Rule Management in Object Oriented Databases: A Uniform Approach

Oscar Díaz†‡
Facultad de Informática†
Universidad del País Vasco
San Sebastián, Spain

Norman Paton§
Computing Science Department§
Heriot-Watt University
Edinburgh, Scotland

Peter Gray‡
Computing Science Department‡
University of Aberdeen
Aberdeen, Scotland

Abstract

Rules have been proposed for providing active behaviour in DBMS. Previous attempts to add rules to Object Oriented DBs have often resulted in a dichotomy between rules and other kind of objects. Here a uniform approach is presented, in which rules are described and handled in the same way as any other object in the system, without any additional mechanisms being introduced. Thus rules can be related to other objects or arranged in hierarchies, and rules can even be defined which are triggered by methods attached to rules themselves. Since rules and classes are both objects, a relationship between these two kinds of objects can be used to provide a class-based index for rules. In this way, the search for applicable rules is considerably reduced. An early implementation and several examples are shown in ADAM, an Object Oriented DB in PROLOG.

Keywords: Active DBMS, Object Oriented DBs, Rule Management, Knowledge Bases, Triggers

1 Introduction

Active databases have been defined as database systems that respond *automatically* to events generated internal or external to the system itself *without user intervention*" [Bauz 90]. System responses are declaratively expressed using event-condition-action rules (ECA rules proposed in [Dayal 88]). ECA rules have an *event* that triggers the rule, a *condition* describing a given situation, and an *action* to be performed if the condition is satisfied. In this way, not only does the system know **how** to perform operations, but also **when** operations have to be performed.

In [BBBC 90] rules are seen as a major feature in future database (DB) systems, and it is remarked that "object-oriented database (OODB) researchers have generally ignored the importance of rules". The research presented here is an attempt to provide insight into rules in an OO context. The focus is on providing **a uniform approach.**

What is meant by a uniform approach is that rules have to be defined and treated in the same way as other objects in the system, without defining any additional mechanisms or auxiliary structures. Rules are seen as "first-class" objects, and are described using attributes and methods. In this way, rule management operations are conceived and implemented as methods. This brings all the advantages of the OO paradigm into rule management: encapsulation, modularity, reusability. In a uniform approach the system should not distinguish rules from other kinds of object. As a result, rules can be related to other objects, and also arranged in hierarchies. Since methods attached to objects can trigger rules, and rules are themselves objects, rules can be defined which are triggered by methods attached to rules. As with any other entity, the meaning of a rule lies in the attributes attached to the rule, and their interpretation by the associated methods. From the point of view of the system however, no distinction should be made. Treating rules as objects also has the advantage that any new facility introduced for objects is automatically applicable to rules (e.g. transaction mechanisms, locking mechanisms, display facilities). This is born out by an implementation in ADAM [Paton 89, Gray 91], an OODB programmed in PROLOG.

Rule evaluation imposes an overhead on every possible event that can be detected by the system. Whereas in relational databases events are generally restricted to be database updates, the approach presented here allows any message to raise an event. Thus, the efficiency requirements for rule support in OO databases are even greater that in relational databases. Here, an attempt is made to enhance system performance by indexing rules by class. A single thread of execution is assumed, and topics such as transactions and optimization are not addressed here.

This paper is organized as follows. A review of related work is given in section 2. In section 3, the components involved in rule management are identified. Issues relating to events in an object oriented context are discussed in section 4. In section 5 the implementation of a rule manager in ADAM is described. Conclusions are presented in section 6.

2 Related work

Research on active behaviour has been conducted in the areas of programming languages, Artificial Intelligence(AI) and DBs. ACTOR [Hewitt 77] was a pioneer programming language in providing objects with active behaviour. Modelling parallel and distributed applications are among the research interests in this area [Ellis 89]. Active behaviour in AI is provided through daemons and active values. So daemons such as *if-needed* or *if-added* are associated with slots to compute their values on demand, or to perform some other test or action.

In relational DBs, active capabilities have been used to enforce integrity constraints, define views, translate update requests and compute derived attributes [Eswaran 75, Stonebr 90, Morgens 84]. In [BBBC 90] rules are seen as a unifying paradigm for providing a broad range of DB facilities. However, in relational DBs, rules are implemented as a distinct layer, and additional mechanisms and structures are required to support rule management.

Several OO systems that support rules are described in the literature [Kotz 88, Dayal 89, Hudson 89, Chakrav 89, Bauz 90]. In [Bauz 90] a review of different mechanisms for supporting rules is given, namely:

- method-based mechanisms: the rule is precompiled into each place in the code where it might be activated. Alternatively, commands could be planted to fire the rule whenever applicable.

- object-based mechanisms: enlarging the object description to indicate which rule to invoke whenever message sending takes place. This is the approach followed in this paper

- external mechanism: additional structures are defined which support checking when some event occurs (e.g. [Bauz 90, Kotz 88])

Several drawbacks can be enumerated for the first approach:

1.- Rules are buried inside methods, and thus it is difficult to enquire about any of the rules attributes, e.g. the condition, the action, or whether it is enabled or not.

2.- Modification of any of the attributes of a rule implies making change in every method supporting the rule.

3.- Since rules can interact, coding of rules within methods requires the programmer to understand all the rules that appear in the method, so that interaction can be handled properly.

4.- The rule definition is scattered in different places, compromising the OO philosophy that encourages all information about a given object to be gathered together.

5.- Method code now includes two things: how the operation itself is implemented and the enforcing of the rule. This severely compromises *method overriding*. Overriding of methods is a useful mechanism in OO systems for customising an operational implementation for special requirements. The problem is that in this case not only is the operation being overridden but also the embedded concept described by the rule (e.g. an integrity constraint).

In [BBBC 90] some of these drawbacks are pointed out and the following conclusion is made: "In our opinion there is only one reasonable solution; rules must be enforced by the DBMS but not bound to any function (i.e. method) or collection". The other two approaches to supporting rules overcome these disadvantages by providing a mechanism supported by the DBMS.

In [Bauz 90] a rule management mechanism is proposed for O_2. Rules are objects having the event as an attribute, and auxiliary structures are defined for storing rule lists which are checked when specific events occur. However, events are not seen as objects in themselves, and thus, system extensibility can be compromised in the sense that composite events or events with special requirements are difficult to introduce. Further, a local mechanism is used to provide rule "inheritance" instead of using a mechanism based on the object hierarchy itself.

In HiPAC [Dayal 88] rules and events are seen as different entities with their own attributes and methods. A sound approach is taken to rule support, paying special attention to transaction management and optimization techniques. However, some of the idiosyncrasies of the OO paradigm have not been considered, such as the primary role that classes play, where methods are part of the class definition.

3 An overview of rule management

The OO paradigm provides a different approach to system design. Whereas procedural design emphasizes the decomposition of the problem into a set of tasks to be executed sequentially, OO design focuses on the entities involved and how they interact. Thus, to provide a rule manager in the context of an OODB, a primary requirement is to identify the significant entities and their interaction.

Briefly described, the function of a rule manager is to provide quick response through the use of *rules*, to *events* generated by some *system*. Three components can be identified in this process:
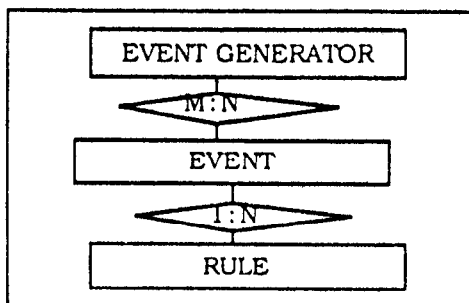
Figure 1: E/R diagram for rule management

- *the rule* describes both when and how the system reacts to an event.

- *the event* is an indicator to signal that a specific situation has been reached to which reactions may be necessary [Kotz 88]. Not all systems consider events as first class objects. For example, events can be treated as simple attribute values. However, this approach can compromise the extensibility of the system to cope with events coming from different sources, or events that need special treatment (e.g. composite events [Dayal 88]).

- *the event generator* can be seen as any system producing events which may need a special response in terms of rule triggering. Events can be generated by the DBMS itself or by any other external system such as a clock or an application program.

Figure 1 shows an Entity-Relationship diagram where these entities are depicted together with the relationships between then. First, a rule can be triggered by an event, but an event can trigger several rules. Second, an event can be generated by several systems and a system can generate several events.

The main interaction between these entities can be described as follows:

1.- an event is produced by any event-generator, and is signalled to the event manager through the message *signal*,

2.- the event manager checks if any rule can be triggered by the event signalled. If so, it sends the message *fire* to the appropriate rules,

3.- when the message *fire* is received by a rule, the rule condition is then checked and if satisfied the rule action is executed

Other kinds of interactions are also possible, such as "awakening" of events as a result of rule creation.

In the following sections the object rule and the object event are defined. Event generators have not been described as objects, although conceptually they are seen as the senders of the signals.

4   Events in an object oriented context

An event is an indicator to signal that a specific situation has been reached to which reactions may be necessary. In relational DBs, an event can be described by the operation together with the moment when this operation takes place (i.e. before or after). For instance, the pair *(insert, before)* could specify that the event arises *before* the operation *insert* occurs. In this context, OODBs present some differences from relational DBs. In OO systems, operations (i.e. methods) are not isolated but are part of the class definition. The class is not just an argument of the method, but the method itself is subordinated to the class. As a result, the same method name can be implemented in different ways in distinct classes, the process known as overloading, or a method can be specialized down the hierarchy by any subclasses, thereby revising the behaviour of the superclass. Now, let us consider the situation shown in figure 2 where an integrity rule to prevent students from being older that ninety is defined. This rule should be fired for instances of the class *student*, *before* the message *put-age* begins execution, i.e. before the student age is altered. Since OO systems allow methods to be inherited from superclasses, the method *put-age* can be defined at the level of the class *person* and inherited by the subclass *student*. Thus, the previous integrity rule has to be invoked not only when an attempt is detected to insert the *age* of a *person*, but also when this *person* happens to be a *student*. Otherwise, the rule should not be invoked even if the message *put-age* is detected.

In other words, the method alone does not completely specify the context of invocation, since a method gets its meaning from a class (subsequently called the active class). Several alternatives are possible for supporting the idea of an active class, for instance:

1.- The active class could be embedded in the condition part of a rule. For example, the previous rule would have as an event the pair *(put-age, before)* and the condition part of the rule would be extended to check that the receiver of the message is an instance of the class *student*. Besides making the context where a rule is invoked difficult to understand, this approach prevents the system from taking full advantage of the active class as an indexing mechanism, as shown later.

2.- The event definition could be enlarged with an active class attribute. Thus, in the previous example, the event would become *(student, put-age, before)*. However the message receiver can be an instance of some subclass (e.g. *postgraduate*), and thus the active class is not its immediate class. Two options are now possible. One is to check if the message receiver is an instance of the ac-
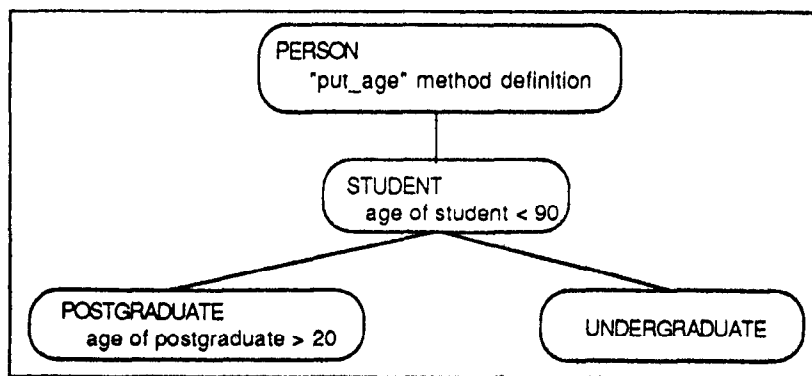
Figure 2: Person hierarchy

tive class (i.e. *student*). This process can turn out to be quite expensive since this checking has to be done for every message sent and for every possible event. Another approach is to generate automatically all possible "inherited" events. For instance, the events *(postgraduate,put-age,before)* and *(undergraduate,put-age,before)* would be generated, providing that *postgraduate* and *undergraduate* are subclasses of *student*. It is worth mentioning that some of the generated events may already be defined (e.g. an integrity rule constraining postgraduate students to be older that twenty). In this case, instead of creating a new event, the set of rules activated by this event has to be extended by the identifier of the *younger-than-ninety* rule. Moreover, if a new subclass is introduced, the appropriate events have to be generated. For instance, if *phd-student* is introduced as a subclass of *postgraduate*, all the events for postgraduate students have to be "inherited" by PhD students. This process can be quite cumbersome and expensive to maintain. In our opinion, the rule identification process should make use of the class hierarchy itself, rather than making use of some additional mechanism.

3.- The rule definition is extended with an active-class attribute. Previous work either does not consider explicitly the role played by the active class, or provides a local mechanism for "inheriting" events. Since rules are truly objects, the extra active-class attribute can be implemented as a two-way relationship between rules and classes. The inverse of *active-class* is declared to the system to be held in the class-rules attribute of a class. For instance, the *younger-than-ninety* rule would have *student* as the value of its *active-class* attribute, and thus the *student* class would have the object identifier of this rule as the value of its *class-rules* attribute. Two important advantages can be drawn from this approach:

 • Rules are indexed by class. The *class-rules* attribute has as its value the set of rules to

be verified when a message is sent to any instance of this class. In this way the search for applicable rules is considerably reduced.

 • The "inheritance" of rules has been moved to the class hierarchy, without defining any additional mechanism. As discussed above, the rules affecting a given instance are not just the ones attached to its immediate class, but also those attached to its superclasses. For example, if the message *put-age* is sent to an instance of the class *postgraduate*, the rules applicable (e.g. representing integrity constraints on the *age* attribute) are those attached to *postgraduate* itself together those attached to *student* and *person*. To handle this situation, the definition of each class has been enlarged with the attribute: activated-by. This attribute is defined just like any other attribute:

```
attribute(att_tuple(activated_by,
    global,set,optional,rule-class,
    [activated_by wof class ::
        class_rules of class
        union
        activated_by of is_a of class]))
```

This definition states that *activated-by* contains objects of type *rule-class*. The constraint, enclosed between brackets and specified using a constraint equation approach [Morgens 84], enforces that the value of the *activated-by* attribute for a given class has to be equal to the union of the rules of the *class-rules* attribute attached to the class and the rules obtained from the *activated-by* attribute attached to its superclasses. It is worth noticing the recursive nature of this constraint. Together with the idea of *weak bound* [1] proposed in [Morgens 84], this accomplishes the right behaviour: when an up-

---

[1] A weak bound -syntactically represented as *wof-* can be seen as the link to be broken to preserve the constraint in case the equality is violated

date is done to the *class-rules* attribute of any class, the update is propagated to the *activated-by* attribute of all its subclasses. This is done automatically by the system as a result of the enforcement of the above constraint, without any further mechanism being required. Further, when a new subclass is introduced, the appropriate rules are "smoothly inherited".

The latter approach is described in detail in the next section, where a rule manager is described for ADAM, an OODB implemented in PROLOG.

5 A uniform approach to rule management in ADAM

5.1 A brief review of ADAM

The OO paradigm encourages reuse and modularity through the subclass mechanism. When a new class has to be introduced in the system, the designer thinks about the differences between, and similarities with existing classes, pointing out what is really new and what can be reused. Although this philosophy has been broadly utilised user-applications, few systems apply it to the definition of the system itself. In ADAM this philosophy is also applied to the definition of the system by the use of **metaclasses**. A metaclass is a class the instances of which are all classes. Metaclasses not only permit classes to be stored and accessed using the facilities of the data model, but make it possible to refine the default behaviour for class creation using specialization and inheritance. In this way, uniformity and extensibility are greatly increased. As a case in point, this paper is about extending ADAM with a rule manager. The use of metaclasses in ADAM is described in more detail in [Paton 90].

In ADAM, objects are considered to be metaclasses, classes or instances. When the system is compiled, the metaclass called *meta-class* already exists. All subsequent classes are created by sending messages to metaclasses such as *meta-class*, which define methods such as *new*, *put-slot* and *put-method*.

New objects, whether they are metaclasses, classes or instances, are created in ADAM by sending the message *new* to the class of which the object is to be an instance. For example, to create a new class called *person* which is an instance of the metaclass *entity-metaclass*, the call shown in figure 3 is made: The argument of *new* is a PROLOG list, the first element of which is the name of the object, and the second element of which is a list of the attributes of the object. An attribute has a name and it is described by several facets: the visibility, the cardinality, the status, the type and the constraints attached to this attribute (empty list in the above example). Methods to retrieve

(get-), to delete *(delete-)* and to change *(update-, put-)* attribute values are automatically created by the system, so that attributes are always handled by these methods.

When *new* is used to create an instance rather than a class, the first element in the list passed to *new* is unified with the system-generated unique identifier of the object, e.g. 4@person [2]. For example, to create an instance of the class *person* in the variable OID, the following message is sent:

```
new([OID,[
        cname([odile]),
        sex([female]),
        born-in([usurbil])
]]) => person.
```

In this way, the same paradigm is used to handle both data and meta-data.

5.2 The event object

Events are not always seen as first-class objects. In [Bauz 90], events are seen as rule attributes, and hence they cannot have attributes or methods of their own. Although this approach may result in performance gains, it can compromise the extensibility of the system for coping with events coming from different places, or which need special treatment.

As with other objects in the system, event definition involves the description of structure (i.e. attributes) as well as behaviour (i.e. methods). An event can be seen as specifying the moment when a rule is to be fired. This moment can be described by the message firing the rule (the **active-method** attribute of the event) and the status of the message (the **when** attribute of the event).

Unlike some previous approaches, a richer and more complex event definition can be created as a result of working in an OO environment, namely:

1.- Events are not restricted to be update operations but can be any message defined in the system (e.g. display, create a new class, move, get-cname).

2.- The possible values describing the status of a message can be enlarged. Previous work has considered just two values: *before* and *after* operation execution. In the OO context, operations are materialized by methods. Besides, *before* and *after*, the range of values has now been extended to take into account situations where the method cannot be found, or other options which reflect the nature of method invocation supported by the underlying

---

[2] The identifier 4@person is an internal identifier. In practice one would use a variable *Baby*, which had been instantiated by another goal, e.g. *get-by-cname([odile],Baby)* => *person*, instantiates Baby with the object identifier of the *person* whose name is *odile*

```
new([person,[
      attribute(att_tuple(cname,global,single,total,string,[]),
      attribute(att_tuple(sex,global,single,total,string,[]),
      attribute(att_tuple(born_in,global,single,optional,string,[])
]]) => entity_metaclass.
```

Figure 3: The definition of the class PERSON in ADAM.

PROLOG evaluation strategy (e.g. backtracking into a method). [Diaz 91a] This broader spectrum of situations attempts to reflect the core role that methods play in OO systems and the variety of situations that can arise during message sending.

In other approaches, the event description includes the arguments to be passed when a rule is fired. In our approach, all the methods' arguments, regardless of whether they are input or output parameters, are passed by the system without any previous declaration. The rule manager makes these arguments available to the condition and action part of the rule through the system-defined predicate *current-arguments*. Examples are given in the next section. Being objects, events can be related to other objects (e.g. with the rules that a given event activates), or arranged in hierarchies. This allows the system to be enlarged to cope with later extensions. In figure 4 a hierarchy is shown where events can be classified into DB events, clock events or application events, depending on the event source. New attributes or specializations of existing methods can be included if required.

Just as with any other object, events can be manipulated and signalled by some event generator as well as created, modified or deleted in a *uniform* fashion. For example, an event can be created by sending the following message:

```
new([OID,[
        active_method([put_age]),
        when([before])
]]) => db_event.
```

This event is raised *before* the method *put-age* is executed.

The classes *db-event*, *clock-event* and *application-event* share some attributes and behaviour which are abstracted at a higher level in *event-class*. Moreover, the procedure which takes place when a new event is created is the same as the one followed for the creation of any other object (e.g. instances of *person*). Nor are the deletion or modification methods distinguished from those used for deleting or modifying other objects. As a result, this behaviour can be inherited from that already provided by the system, i.e. from the *entity-metaclass*.

5.3 The rule object

Rule structure is mainly described by *the event* that triggers the rule, *the condition* to be checked and *the action* to be performed if the condition is satisfied. The condition is a set of queries to check that the state of the database is appropriate for action execution. The action is a set of operations that can have different aims, e.g. enforcing of integrity constraints, user intervention, propagation of methods, etc. Condition and action definitions can refer to the current object to which the rule is applied and to the current arguments of the method firing the rule.

As discussed in the last section, the complete context of invocation is described by the active-class and event attributes. The *event* attribute has as its value the object identifier of an event instance. For instance, the *younger-than-ninety* rule can be defined as:

```
new([OID,[
    event([3@db_event]),
    active_class([student]),
    is_it_enabled([true]),
    disabled_for([1@student,23@student]),
    condition([(
      current_arguments([StudentAge]),
      StudentAge > 90
    )]),
    action([(
      current_object(TheStudent),
      current_arguments([StudentAge]),
      get_cname(StudentName) => TheStudent,
      writeln(['The student ',StudentName,
              'with age ',StudentAge,
              'exceeds the expected age']),
      fail
    )])
]) => integrity_rule.
```

If *3@db-event* is the object identifier of the event shown in the last section, this rule will fire *before* executing the *put-age* method. The condition checks whether the argument of the method (i.e. the age to be introduced) is greater than ninety or not. If the condition is not met, the rule is not applicable and the method can continue. Otherwise, the action is executed. In this case, the action fails after displaying a message, and
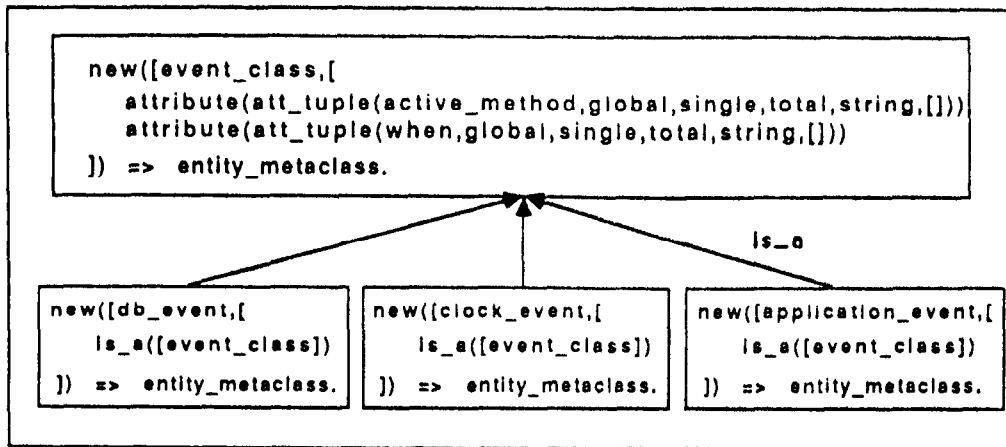
```
new([event_class,[
    attribute(att_tuple(active_method,global,single,total,string,[]))
    attribute(att_tuple(when,global,single,total,string,[]))
]) => entity_metaclass.
```

is_a

```
new([db_event,[
    is_a([event_class])
]) => entity_metaclass.
```

```
new([clock_event,[
    is_a([event_class])
]) => entity_metaclass.
```

```
new([application_event,[
    is_a([event_class])
]) => entity_metaclass.
```

Figure 4: Event hierarchy definition in ADAM

ENTITY_METACLASS
methods: new

SPECIAL_RULE_METACLASS
methods: new (specialized)

is_a

instance

RULE_CLASS
methods: fire
attributes:

event,
active_class,
condition,
action,
is_it_enabled,
disabled_for

instance

USER_DEFINED RULE

INTEGRITY RULE
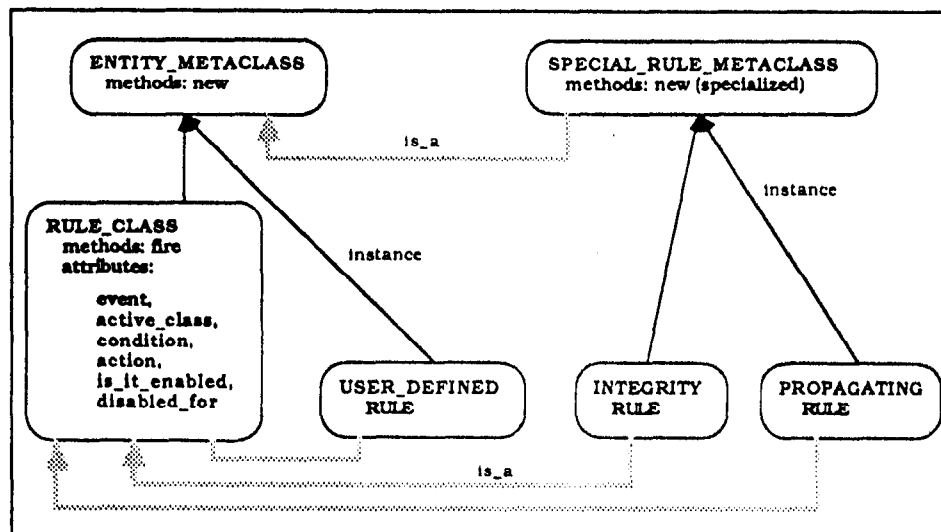
PROPAGATING RULE

is_a

Figure 5: Rule hierarchy

then the invocation of *put-age* does not proceed. The *current-object* and *current-arguments* predicates refer to the current instance to which the rule is applied, and to the current arguments of the method firing the rule respectively. Also the condition result can be passed to the action part through the *condition-result* predicate, the argument of which is instantiated with any value required after condition evaluation.

As well as the event-action-condition description, two more attributes are added to specify the status of the rule itself, i.e. whether the rule is enabled or disabled. The attribute is-it-enabled describes the status at the level of the whole class appearing as the *active-class* value, whereas the disabled-for attribute describes the status for specific instances of the class. In the above example, the rule is enabled for all the students (i.e. the value of *is-it-enabled* is *true*) except for those instances with object identifier 1@student and 23@student. Thus, this rule will not be fired if either

the *is-it-enabled* attribute is *false* or if the object identifier of the current object appears as one of the values of the *disabled-for* attribute.

As part of their structural description, rules can be related to other objects in the system and arranged in hierarchies. Actually, the *active-class* attribute is a relationship between classes and rules that has been used to speed up the system: the inverse of *active-class*, i.e. *class-rules* attribute, is used as a class-based index where the inverse constraint is maintained by the system [Diaz 90]. Other relationships can be defined, even between rules themselves, e.g. a precedence relationship in the order of execution. Arranging rules in hierarchies brings all the advantages of inheritance into rules. In figure 5 the *user-defined-rule*, *integrity-rule* and *propagating-rule* subclasses are shown. User-defined rules are those defined by the user, whereas integrity rules and propagating rules are system-generated rules, namely rules generated

by the system from a declarative specification of integrity constraints and the operational semantics of relationships respectively [Diaz 91a]. Rules for constraint maintenance are an interesting example where several methods and classes can be involved. For instance, to preserve that *the age of a PhD student has to be smaller that the age of his/her supervisor*, methods modifying the *age* (i.e. *put-age*, *delete-age* and *update-age*) either for a *student* or for a *lecturer* have to be considered, assuming that the class *lecturer* is defined and that only lecturers can supervise PhDs. An approach to derive rules for constraint maintenance in this context is presented in [Diaz 91b].

The next question to be addressed is the behaviour of rules. Unlike other objects, rules can be fired, i.e. the condition of the rule evaluated, and if accomplished then the action undertaken. In order to be inherited for all the instances, this rule firing method is defined at the level of *rule-class*. The method **fire** can be specialized to account for further requirements in any subclass. Enabling and disabling of rules is managed through modification of the *is-it-enabled* and *disabled-for* attributes, and no special methods are required.

Finally, rule management is done using the mechanism already provided by the system for handling other kinds of objects. However, special requirements are needed when instances of *integrity-rule* and *propagating-rule* are created and hence, the method *new* has to be specialized for these classes. Owing to the metaclass mechanism available in ADAM, this specialization can be easily and cleanly supported by defining the *special-rule-metaclass*. This situation is shown in figure 5. All rule classes are handled in the same way. However, when the message *new* is sent to the *user-defined-rule* class, the "standard" definition of *new* is inherited, whereas a specialized definition is used when this message is sent to the *integrity-rule* or *propagating-rule* classes.

5.4 Some examples

Two examples are shown in this section, illustrating the use of rules to implement security constraints and operations on derived classes. For legibility, the event attribute of a rule is substituted by the attributes of the event object that would fill this attribute.

In the first example the advantage of following a uniform approach can be seen. Since rules are objects, rules can be defined on the rules themselves. In figure 6 a rule is shown that prevents users other than *graham* from creating user defined rules. It is thus a rule about rules. When an attempt is made to create a rule by sending the message *new* to *user-defined-rule*, this rule is fired, and the identity of the user checked through *current-user*, a predicate which returns the name of the current user.

```
new([_,[
    active_class([user_defined_rule]),
    active_method([new]),
    when([before]),
    is_it_enabled([yes]),
    condition([(
        +\ current_user(graham),
        writeln('You are not authorized to
                 create user defined rules')
    )]),
    action([((fail)]))
]]) => user_defined_rule.
```

Figure 7: Security constraint rule

The second example illustrates the use of rules in metaclasses. Since metaclasses are objects, the rule mechanism can be used to accomplish meta-behaviour. In the SDM [Hammer 81] semantic data model, a class can be derived from another class based on some dynamic criteria. For instance, the *phd* class can be derived from the *postgraduate* class where the criteria could be that the *registration* is "phd". Instances of the *phd* class can be obtained from postgraduates by selecting those whose *registration* = "phd", and operations on the class *postgraduate* have to be "propagated" to *phd*. To provide this kind of behaviour a rule can be defined for each operation to be "propagated" [Amy 89]. In figure 7 a rule is shown to provide this mechanism for the message *new*. When a new postgraduate is created, i.e. when the event *(entity-metaclass,new,before)* is signalled, the condition part of the rule checks if the current instance of *entity-metaclass* (e.g. the *postgraduate* class) has any derived class. If so, the action part of the rule checks if the criteria is verified (through the message *verifying-membership* that has as its argument, the attributes of the new instance) and if satisfied, the corresponding event is signalled. In this way, an improvement in transparency is achieved: the user is unaware whether a class is derived or not. For instance, rules can be defined to fire when a new *phd* instance is created regardless of whether the message *new* was originally sent to the *postgraduate* class.

6 Conclusion

Unlike current DBs, active DBs aim to provide automatic answers to events generated internal or external to the system itself. System responses are declaratively expressed through event-condition-action rules. The research presented here is an attempt to provide an insight into rules in an OO context, stressing uniformity.

Uniformity stems from seeing rules as "first-class" objects described using attributes and methods. In this way, rule management operations are conceived and

```
new([_,[
    active_class([entity_metaclass]),
    active_method([new]),
    when([before]),
    is_it_enabled([yes]),
    condition([[
        current_object(StoredClass),
        condition_result(DerivedClasses),
        findall( DerivedClass,
                 get_by_is_a([StoredClass],DerivedClass) => derived_metaclass,
                 DerivedClasses),
        DerivedClasses \== [ ]
    ]]),
    action([[
        % an event has been detected for a class
        % the same event has to be signalled for each of its derived subclasses
        current_arguments([Args]),
        Args = [_,Atts],
        condition_result(DerivedClasses),
        (member(DerivedClass, DerivedClasses),
          verifying_membership(Atts) => DerivedClass,
          signal([event_tuple(new,before,DerivedClass),Args]),
        false ; true)
    ]]
]]) => user_defined_rule.
```

Figure 6: Derived class rule

implemented as methods. This brings all the advantages of the OO paradigm into rule management. As a result, rules can be related to other objects or arranged in hierarchies, and rules can even be defined which are triggered by methods attached to rules themselves. Treating rules as objects also has the advantage that any new facility introduced for objects is automatically applicable to rules.

Although it has not been the main concern of this paper, efficiency plays a decisive role in active DBs. Several benchmarks have been performed to measure the overhead imposed by the rule management system. The results show that the introduction of rules makes programs on average about *twice* as slow as they are when the rule mechanism is disabled. Such a slow-down is predictable, as rule evaluation imposes an overhead on every possible event that can be detected by the system. However, the *scale up factor* (i.e. how the number of rules affects system performance) has been kept *low* by indexing rules by class. In this way the search for applicable rules is considerably reduced.

Acknowledgements

References

[Amy 89]   I. Amy Chen and D. McLeod *Derived Data Update in Semantic Databases*, 15th. VLDB, pp.225-2235, 1989

[Bauz 90]   C. Bauzer Medeiros and P. Pfeffer *A mechanism for Managing Rules in an Object-oriented Database*, Altair Technical Report

[BBBC 90]   D.Beech, P. Bernstein, M. Brodie, M. Carey, B. Lindsay, L. Rowe and M. Stonebraker *Third-generation data base system manifesto*, in Proc. IFIP TC-2 Conf. on Object Oriented Databases, Kent and Mersman (eds.), North-Holland, 1990

[Chakrav 89]   S. Chakravarthy *Rule Management and Evaluation: an Active DBMS Perspective* SIGMOD RECORD, Vol. 18, No. 3, pp. 20-28, 1989

[Dayal 88]   U. Dayal, A.P. Buchmann and D.R. McCarthy *Rules Are objects Too: A Knowledge Model for An Active, Object Oriented Database System* in Proc. 2nd. Int. Workshop on OODBS, K.R. Dittrish (Ed.), Spring-Verlag, pp. 129-143, 1988

[Dayal 89]   U. Dayal *Active Database Management Systems* SIGMOD RECORD, Vol. 18, No 3, pp. 150-169, 1989

[Diaz 90]   O. Diaz and P.M.D. Gray *Semantic-rich User-defined Relationship as a Main Constructor in Object Oriented Databases* in Proc. IFIP TC-2 Conf on Object Oriented Databases, Kent and Mersman (eds.), North-Holland, 1990

[Diaz 91a]   O. Diaz and N.W. Paton *Sharing Behaviour in an Object Oriented Database using a rule-based mechanism* in Proc. 9th. British National Conference on Databases, BNCOD'91, Wolverhampton (United Kingdom), Butterworth Publishers, 1991

[Diaz 91b]   O. Diaz *Deriving Rules for Constraint Maintenance in an Object Oriented Database* submitted for publication, 1991

[Ellis 89]   C.A. Ellis and S.J. Gibbs *Active Objects: Realities and Possibilities* in Object-Oriented Concepts, Databases and Applications, W.Kim and F.H. Lochowsky (ed.), ACM Press, 1989

[Eswaran 75] K.P. Eswaran and D.D. Chamberlin *Functional Specifications of a Subsystem for Database Integrity* in Proc. 1st VLDB, pp. 48-68, 1975

[Gray 91]   P.M.D. Gray, K.G. Kulkarni and N.W. Paton *Object Oriented Databases: A Semantic Data Model Approach*, Prentice-Hall, 1991

[Hammer 81] M. Hammer and D. McLeod *Database description with SDM: A Semantic Database Model*, ACM Transactions on Database Systems, 6,3 (Sept), pp. 351-386, 1981

[Hewitt 77] C. Hewitt, *Viewing Control Structures as Patterns of Passing Messages*, Artificial Intelligence, Vol.8, pp. 323-364, 1977

[Hudson 89] S. Hudson and R. King *Cactis: a self-adaptive, concurrent implementation of an object-oriented database management system* in Proc. ACM SIGMOD, pp. 237-246, 1990

[Kotz 88]   A.M. Kotz, K.R Dittrich, J.A. Mulle *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism* in Advance in Database Technology, EDBT, Venice, pp. 76-91, 1988

[Morgens 84] M. Morgenstern *Constraint Equations: Declarative Expression of Constraints*

with Automatic Enforcement, Proc. Inter. Con. on VLDB, pp. 153-299, 1984

[Paton 89]   N.W. Paton *ADAM: An Object-Oriented Database System Implemented in Prolog*, Proc. 7th. BNCOD, M.H. Williams (ed.), CUP, pp. 148-161, 1989

[Paton 90]   N.W. Paton and O. Diaz *Metaclasses in Object-Oriented Databases* in Proc. IFIP TC-2 Conf on Object Oriented Databases, Kent and Mersman (eds.), North-Holland, 1990

[Stonebr 90] M. Stonebraker, A. Jhingram, J. Goh and S. Potamianos *On rules, procedures, caching and views in database systems* in Proc. ACM SIGMOD, pp. 281-290, 1990