancel the reservation, and the motel reserves the room within state *Reserving* or immediately returns to state *Idle*, accordingly.

Concrete Solution by using Rules. This asynchronous interaction policy is realized by a rule depicted in Figure 7. The $Esel_C$ specifies an event which is signalled every time a *consortium* object sends the message *tryRes* to a *motel* object from within *newRes* (cf. also Figure 6). The condition is set to *TRUE* since asynchronous interaction requires that the sender has to proceed execution immediately after initiating a request, independent of the receiver's state.

DEFINE RULE async_tryRes **AS**	Rule Name
ON PRE(Consortium, newRes, Motel, tryRes:resObj) **DO**	Condition Event Selector ($Esel_C$)
IF TRUE **THEN**	Condition
ON (**POST** (receiver, notify: from:) **OR** **POST** (receiver, cancelRes:) **OR** **POST** (receiver, confirmRes:)) **OR** ((**POST** (Motel, CLASSMETHOD new) **OR** **POST** (receiver, notify: from:) **OR** **POST** (receiver, cancelRes:) **OR** **POST** (receiver, confirmRes:)); **PRE**(sender, newRes:, receiver, tryRes:)) [left.left.retVal = right.receiver]	Action Event Selector ($Esel_A$)
EXECUTE INSTEAD ASYNC receiver tryRes: resObj.	Action

Fig. 7. Asynchronous Interaction Rule

However, the interaction should be accepted by the `motel` object only if it is in state `Idle`. Thus, the composite event selector $Esel_A$ specifying the acceptability consists of two main components: The first component resembles the $Esel_A$ of synchronous interaction rules. It is responsible for recognizing transitions to the state `Idle`, if the motel is not already in this state. In our example, these transitions are represented by three simple messages which are connected by `OR` operators. The second component is responsible for recognizing the case that the `motel` object is already in the appropriate state `Idle` upon request of `tryRes`. The `motel` object is in state `Idle` either if the object has just been created (classmethod `new`) or if one of the messages which are part of the first component of the $Esel_A$ has been sent. After this, the action has to wait on the request of the triggering method `tryRes`. This is denoted by the sequence operator "`;`". In case that the `motel` object has just been created, a guard ensures that the receiver (`right.receiver`) of the request is identical to that object (`left.left.retVal`). Note that a composite event selector is stored as a tree and its components can be recursively accessed by the keywords `left` and `right`. The *action* of the rule starts the execution of `tryRes` within an asynchronous thread of control (keywords `INSTEAD ASYNC`).

Generic Solution by using Rule Patterns: The Asynchronizer. In analogy to the Blocker rule pattern, the Asynchronizer rule pattern depicted in Figure 8 can be derived from the application-dependent rule shown in Figure 7, accordingly:

DEFINE RULE PATTERN async_<name> **AS**	Rule Pattern Name
ON PRE (<sender>,<senderMethod>, <receiver>,<message0>) **DO**	Condition Event Selector ($Esel_C$)
IF TRUE **THEN**	Condition
ON ((**POST** (receiver, <message>) ['['receiver.state includes: <prestate>']']) { **OR** (**POST** (receiver, <message>) ['['receiver.state includes: <prestate>']']) }*) **OR** (((**POST** (receiver, <message>) ['['receiver.state includes: <prestate> ']']) { **OR** (**POST** (receiver, <message>) ['['receiver.state includes: <prestate> ']']) }*); (sender, senderMethod, receiver, message0))	Action Event Selector ($Esel_A$)
EXECUTE INSTEAD ASYNC receiver message0.	Action

Fig. 8. Asynchronous Interaction Rule Pattern

4 Related Work

There exist different approaches both from the database area and the programming language area related to parameterized rules on the one hand, and to flexible interaction specification on the other hand.

Casati et.al. [Casa95] use rules in the area of workflow management systems. Similarly to [Kapp95ab] rules provide the control and data flow between processing steps, thereby serving as an implementation mechanism for workflow enactment. In addition, *rule templates* are used to define the general structure of rules whose actual code depends on each specific workflow definition. Rule templates are quite similar to rule patterns. They are translated into rules at the time of their application, i.e., when a workflow definition is compiled. Unlike rule patterns where generic components are represented by parameters, the generic components of rule templates are represented by macros. The intention behind rule patterns was to specify business policies and therefore to be independent of a specific application domain. In contrast, the discussion of rule templates is restricted to the domain of workflow management systems.

The *AMOS* active database management system [Risc92, Sköl95] supports *parameterized rules* where the parameter might be either some value or a type. This resembles simple rule patterns consisting of a single rule. However, rule patterns in general go a step further in that they may be composed out of several parameterized rules.

The *composition filter* approach [Aksi93] is an extension to a conventional object-oriented model and offers composable techniques for specifying different aspects of the behavior of objects such as inheritance and delegation, multiple views and roles, synchronization and real-time constraints. New aspects of an object's behavior can be added by introducing new filter types. The behavior of an object can be enhanced through the manipulation of incoming and outgoing messages by using input and output filters. A filter is, similarly to a rule, a first-class object that determines via a *condition*

whether a particular message identified by a *matching part* is either accepted or rejected and what *action* is to be performed in either case. At the conceptual level, composition filters are similar to our approach because they also aim to be adaptable to "constraints and requirements imposed by different application domains" what we call business policies. At the implementation level to the contrary, composition filters are somewhat different. Unlike rules, composition filters are specified statically within a class's definition as instances of a certain filter type. Furthermore, TriGS rules can not only be triggered by message passing events but also by time events and, above all, by composite events. Rule Patterns are comparable to filter types. However, filter types define only the actions which have to be taken when a filter of this type is applied. The situation *when* the action has to be executed has to be specified again and again when applying the filter type to a certain application class. Rule patterns predefine not only the action but also the situation which should trigger the rule independently of a concrete application, thus allowing automatic generation of rules out of the corresponding rule patterns.

The following approaches in object-oriented systems also have intentions similar to our rule patterns, or can possibly be applied for similar purposes. However, coming from the programming and not from the database community they do not consider database aspects such as persistence and transactions. The first two of them have also been presented strictly for sequential programs, but the ideas might be easily adaptable to concurrent systems.

The main purpose of the *Law-governed* approach [Mins87, Mins91] is the expression and enforcement of global regularities within an object-oriented system. It is based on rules, which are typically used to intercept message sends (thus corresponding to ON PRE condition event selectors in TriGS). The collection of all rules belonging to a system is called the Law of the system; rules can pertain to both static and dynamic aspects of the system, and even specify how the Law itself may be changed. The rules are specified in a variant of Prolog, so they can be very powerful. On the other hand, their evaluation can be very expensive (in general, there is no guarantee that the evaluation of a rule will even terminate), so they would be unsuitable for active databases. As a consequence of the global view, message sends can be intercepted only if *every* message sent to every object is passed through the law; in TriGS and similar systems events are caused only by *interesting* message sends. Instead of the parameterization of our rule patterns, the law-governed approach has the conventional unification of logical variables. The Law is always a "flat" collection of rules; there is no structuring facility analogous to composite rule patterns.

The *Contract* [Helm90, Holl92][4] is a construct for explicitly specifying (complex) interactions among groups of objects. A contract is thus analogous to a *composite* rule pattern: both express patterns of behavior that cannot be specified by a method interface or by a single rule. In the development of object-oriented software, contracts are intended to be defined mostly *before* classes; the Demeter system can automatically generate methods for a class from the contracts in which it participates [Lieb96]. This aspect is rather opposite to rules and rule patterns, which are largely intended to specify dynamically changing behavior. Contracts do not allow the behavior of a method to be

4 This is a specific technical meaning different from the well-known idea of "programming by contract" [Meye88].

changed, for instance. However, for the purpose of enforcing certain interactions to occur in a certain sequence, contracts and rule patterns seem to be equally useful.

Transverse activities [Kris93] are a new approach similar to Contracts. Being based on BETA [Lehr93], this work does take concurrency into account. (Any object can be genuinely active in the sense of having its own thread of control, while objects in most active databases could more precisely be called *reactive*.) The continuation of this research may yield results that are useful also for rule patterns.

5 Outlook

The basic idea of rule patterns can be applied to underlying systems that are very different from TriGS. TriGS is a prototype built with small resources, and has some rough edges, in part because the underlying system could not be modified. Good rule patterns can hide a lot of those rough edges. However, most object-oriented database systems are less dynamic and less reflexive than GemStone; in particular, classes in them are not objects themselves. In such a system, it would be sensible to treat rule patterns as static entities, instead of objects. Actually it is not necessary even for *rules* to be first-class objects.

There are some systems, e.g., ODE [Geha92] and Sentinel [Anwa93], in which rules are always declared within class definitions. In the majority of active object-oriented systems as well as in TriGS, rules do not belong to classes but are defined and stored separately from them. This is really the natural choice for rule patterns, which can have a more global purpose than single rules, and where classes can appear as formal parameters. However, when one has patterns, or modules, or other constructs larger than a single class for building the static structure of an object-oriented system, it seems far more promising that also rules and rule patterns could be incorporated in them. We will investigate this issue in the further course of our work.

Finally, let us consider rules and rule patterns within the software life-cycle. At the stage of *analysis*, domain specialists state their requirements more naturally by means of IF ... THEN constructs, i.e., by means of rules. Within *design*, for rules stated during requirements analysis appropriate rule patterns are identified. These rule patterns provide a specification of how the rules found during analysis can be transformed to the design of the system by means of active object-oriented technology. The *implementation* phase comprehends parameter binding on the basis of the rules found during analysis followed by an automatic generation of the corresponding "implementation rules". Since the concept of rules and rule patterns is used within each phase of the software life-cycle traceability of frequently changing requirements for the whole software life-cycle is facilitated.

Future work will include the development of additional rule patterns for business policies other than interaction policies introduced in this paper, and the further implementation of a design environment for active object-oriented applications based on rule patterns. In the long run, designers of active object-oriented applications should not be concerned with decisions on how a recurring design problem is realized by using objects and rules. They should rather be able to compose their applications out of a framework of predefined parameterized rule patterns.

References

[Aksi93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, *Abstracting Object Interactions Using Composition Filters,* Proc. of the ECOOP'93 Workshop on Object Based Distributed Programming, R. Guerraoui, O. Nierstrasz, M. Riveill (eds.), Springer LNCS 791, Kaiserslautern, July 1993

[Anwa93] E. Anwar, L. Maugis, S. Chakravarthy, *A New Perspective on Rule Support for Object-Oriented Databases*, Proc. of the ACM-SIGMOD Int. Conf. on Management of Data, SIGMOD Record, 22 (2), pp. 99-108, June 1993

[Bußl94] C. Bußler, S. Jablonski, *Implementing Agent Coordination for Workflow Management Systems Using Active Database Systems*, Proc. of the IEEE Fourth Int. Workshop on Research Issues in Data Engineering (RIDE), Houston, 1994

[Casa95] F. Casati, S. Ceri, B. Pernici, G. Pozzi, *Conceptual Modeling of Workflows*, Internal Report no. 95.018, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milan, Italy, 1995

[Coad95] P. Coad, D. North, M. Mayfield, *Object Models - Strategies, Patterns & Applications*, Yourdon Press Computing Series, Prentice Hall, 1995

[Copl95] J.O. Coplien, D.C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995

[Diaz91] O. Diaz, N. Paton, P. Gray, *Rule Management in Object Oriented Databases: A Uniform Approach,* Proc. of the 17th Int. Conf. on VLDB, Barcelona, pp. 317-326, 1991

[Ditt95] K.R. Dittrich, S. Gatziu, A. Geppert, *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*, Proc. of the 2nd Workshop on Rules in Databases (RIDS), T. Sellis (ed.), Springer LNCS, Athens, Greece, Sept. 1995

[Eder95] J. Eder, H. Groiss, *Ein Workflow-Managementsystem auf der Basis aktiver Datenbanken (A Workflow Management System Based on Active Databases)*, Geschäftsprozeßmodellierung und Workflow-Management, G. Vossen, J. Becker (eds.), International Thomson Publishing, Bonn, 1995 (in German)

[Embl92] D.W. Embley, B.D. Kurtz, S.N. Woodfield, *Object-Oriented System Analysis - A Model-Driven Approach,* Yourdon Press, 1992

[Fugi92] M. Fugini, O. Nierstrasz, B. Pernici, *Application Development through Reuse: The ITHACA Tools Environment*, ACM SIGOIS Bulletin, 13 (2), pp. 38-47, August 1992

[Gamm94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, 1994

[Geha92] N.H. Gehani, H.V. Jagadish, O.Shmueli, *Composite Event Specification in Active Database Systems,* Proc. of the 18th Int. Conference on VLDB, August 1992

[Gert93] M. Gertz, U.W. Lipeck, *Deriving Integrity Maintaining Triggers from Transition Graphs*, Proc. of the 9th Int. Conf. on Data Engineering (DE'93), IEEE Computer Society Press, Vienna, 1993

[Hare88] D. Harel, *On Visual Formalisms*, Communications of the ACM, 31 (5), May 1988

[Helm90] R. Helm, I.M. Holland, D. Gangopadhyay, *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems*, Proc. of the conjoint OOPSLA/ECOOP Conf., ACM Press, pp. 169-180, Ottawa, Canada, 1990

[Herb95] H. Herbst, *A Meta-Model for Business Rules in Systems Analysis*, Proc. of the 7th Int. Conf. on Advanced Information Systems Engineering (CAiSE'95), J. Iivari, K. Lyytinen, M. Rossi (eds.), Springer LNCS 932, Jyväskylä, Finland, June 1995

[Holl92] I.M. Holland, *Specifying reusable components using Contracts*, Proc. of ECOOP92, O. Lehrmann Madsen (ed.), Springer LNCS 615, Utrecht, 1992

[Kapp94a] G. Kappel, S. Rausch-Schott, W. Retschitzegger, S. Vieweg, *TriGS - Making a Passive Object-Oriented Database System Active*, Journal of Object-Oriented Programming (JOOP), 7(4), July-August, 1994

[Kapp94b] G. Kappel, S. Rausch-Schott, W. Retschitzegger, *Beyond Coupling Modes - Implementing Active Concepts on Top of a Commercial ooDBMS*, Int. Symposium on Object-Oriented Methodologies and Systems (ISOOMS), Springer LNCS 858, 1994

[Kapp95a] G. Kappel, B. Pröll, S. Rausch-Schott, W. Retschitzegger, *TriGS$_{flow}$ - Active Object-Oriented Workflow Management*, Proc. of the 27th Hawaiian Int. Conf. on System Sciences (HICSS'95), 1995

[Kapp95b] G. Kappel, P. Lang, S. Rausch-Schott, W. Retschitzegger, *Workflow Management based on Objects, Rules and Roles*, IEEE Bulletin of the Technical Committee on Data Engineering, 18 (1), March, 1995

[Kapp95c] G. Kappel, S. Rausch-Schott, W. Retschitzegger, *Rule Patterns for Designing Active Object-Oriented Database Applications*, Technical Report, Dept. of Information Systems, University of Linz, June 1995 (also available at http://www.ifs.uni-linz.ac.at/ifs/publications.html)

[Knol94] G. Knolmayer, H. Herbst, M. Schlesinger, *Enforcing Business Rules by the Application of Trigger Concepts*, Proc. Priority Programme Informatics Research, Information Conference Module 1, Swiss National Science Foundation, Bern, 1994

[Kris93] B. B. Kristensen, *Transverse Activities: Abstractions in Object-Oriented Programming*, Object Technologies for Advanced Software, S. Nishio, A. Yonezawa (eds.), Springer LNCS 742, 1993

[Lehr93] O.Lehrmann Madsen, B. Moeller-Pedersen, K.Nygaard, *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993

[Lieb96] K. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, 1996 (to be published)

[Louc91] P. Loucopoulos, C. Theodoulidis, D. Pantazis, *Business Rules Modelling: Conceptual Modelling and Object-Oriented Specifications*, IFIP WG8.1 Working Conf. on the Object-Oriented Approach in Information Systems, F. Van Assche, B. Moulin, C. Rolland (eds.), North-Holland, Quebec City, Canada, October 28-31, 1991

[Meye88] Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988

[Mins87] N. H. Minsky, D. Rozenshtein, *A Law-Based Approach to Object-Oriented Programming*, Proc. of the OOPSLA '87 Conf. (Orlando, Florida, 1987), N. Meyrowitz (ed.), ACM SIGPLAN Notices, 22 (12), pp. 482-493, December 1987

[Mins91] N. H. Minsky, *Law-Governed Systems*, The IEE Software Engineering Journal, September 1991

[Nier93] O. Nierstrasz, *Composing Active Objects*, Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner, A. Yonezawa (eds.), MIT Press, 1993

[Odel94] J.J. Odell, *Specifying requirements using rules*, Journal of Object-Oriented Programming (JOOP), 6 (2), 1994

[Petr94] I. Petrounias, P. Loucopoulos, *A Rule-Based Approach for the Design and Implementation of Information Systems*, Proc. of the 4th Int. Conf. on Extending Database Technology (EDBT'94), M. Jarke, J. Bubenko, K. Jeffery (eds.), Springer LNCS 779, UK, March 1994

[Pree95] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995

[Rein93] B. Reinwald, *Workflow Management in Verteilten Systemen (Workflow Management in Distributed Systems)*, Teubner, Stuttgart-Leipzig, 1993 (in german)

[Risc92] T. Risch, M. Sköld, *Active Rules based on Object Oriented Queries*, IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 15, No. 1-4, Dec. 1992

[Rumb91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[Rubi94] K.S. Rubin, P. McClaughry, D. Pellegrini, *Modeling rules using Object Behavior Analysis and Design*, Object Magazine, June 1994

[Schr95] M. Schrefl, G. Kappel, *Modelling Object Behavior: To use methods or rules or both?* Workshop "Semantics in Databases", Prague, 1995

[Sköl95] M. Sköld, E. Falkenroth, T. Risch, *Rule Contexts in Active Databases - A Mechanism for Dynamic Rule Grouping*, Proc. of the Rules in Database Systems Conference (RIDS'95), Athens, Greece, 1995

[Tsal91] A. Tsalgatidou, P. Loucopoulos, *An Object-Oriented Rule-Based Approach to the Dynamic Modelling of Information Systems*, Proc. of the Int. Working Conf. on Dynamic Modelling of Information Systems, H.G. Sol, K.M. Van Hee (eds.), pp. 165-188, North-Holland, 1991

[Tsic89] D. Tsichritzis, *Object-Oriented Development for Open Systems*, Proc. of Information Processing 89 - IFIP World Computer Congress, G.X. Ritter (ed.), North-Holland, 1989

[Urba92] S.D. Urban, A.P. Karadimce, R.B. Nannapaneni, *The Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database*, Proc. of the 8th Int. Conf. on Data Engineering (DE'92), 1992

[Yone87] A. Yonezawa, E. Shibayama, T. Takada, Y. Honda, *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1*, in: Object-Oriented Concurrent Programming, A. Yonezawa, M. Tokoro (eds.), MIT Press, 1987