NAOS
Efficient and modular reactive capabilities in an Object-Oriented Database System

C. Collet
Christine.Collet@imag.fr

T. Coupaye
Thierry.Coupaye@imag.fr

T.Svensen
Thomas.Svensen@imag.fr

LGI-IMAG, University of Grenoble, BP 53 38041 Grenoble cedex 9, France.

Abstract

This paper describes the design and implementation of NAOS, an active rule component in the object-oriented database system O_2. The contribution of this work is related to two main aspects. The first concerns the integration of the rule concept within the O_2 model, providing a way to structure applications. Rules are part of a schema and do not belong to a class. Program execution and data manipulation, including method calls, can be driven on rules. The second aspect concerns the way NAOS interacts with the kernel of the O_2 system. To support a reactive capability the object manager semantics has been extended, thus providing an efficient event detection. Applications produce events and the subscribed event types react to these events. As a result, rules are triggered.

1 Introduction

The work presented in this paper concerns the integration of active rules in the O_2 object-oriented database system [BDK92]. It is part of the GOODSTEP project[1] whose goal is to provide a platform suited for Software Engineering Environments (SEE) built using the O_2 system. The aim of SEE is to help users through the software engineering process. In SEE data are typically strongly interrelated thus implying sophisticated and safe update propagation mechanisms. Also, data updates may correspond to different levels of granularity: changing an attribute value, editing a procedure, compiling a module, etc. Therefore, in SEE it is necessary to manage links between entities and to support in a uniform way automatic change propagation. High-level operations (i.e., tools invocation) also have to be propagated and controlled. For example, the system may support policies such as the one concerning change of a source code module: "whenever a source code module change is validated (1) call the compiler with the module as parameter and (2) call the linker with the object code of the module as a parameter as well as other object codes associated with this module." Such a policy contributes to controlling, monitoring and assisting teams in performing their activities. In order to support such policies, different approaches have been proposed based on AI techniques, process programming and rules [BK91, OZT90].

In the framework of GOODSTEP, active rules have been incorporated into O_2 as a means of supporting SEE semantics, mainly for: (i) notifying users, i.e., programmers of SEE or end-users, (ii) application access logging, (iii) organizing related application programs, (iv) tools communication, (v) change propagation, and (vi) maintaining data consistency. In NAOS (O_2 Native Active Object System), rules are Event-Condition-Action (ECA) rules. An event of type E is able to trigger multiple rules. For each triggered rule, if the condition C holds, then execute action A. The main features of our work are as follows:

- Rules constitute a programming paradigm which allows applications to be structured. Rules are considered at a higher level than programs, methods and data manipulation. They are defined as elements of a schema, at the same level as class or

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994**

[1]GOODSTEP is the Esprit III project No. 6115.

application definitions. This approach allows to control the execution of more general operations than method calls, i.e., updates of collections, program executions, etc. Rules respect encapsulation and transparency : they are triggered by authorized operations related to persistent or transient entities. Conditions and actions of rules are permitted to operate on persistent or transient entities. Rules can be exported/imported, increasing reusability. For the purpose of programming guidelines, rules are isolated from programs and methods: a rule can be activated or deactivated only through rules.

- The execution of a rule takes place within an application, more precisely within user-defined transactions and programs (read-only transactions) and interacts with one or more databases. Two kinds of rules are considered: *Immediate* rules which have an instance oriented semantics and *deferred* rules which have a set oriented semantics. Rule execution is based on the notion of execution cycles. A cycle describes the execution of a sequence of operations belonging to a user-defined transaction, a program or a rule. Every execution cycle is associated with a *delta structure* containing data related to the triggering operation(s).

- NAOS has three main components: a rule definition module, an event detector and an executive module. The rule definition module offers a specific interface for rule programming. This interface offers direct access to the rule manipulation primitives for allowing dynamic creation or modification of rules within an application. Also, direct access to the event detector allows for dynamic event types subscriptions and thus for rule enabling. With this approach, NAOS can be used by O_2 application developers or software integrators.

  General rule structures are stored as persistent objects while conditions and actions are stored as methods. For efficiency reasons, runtime structures are built for rule execution avoiding multiple access to the object manager when rules are executed. This approach makes it possible to alter rule definitions of a schema without interfering with the execution of applications.

This paper is organized in the following way. Section 2 presents the decisions we made for integrating rules in the O_2 system. It also summarizes the main features of some research prototypes supporting active rules in order to motivate the design and implementation of our rule system. Section 3 concentrates on the resulting rule language. Section 4 discusses our rule execution model. Section 5 presents some aspects of the architecture of our system. Brief conclusions and future research directions are given in Section 6.

2 Integrating rules in O_2

This section first reviews the main characteristics of the O_2 system. The reader may find further details in [BDK92, AC93]. Then, the decisions we made for integrating rules in the O_2 system, are introduced.

2.1 Features of the O_2 Data model

An O_2 *schema* is a set of definitions. The main structure of a schema consists of a set of classes related by inheritance links and/or composition links. A schema also contains definitions of types, functions and applications. An O_2 *base* groups together *objects* and *values* which have been created in compliance with a schema.

An *object* has an identity, a value and a behavior defined by its methods. Objects are class instances and values are type instances. In the remainder of this paper, the entity concept is used for referencing either objects or values. The value of an entity corresponds to the content of an O_2 value or to the value of an O_2 object. A given object can be shared (referenced) by several entities. By default, objects and values created during program execution are not persistent. To become persistent, i.e., stored in a database, an entity must be directly or indirectly attached to a *name*, i.e., a persistent root belonging to a schema.

A *class* definition consists of a type definition and a set of methods. A *type* is defined recursively from atomic types (integer, boolean, char, string, real, and bits) or classes and constructors (tuple, list and set). Methods are coded using the O_2C[2] language which allows to express manipulations on persistent entities as well as non-persistent entities. O_2SQL, the query language of the system, can be used to express boolean expressions on entities as well as SQL-like queries on collections.

An O_2 application is a set of programs whose execution take place in reference to a schema and to one or more associated databases. When an application starts up, it opens a *read-only transaction*. Programs of the application executed within this transaction can manipulate transient entities (apply methods, modify values, etc.) but is restricted to read-only access to persistent entities. A *(read-write)transaction* must be initiated before updating persistent entities in order to ensure consistency. Transactions in O_2 are flat atomic transactions. Commands such as `transaction`,

[2] O_2 and all product names derived from it (O_2C, O_2API, O_2SQL) are registered trademarks of O_2 Technology.

validate, commit and abort are provided for managing transactions.

Encapsulation is provided at different levels. First, properties (attributes or methods) are private to their class by default. Also, programs are private to their application. Encapsulation is also provided at the schema level as elements of a schema cannot be used by another schema. In order to increase reusability, O_2 provides an import/export mechanism.

The following is an example of O_2 schema. The Person class describes persons; The Employee class is a subclass of Person and describes employees of a company. The Task class characterizes a task of a project including the employees participating in this task and the set of documents (modules) produced for realizing the task. An instance of class Document is basically a text; it belongs to a task and can refer to (or be referred to by) other documents. The_boss and GP are names of instances of classes Employee and Task, respectively. Application project_management includes a public program create_Task for creating a new instance of Task. An instance of Employee is given to this program when it is invoked by the manage_Task program.

```
class Person public type tuple (
        name: tuple(family_name: string,
                    first_name: string),
        spouse: Person,
        age: integer)
end;
class Employee inherits Person
public type tuple (
        profile: list(string),
        salary: integer))
end;
class Task public type tuple (
        name: string,
        manager: Employee,
        group: set(Employee),
        documents: set(Document))
end;
class Document type tuple (
        name: string,
        content: Text,
        project: Task,
        used_by: set(Document),
        uses: set(Document))
method public compile
end;
name The_Boss: Employee;
name GP: Task;
application project_management
program get_information_Manager(m:Employee),
          get_information_Task(t:Task),
          public create_Task(m:Employee),
          public manage_Task,
          ...
end;
```

2.2 Design choices and motivations

From the simple ECA paradigm, different kinds of active rules and therefore different kinds of active systems have already been proposed. The features of these systems depend upon whether the rules are defined for a general purpose database system or for specific applications [ACC+93]. In designing our rule system we took into consideration works on database production rules which have focused on (i) relational systems: Postgres[SJGP90, SK91], Starburst[LLPS91, AWH92] and Ariel[Han92] and (ii) object-oriented systems: HiPAC [DBM88, DHL90, Day88, HLM88, MD89, Cha89], Ode [GJ91, GJS92b, GJS92a], O_2 [MP91][3], SAMOS [GGD91], Sentinel [AMC93] and TriGS [KRRV94]. We also consider specific approaches for incorporating rules in software engineering environments: Marvel [BK91], ALF [OZT90] and Adele [EBAM92].

The following provides a motivation for our approach based on the characteristics and limitations of the existing systems, especially the object-oriented ones. Issues in designing a rule language are considered in Section 2.2.1. The rule processing semantics with respect to transaction management is discussed in Section 2.2.2. The integration of rules with object-oriented concepts is discussed in Section 2.2.3. Issues in rule implementation are discussed in Section 5.

2.2.1 Rule definition

The main aspects for defining a rule language concern event specification, condition and action definitions, and the binding between rule parts. This section also describes the operations we allowed on rules.

Event specification

The event part of a rule specifies a type of events. These events are usually divided into two categories: primitives (called basics in Ode) and composites [Day88, GGD91, GJS92a]. Three kinds of primitive events are considered: internal events related to database operations, temporal events and external events. Composite events are made up of other composite or primitive events. In HiPAC, SAMOS and Sentinel, events are mostly related to the state of objects and to the actions applied to them. They may occur whenever a database operation takes place, such as an access to an object, an attribute value update, a method execution or the call of a transaction primitive. In SAMOS, temporal events and abstract events (named and defined by the application) are considered

[3] In order to distinguish between this earlier work done in the framework of the O_2 prototype and our rule system NAOS, we will refer to the prior as ProtoO_2.

as well. SAMOS's abstract event is called external notification in HiPAC. In Ode events are not considered (propositions in [GJS92b] have not been integrated in the Ode prototype). ProtoO_2 supports only message and time related events (neither composite events nor events related to transactions are provided). In TriGS, only message events are considered.

Although composite events and temporal events have been studied, they are not part of the current implementation of NAOS and therefore, this paper only concerns primitive event types related to (i) read/write operations on entities or parts of entities, and (ii) code execution (method execution, and program/application/transaction processing). Event types are parts of rule structures. They can be seen as class attributes as in [BM91]. However, within NAOS, event types are objects and they can be dynamically created, modified or deleted. As we will see events (instances of event types) also exhibit the properties of objects as in [AMC93] and may become persistent.

Conditions

Conditions are made up of predicates over the database state. These predicates involve query language expressions (the simplest being retrieving an attribute value), method calls and simple logical expressions. In some systems, such as Ode, conditions are part of the event specification as a mask which qualifies the event and can refer to parameters of the method call defining the event. In TriGS, conditions are queries against the triggering object or the parameters of the triggering method. In NAOS, conditions are predicates over persistent or transient entities, defined as O_2 queries. The target of a query is the actual database or the data associated with the triggering operation (an entity, the parameters of a method or a program). Also queries may use methods. However, we cannot eliminate side-effects produced by update of entities while evaluating a condition like in SAMOS, but the rule system ensures that the evaluation of a condition cannot trigger another rule. This is done by deactivating the detection of all events while evaluating a rule condition.

Actions

In the context of object-oriented systems, actions are messages to objects, database operations or procedure calls. Depending on the level at which a rule is defined, the scope of its action is a class or a hierarchy of classes. In NAOS, actions are pieces of O_2C code that may operate on persistent and transient entities. The action scope of a rule is defined by the class hierarchy of the schema to which the rule belongs. It is not restricted, like in Ode, to a class. As in HiPAC, our approach gives a far better flexibility because we can either have very simple actions (e.g. a transaction's abort, a method invocation, the signal of a user-defined event) or arbitrary composition of actions which may trigger other rules. A rule can also contain a call to a program. Such a rule can be executed only in the framework of the application to which the program belongs. This means that the rule will have to be deactivated for any other application.

Event-condition-action binding

There are different notions of binding. In HiPAC and Sentinel, conditions and actions may refer to parameters specified in the event part of the rule. These parameters generally refer to the object on which the triggering operation occurred. However, no mechanism is provided for accessing the "old" value of the object. In HiPAC, an additional mechanism is used for referencing in actions the results of queries of conditions. In Sentinel, it is not clear that actions can refer results of conditions. In Ode there is no mechanism for condition-action binding. Only the object concerned with the event is accessible. In SAMOS data related to the triggering operations are not visible in actions. In NAOS, data related to events are stored in *delta structures* whose type depends on the kind of the triggering operation and the type of the rule. The rule language provides a means to name these structures and to manipulate them in conditions and actions. Actions can also refer the result of queries of conditions.

Operations on rules

NAOS offers specific operations for manipulating rules. By means of the `create`, `delete`, `modify`, `display`, `rename` commands, rules can be manipulated as well as other elements of an O_2 schema. When an application of a schema is executed, the rules belonging to this schema are enabled. However, all these rules may not be relevant to the application execution. The `enable` and `disable` operations allow a rule to be activated or deactivated, respectively. The syntax used for specifying such operations is similar to the one used for applying a method to an object. For example, `R->disable` will deactivate rule `R`. This correspond to unsubscribe its corresponding event type. To conclude, in order to treat only events of interest for an application, a rule may have to be triggered after the start of the application for deactivating some rules.

2.2.2 Rules and transactions

In most rule systems, user-defined transactions trigger rules and it is natural to view a triggered rule as a sub-transaction of the triggering transaction. E-C and C-A coupling modes proposed in HiPAC define how an

event, a condition and an action, i.e., sub-transactions relate to each other and the transaction having triggered the rule. The links with transaction processing can be defined in different ways. For example when a triggered sub-transaction (a condition) is scheduled right after the occurrence of an event, the E-C coupling mode is said to be *immediate*. On the other hand, the triggered sub-transaction can be scheduled for execution at the end of the triggering transaction. This is called a *deferred* coupling mode. Several combinations of these coupling modes have been defined in HiPAC and SAMOS. Coupled and decoupled executions of triggered transactions have also been considered. In Ode a week E-A coupling mode is provided: the action is executed after (but not immediately after) the event. In ProtoO_2, only immediate E-C and C-A coupling modes are provided. In Sentinel, immediate and deferred rules are proposed but the links between these kinds of rules and the transactions are not clearly described. TriGS supports immediate and deferred coupling modes. Furthermore, it provides a separate coupling mode. In such a mode, a rule is executed within a new transaction, independently of the triggering transaction. No dependencies are considered between the triggering method and the separated rule.

In NAOS the O_2 transaction model serves as a basis for defining when and how rules are executed. Conditions and actions are coupled sub-transactions of the triggering transaction. As the O_2 model does not provide a nested transaction mechanism, these sub-transactions are not real transactions : they are units of execution for rules, so called execution cycles. The initial cycle is defined as a sequence of operations before the actual execution of a rule. This allows us to consider *immediate rules* triggered by operations which take place in a read-only transaction and/or a read-write transaction, and *deferred rules* triggered by operations which take place in a read-write transaction. Every execution cycle is associated with an event history used to build the delta structure of each rule considered in the cycle.

2.2.3 Rules and Object-Oriented concepts

In NAOS, rules are components of a schema and are defined at the same level as classes and applications. This approach offers the possibility of specifying rules triggered by events concerning one or more entities, possibly from different types. We do not provide external/global and internal/local rules as in SAMOS and TriGS which can be confusing from the programmer's point of view. Further, providing rules as part of a class definition leads to class update problems when modifying rules.

In most of the rule systems based on object-oriented systems, rules are first class objects. However, the integration of the rule concept with the classical properties (encapsulation, inheritance, overriding and overloading) of object-oriented concepts is not clearly discussed. In NAOS, two levels are clearly considered: the model level where rules are not objects and the implementation level where rules are objects. When defining rules in a schema using the rule language, the programmer does not consider rules as objects or methods. However, using the Rule Programming Interface, the programmer sees rules as objects. The only operations available on these rule objects are those introduced in Section 2.2.1.

Considering encapsulation, rules of NAOS respect encapsulation: only public operations (methods, programs, update, etc.) can generate events. The O_2 persistency transparency principle is also respected when programming rules. Rules can be triggered by operations on entities which may or may not be persistent. As a part of a schema, a rule definition can refer all the other definitions of the schema(classes, types, name objects or values, etc.). Rules can be imported and re-used in another schema assuming the elements used in its definition are imported as well.

For the purpose of inheritance, rules are classified by their relationship to classes. In fact only event types are considered and propagated across the class hierarchy. Let us consider the event type, say `E = op(C)` of a rule r. Such an event type characterizes an operation op on an instance of C. If one of the sub-classes of `C` is `C1` then the event type `E` is inherited in `C1`. As a result, when an operation `op` occurs on an instance `C1`, rule r is triggered and any rule r_1 with event type `op(C1)` is also triggered.

For the purpose of overriding and overloading, let us consider rule r with the event type `E` as defined above, the condition `C` and the action `A`. As in TriGS, the signature of such a rule can be represented as the function `r:E->(C->A)`. Overriding means overriding the event type `E` and the type of `(C->A)` in a subclass of `C`. O_2 inheritance is defined with a subtyping semantics. Changing the type of `E` means that the triggering class may be changed by any of its subclasses. The current prototype allows the programmer to define the rule `r':E1->(C1->A1)` where `E1= op(C1)` is a sub event type of `E` and `C1->A1` overrides `C->A`. As a result, when an operation `op` occurs on an instance `C1`, rules r' and r are triggered. More work is necessary for allowing the overriding and overloading of rules as the new rule r' cannot have the same name as rule r. However, priorities can be defined between r' and r and, rule r' may be disabled in action of rule r. With this approach, only rule r' will be executed but it is the programmer's responsability to enforce the overriding.

From an application development point of view, rules are considered at a higher level than O_2 programs, methods and data manipulations. This means for example that the beginning or the end of a program execution may trigger rules. But programs cannot explicitly manipulate rules except when they use the rule programming interface. Therefore, when using our rule language, a rule can be manipulated only through rules, e.g., the activation and the deactivation of rules cannot be done directly in the body of a program but only in actions. With this approach the programmer is forced to specify the rules well separated from methods and programs thereby avoiding undesirable linkage between two programming paradigms.

3 Rule language

3.1 General structure

The overall structure of a rule definition is as follows:

```
[create] rule <rule name>
[precedes <list of rule names>]
[coupling <coupling mode>]
[in <execution mode>]
on <event type>
[with <name of the associated delta structure>]
[if <condition>]
do [instead] <action>
```

- *rule name*
  Each rule has a name in the schema it belongs to.

- *list of rules*
  As one event may trigger several rules there is a need for ordering between the rules of a schema. A default total ordering is based on the order in which rules were defined. As in [ACL91], priorities between rules allows to define a total ordering for rule processing. This order endows the rule system with deterministic behavior.

- *coupling mode*
  The *<immediate,immediate>* combination means that a condition is evaluated right after event detection and, if it holds, action is immediately scheduled for execution. This (default) combination characterizes *immediate* rules which respond to operations on a single entity. The *<deferred, immediate>* combination means that the condition evaluation and action execution take place after the last operation of the triggering transaction, but before it validates or commits. This combination characterizes *deferred* rules which respond to aggregate and cumulative changes to an entity. Deferred rules are set-oriented. Note that the C-A coupling mode can be omitted as it is always *<immediate>*.

- *execution mode*
  Execution mode has to be given only for immediate rules. It specifies whether an immediate rule has to be triggered by an event which occurs within a read-only transaction(*r_trans*), a read-write transaction(*rw_trans*), or both(*r&rw_trans*). For deferred rules, the execution mode is always *rw_trans* and can be omitted. It is clear that the `in` clause may be not necessary when considering composite events with the sequence operator. In that case, the first event type of the sequence is an application event type `transaction_begin` (see bellow).

- *event type*
  Events may be connected to manipulating database entities or executing transactions, programs or applications. An event type characterizes a particular situation detectable by the O_2Engine, the kernel of the O_2 system. Primitive event types proposed in NAOS have been described in [CHCA94]. *Entity manipulation event types* characterize events which are produced when manipulating entities, i.e., when objects are created (new) or deleted, values are modified, entities become persistent or transient, and messages are send to objects. The rules defined with these event types may either be immediate or deferred. *Application event types* are related to the execution of O_2 applications. These event types characterize the events generated by the beginning and end of application, program or transaction execution respectively. The rules triggered by events of these types are always immediate rules as deferred execution gives no meaning in this case. NAOS also considers *user-defined event types* that characterize situations which are not necessarily associated with entity manipulation operations, program executions, etc. Such event types are defined independently from a rule definition. They belong to an O_2 schema and are uniquely identified by their names. A user-defined event is explicitly generated by using the `signal` operation in a code.

  An event type specification also describes the moment of generation (*before* or *after*) when an event of this type has to be generated with respect to the actual triggering operation. The default generation moments is always *after* if this is possible for the operation concerned.

- *delta structure name*
  The `with` clause of a rule enables programmers to name the delta structure which will be associated with the rule at runtime. This delta structure contains data related to the triggering operation(s).

According to whether the rule is immediate or deferred the structure is known as a delta element or a delta collection. Section 4.2 introduces the types of the delta structures and the operations such as **new**, **old**, **current** provided for referencing data contained in a structure without having to know about its type.

- *condition*
  The condition is a formula composed of predicates on objects and values. A predicate is an O_2SQL query. A predicate is true when the corresponding query's result is true or non-empty. A predicate may refer the delta structure of the event part. The result of a query can be denoted by a variable declared in the **if** clause.

- *action*
  The action can be an executable O_2C code; it may abort the current transaction in which the corresponding event occurred. The most simple form of an action is a method applied to the object concerned by the corresponding event. More generally, an action may use the delta structure of the event part of the rule and the result of the query of the condition.

- *instead*
  When a rule is executed it may sometimes be desirable to cancel the triggering operation. However, the cancellation of a triggering operation only makes sense if rule execution is *immediate*. Further, as the O_2 transaction manager does not provide nested transactions, the cancellation is reasonable only if the rule involved is triggered prior to execution of the triggering operation. Therefore, only rules with *before* event types are considered. The cancelling of a *before* event is materialized as in [SK91] by **instead** as part of the **do** clause of a rule. When a rule with such a clause is executed, the triggering operation is cancelled and the actions of the **do instead** clause are executed.

3.2 Examples of rules

The following rules belong to the O_2 schema of Section 2.1. They are used to control salaries of employees participating in the GP's task and to propagate the execution of a compile method on a document.

```
create rule Update_employee_salary
coupling immediate
in rw_trans
on before update Employee->salary with e
if new(e)->salary > 2 * e->salary
do instead {
    notify_increase_salary(The_Boss, e);
}
```

```
create rule compile_propagate
coupling deferred
on method_end Document->compile with d
if range of doc_to_compile is set(Document)
    select doc->used_by
    from doc in d
    where doc->used_by != set()
do {o2 Document dd;
    for(dd in flatten(doc_to_compile)) dd->compile;
}
create rule create_Task_authorization
coupling immediate
in r_trans
on program_begin create_Task(m)
   in application project_management with p
if arg(p)->m != The_Boss
do instead { display("You are not authorized
    to create a task");
}
```

The event type of the first rule is **BEFORE UPDATE (Employee, salary)**. It characterizes the modification of the salary attribute of an Employee's instance. This instance may be persistent or transient as the rule will be executed within a read-write transaction. When rule **Update_employee_salary** is triggered, its associated delta element is made up of the Employee's instance whose **salary** attribute has to be modified, and the new value for this attribute. **e** denotes the instance of Employee before modification and **new(e)** denotes the instance of Employee after modification. The condition holds if the updated employee belongs to the GP group and has a salary with a value twice its previous value. If the condition holds, the employee named **The_Boss** is notified instead of updating the salary of **e**.

The **Compile_propagate** rule specifies a propagation policy for document compiling. The event type of this rule is **AFTER METHOD_END compile(Document)**. At the end of a transaction, possibly including multiple compiling of different documents, the rule is triggered. Its associated delta collection **d** is a set of tuples, each of them describing the execution of a compile method on a document **doc**. The condition holds if at least one compiled document is used by other documents. The results of the query, **doc_to_compile** is a set of sets of "used_by" documents. The action of the rule flattens this set. Then, every document of the resulting set is compiled.

The **create_Task_authorization** rule specifies an authorization checking for the execution of program **create_Task(m)**. The event type of this rule is **BEFORE PROGRAM_BEGIN create_Task (project_ management, m)**. The rule is only triggered when the **create_Task** program is called within a read-only transaction of the **project_management** application. When the rule is triggered, its associated delta element **p** describes the actual call of **create_Task**. **arg(p)** denotes a tuple de-

scribing the actual parameters of the program. If attribute `m` of this tuple does not reference the object `The_Boss`, the execution of the program is refused.

4 Rule execution

The rule execution model covers the aspects of (i) coupling modes, (ii) multiple rules triggered by the same event, (iii) cascading rule execution, (iv) delta structures and (v) net effect of events. Section 3.1 introduced the *coupling modes* we provide and the *precedence* relationship for managing multiple rule executions. In this section *cascading* will be explained for immediate and deferred rules respectively. Then we will briefly present *delta structures* and *net effect*.

4.1 Cascading execution

Executions of immediate and deferred rules take place in execution cycles. An execution cycle describes the execution of a series of operations which belong to a transaction, a program or to the condition and action part of a rule. Whatever the coupling mode under consideration, the rules triggered are always executed in a new execution cycle distinct from the one to which the triggering operation belongs. Furthermore, if more than one rule is executed in a single cycle, they are executed in an order corresponding to their respective priorities. Figure 1 and 2 show the execution of rules r_1, r_2 triggered by event `e1` of type `E1`, r_{1a} and r_{1b} triggered by event `e2` of type `E2`. Rule r_{2a} is triggered by event `e4`. These rules are defined under the following precedence (<) relationships: $r_1 < r_2$ and $r_{1a} < r_2 < r_{1b}$.

4.1.1 Immediate rules

Immediately triggered rules are executed depth first. This approach is closely related to the one proposed in [WCL91] but rules considered in NAOS respond to operations on a single entity. The sequence of operations executed up to the triggering event defines an initial execution cycle. Then, every subsequent rule execution defines a new nested execution cycle.

In Figure 1, the rules are considered to be immediate assuming a transaction in which an event `e1` of type `E1` occurs. The operations of the current transaction executed before `e1` defines *cycle 0* the initial execution cycle. In the case of an ***after*** event type, this cycle also includes the triggering operation. When `e1` occurs, rules r_1 and r_2 are triggered. r_1 having the highest priority is executed first and defines a new execution cycle, *cycle 1(a)* in Figure 1. In this cycle, the condition of r_1 is checked and assuming it is true, the action part is executed. As one can see in Figure 1, r_1 produces event `e2` of type `E2` which in turn triggers rules r_{1a} and r_{1b} (r_{1a} precedes r_{1b}). Both rule executions define (sub) execution cycles, *cycle 2(a)* and *cycle 2(b)*, respectively. The depth first execution order implies that events produced in an execution cycle are treated without considering rules already triggered but still to be executed. For example, when event `e2` occurs, selection of rules concerns r_{1a} and r_{1b} and we consider the precedence relationship between these two rules only. We do not add r_{1a} and r_{1b} to the initial set of triggered rules. Finally, r_2 is executed taking into account the composition of the initial execution cycle (*cycle 0*) and the r_1 execution cycle(*cycle 1*(a)) with its subcycles *cycle 2(a)* and *cycle 2(b)*.

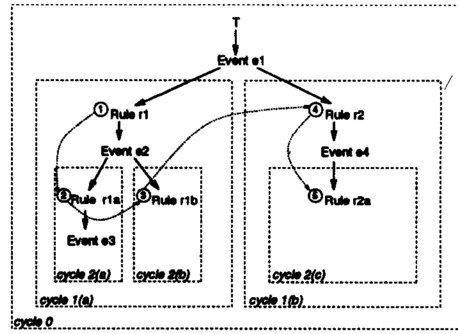

Figure 1: Execution cycles for immediate rules

4.1.2 Deferred rules

Deferred rules are executed at the end of the transaction in which the triggering event occurs but before its commit or validate. Operations of the transaction constitute *cycle 0*, while *cycle 1* will contain the execution of the rules triggered in *cycle 0*. Thereafter, *cycle n+1* executes the rules triggered in *cycle n*, thereby enforcing the width first execution order.

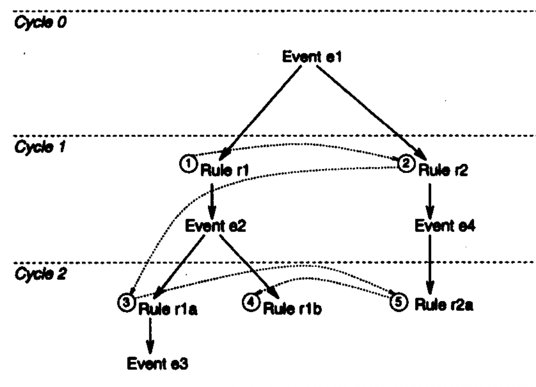

Figure 2: Execution cycles for deferred rules

In Figure 2 all rules are considered to be deferred. Event `e1` occurring in *cycle 0* triggers rules r_1 and r_2 which are then scheduled for execution in *cycle 1*. The execution of rule r_1 triggers the two rules r_{1a} and r_{1b} , but these are not executed until all the rules of

cycle 1 are finished. In other words, rule r_2 is executed before any of these, producing event `e4` which triggers rule r_{2a}. *Cycle 2* is then initiated to execute the rules in the order r_{1a}, r_{2a} and r_{1b}. Assuming that these rules do not trigger any deferred rules, *cycle 3* will never be created. Rules r_{1a} and r_{1b} see the effects of the operations executed in the initial cycle (*cycle 0*) and in the r_1 and r_2 execution cycles (*cycle 1*). More generally, deferred rules of a cycle see the effect of operations performed since the beginning of the transaction. When there are no more deferred rules to be considered, the transaction is validated or committed.

The examples given are rather simple. One may have immediate rules being triggered during the execution of deferred rules. Thus, a rule may be characterized as "the rule deferred cycle 2, immediate cycle 4". The nesting may in theory be infinite, there are no limitations on the number of immediate cycles, or deferred cycles, nor is there any theoretical limit to the number of rules that may be triggered by any one event.

4.2 Delta structures

4.2.1 Delta elements

The execution environment of an immediate rule is known as a delta element. It contains (i) the entity which is concerned with the operation producing the event and, (ii) the inserted, deleted or updated data or the actual parameters of a method or a program.

Data contained in a delta element is accessible even though the programmer does not know the details of the type of this element. The name of a delta element (cf. Section 3.1) may be used to construct views which give a simplified description of the information contained in the designated delta structure. In order to construct these views five operators are proposed, i.e., `new`, `old`, `current`, `delta` and `arg`. The `current` operator is the one assumed if none is specified. The `arg` operator is used with event types related to methods and programs. The `delta` operator is used with event types characterizing insertion/deletion of elements in/from sets, lists or bags.

Let us consider rule `Update_employee_salary` of Section 3.2. The delta element associated with this rule is `e` with type:

```
tuple(ENTITY: Employee,
      COMPONENT: integer)
```

The condition of this rule could also have been written `new(e)->salary > 2 * current(e)->salary`. At runtime, `current(e)` refers the ENTITY part of `e` and `new(e)->salary` refers the COMPONENT part of `e`.

4.2.2 Delta collections

A deferred rule responds to cumulative changes to entities. When a deferred rule with event type `E` is executed, the system considers every event of type `E` which has occurred during the previous execution cycle(s). These events may concern the same entity or different entities. Therefore, the execution environment of a deferred rule reflects the changes that have occurred on a set of entities. These changes are necessarily of the same type. To each entity is associated the modified, inserted or deleted data. The resulting delta collection is in other words also a set of delta elements. The operators we introduced in the previous section can also be used to refer data of delta collections. In that case, the operators build set of entities.

4.3 Net effect of events

Two of the main tasks of a rule system are (i) to determine which rules have to be executed and, (ii) to build their corresponding delta structures. These two tasks are realized considering the net effect of a sequence of operations performed in the triggering transaction. For instance, if a rule is triggered by the creation of an entity, but this same entity happens to be deleted before the actual execution of the rule, the rule should not be executed. Further, during a cascading execution of rules, such as the one in figure 1, the execution of a rule (e.g., r_{1b}) may nullify the effect of an event (`e1`) having triggered a rule (r_2) so that the latter no longer has reason to execute. Also the execution of a rule may change the value of the entity on which the triggering event occurred. For example, when rule r_2 in figure 1 is executed, it sees the net effect of all operations executed on the entity in previous execution cycles. NAOS computes the net effect of events based on the classical composition of pairs of operations applied to the same entity. If the net effect was not taken into account, we would have a rule system in which some rules would be executed while they should not and in which incorrect results could be obtained because of inconsistent execution environments.

5 Implementation

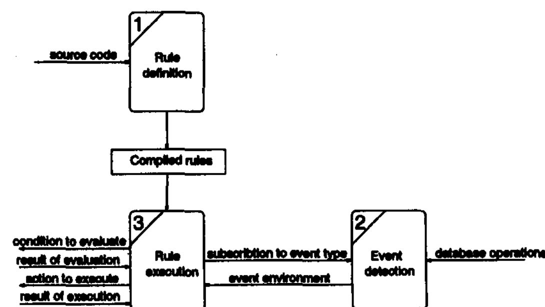

Figure 3: Architecture of the rule system

The overall structure of the implementation is depicted in Figure 3. Module 1 (Rule definition) creates the per-

sistent representations of a rule (*Compiled rules*). It basically compiles rule definitions into O_2 objects and O_2C methods. It also offers the possibility of displaying, renaming, modifying or removing previously defined rules. Module 2 (Event detection) detects events. When a schema is chosen, subscriptions are sent to the O_2Engine for all event types associated with the rules of this schema. Then, it starts up the "Rule execution" module. The latter is then able to process rules in response to events of the subscribed types occurring during the execution of an application. For each of the detected events this module constructs the delta structure and sends it to the execution module. Module 3 (Rule execution) receives detected events from module 2 and executes the concerned rules taking into account the coupling modes, cascading in the sense of execution cycles, priorities between rules, and the calculation of net effect.

5.1 Rule definition

The rule definition module is a modular tool which offers a language for rule programming and a rule programming interface for dynamic creation or modification of rules within an application. The programming interface is useful for programmers who want to use reactive processing for implementing specific languages or tools, e.g., declarative integrity constraint languages, tools communication or change propagation facilities. Figure 4 shows the architecture of the rule definition module.

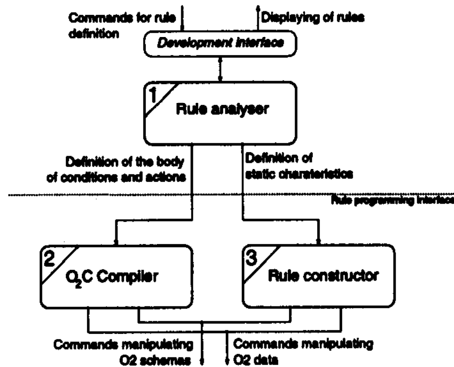

Figure 4: Architecture for the rule definition module

The top layer allows rule definitions to be written using the syntax shown in Section 3. It principally contains module 1, the rule analyzer, which realizes the classical tasks of a language analyzer. From the source code written by the programmer it produces two outputs, (i) the static characteristics of a rule that will specify, for instance, when and how the rule should be executed and (ii) the O_2C methods representing the conditions and actions of the rules. These methods belong to two classes named `Conditions` and `Actions`. These two classes do not have attributes, they simply act as place-holders for the conditions and actions.

The bottom layer represents the minimum requirements of the rule definition module. It contains the O_2C compiler and the rule constructor. The O_2C compiler provides an executable version of the condition and action, while the rule constructor takes the static characteristics of a rule as input and creates the corresponding O_2 objects. These objects are stored in persistent O_2 lists ordered by the priority of their corresponding rules. These two modules together allow the creation and manipulation of rule definitions through a rule programming interface. This interface is a set of C functions, based on the O_2 Application Programming Interface (O_2API).

To conclude, the representation of rules as persistent objects in O_2 lists provides three main advantages. First it allows the use of clusters and indexes which provide easy and efficient selection of rules through O_2SQL for manipulating rules. Second, at runtime, it allows a fast rule set initialization because the lists containing the rules are ordered (cf. Section 5.2). Also at runtime, rules are basically compiled O_2C code which allows for efficient evaluation of conditions and execution of actions.

5.2 Event detection and rule execution

The event detector is a most vital part of NAOS for two major reasons. First of all, there can be no rule executions without events, but at the same time it is the part most susceptible to a huge performance penalty unless a very efficient checking technique is employed. The event detection module of NAOS is incorporated into the O_2 engine to minimize the overhead of event checking. Also, to speed up actual rule execution, a C++ snapshot of the rule definitions is created when an application is executed and more precisely when a schema is set. With this approach, there is only one access to the object manager for each rule.

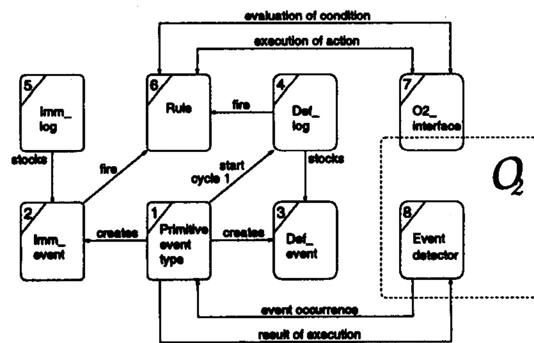

Figure 5: Event detection and rule execution

The event detector is based on a subscription mechanism. As we already said when a schema becomes

active, a subscription is made for each of the event types concerned. For each subscription, the address of a handling function is supplied, thereby making for a dynamic subscription mechanism that may also be used by applications other than our rule system. This function will necessarily relate to our event types and delta structures. The O_2 object manager regards all operations on objects as potential occurrences of the subscribed event types.

After the subscription process is finished, the event detector, module 8, starts its surveillance of database operations. Only at the arrival of an event of a subscribed event type will further actions take place, i.e., the appropriate delta structure is sent to the function supplied at subscription time.

The call of a function associated to an event type starts the rule execution module. As shown in figure 5, a number of C++ classes, modules 1 through 7, have been conceived to implement all the features of our rule model. Module 1 manages a class hierarchy representing all possible event types. In each object representing an event type, there is an ordered list of rules which can be triggered by this event type, what we call "rule indexing". Rule indexing can be seen as an transposition of the Rete algorithm [For82] which is used in Artificial Intelligence for object pattern matching. It avoids running through all the rules to find the one to be triggered when an event occurs. Module 2 and 5 takes care of immediate rule execution while module 3 and 4 are in charge of execution of deferred rules. Class `Rule`, module 6, is actually a run-time snapshot of the persistent O_2 rule definitions, created to increase performance and improve accessibility.

6 Conclusion

This paper introduced the NAOS component for the O_2 Database System. It described the model, the language and its first implementation. In NAOS, event types are related to manipulation of entities and code execution (methods, programs and transactions). Also, an event associated with the manipulation of a certain attribute can be tracked and not only the manipulation of the entire entity. Consequent work has been done concerning ECA binding by introducing the notion of delta structures and how to access these delta structures in conditions and actions.

Performance has widely been taken into account in the implementation of NAOS, as it is an important issue in proving the usefulness of an active rule system: (i) the event detector is part of the O_2Engine and treats only events for which the event type has been subscribed to, (ii) the executive module uses a dynamic C++ snapshot of the persistent O_2 objects representing the rules and the trigger indexing technique to find the rules to be executed, and (iii) condition and action parts of rules are compiled into O_2C methods. This allows for efficient evaluation of conditions and execution of actions.

In the immediate futur we plan to expand our event detector to take into account temporal and composite events, and to consider a more flexible transaction model, as the one proposed in [ADF+93], for rule execution. We also wants to investigate how to provide some parallel execution of rules. Further research directions includes (i) investigating the notion of inheritance and overriding in NAOS, (ii) specifying and implementing a rule programming environment including debugging and visualization tools. These tools may be considered as a partial answer to theoretical problems such as termination and more generally understanding of rule behavior.

Acknowledgements

This work grew out of earlier research with P. Habraken; discussions with him were helpful and greatly appreciated. We also want to thank A. Chabert for the coding of the rule definition language analyzer and M. Adiba, P. Dechamboux and C. Roncancio for useful discussions about our work.

References

[AC93] M. Adiba and C. Collet. *Objets et Bases de Données : Le SGBD O_2*. Hermès, 1993.

[ACC+93] M. Adiba, C. Collet, T. Coupaye, P. Habraken, J. Machado, H. Martin, and C. Roncancio. Trigger Systems: Different approaches. Rapport de Recherche Aristote-SUR007, LGI-IMAG, France, June 1993.

[ACL91] R. Agrawal, R. Cochrane, and B. Lindsay. On Maintaining Priorities in a Production Rule System. In *Proc. of the 17th International Conference on Very Large Data Bases*, pages 479–487, Barcelona, Spain, September 1991.

[ADF+93] T. Atwood, J. Duhl, G. Ferran, M. Loomis, and D. Wade. *Object Database Standard: ODMG-93*. Kaufmann, San Mateo, California, 1993.

[AMC93] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proc. of the 1993 ACM-SIGMOD*, pages 99–108, Washington, DC, May 1993. ACM press.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proc. of the 1992 ACM-SIGMOD*, pages 59–68, San Diego - USA, May 1992. ACM Press.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database - The story of O_2*. Morgan Kaufmann, 1992.

[BK91] N.S. Barghouti and G.E. Kaiser. Scaling up Rule-Based Software Development Environments. In *Proc. of the 3rd European Software Engineering Conf., ESEC'91*, Milan - Italy, October 1991.

[BM91] C. Beeri and T. Milo. A Model for Active Object Oriented Database. In *Proc. of the 17th International Conference on Very Large Data Base*, pages 337–349, Barcelona, Spain, September 1991.

[Cha89] S. Chakravarthy. Rule Management and Evaluation : An Active DBMS Perspective. *SIGMOD Record*, 18(3):20–28, September 1989.

[CHCA94] C. Collet, P. Habraken, T. Coupaye, and M. Adiba. Active rules for the Software engineering platform GOODSTEP. In *Proc. of the 2nd International Workshop on Database and Software engineering - 16th international conference on Software Engineering*, Sorrento, Italy, May 1994.

[Day88] U. Dayal et al. The HIPAC Project: Combining Active Databases and Timing Constra ints. *SIGMOD Record*, 17(1), March 1988.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules Are Objects Too : A Knowledge Model For An Active, Object-Oriented Database System. In *Proc. 2nd International Workshop on Object-Oriented Database Systems*, pages 129–143, September 1988.

[DHL90] A. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 204–214, Atlantic City, USA, May 1990.

[EBAM92] J. Estublier, N. Belkhatir, M. Ahmed-Nacer, and W.L. Melo. Process Centered SEE and Adele. In *Proc. of the 5th Int. Workshop on CASE*, Montreal - Quebec, July 1992.

[For82] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *J. Artificial Intelligence*, 19:17–37, 1982.

[GGD91] S. Gatziu, A. Geppert, and K.R. Dittrich. Integrating Active Concepts into an Object-Oriented Database System. In *Proc. of the 3rd International Workshop on Database Programming Languages: Bulk Types & Persistent Data*, pages 399–415, Nafplion, 1991. Morgan Kaufmann.

[GJ91] N. Gehani and H.V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proc. of the 17th International Conference on Very Large Data Base*, pages 327–36, Barcelona, Spain, September 1991.

[GJS92a] N. Gehani, H.V. Jagadish, and O. Shmueli. Composite Event Specification in an Active Databases: Model and Implementation. In *Proc. of the 18th International Conference on Very Large Data Base*, pages 327–338, Vancouver, USA, 1992.

[GJS92b] N. Gehani, H.V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, San Diego, USA, 1992.

[Han92] E. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proc. of the ACM-SIGMOD*, pages 281–290, June 1992.

[HLM88] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model for Active Database Management Systems. In *Proc. 3rd International Conference on Data and Knowledge Bases*, pages 171–179, June 1988.

[KRRV94] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and S. Vieweg. TriGS making a Passive Object-Oriented Database System Active. *JOOP - To be published*, 1994.

[LLPS91] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to Starburst: objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, October 1991.

[MD89] D. McCarthy and U. Dayal. The Architecture of An Active Data Base Management System. In *Proc. of the ACM SIGMOD*, pages 215–223, May 1989.

[MP91] C.B. Medeiros and P. Pfeffer. Object Integrity Using Rules. In *Proc. of the ECOOP - LNCC 512*, pages 219–230, 1991.

[OZT90] F. Oquendo, JD. Zucker, and G. Tassart. Suppport for software tool integration and process-centered software engineering environments. In *Proc. of the third International workshop on Software Engineering and its Applications*, Toulouse, France, December 1990.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. of the ACM SIGMOD*, pages 281–290, Atlantic City, USA, May 1990. ACM Press.

[SK91] M. Stonebraker and G. Kemnitz. The Postgres next generation database management system. *Communications of the ACM*, 34(10):78–93, October 1991.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proc. of the 17th VLDB*, pages 275–285, Barcelona - SP, September 1991.