The VLDB Journal
© Springer-Verlag 1997

EXACT: an extensible approach to active object-oriented databases

Oscar Díaz, Arturo Jaime

Departamento de Lenguajes y Sistemas Informáticos, Universidad del País Vasco / Euskal Herriko Unibertsitatea, Apd. 649, 20080 San Sebastián, Spain; e-mail:<jipdigao,jipjaela>@si.ehu.es

Edited by Y. Vassiliou. Received May 26, 1994 / Revised January 26, 1995, June 22, 1996 / Accepted November 4, 1996

Abstract. Active database management systems (DBMSs) are a fast-growing area of research, mainly due to the large number of applications which can benefit from this active dimension. These applications are far from being homogeneous, requiring different kinds of functionalities. However, most of the active DBMSs described in the literature only provide a *fixed, hard-wired* execution model to support the active dimension. In object-oriented DBMSs, event-condition-action rules have been proposed for providing active behaviour. This paper presents EXACT, a rule manager for object-oriented DBMSs which provides a variety of options from which the designer can choose the one that best fits the semantics of the concept to be supported by rules. Due to the difficulty of foreseeing future requirements, special attention has been paid to making rule management easily extensible, so that the user can tailor it to suit specific applications. This has been borne out by an implementation in ADAM, an object-oriented DBMS. An example is shown of how the default mechanism can be easily extended to support new requirements.

Key words: Active DBMS – Extensibility – Object-Oriented DBMS – Metaclasses

1 Introduction

Database management systems are at the heart of current information system technology. They provide reliable, efficient and effective mechanisms for storing and managing large volumes of information in a multiuser environment. In recent years, there has been a trend in database research and practice towards increasing the proportion of the semantics of an application that is supported within the database system itself. Temporal databases, spatial databases, multimedia databases and database programming languages are examples of this trend. *Active databases* can be considered as part of this tendency, where the semantics that are supported reflect the *event-driven behaviour* of the domain.

Event-driven behaviour, unlike call-driven behaviour which is executed when explicitly invoked, is considered as a reaction to the occurrence of an event such as a data manipulation operation, an interruption, a clock signal, etc. Such an event signals the potential occurrence of a relevant situation to which reactions might follow. Non-active databases rely on embedding the checking for this relevant condition either in application programs updating the database or in a special-purpose program which polls the database periodically to check the database state. The former approach leads to redundancy, distribution and difficult maintainability as a result of the check being replicated and/or distributed among several application programs. As for the second approach, the drawback lies in ascertaining the polling frequency: if too high, a great overhead can be caused; if too low, the relevant condition may not be detected in time, a fact which could be crucial for some applications.

Active databases support the above application by moving the reactive behaviour from the application (or polling mechanism) into the DBMS. This implies that the active DBMS has to provide some mechanism for users to describe the reactive behaviour (generally referred to as the *knowledge model*), as well as support for monitoring and reacting to relevant circumstances (generally referred to as the *execution model*)

Event-driven behaviour can support traditional database functionality (e.g. integrity constraint maintenance, support for derived data, monitoring of data access and evolution, access control), as well as application-based tasks (e.g. network management, air-traffic control, tracking, etc). Making DBMSs active allows a broad range of applications to be moved from user programs or some sort of polling mechanism to the database. However, this diversity makes it difficult to find a common knowledge model and execution strategy which is suitable no matter what application is to be supported. Thus, as the number of applications requiring event-driven behaviour increases (and this seems to be the tendency), flexibility become an essential feature of future active DBMSs.

As for the knowledge model, event-condition-action rules (hereafter ECA rules) have been proposed for both object-oriented (OO) and relational systems, both commercial (e.g. ORACLE, INFORMIX, INTERBASE) and advanced (Widom and Ceri 1996). 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 (as in OO databases), but also *when* operations have to be performed. In OO databases, ECA rules have been modeled as *first-class* objects, and as such they benefit from the advantages of the OO approach (Dayal et al. 1988, Díaz et al. 1991). Among them, specialization is the most useful in this context.

As for the execution model, requirements can vary depending on the concept supported by the rule. The semantics of this concept will influence, for example, whether all rules reacting to a given event should be fired when this event is detected, or what reaction follows once a rule fails. However, most of the systems described in the literature only provide a *fixed* execution model for rules, hardwired in the code [e.g. ODE (Gehani and Jagadish 1991), O_2 (Medeiros and Pfeffer 1990), Starburst (Widom et al. 1991), Chimera (Ceri et al. 1996), SAMOS (Gatziu and Dittrich 1994), Sentinel (Chakravarthy et al. 1994)]. This can lead to ad-hoc solutions or even worse, to complicate the condition part of the ECA rule with elements whose only purpose is to achieve the required solution[1].

This paper presents EXACT, an EXtensible approach to ACTive OO DBMSs, which allows the user to choose the execution model that best fits the semantics of the concept to be supported. Due to the difficulty of foreseeing future requirements, special attention has been paid to making execution model flexible enough, so that the user can tailor it to suit specific applications.

Two contentions support this work, namely:

1. It is the user who, besides defining the rules, should specify how these rules have to be executed. The user knows the semantics of the concept to be supported which includes not only the definition of the basic rules but also control information about how these rules have to be exploited.
2. Control information rarely refers to individual rules, but is shared by a set of rules supporting the same concept or functionality (e.g. integrity constraint maintenance, derived data, etc). Hence, this control information should be removed from single rules and moved to a higher level.

Thus, our aim is first to find a set of dimensions to characterize the execution model and second, a mechanism exhibiting the following features:

– *flexibility:* different sets of rules can have different execution models, depending on the semantics of the concept to be supported,
– *declarativeness:* to encourage user implication, the execution model definition should be specified through a set of parameters rather than by encoding how the desired execution strategy is actually achieved,
– *extensibility:* the standard execution model can be easily extended by refining existing features or introducing new ones.

To provide such features, a mechanism based on metaclasses has been used. Metaclasses allow the advantages of the OO paradigm to be applied to the DBMS, since the system itself is described using classes and methods. In this way, not only are rules described using an OO approach but so is the rule-processing strategy.

These ideas have been borne out by EXACT, a fully developed system built on top of ADAM (Paton 1989, Díaz and Paton 1994), an OODBMS implemented in ECLiPSe Prolog[2] with disk-resident and multiuser features provided through MegaLog (Bocca 1991).

It is the successor of a rule system described in Díaz et al. (1991), where the focus is on providing rules as *first-class* objects: rules are defined and treated as any other object in the system. Here however, the main issue is not rule definition as such but the rule's execution model, where topics such as scheduling, conflict resolution strategies or coupling modes are addressed. The advantages of the approach are illustrated by an example, where an initial rule management strategy is enlarged to support new requirements.

The rest of this paper is organized as follows. The need for flexible rule systems is first presented. Section 3 addresses the description and support for the knowledge model. However, the main contribution of the paper is based on the description of the execution model and its implementation in EXACT, both topics covered in Sect. 4. An example illustrating the advantages of the approach is shown in Sect. 5. Related research is the focus of Sect. 6. Finally, conclusions are presented.

2 Why flexible rule systems?

Flexibility means the capability of a system to cope with different application requirements. No drastic changes should be involved – the basic model should be retained, but extensions made to the core to cater for new requirements. To motivate the need for flexibility, a set of applications exhibiting event-driven behaviour are described. The emphasis is upon the different requirements posed to the active mechanism by what needs to be represented (i.e. the knowledge model) and in how it has to be handled (i.e. the execution model). For a more complete account and examples see Paton et al. (1993).

2.1 Integrity constraint maintenance

• **Purpose.** Integrity constraints can be seen as restrictions which must hold among different pieces of information to keep the database consistent. Constraint maintenance was one of the first applications to be supported using active mechanisms. Work has been described both for the relational model (Ceri and Widom 1990) and the OO model (Urban and Desiderio 1991; Díaz 1992).

[1] A similar situation arises in production systems where the simplicity of the recognize-and-act cycle forces the user to extend the rule condition with control information to achieve, for instance, iterations or sequences of production rules.

[2] ECLiPSe is a trademark of ECRC (European Computer-Industry Research Center).

• **Requirements for the knowledge model**

- description of the integrity constraint to be monitored,
- support for exceptions, i.e. objects for which the integrity constraint does not apply. CAD/CAM applications commonly require this feature where the constraint applies to the whole class, except for some given objects (Buchmann and Dayal 1988),
- description of the possible reaction to follow if the constraint is violated. This can be abortion of the transaction, a compensating action to restore the database to a valid state, or just a warning depending on the level of importance of the constraint.

• **Requirements for the execution model**

- *all* integrity constraints affected by a database update should be checked in turn *until one is falsified*. Once a constraint is violated, there is no point in prolonging the checking process,
- due to the former point, if different integrity constraints have to be checked, an order can be established based on the complexity of each constraint: the higher its complexity, the lower its priority. This assumes that each false check aborts the transaction,
- once a database update occurs which affects one or more constraints, the integrity checking can begin either immediately, be deferred till the transaction ends, or both. The appropriate option depends on the kind of constraint to be maintained,
- if a constraint is violated and no repairing action is available, the transaction must be aborted.

2.2 Derived-data support

• **Purpose.** The term *derived data* can be defined as data which is obtained from other data (i.e. the derivers) through calculations, but which looks like they are stored. Derived data has been used to support multiple views, capture data semantics or hide database evolution. Recently, some authors have proposed active rules as a mechanism for supporting derived data. In Ioannidis and Sellis (1992), a proposal is made for a general framework for the study of conflict resolution when alternative derivation criteria are available. In Etzion (1993), a rule approach is also used. Here, the focus is on finding flexible consistency modes between the derived data and its derivers.

• **Requirements for the knowledge model**

- the derivation expression involving the derived data and the derivers

• **Requirements for the execution model**

- if the derived data can be calculated following different expressions, only one should be used. The choice can be based on accurateness and reliability,
- if, once a data expression is selected, the process runs into problems (e.g. the rule fails due to a lack of suitable data), the system can try the next derivation expression, if available,
- if derived data is materialized (i.e. stored) rather than calculated on demand, an update on any of its derivers causes an update on the derived data. The latter update can occur at different moments, depending on the semantics of the application. Either the database state should always be consistent with the derivation criteria and, therefore, derived data should be updated immediately, or derivation expressions are complex and time-consuming to enforce; thus, a suitable solution could be to calculate the derived data once at the end of the transaction, regardless of the number of updates in its derivers.

2.3 Dynamic display support

• **Purpose.** Graphical database interfaces allow some portion of the data stored in the database to be displayed for browsing or manipulation. However, there is no guarantee that while this data is presented on screen the extension of the database will remain unchanged. Thus, changes to database objects which are depicted on screen can lead to inconsistencies between the data which are stored and that which are displayed. Dynamic displays remove such inconsistencies by propagating changes to the state of the database to the different interfaces where the affected data is being displayed. In Díaz et al. (1994), an approach is presented to support this application by using active rules.

• **Requirements for the knowledge model**

- description of the data to be monitored, i.e. the data on the display. Unlike previous applications where the description of the data to be monitored can be established at the class level (e.g. salary of employees), here, it would be very costly to monitor the entire class, since, frequently, only few instances are on display. Thus, we are faced with the need of defining the monitor at the instance level rather than at the class level.
- specification of the reaction to follow if the data are updated. The reaction consists of propagating the update to the displays where these data is being shown. Thus, the external or internal identifier of the displays is required to propagate the update.

• **Requirements for the execution model**

- if an update on a display does not end successfully (e.g. due to communication problems), a message can be issued, and the propagation process be continued with the next display. Failure does not cause an interruption of the whole process.
- the semantics of the displayed data determines the frequency of display updating: for critical data (e.g. stock market), updates on the display should be made immediately once the data has been modified; for non-critical data (e.g. booking applications), display updating can be delayed till the end of the transaction. Here, the final result is provided rather than keeping the screen updated as changes happen. If displaying is a slow process, concurrency can be enhanced by running this process in a separate but commit-dependent transaction (i.e. the displaying transaction commits only if the updating transaction ends successfully). However, since this separate

transaction has some effects visible from outside, the user should be somehow warned that the update is not permanent till the updating transaction commits (e.g. by showing transient and committed data in different colors). If the transaction finally fails, a contingency action should follow, where the data formerly on the display are re-established.

3 Knowledge model description: an OO approach

If complete flexibility is to be provided, both the knowledge model and the execution model have to be specified using an OO approach. The OO approach, through specialization and modularity, is well-placed to achieved our aim of flexibility. 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 an event-driven behaviour in the context of an OODBMS, a primary requirement is to identify the significant entities and their interactions.

A common approach to the knowledge model of event-driven behaviour is through ECA rules. 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. Thus, the description of event-driven behaviour is supported by two entities: *the event*, which is a description of a specific situation to which reactions may be necessary, and *the rule*, which describes both when and how the system reacts to an event.

Following an OO approach, an entity description includes both structural (i.e. attributes) and behavioral (i.e. methods) features. The next subsection gives a detailed description of the rule and event entities. Here, we assume users define active behaviour by creating the appropriate event and rule objects. No rule language as such is provided.

3.1 The rule object

Representing rules as objects is currently common practice. This approach can be illustrated by the pioneering work of HiPAC (Dayal et al. 1988), where ECA rules are proposed. In EXACT, the structural description of rules involves the following attributes:

– the **event**, which holds the object identifier of the event causing the rule to be fired.
– the **condition** to be verified. The condition is a set of queries to check that the state of the database is suitable for action execution.
– the **action**, which is a set of operations that can have different aims, e.g. enforcement of integrity constraints, user intervention, propagation of methods, etc. Condition and action definitions can refer to the event occurrence where the parameters of the event can be obtained (e.g. the method selector, the method arguments, the receiver, the sender) using the system-provided predicate *event_occurrence*. The result of the condition can also be passed to the action part through the *condition_result* predicate.
– **priority** which is a number reflecting the importance of the rule
– **is_it_enabled** is a boolean attribute that describes the status of the rule, i.e. whether the rule is activated or not. The rule can be ‘switched on’ or ‘switched off’ just by updating the value of this attribute.

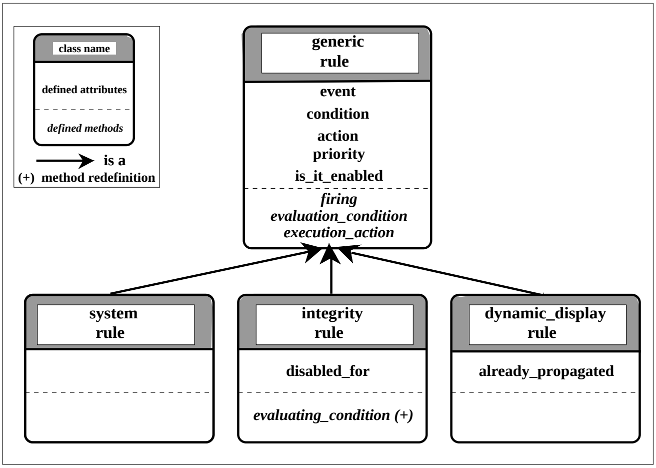

Fig. 1. The rule hierarchy in EXACT

As for the behaviour, it is achieved through the following methods:

– **firing**, which involves the evaluation of the rule and, if satisfied, the execution of its action.
– **evaluating_condition**, which checks whether the rule's condition is satisfied.
– **executing_action**, which executes the rule's action.

As any other object, rules are created by sending the message *new* to its class. For example, to create an instance of the class *generic_rule*, whose identifier is returned in the variable *RuleOID*, the following message is sent:

```
:- new([RuleOID,[
       event([8#ms_event]),
       condition([ % ADAM query]),
       action([ % ADAM program]),
       priority([4]),
       is_it_enabled([yes])
   ]]) => generic_rule.
```

where *8#ms_event* is the object identifier of the event whose occurrence makes this rule to be fired.

All these features are collected in the class *generic_rule*, which can subsequently be specialized to account for new requirements: subclasses *system_rule*, *integrity_rule* and *dynamic_display_rule* are introduced where method (e.g. *evaluating_condition*) and/or attributes (e.g. *already_propagated*) were added or specialized. A detailed description of each subclass is out of the scope of this paper. Figure 1 shows part of the current rule hierarchy in EXACT.

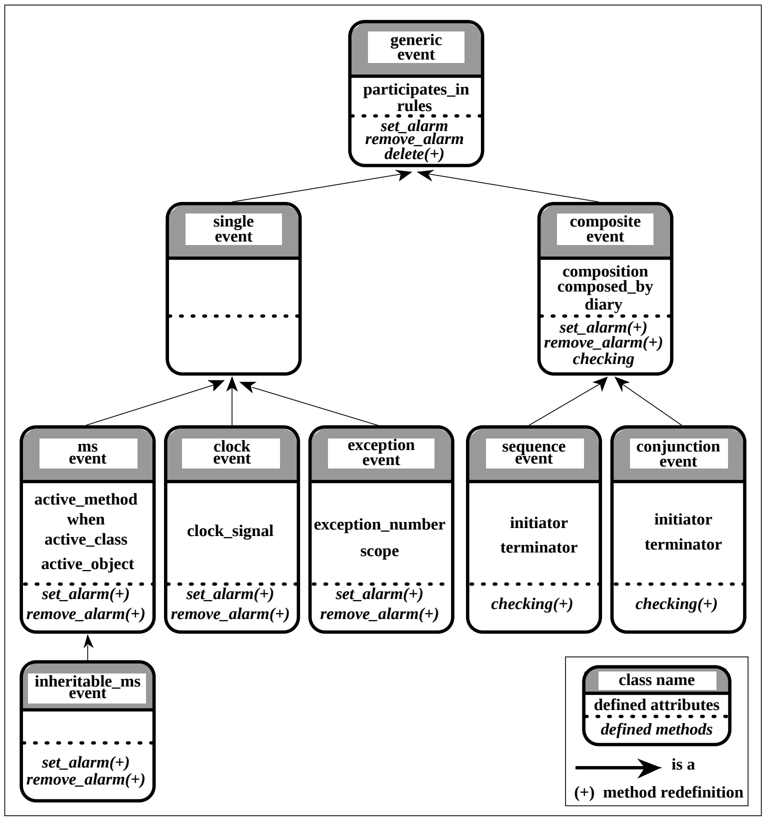

Fig. 2. The event hierarchy in EXACT

3.2 The event object

Events are not always seen as first-class objects. In Medeiros and Pfeffer (1990), events are seen as rule attributes and, hence, cannot have attributes or methods of their own. Although this approach may result in performance gains, it can compromise the ability to extend the system to cope with events coming from different places, or which need special treatment. Furthermore, several reasons can be identified for having the entity *event* different from the entity *rule*: events have their own attributes and behaviour, events can be composite and, finally, events can be specialized, giving rise to the need for hierarchies.

Any kind of event can fire rules and participate in the specification of composite events, features which are respectively captured by the attributes:

– **rules**, where the set of rules activated by the event is kept. It represents the inverse of the attribute *event* defined for the *generic_rule* class, and
– **participates_in**, which holds the object identifiers of the composite events in which the event is involved.

Moreover, events have their own behaviour, namely:

– **set_alarm**, which makes the corresponding event detectable. That is, it places an 'alarm' in the corresponding event generator. This method is used not when the event is created, but once the first rule having this event is activated, so that events whose associated rules are switched off for long periods do not cause any overhead. Thus, this method is internally invoked when the *is_it_enabled* rule attribute is set to *'true'*,
– **remove_alarm** which stops the monitoring of the event. This happens once no rule is left in the system with this event,
– **delete** which specializes standard object deletion, since removal of an event object implies ending the tracking of this event.

As shown in Fig. 2, this common description is kept in the *generic_event* class. This class needs to be specialized, since different attributes and/or methods may be required for event description, depending on the situation to be monitored. An initial classification distinguishes between *single_events* (i.e. primitive events) and *composite_events*.

3.2.1 Primitive events

Primitive or single events are classified according to their event generators. For example, *message-sending events* are produced by the DBMS when a message is sent, *clock events* are generated by the UNIX clock and *exception events* are signaled by the underlining interruption mechanism. Thus, the description of an event depends on the situation to be monitored. In Fig. 2, part of the current event hierarchy is shown. Here, only the description of *message-sending events* is addressed.

Message-sending events are described by four attributes which are kept in the *ms_event* class, namely:

– **active_method,** which holds the name of the method whose invocation produces the event. In an OO context, *active methods* are not restricted to be update operations but can be any method defined in the system (e.g. *display*, *get_age*, create a *new* class or *delete* an instance).

- **when**, which describes when the rule should be fired relative to the execution of the *active_method*: *before* or *after* method execution.
- **active_class.** 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. This attribute keeps the class on which the method is defined. In an early implementation, this attribute was part of the rule object definition. Despite compromising correct *composite event* definition, this design choice stemmed from the desire to improve the efficiency of the rule manager by providing a class-based index to rules. In the current implementation, rules are indexed by the whole event rather than the *active_class* alone. In this way, the system performance is maintained without compromising the definition of *composite events*.
- **active_objects**, which holds the instances to which the message should be sent for the event to be triggered. It is an alternative to the *active_class* attribute. This attribute supports the case where active behaviour is shown by only a few instances rather than the whole class.

As with any other object in the system, a DBMS-generated event is created by sending the message *new* to its class. For example, an event to be detected *before* sending the message *put_age* to a *student* instance, would be created by the following instruction:

```
:- new([EventOID,[
        when([before]),
        active_method([put_age]),
        active_class([student])
]]) => ms_event.
```

As for behavioral features, the *set_alarm* and *remove_alarm* methods need to be specialized for each kind of single event so that the idiosyncrasies of the respective event generators can be considered.

3.2.2 Composite events

Composite events are defined as a disjunction, conjunction, sequence, etc. of other events, either primitive or composite. EXACT has a poor support for composite events where only the sequence construct is provided. Here, composite events are included only to provide a whole picture of the system. Semantic (e.g. consumption modes) and efficiency issues relating to the detection of such events are still unresolved problems which have not been the concern of this research. For a discussion of this topic and complex event definition languages, see Gatziu and Dittrich (1994), Gehani et al. (1992), and Chakravarthy et al. (1994).

In EXACT, composite events are described through the following attributes:

- **composition**, having as value a sequence of events either single or composite. It is specified by the user,
- **composed_by**, which holds the events (single or composite) that comprise the event. It is a local attribute, i.e. only handled by methods associated with the class *composite_event*. The value of this attribute is obtained from the value of the *composition* attribute. It is thus the inverse of the *participates_in* attribute,
- **diary**, in which the event occurrences already detected and involved in the definition of the composite event are recorded. It is used to detect when the whole composite event arises. More complex and complete approaches use Petri nets to support this concept of diary (Gatziu and Dittrich 1994).

As for behavioral features, whereas primitive events are directly detected by an event generator, composite events have to be detected by the event manager itself. Primitive events such as *'before sending the message display to a student'* or *'it is 12th October 1992'* are detected by the DBMS and the UNIX system, respectively. However, the composite event *'before sending the message display to a student on 12th October 1992'* can only be detected by the event manager with the assistance of the diary of events that have already occurred. This is achieved through the method **checking** which has an event as its argument and determines whether the composite event to which it has been sent has arisen or not. The attributes **initiator** and **terminator** support this task. See Chakravarthy et al. (1994) for a complete account. These features are collected in the *composite_event* class, and are specialized for each composition primitive as shown in Fig. 2.

4 Execution model description: on OO approach

Commonly, the execution model is fixed and hard-wired into the DBMS, an approach which compromises its extensibility, flexibility and declarativeness. EXACT attempts to enhance these features first by using an OO approach to describe the execution model, and secondly by identifying a number of parameters which can be used to characterize the execution model. Instead of providing code, the user can declaratively specify the required execution model by choosing among a set of pre-established alternatives. If none of these alternatives accommodates the problem, extensions at the behavioral level (i.e. specialization of methods) will be required.

Following an OO approach, an overview of rule processing and the participating entities is first presented. Next, a set of dimensions to characterize rule processing is given, and finally, how EXACT supports rule processing is shown.

4.1 Execution model description

Broadly speaking, the execution model indicates how the system provides quick response through the use of rules to events generated by some system. The following elements can be identified in this process:

- *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.
- *the event manager* is in charge of the management of events. This includes the setting up of alarms in the appropriate event generator, as well as the detection of composite events.

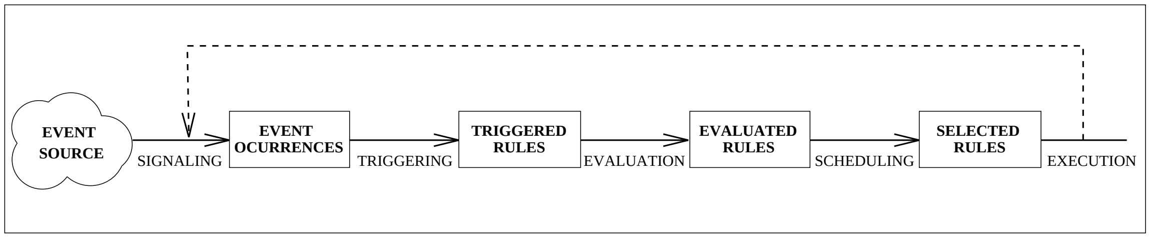

Fig. 3. Execution model algorithm steps

- *the event detector* is the mechanism responsible for identifying events. It signals the happening of event occurrences. Normally, it coincides with the event generator.
- *the rule manager*, where the creation, ordering and number of rules to be executed are considered.

For the sake of clarity, two further elements can be introduced, namely:

- *the event occurrence* is an indicator to signal that the situation described by an event has been reached. Usually, the event occurrence includes the current instantiation of the parameters of the corresponding event (e.g. the actual arguments of a message).
- *the rule instantiation* reflects that a given rule has been awakened by an event occurrence, and its description can be given as a pair *(rule-oid,event-occurrence)*.

The main interactions among the previous entities are those corresponding to the following activities:

- creation of an event object, which does not differ from the creation of any other object. When the first rule having this event is activated, the event detector will be notified to monitor it.
- deletion of an event, which can only be achieved by previously deleting all rules attached to it. Deletion has to be notified to the event detector to end the monitoring of this event.
- creation of a rule, which differs from the normal creation procedure in two issues. First, a call can be made to the rule manager to assign the static rule priority (see below). Second, if the rule is *awaken* when it is created, the system checks whether the associated event is being monitored. If not, the system begins to monitor it.
- deletion of a rule, which besides the standard procedure for deleting objects, stops monitoring the rule's event if no other rules use that event.
- happening of an event occurrence. Loosely speaking, this corresponds to what is known as the execution model algorithm.

The execution model algorithm can be seen as based on the following phases (see Fig. 3):

1. the *signaling* phase refers to the appearance of an *event occurrence* caused by an event source,
2. the *triggering* phase takes the events produced so far, and triggers the corresponding rules. The association of a rule with its event occurrence forms a *rule instantiation*,
3. the *evaluation* phase evaluates the condition of the triggered rules. The *rule conflict set* is formed from all rule occurrences whose conditions are satisfied,
4. the *scheduling* phase indicates how the rule conflict set is processed,
5. the *execution* carries out the actions of the chosen rule instantiations. During action execution other events can in turn be signaled, which may produce *cascaded* rule firing.

These phases are not necessarily executed contiguously, but depend on the **Event-condition** and **Condition-action** coupling modes. The former determines when the condition is evaluated relative to the event that triggers the rule. The *Condition-action* coupling mode indicates when the action is to be executed relative to the evaluation of the condition. The options for coupling modes most frequently supported are immediate, deferred and detached (Dayal 1989). Here, we also propose to use a variation of the deferred mode: the *user-invoked* coupling mode, whereby the condition (action) is evaluated (executed) at a user-specified time after the event (condition) has been signaled (evaluated). A similar effect is also supported by Starburst (Widom and Finkelstein 1990), where users can invoke rule processing within a transaction by issuing special commands: the *process rules*, *process ruleset S* and *process rule R* commands invoke the rule processing for the whole triggering rule set, a given subset *S* or a unique rule *R*, respectively. However, it is worth noticing that, in our proposal, the possibility for a rule to be considered for execution at user-required time is a property of the rule class, to be fixed by the rule's author rather than being decided by the application programmer as in Starburst.

The description shown in Fig. 3 is general enough to be valid no matter what application is considered. In EXACT, variations on this general framework can be characterized along the following modes: scheduling mode, conflict-resolution mode and error-recovery mode.

Some of the features considered here are clearly not new. Although they have to be faced by the system designer, these characteristics are commonly hard-wired in the system code. Our aim is to make them explicit and, more importantly, available to the final user who knows the application functionality which should be supported by the execution model. The alternatives presented are far from exhaustive, but only those currently provided by EXACT are included.

4.1.1 Scheduling mode

The conflict set is built upon the rule instantiations which are triggered simultaneously and whose conditions are satisfied by the current database state. The question is how many of these rules have to be triggered. Possible answers are:

– **all_rules,** here every rule is considered. This behaviour is exhibited by rules supporting integrity maintenance: an update on the database is correct once *all* constraints affected by this update are validated or appropriated actions are found to recover a valid state.
– **only_one_rule,** this time only one rule is considered. This option can be used to support derived data. Each rule can be seen as supporting a different derivation equation but only one is used. The rule to be chosen is decided based on the conflict resolution mode.

There is also the question of *rule cascading*, i.e. what happens if event occurrences have arisen in the middle of a rule's action. An option is to suspend the current call and recursively initiate a new instantiation of the execution algorithm where new rule occurrences triggered by this event occurrence are considered. Another alternative is to gather all event occurrences in a *bag* which would be considered in a later step, but the current call is not interrupted. POSTGRES and Starburst can be seen as examples of the recursive and iterative approach, respectively. EXACT follows a recursive approach.

4.1.2 Conflict_resolution mode

This mode addresses the question of how is the next rule to be fired chosen if several rule instantiations are available? This topic has received much attention among the expert system community, as it is considered fundamental to understanding and controlling the behaviour of the whole system. Indeed, rule order can strongly influence the result, and reflects the kind of reasoning followed by the system. In databases, several reasons motivate the availability of a clear and powerful mechanism for conflict resolution, namely:

– rule order can influence the final database state, and have an important impact on performance. Ensuring confluent rule sets (i.e. rules where the order in which they are fired does not have any influence on the final database state) is a matter of active research (Aitken et al. 1992);
– the number of rules can be large, leading to complex interactions which are difficult to foresee and understand,
– a potentially larger number of users than in expert systems can add rules to the system. Then, it is paramount for the rule's authors to have clear guidelines and understanding concerning the control of rules.
– different applications impose distinct strategies for conflict resolution. Although this point is also relevant to expert systems [e.g. OPS5 (Brownston et al. 1985) provides two conflict resolution strategies, known as LEX and MEA], it is even more important for active databases where the applications to be supported can be very diverse. An enlightening example can be found in Ioannidis and Sellis (1992) where virtual attributes can be obtained from distinct derivation expressions in the form of rules. When several derivation expressions are available for the same virtual attribute, a conflict resolution strategy (referred to as resolution criterion) is applied based on the semantics of the attribute itself. Different criteria can be followed: a value-based criterion, where the value assigned to the virtual attribute is determined based on properties of the generated values themselves, and a rule-based criterion, where the value assigned is chosen based on properties of the rule that generates the value.

Despite these remarks, no active DBMS provides a flexible, easy-to-customize conflict resolution mechanism. To the best of our knowledge, conflict resolution in the active DBMSs described in the literature is always fixed, hard-wired into the implementation. So far, rule priority is the only mechanism available to reflect the application semantics in the conflict resolution phase, but even how priority is being used is not free from some criticism, as presented below.

In most of the active DBMSs described in the literature, the order in which simultaneously triggered rules are processed is based on an attribute *priority* given by *the user* for each *single rule* when *it is created*. This approach suffers from the following drawbacks:

1. The priority is assigned by the rule designer after some consideration. The nature of this process is not recorded in the database, so it can happen that rules supporting the same functionality are assigned priorities based on different criteria followed by the various authors of the individual rules. As an example, consider integrity constraints. On the assumption that a constraint is violated, the update is rejected, then, important performance gains can be achieved by ordering constraint checking based on the complexity of the constraint to be verified. However, normally, this criterion is synthesized in a priority number given by the user. In a multiuser environment where competing integrity constraints can be defined by different users, to have a shared and clear understanding of the rationale for priority assignment would greatly enhance the coherence of the system. If priority assignment is supported by the system, not only is the soundness of the system improved but it also would relieve the rule designer of this burden.
2. A per-rule priority mechanism may not be appropriate. In large rule sets, rule-based priority can be a too low-level mechanism, making it difficult to ascertain the repercussion of assigning a given priority to the whole flow of control. Different levels of priority can help here, where higher levels are based on the functionality being supported rather than being based on the rules themselves – after all, rules are just an implementation mechanism. For example, rules supporting integrity constraints can have a higher priority as a whole than rules supporting dynamic displays. That is, the priority is established at the conceptual level: integrity maintenance has a higher priority than keeping displays coherent. Here, rule-based priority can be used to establish the order among rules *supporting the same application*. This provides different levels of abstraction to establish priority among rules, helping in understanding, structuring and controlling rule processing. Furthermore, concept-based priorities constitute a clearer way of stating the rational behind the setting-up of priorities. The priority is established at the level of the concept (e.g. constraint maintenance), and afterwards, if required, at the level of the rule (e.g. salaries cannot decrease). However the latter is subordinated to the priority of the concept. Such an approach allows reasoning about

priorities to be done at the conceptual level (i.e. the rule class) rather that at the implementation level (i.e. the rule instances).

3. Priority is set at rule creation time. However, for some concepts to be supported using rules, the parameters to be considered which establish a rule's order cannot be known until execution time. For instance, in real-time databases, the workload of the system can be decisive when choosing the most appropriate rule. Production systems such as OPS5 are another example where rule order is based not only on a priority, but on a dynamic parameter: the recency of the elements instantiating the condition.

EXACT attempts to overcome these disadvantages by means of the following mechanisms:

– a concept-based priority which is decided by the designer according to the semantics and time requirements of the concept to be supported using rules;
– a rule-based priority which is subordinated to the concept-based one (the *priority* attribute);
– a priority function which sets the rule's priority. This function is taken into account at rule creation time, and aims to make explicit and coherent the criteria considered in assigning priorities. Being invoked at compile time, this function does not cause any overhead. This function is defined at the class level and therefore takes into consideration concept-based features;
– a conflict resolution strategy to be considered at execution time. Besides a rule's priority, other parameters obtained at execution time (e.g. current workload, recency of the instantiation, etc.) are taken into account. By default, only a rule's priority is considered. Except for the default, the checking of parameters at execution time does incur some performance penalty.

4.1.3 Error_recovery mode

During the *execution phase*, where actions of the chosen rule occurrence are carried out, this dimension contemplates what to do if an error condition is produced during this phase. Most systems just abort the transaction, as this is the standard behaviour in databases. However, other alternatives can be more convenient, as suggested in Hanson and Widom (1993), namely, to terminate execution of that rule and continue rule processing, to return to the state preceding rule processing and resume database processing, or to restart rule processing. Here, the following alternatives have been considered:

– **abort**, which cancels the whole execution;
– **ignore_this_rule**. The execution of the rule is resumed and rule processing continues. As an example, this option can be appropriated to support dynamic displays where rules are used to propagate database updates to the displays where the affected data is being displayed. If a communication problem arises while a rule is being fired, the corresponding display is left unchanged, but rule processing continues. This option requires the rule to be fired within a subtransaction, so that if an error arises the initial state can be recovered;
– **contingency_action**. The condition error causes control to be transferred to the contingency action, which ends either with *abort* or *retry*. The latter means that the condition error was recovered and that control is returned to the point where the error arose. The user must provide the contingency action.

4.2 Execution model support in EXACT: the entities

Now the issue is not the description of isolated rules but the execution model of *a set of rules* taken as a unit.

The idea of *set* is supported in OO systems by the concept of *class*. A class has a twofold definition. On the one hand, it describes the common features shared by its instances (the class as a template), and on the other hand, a class can be seen as collecting together a set of instances[3] (e.g. the *average_age* of people). It is worth noticing that these properties of the set-as-a-unit apply not only to the structural features of the set – as it is commonly found in semantic data models such as SDM (Hammer and McLeod 1981) – but also to its behavioral features. These properties can then be shared by other sets, i.e. by other classes. Hence, in the same way that the common description shared by a set of instances is abstracted at a higher level to form the class, the common description shared by a set of classes, now seen as the unit set-of-instances, can be abstracted at a higher level: *the metaclass*. A metaclass is a class whose instances are also classes. Metaclasses not only permit classes to be stored and accessed using the facilities of the data model, but also make it possible to refine the default behaviour for class creation using specialization and inheritance. In this way, uniformity and extensibility are also available for the data model.

The execution model can be seen as describing the pattern followed by a set of rules. Thus, the objects describing the execution model, i.e. *event_manager* and *rule_manager* objects, are supported as metaclasses, since they describe how to manipulate a set rather than an individual rule.

4.2.1 The event_manager object

The detection of composite events is the only behaviour attached to the *event_manager* in the current implementation. This is supported as part of the metamethod *signal*. Creation of events does not differ from the standard object creation procedure, so the method **new** provided by the system is used.

4.2.2 The rule_manager object

In the same way that the *generic_rule* class holds how to describe single rules, the *rule_manager* keeps how to describe the execution model for a rule class. Such description is achieved through the previously introduced dimensions which are supported by the following attributes (better said, meta-attributes): the **e-c_coupling** attribute, which stands

[3] In Smalltalk, these properties are represented by the so-called *class variables*.

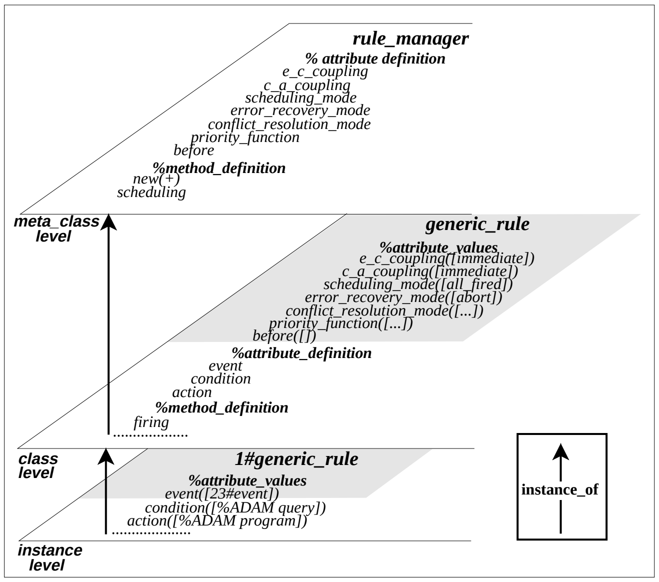

Fig. 4. The three-level definition in EXACT

for the event-condition coupling mode[4]; the **c-a_coupling** attribute, which represents the condition-action coupling mode; the **scheduling_mode** attribute; the **error_recovery_mode** attribute; the **conflict_resolution_mode** attribute, which holds the function and parameters to be used to decide the next rule instantiation to be fired from conflict set; the **priority_function** attribute, which holds the function to be followed to assign rule-based priority[5], and the **before** attribute, whose value is another rule class. It supports a relative application-based priority. Notice that, unlike previous approaches, coupling modes are specified at the class level rather than at the instance level (i.e. rule instances). This is akin to our contention that control information rarely refers to single rules, but rather depends on the concept being considered, which is reflected in the rule class.

As for the behavioral features, three main meta-methods are provided: **new, triggering** and **scheduling**. The former is used to create rule instances, which is a specialization of the standard creation procedure provided by ADAM. Besides the default creation, introducing a new rule instance of class *C* requires establishing its priority. If none has been given by the user, the rule-based *priority_function* of class *C* is applied.

Other methods (e.g. $evaluating\_all\_conditions$) have been defined, but they are not used globally (i.e. they do not belong to the class interface). Their only purpose is to enhance modularity and re-use. Indeed, being supported as methods, these phases can subsequently be specialized to cope with new, unforeseen requirements.

A rule instance provides the values for the attributes described in its class (e.g. the rule's condition). Likewise, a rule class provides the values for the meta-attributes described in its metaclass (e.g. the class' scheduling mode). Therefore, there is a three-level definition as shown in Fig. 4: 1) the metaclass level, where the dimensions of the execution model shared by the rule classes are factored out in metaclasses such as *rule_manager*, 2) the class level, where the characteristics of single rules are factored out (i.e. the knowledge model) and their execution model is defined (e.g. the *generic_rule* class) and finally, 3) the instance level, where individual rules are defined (e.g. *1#generic_rule*).

Notice that, while the knowledge model is inherited from the superclass to its subclasses, the execution model must be specified for each class, regardless of its superclass. The execution model of class *C* applies to the rule instances which have *C* as its immediate class, in the same way that the value of the *average_age* for the *person* class is not inherited by its subclass *student*, which would have its own *average_age*.

A parametrized execution model is provided by the core of EXACT. When a new rule class is created, a customized version of this standard model is generated from the above meta-attributes by the system. This version overrides the standard execution model for this new rule class.

If these meta-attributes do not convey the functionality required, the user can himself specialize the methods which support the execution model. In implementation terms, new submetaclasses can be defined where inherited methods can be specialized or overridden. Despite the burden put on the user, this approach allows the possibility of specializing the standard execution model to accommodate the special requirements of the concept to be supported using rules.

4.3 Execution model support in EXACT: the process

Once the objects participating in the description of the execution model have been introduced, rule processing, as shown in Fig. 3, can be described as a message interaction process between these objects. Rather than an intricate algorithm, rule execution is conceived as message interchange among those entities. Figure 5 shows such interaction namely:

1. Signaling phase. When the occurrence of a primitive event is realized by the event detector, a *signal* message is sent to the event manager, which checks whether some composite event occurrences may also arise (through the *checking* method).
2. Triggering phase. Once the primitive and composite event occurrences have been detected, the event manager informs the rule manager through the method *triggering*. Then, the rule manager obtains the set of rule instantiations (i.e. the triggered rules).
3. Evaluation phase. Triggered rule's conditions are evaluated through the $evaluating\_all\_conditions$ method, which, in turn, calls $evaluating\_condition$ for each of the rule instantiation. Those rule instantiations whose conditions have been satisfied form the conflict set.
4. Scheduling phase. The conflict set is split up based on the rule class (i.e. the concept supported), so that each subset can be managed according to the requirements of each concept (i.e. integrity maintenance, view support, etc.). This situation is shown in Fig. 5 as a set of overlapping rectangles. Each conflict set is passed as a parameter to the *scheduling* method, which schedules and fires the rule

[4] The detached coupling is not implemented.

[5] This function, defined by the user, is invoked at rule creation time to establish priorities based on structural features of the rule (i.e. attribute values).This function can be supported as an attribute due to the features of Prolog, the underlying language. In other environments it can be supported as a method.

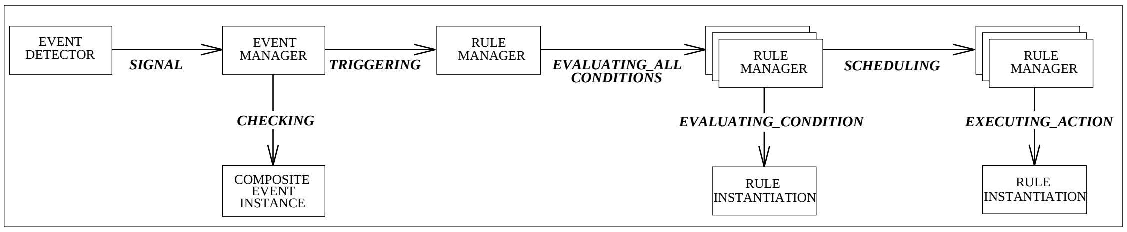

Fig. 5. Rule processing as message interaction among objects

occurrences according to the specific semantics of their classes. The order in which each conflict set is processed depends on the priority of the rule class (i.e. the *before* meta-attribute).

5. Execution phase. Firing a rule instantiation implies invoking the *executing_action* method. The order in which each rule in the conflict set is fired depends on the priority of the rule instance (i.e. the *priority* attribute). If, during action execution, an event arises, then the *signal* method is invoked and a recursive call to the whole process is initiated, i.e. cascade rule firing.

These steps are not necessarily executed sequentially, but are dependent on the coupling modes chosen. In implementation terms, the event-condition coupling refers to when the *evaluating_all_conditions* method is executed relative to the end of the *triggering* method, whereas the condition-action coupling is concerned with when the *executing_action* method is performed relative to the end of the *scheduling* method. Since the coupling modes adopted depend on the rule class, they cannot be hard-coded within the methods themselves, since such couplings vary from rule class to rule class in order to accommodate the semantics of the corresponding concept.

To overcome this problem, an approach similar to the one proposed to support flexible control flow (Dayal et al. 1990) has been used. A rule can be seen as "coupling" the method involved in the rule's event and the methods which appear in the action part. Since the basic execution model functionality is supported by methods, and methods can be invoked from rule conditions and actions, a cleaner and more flexible way to support the coupling is through meta-rules.

Hence, the event-condition coupling realized as the coupling between the $triggering$ and the $evaluating\_all\_conditions$ methods and the condition-action coupling achieved as the coupling between the $scheduling$ and the $executing\_action$ methods are obtained through meta-rules. For instance, a $user\_invoked$ event-condition coupling mode is achieved by the following rule:

```
event:
  sequence(event1([after,triggering,
                   rule_manager]),
      event2([after,evaluate_rule_set,
                   RuleClass]))
condition:
   true
action:
   send evaluating_all_conditions
        to rule_manager
```

The situation which causes $evaluating\_all\_conditions$ to be invoked is described as a sequence of two events: *event1* is raised after successful completion of the *triggering* method; *event2* is produced by $evaluate\_rule\_set$, a special dummy method which causes all rule instantiations of $RuleClass$ to be evaluated. It is discretionarily used within the transaction.

As a further example, a $deferred$ condition-action coupling mode is obtained by the rule:

```
event:
  sequence(abstract_event1([EventParams,
                  RuleTriggeringSet]),
               EOT)
condition:
   true
action:
   for_each RuleOid in
            RuleTriggeringSet do
        ...
        send executing_action
             to RuleOid
        ...
```

where EOT signals the end of the transaction, and $abstract\_event1$ is an abstract event raised within the $scheduling$ method for each class of rule instantiation. The sequence of these two events describes the situation where $executing\_action$ is invoked. Therefore, the condition-action coupling mode is not hard-coded in the $scheduling$ method. Instead, the $scheduling$ method indicates that the situation ready-to-be-fired has been reached through raising the abstract event, $abstract\_event1$. This event can then be combined with nothing, an $end\_of\_transaction$ event, or a $process\_rule\_set$ event to achieve an immediate, deferred, or user-invoked coupling mode, respectively[6].

Unlike in other approaches (e.g. ODE, CHIMERA) where deferred rules are obtained by enlarging the rule's event with the $end\_of\_transaction$ (EOT) event, in EXACT, instance rules are preserved as defined by the user. Indeed, coupling modes are seen as part of the rule class execution model and supported as the "coupling" between the meta-methods which implement the execution model of the rule class. Furthermore, far fewer composite events need to be tracked. As an example, consider a database with 60 deferred in-

[6] A question could arise about how meta-rules themselves are supported. The same mechanism (i.e. defining meta-meta-rules) cannot be used, else an infinite loop would be provoked. Thus, meta-rules are grouped in the *system_rule* class, where the meta-methods (e.g. *triggering*) are specialized. Since the coupling modes for meta-rules are immediate, this specialization consists of hard-coding coupling modes within the meta-methods (e.g. the method *triggering* explicitly calls *evaluating_all_conditions*).

tegrity constraints. In EXACT, only one composite event is tracked, since all the deferred constraints are scheduled as a set, whereas the traditional approach requires the monitoring of 60 composite events.

5 Advantages of this approach: an example

In this section, an example is shown to illustrate the extent to which the goal stated in the introduction, i.e. enhancing the flexibility, declarativeness and extensibility of a rule system, has been achieved.

The point to be noted is that now the designer can adapt both the knowledge and execution model to reflect more accurately the concept semantics. The designer should begin by listing the requirements that the support of the concept poses for both models, and then study what is new and what can be re-used. As an example, this section shows how to accommodate the requirements for integrity constraint maintenance.

The knowledge model should allow the description of the integrity constraint to be monitored and the support for both exceptions and compensating actions. The description provided by the system, and kept in the *generic_rule* class, can accommodate most of the requirements for defining integrity constraints. It has an event which can be any of the updates potentially violating the constraint, a condition where the constraint is validated, and an action which holds the reaction to be followed if the constraint is violated. However some applications (Buchmann and Dayal 1988) also need to specify exceptions, i.e. objects for which the integrity constraint does not apply. Such requirement can be supported by specializing *generic_rule* into a subclass *integrity_rule* where a new attribute *disabled_for* is introduced, which holds a list of the exceptional objects. Moreover, condition evaluation is extended to check whether the object involved is among the exceptional objects. This is achieved by redefining the *evaluating_condition* method. Figure 6 shows the EXACT definition for this rule class.

```
new([integrity_rule,[
   is_a([generic_rule]),
   e-c_coupling([immediate]),
   c-a_coupling([immediate]),
   scheduling_mode([all_fire]),
   error_recovery_mode([abort]),
   priority_function([( ... )]),
   before([dynamic_display_rule]),

   attribute(att_tuple(
      disabled_for,global,set,optional,object,[]
   )),
   method((
      evaluating_condition(global,[],[plog],plog,
            [[Params],ConResult]) :- ...
   ))
)]] => rule_manager.

new([deferred_integrity_rule,[
   is_a([integrity_rule]),
   e-c_coupling([deferred]),
   c-a_coupling([immediate]),
   scheduling_mode([all_fire]),
   error_recovery_mode([abort]),
   priority_function([( ... )]),
   before([statistical_rule])
)]] => rule_manager.
```

Fig. 6. Extending EXACT with integrity rules

The execution model requirements are sketched in Sect. 2.1 specifically,

- once a database update occurs which affects the constraint, the integrity checking can begin immediately. If this is the desired behaviour, it can be reflected by assigning *immediate* as the value of the *e-c_coupling* and the *c-a_coupling* attributes.
- *all* integrity constraints affected by a database update should be checked in turn until one is falsified. This can be accommodated by assigning the values *all_rules* and *abort* to the attributes *scheduling* and *error_recovery_mode*, respectively.
- if different integrity constraints have to be checked, an order can be established based on the evaluation cost of each constraint: the higher its cost, the lower its priority. This is supported by assigning a function, which works out the priority based on the complexity of a rule's condition and action, kept as a value of the *priority_function* attribute. Rule's complexity can be calculated from the number of message-sending instructions which appear in the condition and the action. This function is invoked at the rule's creation time if no priority is explicitly given by the user.
- since integrity rules have a higher priority as a whole than, for example, dynamic-display rules, the *before* attribute will hold *dynamic_display_rule* as its value.
- as for the conflict resolution strategy, only the rule's priority is considered, since no specific policy is given.

As shown in Fig. 6, the *integrity_rule* class keeps the values for these meta-attributes, which declaratively define the execution model chosen. Broadly speaking, the knowledge model is supported by attribute/method definitions, whereas the execution model is described by values given to the meta-attributes.

Now consider that deferred integrity constraints are also required. These constraints exhibit the same knowledge model and most of the features of the execution model of the previous integrity rules. The only difference stems from the event-condition coupling mode (i.e. the *e-c_coupling* parameter) which now is *deferred*. This can be supported by defining a subclass **deferred_integrity_rule** which inherits the knowledge model of the *integrity_rule* class and specifies its own execution model. Figure 6 shows this situation. Notice that the condition-action coupling mode (i.e. the *c-a_coupling* attribute) is *immediate*, as the condition evaluation is already deferred. Also, the concept-based priority (i.e. the *before* attribute) has been changed, since now the potentially conflicting rules are those also evaluated at the end of the transaction (e.g. *statistical rules*).

6 Related work

Dimensions of active behaviour are scarcely addressed in the literature. Attempts to classify active systems have been investigated in Paton et al. (1993) and Widom (1994). Paton et al. (1993) provide a set of dimensions along which active systems (both relational and object-oriented) can be classified, whereas Widom (1994) presents a comparison between the active and deductive approach to rule support in database systems.

However, little experience is reported on how to accommodate and explicitly support a given set of dimensions. An exception is the work of Fraternali et al. (1994), where a semantic model for active databases is proposed. Unlike EXACT, rule management is only characterized by the coupling mode. However, while EXACT presents a very simple event management policy – where simple events are mainly considered – Fraternali et al. consider two additional modes of event handling: the *consumption mode* and the *net effect* mode. The former addresses what happens to events after triggered rules have considered them: "are they 'forgotten', or can they still trigger the rule, or can they be referred to in some subsequent consideration or execution of the rule?" (Fraternali et al. 1994). Two options are considered: *consuming*, where events are no longer visible after rule consideration, and *preserving*, where events are kept visible after rule consideration. The issue of event consumption policies is also addressed in the pioneering event language SNOOP (Chakravarthy and Mishra 1991).

The net effect mode poses the question of what happens if the triggering event of a rule is invalidated by some other event before the rule has been considered. Either a rule is triggered by the bare occurrence of an event, regardless of what happens afterwards, or a rule triggering must be revised to check for invalidating events (e.g. an update event occurrence and a posterior delete event occurrence on the same object would result in no event occurrence at all). Similar considerations used to determine the net effect are observed by other active systems, e.g. Starburst.

The previous ideas are supported in the database language Chimera (Ceri et al. 1996), where event-condition-action rule syntax is extended with two keywords for the user to specify the rule activation mode (i.e. coupling mode) and the event-handling mode.

Rules in this context are later translated into an internal format, where the coupling mode or the event consumption policy are realized. For example, a deferred rule implies to extend the internal rule event with an *end_of_transaction*-like event, and also the internal rule condition extends the initial condition to reflect the event consumption policy. By encoding these policies outside the initial rule, different event consumption policies can be easily obtained by changing the translating schemas.

By contrast, the EXACT approach relies on inheritance to accommodate (i.e. specialize or override) the evolution of the execution model, where dimensions are specified at the class level, as opposed to Chimera, where it is given at the rule level. And most important, EXACT allows rule sets to co-exist with different execution strategies.

The need for flexible systems is also addressed in Hull and Jacobs (1991), where it is stated that 'it appears that different rule-application semantics will sometimes be appropriate even within a single database... It seems unlikely that a fixed collection of choices will suffice however, specially as active databases become increasingly sophisticated'. For Hull and Jacobs (1991), the differences in semantics stem from choices concerning when rules should be fired, how they should be fired and how their effects should be combined. Our work extends these dimensions and presents an approach to realize such flexibility.

7 Conclusions

There are a growing number of applications that can be supported using event-driven behaviour. However, these applications are far from being uniform and thus impose different requirements on both the knowledge model and the execution model. Therefore, a flexible rule system that is tailorable to suit special requirements can be most useful.

Using rule objects to represent the knowledge model is now current practice. However, the execution model is being embedded in the system code. This makes the execution model difficult, if not impossible, to change or customize. In this paper, an approach has been presented which supports tailorable rule execution models in OODBs by using metaclasses. Three main advantages stem from this approach:

– *flexibility:* different sets of rules can have different execution strategies, where each set supports a distinct application (e.g. integrity constraint maintenance).
– *declarativeness:* easiness-of-use is enhanced by describing the rule execution model through a set of parameters. In this way, the user just provides the parameters for each dimension that best fit the concept requirements, and the system customizes the appropriate methods to support this execution model.
– *extensibility:* being supported as methods, the rule execution strategy can be easily extended by refining already provided methods or introducing new ones.

Extensibility is achieved through metaclasses. This mechanism is not as extensible as other approaches, including persistent languages or the use of a storage and optimizer manager, because they let designers extend the data model only, not the DBMS kernel. Thus, the extensibility of EXACT is constricted for the underlying DBMS (e.g. the transaction mechanism). However, such systems are much easier for designers to use, because metaclasses let them use the object-oriented data model to extend itself and avoid the considerable coding involved with other approaches (Díaz and Paton 1994).

Modularity can be seen as a by-product of this approach. Active DBMSs can be jeopardized if appropriate structuring mechanism are not available to handle the increasing complexity and number of rules than future active systems are expected to hold. Similar problems to those found in artificial intelligence can be encountered, where deception among production system practitioners is largely due to the lack of structuring mechanisms available to cope with large rule sets. The approach presented in this paper provides some help, as active rules are grouped according to the functionality supported, where control can be customized to obtain the desired requirements.

It is our experience that this approach greatly enhances the rule system ability to cope with heterogeneous applications in active DBMSs.

Acknowledgements. We are in debt with Norman Paton for his help with ADAM. The authors would like also to thank the ACT-NET colleagues for fruitful discussions on active databases. This work has been supported by the Department of Industry, Agriculture and Fishing of the Basque Government under contract G295UN012. Active database research at San Sebastián has benefited from funding from the EU Human Capital and Mobility Network ACT-NET.

References

Aitken A, Widom J, Hellerstein J.M (1992) Behaviour of database production rules: Termination, confluence, and observable determinism. In Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp 59–68

Bocca J (1991) MegaLog: A platform for developing knowledge base management systems. In Proc. 2nd. Intl. Symposium on Database Systems for Advanced Applications (DASFAA'91)

Brownston L, Farrel R, Kant E, Martin N (1985) Programing Expert Systems in OPS5: An Introduction to Rule-Based programming. Addison-Wesley, Reading, Mass.

Buchmann AP, Dayal U (1988) Constraint and exception handling for design, reliability and maintainability. In: Fulton RE (ed) Managing Engineering Data: Emerging Issues (ASME), pp 95–100

Ceri S, Fraternalli P, Paraboschi S, Branca L (1996) Active rule management in Chimera. In: Widom and Cheri (1996)

Ceri S, Widom J (1990) Deriving production rules for constraint maintenance. In: Proc. 16th Intl. Conf. in Very Large Data Bases. Morgan Kaufman, San Mateo, Calif., pp 567–577

Chakravarthy S, Krishnaprasad V, Anwar E, Kim S-K (1994) Composite events for active databases: Semantics, contexts and detection. In: Bocca J, Jarke M, Zaniolo C (eds) Proc. 20th Int. Conf. on Very Large Data Bases.Morgan-Kaufmann, San Mateo, Calif., pp 606–617

Chakravarthy S, Mishra D (1991) An event specification language (SNOOP) for active databases and its detection. Technical Report, University of Florida

Dayal U (1989) Active database management systems. SIGMOD RECORD 18(3):150–169

Dayal U, Buchmann AP, McCarthy D.R (1988) Rules are objects too: A knowledge model for an active object oriented database system. In: Dittrich KR (ed) Proc. 2nd Intl. Workshop on OODBS. Springer, Berlin Heidelberg New York, pp 129–143

Dayal U, Hsu M, Ladin R (1990) Organizing long-running activities with triggers and transactions. In: Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp 204–214

Díaz O (1992) Deriving rules for constraint maintenance in an object-oriented database. In: Ramos I, Tjoa AM (eds) Proc. Intl. Conf. on Databases and Expert Systems DEXA. Springer, Berlin Heidelberg New York, pp 332–337.

Díaz O, Gray PMD, Paton N (1991) Rule management in object oriented databases: a uniform approach. In: Sernadas A, Lohman GM, Camps R (eds) Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona. Morgan Kaufmann, San Mateo, Calif., pp 317–326

Díaz O, Jaime A, Paton N, al Qaimari G (1994) Supporting dynamic displays using active rules. SIGMOD RECORD 23(1):21–26

Díaz O, Paton N (1994) Extending ODBMS using Metaclasses. IEEE Software 11(3):40–47

Etzion O (1993) Flexible consistency modes for active databases aplications. Information Systems 18(6):391–404

Fraternali P, Montesi D, Tanca L (1994) Active database semantics. In: Proc. ADC'94 Fifth Australasian Database Conference. University of Canterbury, New Zealand

Gatziu S, Dittrich K.R (1994) Events in an active object-oriented database. In: Paton NW, Williams MH (eds) Proc. 1st Int. Workshop on Rules in Database Systems. Springer, Berlin Heidelberg New York, pp 23–39

Gehani NH, Jagadish HV (1991) Ode as an active database: Constraints and triggers. In: Sernadas A, Lohman GM, Camps R (eds) Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona. Morgan Kaufmann, San Mateo, Calif., pp 327–336

Gehani NH, Jagadish HV, Shmueli O (1992) Composite event specification in active databases: Model and implementation. In: Proc. 18th Intl. Conf. on Very Large Data Bases. Morgan Kaufmann, San Mateo, Calif., pp 327–338

Hammer M, McLeod D (1981) Database description with SDM: A Semantic Database Model. ACM Transactions on Database Systems 6(3):351–386

Hanson E.N, Widom J (1993) An overview of production rules in database systems. Knowl Eng Rev 8(2):121–143

Hull R, Jacobs D (1991) Language constructs for programming active databases. In: Sernadas A, Lohman GM, Camps R (eds), Proc. 17th Intl. Conf on Very Large Data Bases, Barcelona. Morgan Kaufmann, San Mateo, Calif., pp 455–467

Ioannidis Y.E, Sellis T.K (1992) Supporting inconsistent in database systems. J Intell Inf Syst 1(1):243–270

Medeiros C, Pfeffer P (1990) A mechanism for managing rules in an object-oriented database. Technical report, Altair Technical Report

Paton N (1989) ADAM: An object-oriented database system implemented in Prolog. In: Williams MH (ed) Proc. British National Conference on Databases. Cambridge University Press, pp 147–161

Paton N, Díaz O, Williams MH, Campin J, Dinn A, Jaime A (1993) Dimensions of active behaviour. In: Williams M, Paton N (eds) Proc. 1st Intl. Workshop On Rules In Database Systems. Springer-Verlag, Workshops in Computing series, pp 40–57.

Urban SD, Desiderio M (1991) Translating constraints to rules in CONTEXT: A CONstrainT EXplanation tool. In: Kent W, Meersman RA, Khosla S (eds) Object-Oriented Databases: Analysis, Design and Construction. North-Holland, Amsterdam, pp 373–392

Widom J (1994) Deductive and active databases: Two paradigms or ends of a spectrum? In: Williams M, Paton N (eds) Proc. of the 1st Intl. Workshop On Rules In Database Systems. Springer-Verlag, Workshops in Computing series, pp 306–315

Widom J, Ceri S (eds) (1996) Active Database Systems. Morgan Kaufmann, San Mateo, Calif.

Widom J, Finkelstein SJ (1990) Set-Oriented Production Rules in Relational Database Systems. In: Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp 259–270

Widom J, Cochrane RJ, Lindsay BG (1991) Implementing set-oriented production rules as an extension to starburst. In: Sernadas A, Lohman GM, Camps R (eds) Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona. Morgan Kaufmann, San Mateo, Calif., pp 275–286.

This article was processed by the author using the LaTeX style file *pljour2* from Springer-Verlag.