Design and Implementation of Chimera Active Rule Language*

Giovanna Guerrini

DISI - Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy
guerrini@disi.unige.it

Danilo Montesi

DSI - Università di Milano, Via Comelico 39/41, 20135 Milano, Italy
montesi@dsi.unimi.it
School of Information Systems - University of East Anglia, Norwich NR4 7TJ, UK
dm@sys.uea.ac.uk

Abstract

This paper describes the implementation of active rules in the Chimera object-oriented database system. The Chimera active rule language is very rich, since it provides alternative semantics for active rules and combines several innovative features. We show how a run-time support for that active rule language is designed and implemented through the necessary data structures, functional components and algorithms. The paper also compares Chimera active rule implementation with other active database system implementations.

Keywords: Active databases, trigger support, design and implementation.

1 Introduction

Active database systems [0] provide active rules that execute actions in response to specific events. An active rule is a syntactical construct to define the reaction of the system and is usually specified by means of the *event-condition-action* (ECA) paradigm: events are monitored and their occurrence activates (triggers) the rule, a condition is a declarative formula that must be satisfied in order to execute the action; an action is a sequence of database operations (possibly causing state changes). The introduction of active rules in a DBMS

* This work has been supported by ESPRIT project P6333 IDEA.

is important mainly for integrity constraint enforcement, for view materialization, for auditing and security issues and for different kinds of knowledge base processing. Thus, active databases are different from traditional databases (passive ones) in which the only operations performed (queries or updates) are those explicitly requested from the applications or the users. This view leads to see the active features as an orthogonal dimension in database design and implementation. Indeed, research on active databases has begun on the relational model, but active rules may be added to a database independently from the underlying data model. Several systems supporting active rules in an object-oriented data model have indeed been proposed [0,0,0,0,0,0] and some prototypes have been developed. Unfortunately, to the best of our knowledge, there is no detailed description about the design and implementation of active rules in object-oriented databases.

The aim of this paper is to describe the design and implementation of the active rule language of Chimera [0,0] through data structures, functional components and basic algorithms. Since Chimera relies on an object-oriented data model, this paper provides a new insight on the design and implementation of active object-oriented database systems. Chimera integrates an object-oriented data model, a declarative query language based on deductive rules and an active rule language for reactive processing [0]. The reactive component constitutes the most innovative and challenging feature of Chimera. This is the reason to consider only the design and implementation of this part. Chimera supports the usual *set-oriented* active rules computation [0]: they react to sets of changes to the database and may perform sets of changes. In most active object-oriented databases, active rules are associated with objects through methods. Rules are triggered by method activations and are used as devices for testing pre and post-conditions for methods applications to individual object instances. The Chimera approach is substantially different: it uses set-oriented active rules, activated as the effect of several, logically indistinguishable events affecting multiple objects. This approach is consistent with the rest of Chimera which supports a set-oriented declarative query and update language. Moreover, active rules in Chimera have several innovative features; indeed, they support: optional composition of event effects, when the same object is the target of multiple operations; different models for processing events; different activation times; mechanisms for accessing intermediate states of affected objects during transaction execution. The above features are orthogonal to each other and are combined for the first time in the Chimera language allowing different semantics to be specified for different active rules.

In this paper we describe how Chimera active rules have been implemented in the Chimera prototype developed at Politecnico di Milano, on top of the ALGRES extended relational database environment [0]. The prototype is focused on the integration of object-orientation and active behavior, thus it deals with compilation and implementation of triggers and the development of a run-time component to provide the support for trigger execution. In this paper we describe the structures to be maintained and the main operations to be performed in order to support all the features of Chimera active rule language, thus we report the methods and the techniques that allow efficient implementation of the combined novel aspects of the Chimera active rule language. The main advantage of the described architecture is that it allows to handle event consumption and preservation, event composition, evaluation of event formulas and of formulas on past database states in a very efficient way. This has been obtained thanks to the synchronization of the structures handling different information through a transaction-level timestamping mechanism, and thanks to appropriate auxiliary index structures. In particular, the event tree allows a very fast detection of the rules triggered by the occurrence of an event, and a fast composition of event effects.

The contribution of the paper is then that of describing an architecture for implementing set-oriented active rules on an object-oriented database system. The richness of the Chimera active rule language causes that architecture to cover most of the features supported by other active rule languages. The proposed architecture brings some similarities with the ones developed for other set-oriented active rule languages, namely Starbust [0] and NAOS [0]. However, as we will discuss in detail in Section 7 it extends them, since the Chimera language is richer, and it has some advantages over them, mainly due to the maintenance of an explicit event base, synchronized with other structures by means of the timestamping mechanism and appropriately indexed, rather than of the transition/delta tables employed in those systems.

The remainder of the paper is organized as follows. Section 2 introduces Chimera active rule language. Section 3 presents the overall architecture of the rule processor, Section 4 explains how information is recorded and coordinated during block execution [1]. Rule selection and execution are described in Section 5. Section 6 illustrates the described implementation through an example. Section 7 provides a comparison with other active system implementations taking into account the differences among the underlying rule languages. Finally, Section 8 concludes the work.

2 Chimera Active Rule Language

In this section we describe the syntax and the execution model of active rules in Chimera.

[1] A block is an execution unit, that can be either a transaction unit in usual processing, or an active rule in reactive processing.

2.1 Language

Active rules in Chimera are called triggers. Each trigger consists of four components: events, condition, actions and priority. Moreover each trigger is characterized by trigger options defining the processing and the event consumption modes. The definition of a trigger is:

```
define     [TriggerOptions] trigger TriggerName [for ClassName]
events     TriggerEvents
condition  ConditionFormula
actions    Actions
[PriorityOptions]
```

Active rules may be defined in the context of a single class, in which case they are called *targeted* rules, or in the context of multiple classes, in which case they are called *untargeted* rules. Targeted triggers use only the concepts defined in the class they refer to and are included in the signature of the class. This notion is relevant for schema design and modularization, but there is little syntactic difference and no semantic difference between targeted and untargeted triggers. In the following we discuss in details the trigger components.

Events

Events define the conditions under which an active rule is *triggered*. Such conditions can be update operations or queries. In case of events as update operations performed over instances of object classes, events are denoted by the name of the update operation and the target (class name, possibly attribute name) of the operation. Primitive update operations available in Chimera are object creation (both temporary and persistent), deletion, update, object migration in inheritance hierarchies and their change of persistence status. In case of events as queries performed over object classes, events are denoted by the name of the target of the query (either a class or an attribute of a class).

Example 1 *Given a class* `employee` *containing an attribute* `salary` *examples of events are*

`create(employee)`, `modify(employee.salary)`, `query(employee.salary)`

In triggers targeted to class `employee` *those events are shortened as* `create`, `modify(salary)`, `query(salary)`*, respectively.*

Condition

The condition is a formula that serves the purpose of monitoring the execution of the reaction part. Condition of triggers may contain, in addition to conjunction of atoms, *event formulas* and references to *old states*. Event formulas are particular formulas supported by the declarative language of Chimera, built by means of the binary predicates *occurred* and *holds*. Syntactically, these predicates have two arguments: an event name and a variable name. The variable appearing as second argument of the event formula ranges over the objects of the class affected by the event, and it becomes bound to the identifiers (OIDs) of objects which were subject to the event; each OID bound by the computation of an event formula is called an *event instance*. The distinction between predicates *occurred* and *holds* is that in the former case all events which originally caused rule triggering are bound, while in the latter case some events are excluded: precisely, those events whose effect was compensated by subsequent events on the same object, thus computing the net effect of event instances. Compensations are performed as follows [2]:

- a sequence of *create* and *delete* primitives on the same object, possibly with an arbitrary number of intermediate *modify* primitives on that object, has a null net effect;
- a sequence of *create* and several *modify* primitives on the same object has the net effect of a single create operation;
- a sequence of several *modify* and a *delete* primitive on the same object has the net effect of a single delete operation on that object.

Example 2 *Referring to class* `employee` *of Example 1 above, an example of condition, in a trigger targeted to class* `employee`*, is the following*

```
occured(create,X), X.salary > 4000
```

The formula is satisfied by those objects `X` *created as members of class* `employee` *whose (current) value for attribute* `salary` *is greater than 4000.*

By contrast, a condition of the form

```
holds(modify(salary),X), X.salary > 4000
```

is satisfied by those objects `X` *of class* `employee` *not affected by creations or deletions, whose attribute* `salary` *has been modified and (currently) has a value greater than 4000.*

[2] Currently the *holds* predicate only applies to the primitives *create, delete* and *modify*, which are the most significant. However, other update primitives of Chimera, such as *specialize* and *generalize*, could be considered for net effect as well.

References to past database states are allowed in active rule conditions by the function *old*, that can be applied to atomic formulas, indicating that the respective formula is to be evaluated in a previous database state. The chosen state depends on event consumption mode (see below). If the rule is event-preserving then the old state refers to the state prior to the transaction start. If the rule is event-consuming, then the old state is the one produced by the last rule execution; if the rule has never been executed during the transaction, then the old state refers to the state prior to the transaction start.

Example 3 *Referring to class* `employee` *of Example 1 above, the condition, in a trigger targeted to class* `employee`*,*

```
occured(create,X), old(X.salary) > 4000
```

is satisfied by those objects `X` *created as members of class* `employee` *whose past value for attribute* `salary` *was greater than 4000.*

Actions

The action is a sequence of database operations, including update or display primitives, class operations or transactional commands. Condition and action may share some atomic variables, in which case the action must be executed for every binding produced by the condition on the shared variables. Moreover, operations that constitute the action are executed in strict sequence, because each of them may have side effects.

Example 4 *Referring to class* `employee` *of Example 1 above, consider a trigger targeted to class* `employee`*, with condition*

```
occured(modify(salary),X), integer(Y), integer(Z),
Y = X.salary - old(X.salary), Y > 5000,
Z = old(X.salary) + 5000
```

and action

```
modify(employee.salary),X,Z)
```

The action is executed on all objects bound to variable `X` *by the condition, and the corresponding value of variable* `Z` *is assigned to their* `salary` *attribute.*

Priority

An ordering among rules can be specified in the `PriorityOptions` clause of the trigger definition, to control run-time trigger. Priorities can be specified as follows:

– through a statement `after TriggerList`, specifying the list of triggers after which the trigger being defined must be executed (that is, listing triggers with higher priorities);
– through a statement `before TriggerList`, specifying the list of triggers before which the trigger being defined must be executed (that is, listing triggers with lower priorities).

Such specifications define a partial order on triggers, acyclicity of the precedence relation between triggers is checked when a new trigger is defined.

We have seen that a trigger has options which allows to define the *processing mode* and the *event consumption mode*. Let us see them in details.

Processing Mode For each trigger a processing mode is specified. The processing mode of a trigger may be *immediate* or *deferred*. Immediate triggers are processed at the end of the transaction unit or reaction in which triggering occurs. Deferred triggers are processed at the end of the transaction (after the commit command). Default for processing mode is set to deferred. Note that Chimera does not support detriggering of triggered deferred rules because of net effect of events, as Starbust [0] or NAOS [0]. Thus, once triggered, a deferred rule is always executed at the end of the transaction. Chimera net effect, therefore, only affects the bindings returned from event formulas in rule conditions, while Starbust and NAOS one also affects the triggering of rules.

Event Consumption Mode Two distinct event consumption modes are possible for each trigger; this feature is relevant when a given trigger is considered multiple times in the context of the same transaction. Events can be *consumed* after the consideration of a rule, therefore, each event instance is considered by a rule only for the first execution, and then disregarded. Alternatively, events can be *preserved*, i.e., all events since the transaction start are considered at each rule consideration. Default for event consumption mode is set to consuming.

The notions of immediate/deferred and event consuming/preserving are orthogonal and they are motivated by specific applications. The large number of alternatives for triggers is motivated by their wide spectrum of applications. Event consuming deferred rules (with the use of net effect) are suited for checking static integrity constraints at transaction commit, allowing the database to be invalid at intermediate states of the transaction. Event preserving rules are required for checking of dynamic integrity constraints, event preservation is required to avoid sequences of events collectively leading to the violation of the constraint, but not individually. Immediate rules are required for view materialization or data derivation (changes to basic data are propagated immediately to derived data) or to check integrity constraints that cannot be violated even at intermediate states. Events without net-effect composition are required by triggers used for book-keeping.

Example 5 *Assume the current database has a class* `employee`*, with attributes* `name`*,* `salary` *and* `mgr`*. Suppose moreover that a subclass* `specialEmployee` *of* `employee` *is defined. Consider the following targeted trigger* `R1`

```
define immediate  trigger immAdjustSalary for employee
  events          create
                  modify(salary)
  condition       Self.salary > Self.mgr.salary
  actions         modify(employee.salary,Self,Self.mgr.salary)
```

The trigger is immediate and event consuming, it does not employ neither event formulas nor references to old states in its condition. The effect of this trigger is to disallow (also at intermediate states) that an employee earns more than its manager.

Now consider the following targeted trigger `R2`

```
define immediate  trigger recordAccesses for employee
  events          query(salary)
  condition       occurred(query(salary),X),
                  old(X.salary) > 40000
  actions         recQueried(X)
  after immAdjustSalary
```

The trigger is immediate and event consuming, it employs an event formula and a reference to an old state in its condition. The effect of this trigger is to invoke the execution of the procedure `recQueried(X)` *for each employee* `X` *which earns more than* `40000` *and whose salary has been accessed.*

Finally consider the following targeted trigger `R3`

```
define trigger spEmp for employee
  events       create
  condition    holds(create,X),
               X.salary > 40000
  actions      specialize(employee,specialEmployee,X)
```

The trigger is deferred and event consuming, it employs an event formula in its condition. The effect of this trigger is to specialize by inserting in the class `specialEployee` *each employee inserted (and not deleted) during the* *transaction which earns more than* 40000. *This specialization is performed at transaction commit.*

Finally, the following example shows the usefulness of event-preserving triggers.

Example 6 *Referring to the classes of Example 5, consider the following trigger, that selects all employees who get, in the course of the transaction, a high salary raise (possibly caused by small salary raises due to individual modify operations). Note that the rule is event-preserving, therefore all modifications since the transaction start are accumulated at each rule consideration; further, note that the condition part evaluates the salary difference between the state before transaction execution and the new state determined at active rule processing time. The reaction consists in calling the procedure* monitorSalary.

```
define event-preserving trigger modifySpecialEmp for employee
  events        modify(salary)
  condition     occurred(modify(salary),X), integer(Y),
                Y = X.salary - old(X.salary), Y > 5000
  actions       monitorSalary(X,Y).
```

Further details about the Chimera active rule language can be found in [0,0].

2.2 Execution Model

Before introducing the execution model we specify that in Chimera a transaction is a sequence of calls to the query and update primitives. Query primitives are used either to display the content of the database or to provide bindings to variables. Transaction units, which we refer to as transaction lines, define the scope of variables. Thus, a Chimera transaction is a sequence of transaction lines, appropriately delimited. Each transaction line contains itself queries and update primitives; it may contain as well procedure invocations. Transaction lines define the scope of variables which are shared by different operations and play a relevant role in fixing the semantics of database triggers, as described below. An example of transaction line is:

```
select (X where employee(X), X.dept = 13),
modify(employee.salary, X, X.salary + 100);
```

In the transaction line above, X is bound by the execution of the select operation to a set of OIDs, and these identify the employees whose salary is modified. Note that the condition-action sequence of an active rule is very similar to a query-modify sequence in a transaction line. Thus triggers may be regarded as transaction lines from the execution viewpoint.

When one of the events of an active rule occurs, the rule is said to be *triggered*; several rules may be triggered at the same time. Trigger processing time depends on the trigger processing mode: at the end of the transaction line or at transaction commit. Although initiated at different times, rule processing of immediate and deferred triggers is conducted in the same way as an iterative execution of rule processing steps, each of which in turn consists of three phases, called rule selection, consideration and execution.

- *rule selection* consists in choosing non-deterministically one of the rules at highest priority;
- *rule consideration* consists in the evaluation of the condition, which is a declarative formula; at this point the selected rule becomes untriggered;
- *rule execution* occurs if the condition is true or produces some bindings, and it is performed by sequentially executing the operations in the reaction part of the rule.

Trigger execution consists of queries and updates, which may in turn trigger other rules; rule processing continues until no more rule is triggered. Clearly, the possibility of infinite rule processing due to chains of active rules that trigger each other exists in Chimera. Whenever a transaction unit is completed, active rule processing is applied to immediate triggers until a fixpoint is reached (called a *quiescent state* with respect to immediate triggers); at commit time, active rule processing is applied to *all* triggers until a *final state* is reached. The transactional command *savepoint* forces rule processing over all triggers (including deferred ones); rule processing started by a *savepoint* command produces an intermediate transaction state which is quiescent with respect to all triggers.

3 Architecture of the Active Rule Processor

In this section we present the overall architecture of the active rule processor in the Chimera prototype developed at Politecnico di Milano. The prototype is built on the ALGRES environment [0] but it is highly independent from ALGRES, being thus portable to any other target system. Therefore ALGRES features are encapsulated through the definition of a Virtual Interface of Chimera (VICHI) which supports a set of services. An overall description of the prototype may be found in [0,0,0], while details on the components of the processors are presented in the following sections.

Figure 1 introduces the overall architecture for reactive processing, showing both data structures and functional components. We denote with rectangles the structural components and with ovals the functional components of the architecture. Plain arrows represent information flow, while dashed arrows

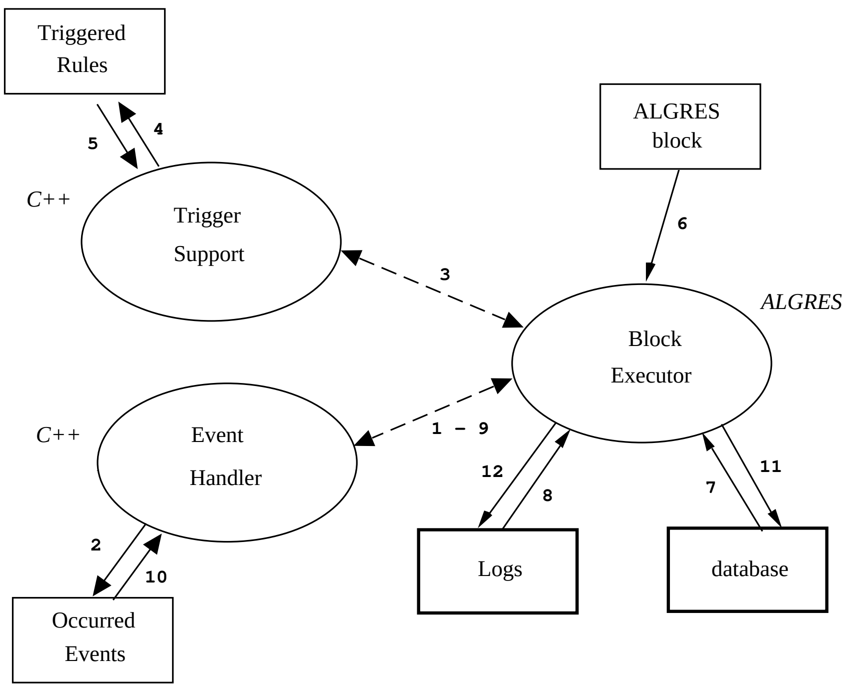

Fig. 1. Reactive processing architecture

represent control flow among modules. An incoming plain arrow to a data structure represents writing of information into the data structure, while an outgoing plain arrow from a data structure represents reading of information from it.

The data maintained by the rule processor are:

- *Occurred Events:* This is a global event base which records all the event instances relevant to a particular transaction.
- *Triggered Rules:* This table keeps track of the triggered rules and records the evolution of event processing for each individual rule. Actually, the table has an entry for each defined rule and it also maintains general information about rules, duplicating, for efficiency reasons, information stored in the rule catalog in the database.
- *Log:* This is a structure defined only for those classes on which at least one rule containing an *old* predicate is defined; Logs maintain all past states of objects of that class[3].

The three main functional components are:

- *Block Executor:* It retrieves the correct ALGRES block and invokes an AL-

[3] Each class has associated a log including all the attributes of the class. To improve efficiency, we have developed a *logging on demand* mechanism that enables logging for a class only if there is a rule with an *old* predicate referencing objects of that class.

GRES Executor for processing the relevant compiled code. Each transaction line corresponds to a single ALGRES block, while the compilation of a rule produces a pair of blocks, one corresponding to the condition and the other one to the action[4].

- *Event Handler:* It is responsible of event handling, that is of appropriately storing the events occurred during block execution in the Occurred Events table and for evaluating on this table event formulas. It is activated at the end of each block execution for storing the occurred events, and at the beginning of the evaluation of rule conditions containing event formulas.
- *Trigger Support:* It is responsible for selecting the next rule for consideration and for managing all the dynamic information on rules. It is activated at the end of each transaction unit execution as well as at *commit* or *savepoint* evaluation; it stops when there are no more rules to be selected for consideration (a quiescent or final state has been reached).

In the following we briefly illustrate the interactions among reactive processing components, referring to the numbers that label edges in Figure 1. For each transaction line in the transaction, the Block Executor executes it and then the Event Handler is activated (edge 1), for properly updating the Occurred Events table storing the events occurred during the execution of the block. As we will see in a greater detail later, when the Event Handler inserts the occurred events in the table (edge 2) it contemporaneously determines the rules triggered by the events, which are passed (as a return value) to the Block Executor. Now the Trigger Support is activated (edge 3) and it fills the Triggered Rules table according to the triggered rules that are passed to it (edge 4). Then, the Trigger Support selects from the Triggered Rules structure a triggered rule to be executed thus detriggering the rule (edge 5). Now, the Block Executor is invoked to execute the ALGRES blocks corresponding to the selected rule condition and action in compiled form (edge 6). During the evaluation of rule condition the Block Executor may access the database (edge 7), it may access the log for evaluating formulas on past states (edge 8), and it may moreover temporarily call the Event Handler (edge 9) to access the Occurred Events structure when evaluating event formulas (*occurred/holds* predicates) (edge 10); after this evaluation, the Event Handler returns the control to the Block Executor together with a set of bindings. Then, if the condition is satisfied, the action is executed. The execution of the action may cause update to the database (edge 11) and to logs (edge 12), since Chimera transactions and rules are compiled in such a way that the execution of the compiled code appropriately logs object states, if the executed updates are on classes requiring logging. After executing the rule, the Block Executor calls the Event Handler to store the occurred events passing to it the occurred events and the related oids; the Occurred Events structure is thus updated

[4] This allows ALGRES to execute the condition part of a rule without executing the corresponding action.

by the Event Handler as a consequence of any update in rule action. Then, the control returns to the Trigger Support (together with the identifiers of the rules triggered by the events occurred), the Triggered Rules structure is updated and the previously described steps are iterated till there is a triggered rule. When a quiescent state is reached, the Trigger Support stops and control returns to the Block Executor which executes the next transaction line.

The above described steps refer to immediate trigger processing. In case the transaction line under consideration contains the *commit* or the *savepoint* command, no new events are generated but the Trigger Support is activated in a different modality, so that all the deferred rules in the Triggered Rules table are eligible for execution.

4 Information Recording

The information that should be maintained during the life of a transaction in order to support the semantics of triggers in Chimera regard three entities: Logs, Occurred Events and Triggered Rules. The Logs hold a partial difference between the database instance at begin transaction and the current one. The Occurred Events structure records the events that have occurred since the transaction started. Finally, the Triggered Rules structure records information about the execution state of the rules triggered as a reaction to events. The use of a transaction-level timestamp seems a natural way to achieve the necessary synchronization between objects, events and rules in order to guarantee the correct implementation of the different semantic features of triggers. Our (logical) timestamp is an integer that is set to zero at the beginning of the transaction and is increased by one after each event occurrence (update operation or query). The timestamp mechanism should satisfy the following conditions:

- Event timestamps have no duplicates, and are assigned in sequential order. At each update or query the timestamp is increased and the same timestamp is assigned to all the event instances involved (an event may involve different OIDs if the operation is set oriented).
- Rule timestamps denote the last *inflection point* of the rule, that is, the last consideration of the rule, meaning that the events having a smaller or equal timestamp have already been considered in previous evaluations of the rule. In the case of event-consuming rules, timestamps have a non-default value, while for event-preserving rules the timestamp always denotes the beginning of the transaction.
- Object timestamps are included in Logs but not in the class extension. They denote the event that has transformed that particular object instance in an old state of the object, thus forcing its logging. They are the mean for

accessing an old state of objects in the evaluation of a rule condition. It is possible to have different objects having the same timestamp.

This timestamping mechanism is a very relevant component of our implementation since it allows to handle, in a conceptually simple yet efficient way, many semantic features of Chimera triggers. The semantic features of triggers that can be implemented by means of the timestamping mechanism include event consumption, determination of rule inflection points, access to past versions of object state (evaluation of the *old* predicates) and evaluation of event formulas. Event consuming rules are implemented by updating the rule timestamp at condition evaluation time. Rule timestamp is set to the successor of the last event timestamp, then *consuming* the considered events. By contrast, event preserving rules always have a null timestamp, so that at each evaluation the rule considers all events occurred during the transaction. In this way the rule timestamp can be viewed as a pointer to the Occurred Events structure, pointing to the first candidate event for rule condition evaluation.

An *old* predicate occurring in a rule condition is evaluated by looking at the Log for the appropriate class and then finding the last entry for the given OID having a timestamp less than or equal to that of the rule. As a particular case, if the rule is event preserving, the old state is the entry in the Log with the lowest timestamp for the given OID. Indeed, the old state of an object is not univocally determined but depends on the rule that evaluates it. Thus, for properly evaluating an old state of an object in rule conditions, when logging the state a timestamp is included to distinguish among different instances of the object that may exist in the Log.

As far as event formulas are concerned, a *holds* predicate requires the evaluation of the net effect from the last inflection point of the rule, looking at the Occurred Events structure to determine the composite effect of the events having a timestamp greater than that of the rule. In case of event preserving rules the evaluation of the net effect on the entire structure is performed. Analogously an *occurred* predicate requires to get the rule inflection point timestamp and then to scan the Occurred Events structure considering only events having a timestamp greater than that of the rule. In case of event preserving rules the entire structure is scanned.

Example 7 *Consider a transaction containing as first transaction line the following:*

```
create(employee, "John Smith", 37000, null, Oid);
```

The timestamp is set to `0` *at begin transaction and it is set to* `1` *at the execution of that transaction line. A* `create(employee)` *event timestamped by* `1` *is then stored in the Occurred Event structure.*

Consider an event-preserving rule `Rp` *and an event-consuming rule* `Rc`*, both triggered by the event* `create(employee)`*. At begin transaction both rules are associated with timestamp* `0`*. after the completion of the first transaction line both rules are triggered. Once selected for consideration, the timestamp of* `Rc` *is set to* `1`*, whereas the timestamp of* `Rp` *does not vary, i.e. it remains* `0`*. This means that* `Rp` *will be triggered at each reactive processing activation in the transaction, since there is (at least) one event, whose timestamp is* `1` *that triggered it and it must consider. This also means that* old *predicates in* `Rp` *conditions will be evaluated referring to the state before transaction start (that is, log instances timestamped by* `0`*), and that, during the evaluation of event formulas, all events occurred in the transactions are considered (since all of them have timestamps greater than* `0`*). By contrast, rule* `Rc` *is triggered again only if another* `create(employee)` *event occurs. In its (eventual) subsequent condition evaluation event formulas and* old *predicates will refer to the database state at time* `1`*.*

As a further remark, we point out that in our active database prototype we have no internal mechanism for detecting and automatically propagating events into the event base; therefore we need to explicitly program such event manipulations. Updating the event base with generalization hierarchies is particularly critical, because events at given levels of the hierarchy may propagate at different levels. We have chosen of performing the analysis and generation of event instances by means of code that is generated at compile time (an alternative solution, probably less efficient, would be to propagate event instances at run time).

4.1 Data Structures

In the following we briefly describe the main features of the data structures constituting the active rule processor. A pictorial description of those structures, illustrating their fields, is presented in Figure 2.

Occurred Events

This structure has the role of event base. The structure has the following fields. It stores the event type in the fields OPERATION (the update/query operation), CLASS (the class over which the operation is performed), ATTRIBUTE/NEWCLASS (the individual attribute over which the operation acts, in case of `modify` or `query` operations, or the new assigned class in case of object migration operations). Moreover it has a field for storing the TIMESTAMP at which the event occurred and a field for the OID of the object involved in the event. The structure has an entry for each event instance. The structure

Occurred Events

OPERATION	CLASS	ATTRIBUTE	TIMESTAMP	OID

Triggered Rules

RULEID	TIMESTAMP	PRIORITY	PROCESSING	CONSUMPTION	FLAG

Class C Log

OID	A_1	...	A_n	TIMESTAMP

Event Tree

EVENTTYPE	TIMESTAMP	RULELIST

Fig. 2. Fields of the main data structures

is completely handled by the Event Handler and it is accessed for evaluating predicates *occurred* and *holds* in rule conditions (on explicit request from the Rule Executor). It is a strictly increasing data structure: only insertions are applied to it, no entry is modified and no entry is deleted until the end of the transaction. The structure lifetime is a transaction.

Triggered Rules

This structure stores information about the triggered rules to be processed. This information is stored in the following fields. RULEID is the rule identifier of the rule (considered unique in the entire database). TIMESTAMP is the timestamp of the last inflection point of the rule (always `0` if the rule is preserving). PRIORITY is the priority level of the rule, used for selecting the rule to execute; PROCESSING is the rule processing mode, while CONSUMPTION is the rule consumption mode. Moreover a FLAG is used to determine the rules to be processed, in fact the triggered rules data structure contains all the rules that have been triggered, but those with flag `OFF` have already been processed. When a rule is triggered, it is inserted with flag `ON` in the triggered rules data structure. If an entry for the rule is already present in the structure the entry is not duplicated, but if the entry has flag `OFF` its flag is switched to

`ON`, because the rule is re-triggered. When a rule is selected for execution, after condition evaluation the rule is detriggered so its flag is switched to `OFF` (at this time we perform also timestamp updating). The structure (whose lifetime is a transaction) is completely handled by the Trigger Support. No deletions of entries are performed, but only updates and insertions. It is used for storing information about the triggered rules to be processed and relating each rule which has been triggered at least one time in the transaction with the timestamp of its last inflection point. Moreover, it duplicates, for efficiency reasons, some of the static information about rules stored in the rule data dictionary. We remark that the data structure contains at most an entry for each rule of the database. We have chosen to store all the rules that have been triggered in the transaction instead of only the triggered rules that have not yet been processed. This choice is due to the fact that, to properly evaluate event formulas, we need to store the timestamp of the last inflection point of each consuming rule which has been triggered at least one time in the transaction.

An efficient access to the Triggered Rules table is got through the use of an hash table whose key is the rule identifier. Moreover, a queue structure on this table is also used, to get the rules ordered by priority.

Logs

Logs are used to maintain information about the past states of the database. They are used only for keeping old object states, needed for the evaluation of *old* predicates in active rule conditions, as a consequence the log is not a log in the usual database sense: it does not keep trace of all database updates, but only of those relevant to this evaluation (the state changing ones). We recall that to improve efficiency we consider the development of a *logging on demand* mechanism that enables logging for a class only if there is a rule with an *old* predicate referencing objects of that class. The active rules compiler maintain a table with the classes that should be logged. The update primitives compiler generates, for the modify primitives, the code to implement logging subject to the existence of an entry for the given class in the table of classes to be logged. This is a sort of parametric approach because the addition of a rule with an old predicate over a class automatically enables logging for the class without requiring recompilation of the update primitives associated with the class. The structure maintains the information in the following fields. OID is the object identifier of the object, $A_1, \ldots, A_n$ are the attributes of the class to which the log refers, the correspondent fields store the old state of the object, TIMESTAMP is the timestamp of the event that produced the logging. Note that as a Log contains the difference between the initial data base instance and the current one, it is clear that at begin transaction it should be empty and at end transaction it is no longer meaningful. The structure is updated by the Block Executor during usual as well as reactive processing. Logs are strictly increasing data structures to which only insertions are applied. They are accessed by the Block Executor for evaluating rule conditions depending on *old* states.

Event Tree

To get an efficient determination of the triggered rules from the set of occurred events an *event tree*, that is, a tree of event types, is also maintained by the Event Handler. This tree is initialized at the beginning of the session by inserting the event types triggering some rules, while it is updated as soon as events of new event types occur during the session. The event tree is the structure that allows a fast access both to the Occurred Events table and to the Triggered Rules table. Indeed, through the entry in the tree for a given event type we know the last event occurrence for the event type and which rules must be activated on the occurrence of an event with this type.

When the Block Executor has completed the block execution, calls the Event Handler to store the events that have occurred during block execution. Then these events must be inserted in the table. Thus, for each individual event, the corresponding event type is searched for in the event tree. After this search we get all the rules triggered by the event and we also get the timestamp of the last occurrence of an event with the considered event type and thus we are able to keep the events of the same type linked. Linking the events of the same type in a list ordered on the event timestamp allows a fast evaluation of event formulas. This fast evaluation is performed through a reverse scanning of the list. The information on the last occurrence of an event with the specified event type is updated to take into account the last occurred event. If the search for the event type in the tree fails, then no events with the considered event type have occurred during the session. Since the tree has been initialized with all the event types triggering rules, we know that such an event does not trigger any rule. However, the event might be used in the evaluation of event formulas in some rule condition. This happens because Chimera permits the use of event formulas on any arbitrary event, without restricting event formulas to triggering events. Thus, we currently store all events; a possible optimization consists in collecting the information about predicates used in event formulas and inserting these event types into the event tree upon initialization, then we can log only the events whose types belong to the event tree.

Example 8 *Consider the triggers of Example 5. The events that triggers some rules are* `create(employee)`, `modify(employee,salary)`, `query(employee, salary)`. *Then, in principle, an event* `delete(employee)` *needs not to be stored, since it does not trigger any rule. However, since this event affects*

Trigger Support

1: update Triggered Rules;
while a rule exists in Triggered Rules s.t. its FLAG is `ON`
do
  2: select a rule r with max PRIORITY in Triggered Rules;
  3: update the entry for r in Triggered Rules setting FLAG = `OFF` and,
    if r is consuming, TIMESTAMP = current timestamp;
  4: call the Block Executor to execute rule r condition;
  if the Block Executor returns a non empty set of bindings B
   (i.e., the condition evaluation succeeded)
  then
    5: call the Block Executor to execute rule r action on the bindings B;
    6: update Triggered Rules
  endif
enddo

Fig. 3. Trigger Support activity

the evaluation of predicate `holds(create,X)` *in trigger* `R3` *condition, since it can nullify the effect of a creation, it needs to be stored. By contrast, an event* `modify(employee,mgr)` *needs not to be stored, since it is not employed in any trigger conditions, nor it modifies the net effect of events employed in trigger conditions.*

5 Rule Selection and Execution

In this section we outline the main steps of reactive processing specifying the execution model of rules in our prototype.

5.1 Rule Selection

The overall activity of run time support in the context of reactive processing is outlined by the cycle in Figure 3. The cycle in Figure 3 is to be applied in the case of reactive processing started at transaction commit or at a *savepoint*, when all triggered rules have to be considered. In the case of reactive processing started at the end of a transaction line (when only immediate rules are to be considered) the operations done by run time support are similar, but Step 2 is done only among rules with PROCESSING MODE = `immediate`, and the cycle is done while a rule exists in Triggered Rules s.t. its FLAG is `ON` and its PROCESSING MODE is `immediate`.

During rule selection (Step 2), when looking for a rule with the greatest priority [5], if two or more rules with equal priority are found, the choice of which of them to execute is taken non deterministically according to [0]. The rule priorities that the Trigger Support gets, together with all other information about rules, when it fetches the rule data dictionary, are static priorities. These priorities are computed starting from the graph that represents the rule partial ordering derived from `before/after` clauses in rule definitions. These priorities are translated into a total order (integer numbers) which is nondeterministically chosen by the Trigger Support. This total order is obviously consistent with the partial one derived from class definitions. The user may however require to the Trigger Support to recompute rule priorities starting from the partial ordering graph, so as to enable the user to indicate which of the rules with the same priority should be executed first.

Moreover, note that:

- Step 1 is needed to keep into account events occurred during usual processing, between different activations of reactive processing;
- Step 2 is efficiently handled through the use of the priority queue that keeps the Triggered Rules structure ordered by priority;
- Step 4 is performed by a call $evaluate(r, ts, B)$ with r rule id of the selected rule, ts its timestamp and B set of bindings satisfying the condition, returned by the Block Executor;
- Step 5 is executed only if the evaluation of the rule condition produced at least a binding (that is, $B \neq \emptyset$); it is performed by a call $execute(r, B, R)$ with r rule identifier, B set of bindings on which the rule must be executed, R set of triggered rules returned by the Block Executor; this call also results in getting the set of events occurred during the execution of rule action to be properly added to the Occurred Events structure by the Event Handler.

Steps 1 and 6, that is, the Triggered Rules structure updating, are performed by the Trigger Support by simply inserting the rules returned from the Event Handler after any Occurred Events structure updating. Indeed, as a consequence of the insertion of events in the Occurred Events structure the set of

[5] We remark that this search is trivial since it makes use of the queue structure on the Triggered Rules table.

```
procedure update_triggered_rules(R)

  for each r ∈ R do
    if r.RuleId is not in Triggered Rules then
      1: insert in Triggered Rules the tuple
        r.RuleId, 0, r.Priority, r.Processing, r.Consumption, ON
    else
      if the entry in Triggered Rules for r.RuleId has Flag = OFF then
        2: set the Flag field for this entry to ON
      endif
    endif
  endfor
```

```
procedure re_trigger_preserving

  for each r in Triggered Rules do
    if r.Consumption = preserving then
      set the Flag field of r's entry in Triggered Rules to ON
    endif
  endfor
```

Fig. 4. Updating the Triggered Rules structure

rules triggered by the occurrence of that event is determined and passed to the Trigger Support. This allows an efficient determination of triggered rules since the Trigger Support avoids to examine the Occurred Events data structure, the rule data dictionary and the current version of the Triggered Rules data structure, to determine for each rule whether some triggering events have occurred. Thus, the event tree avoids running through all the rules to find the ones to be triggered when an event occurs.

Let us examine that in more detail. We have seen in Section 4 that the entry in the event tree for a given event type contains the rules triggered by that event type. Thus, those rules are passed to Trigger Support to be inserted in the Triggered Rules structure. Moreover, to properly handle preserving rules (which must reconsider all events occurred since the beginning of the transaction), at each reactive processing activation (Step 1) we set the FLAG of all the preserving rules already in the Triggered Rules structure to `ON`, without searching for any event in the Occurred Events structure. In fact, if a preserving rule has been triggered once in a transaction, it is always triggered in the transaction, and at each reactive processing activation it has to be evaluated.

Thus, let R denote the set of rules passed to the Trigger Support by the Event Handler (thus, the rules -either consuming or preserving- triggered by events occurred in the last transaction unit). Then, the updating of the Triggered Rules structure is performed by the procedure $update\_triggered\_rules(R)$ shown in Figure 4. In that procedure, case 1 corresponds to a rule that has not yet been triggered in the transaction, while case 2 corresponds to a rule that has already been triggered in the transaction, but that has already been processed. The rule is thus re-triggered and needs to be processed again. Note that rules are always inserted in the Triggered Rules table with a null timestamp, to model the fact that they have not yet been considered during the transaction. In Step 1, besides executing that procedure, the Trigger Support sets to `ON` the flag of each preserving rule already in the Triggered Rules structure, to re-trigger each preserving rule that has been triggered at least once in the transaction. To do that, it executes procedure $re\_trigger\_preserving$ of Figure 4.

5.2 Rule Execution

The Block Executor must handle two specific requests from the Trigger Support, that is $evaluate(r, ts, B)$, for evaluating a rule condition, returning in B the set of bindings satisfying the condition, and $execute(r, B, R)$, for executing a rule action, returning in R the set of rules triggered by the execution of the rule action. The first request is handled as follows. If the condition contains an event formulas, (i.e. `occurred(create(employee),X)` or `holds(create(employee),X)`) the Block Executor first of all calls for a service from the Event Handler, which evaluates the formula by simply accessing the Occurred Events data structure; the Event Handler returns to the Block Executor a set of bindings for the variable `X`. In case of simple event formulas (*occurred* predicate) the OID set returned simply includes the identifiers of objects on which a non consumed event of the specified type has occurred. In case of event formulas with net effect (*holds* predicate) the set is computed by difference between OID sets (the set of OIDs for which a non consumed event of the specified type has occurred minus the set of OIDs for which a non consumed event with a "complementary" event type has happened). The returned OIDs are a subset of those selected from the Occurred Event table. This evaluation is very fast, since it is performed by accessing only main memory structures. In particular, the list of events of a given type is reversely scanned starting from the last one, referenced from the Event Tree. Furthermore, the early evaluation of event formulas allows to immediately suspend the evaluation of the condition if the OID set returned by the Event Handler is empty.

Formulas on the current state (i.e. `X.salary > 500`) are handled by the Block Executor simply evaluating a query on the database, while formulas on old states (i.e. `old(X.salary) > 500`) are handled by the Block Executor accessing the appropriate logs and, if the rule is consuming, making use of the timestamp of the last rule inflection point; if the object to which the formula refers has not yet been modified within the transaction, its old state coincides with the current one, thus the database is accessed.

As far as the request $execute(r, B, R)$ is concerned, let us remark that the Block Executor, after completing the execution of each block (with the only exception of rule conditions, that do not generate event) activates the Event Handler to store the set of events occurred when executing the block. The Occurred Event structure is thus appropriately filled. Moreover, as a result of that updating, the set of rules triggered by the occurred events is also determined (as seen in Section 5.1) and passed, as a return value, to the Block Executor.

6 An Illustrative Example

Let us consider the rules of Example 5. The information computed at rule compilation time and recorded in the rule data dictionary is the following:

Rule Data Dictionary

RuleId	Events	Consumption	Processing	Priority
R1	create(employee) modify(employee,salary)	consuming	immediate	3
R2	query(employee,salary)	consuming	immediate	2
R3	create(employee)	consuming	deferred	1

Note that we have omitted the field CODE, which, for each rule, contains a pointer to the compiled rule body ALGRES code. Note that a partial order among rules has been deduced from the priority declarations and the default of assigning a greater priority to an immediate rule over a deferred one. Note, moreover, that at rule compiling time and transaction compiling time, event types are appropriately codified, assigning them an identifier. However, for the sake of clarity, in the example, we do not consider codified event types but explicit ones.

Moreover, the Event Tree index is initialized to contain the following entries:

Event Tree

EVENTTYPE	TIMESTAMP	RULELIST
`create(employee)`	0	R1, R3
`modify(employee,salary)`	0	R1
`query(employee,salary)`	0	R2

Finally, since rule `R2` contains an *old* predicate, updates to the objects of class `employee` will be logged in an Employee Log.

Suppose that the following transaction is executed:

```
begin transaction
      create(employee, "John Smith", 37000, null, Oid);
      select(X where employee(X), X.salary > 35000),
      create(employee, "Paul Young", 45000, X, Oid')
commit
```

where ";" is the syntactical delimiter of transaction lines. Suppose moreover that the new generated oids are respectively `14` and `39`. The transaction consists of two transaction lines. Now let us look how the different components of the architecture evolve when executing the transaction.

After the execution of the first transaction line the database relation for class `employee` contains a tuple with OID = `14`, NAME = `John Smith`, SALARY = `37000` and MANAGER = `null` and a tuple with OPERATION= `create`, CLASS = `employee`, TIMESTAMP = `1` and OID = `14` is inserted in Occurred Events. Moreover, the entry with EVENT TYPE = `create(employee)` in the Event Tree is updated setting TIMESTAMP = 1 and the rules `R1` and `R3` are returned to the Trigger Support.

At this point reactive processing starts. The Triggered Rules structure, empty at transaction beginning, is updated inserting those rules and thus becoming:

Triggered Rules

RuleId	TimeStamp	Priority	Processing	Consumption	Flag
`R1`	`0`	`3`	`immediate`	`consuming`	`ON`
`R3`	`0`	`1`	`deferred`	`consuming`	`ON`

Only immediate rules must be considered, therefore the only rule which can be selected for execution is `R1`. Its condition is evaluated, accessing the database, but the evaluation fails. The entry corresponding to rule `R1` in the Triggered Rules data structure is updated setting its Flag field to `OFF` and its TimeStamp field to `1`, then reactive processing stops.

The second transaction line is then executed,the database relation for class `employee` becomes:

Employee

Oid	Name	Salary	Manager
`14`	`John Smith`	`37000`	`null`
`39`	`Paul Young`	`45000`	`14`

while the Occurred Events structure becomes:

Occurred Events

Operation	Class	Attribute	TimeStamp	Oid
`create`	`employee`		`1`	`14`
`query`	`employee`	`salary`	`2`	`14`
`create`	`employee`		`3`	`39`

and the entries of the Event Tree index are updated as follows:

Event Tree

EventType	TimeStamp	RuleList
create(employee)	3	R1, R3
modify(employee,salary)	0	R1
query(employee,salary)	2	R2

Note that events timestamped by 1 and 3 are linked, since they form the list of events of type create(employee) occurred since the beginning of the transaction. Finally, rules R1, R3 and R2 are returned to the Trigger Support.

At this point another transaction line has been executed and reactive processing starts. The Triggered Rules structure is updated as follows:

Triggered Rules

RuleId	TimeStamp	Priority	Processing	Consumption	Flag
R1	1	3	immediate	consuming	ON
R3	0	1	deferred	consuming	ON
R2	0	2	immediate	consuming	ON

i.e., rule R1 is retriggered, while rule R2 is triggered for the first time in the transaction.

Only immediate rules must be considered, so rules R1 and R2 can be selected for execution. Rule R1 has greater priority and therefore it is selected. Rule R1 condition is evaluated and the entry corresponding to R1 in the Triggered Rules structure is updated setting to OFF its Flag field and to 3 its TimeStamp, i.e., rule R1 is detriggered.

Rule R1 condition holds, so its action is executed, modifying the employee salary. The database relation for class employee is updated as follows:

Employee

Oid	Name	Salary	Manager
14	John Smith	37000	null
39	Paul Young	37000	14

Moreover the Log for class `employee` is filled in the following way (we recall that class `employee` requires logging since rule `R2` makes use in its condition of an *old* predicate on an attribute of that class).

Employee Log

OID	NAME	SALARY	MANAGER	TIMESTAMP
39	Paul Young	45000	14	4

Moreover, a tuple with OPERATION = `modify`, CLASS = `employee`, ATTRIBUTE = `salary`, TIMESTAMP = `4` and OID = `39` is added to the Occurred Events structure, the entry for EVENTTYPE = `modify(employee,salary)` in the Event Tree is updated setting the TIMESTAMP to `4` and rule `R1` is returned to the Trigger Support.

The Triggered Rules structure is updated as follows:

Triggered Rules

RULEID	TIMESTAMP	PRIORITY	PROCESSING	CONSUMPTION	FLAG
R1	3	3	immediate	consuming	ON
R3	0	1	deferred	consuming	ON
R2	0	2	immediate	consuming	ON

Rule `R1` is selected again, its condition is evaluated, and its tuple in the Triggered Rules structure gets its FLAG field to `OFF` and its TIMESTAMP field to `4`.

`R1` condition does not hold, thus rule `R2` is selected for execution and its condition evaluated. The Triggered Rules structure becomes:

Triggered Rules

RULEID	TIMESTAMP	PRIORITY	PROCESSING	CONSUMPTION	FLAG
R1	4	3	immediate	consuming	OFF
R3	0	1	deferred	consuming	ON
R2	4	2	immediate	consuming	OFF

In evaluating the condition, `occurred(query(salary),X)` is evaluated on Occurred Events and produces the binding `X = 14`. The predicate `old(14.salary)` is thus evaluated on the state with timestamp = `0` (which corresponds to `R2` inflection point; rule `R2` is indeed being considered for the first time in the transaction). Object `14` has not been modified during the transaction. Therefore, its old state coincides with the current one, and the value for the attribute `salary` is `37000`. Thus, the condition does not hold.

No immediate rules are now triggered. Therefore reactive processing stops and `commit` is executed. Reactive processing is activated again and all rules are to be considered. The only triggered rule is `R3` so it is selected and its condition evaluated. The predicate `holds(create,X)` is evaluated on the Occurred Events structure, considering all events having a timestamp greater than `0` (which is the timestamp of `R3` inflection point). The evaluation produces the bindings `X = 14` and `X = 39`. Object `14` has `salary` = `37000` and object `39` has `salary` = `37000`, thus condition evaluation fails. The Triggered Rules structure is updated as follows:

Triggered Rules

RuleId	TimeStamp	Priority	Processing	Consumption	Flag
R1	4	3	immediate	consuming	OFF
R3	4	1	deferred	consuming	OFF
R2	4	2	immediate	consuming	OFF

There are no more triggered rules, thus reactive processing stops. The Occurred Events, Triggered Rules and Logs structures are emptied.

7 Comparisons with other Active System Implementations

In this section we compare the described implementation for Chimera active rules with the implementation of reactive features in well-known active database systems (for a comparison among some of these systems see [0]). Most of the active database systems implemented are based on the relational model. Thus, we first consider implementations of relational active database systems. The rule systems considered are Starbust [0], Postgres [0] and Ariel [0]. Our comparison can only be "parametric" with respect to the data model since it takes into account the differences among the rule languages. Then we consider object-oriented active database systems. The systems considered are Ode [0], NAOS [0], TriGS [0], REACH [0], Sentinel[0].

7.1 Relational Active System Implementations

Starbust rule system supports only deferred consuming rules with net effect computation. Rule language allows to refer to transition tables, which maintain tuples inserted, deleted or updated during the transaction. Such tables do not maintain the complete history of the transaction, but only its net effect. Like in Chimera, rules are related to their last consideration time. The transition tables of each rule consider the operations of the transaction that are subsequent to the last rule consideration. Note that net effect in Starbust is quite different from the one of Chimera. Starbust net effect indeed affects the triggering of rules, while Chimera one only affects the bindings returned from event formulas in rule conditions. Starbust implementation is based on a Transition log, which records all the operations occurred during the transaction that are relevant to rules. This structure is used for net effect computation and to build transition tables for each rules.

The main differences between Chimera and Starbust architectures are concerned with events and log handling. A first difference relies in the decoupling of the information that Starbust stores in the Transition Log in two Chimera structures, i.e. the event base and the logs. This decoupling is motivated by several reasons. First of all, in Chimera the event base (Occurred Events) stores only the object identifiers of the objects involved in the event (event instances), not the entire state. Past object states are saved, only if they may be accessed through an old predicate, in the appropriate log. Thus, Chimera allows a more compact technique to store information, due to OID exploitation. Note however that Starbust consider a static analysis technique of rules to detect what to store in Transition Log (deduced from triggering events and transition table references). In Chimera, by contrast, we store in logs only necessary states, but all events are stored in the event base. This is due to the fact that event formulas are not restricted to triggering events. However, we are currently investigating the possible benefits of storing only the potentially relevant events. The choice of relevant events can be done through static analysis. In Starbust the Transition Log is used both to determine the triggered rules and to build the transition tables. The first operation is performed in Chimera on the event base. In Chimera this operation is much simpler and is performed very efficiently by making use of the event tree. By contrast, in Starbust one must deduce from the Transition Log if there is an event which is relevant for the rule, which has not yet been consumed and whose effect has not been invalidated by another event. In Starbust transition tables are built from the Transition Log at run-time, extracting a tuple at a time, and computing the net effects of operations performed after the last rule consideration. By contrast, in Chimera old predicates are evaluated by simply searching a tuple in the appropriate log.

Postgres rule system is tuple-oriented instead of set-oriented as Chimera and Starbust. Rules are all (super)immediate, that is, after each single operation rules are activated. Rules are all consuming (each rule considers all events a single time) and obviously no net effect is performed. There is no parameter passing between condition and action (they are always executed on a single tuple at a time), but each time an individual tuple is accessed, updated, inserted or deleted a CURRENT tuple and a NEW tuple are present in the system, thus it is possible to refer to them in the condition (which is evaluated on the CURRENT tuple). In Postgres two alternative implementations of rules are provided: tuple-level marking and query rewriting. The idea behind marking is to place markers on all tuples for which rules apply; if a marker is encountered during execution then the rule processor is called. Markers must be maintained through modifications. By contrast, query rewriting is a static rewriting of queries to keep into account rule activations caused by the query; a query is then transformed in a set of queries by means of a rewriting algorithm. Both these approaches are heavily different from Chimera implementation, because of the deep differences between the languages.

In Ariel implementation the emphasis is placed on efficient testing of rule conditions. Ariel rule language is characterized by an optional event specification, that is, Ariel supports both event based and pattern based conditions. In Ariel a notion of transition is defined, where a transition is a sequence of operations enclosed in a do-end block. Rules are all activated at the end of each block, and net effect of events is always performed. It is allowed in rule condition to refer to the state prior the transition start (thus the previous state is fixed for all rules). Being the specification of events optional, Ariel provides an efficient evaluation of pattern based conditions. This evaluation is handled using a discrimination network. Also event detection is performed on this network. In Ariel the Chimera distinct phases of verifying if a rule is triggered and evaluating a rule condition are thus merged in a single *matching* phase; Chimera event handling and triggered rule handling mechanisms have therefore no equivalent in Ariel implementation. Chimera does not exploit discrimination networks. However, the determination of the triggered rules is a very fast operation, thanks to the event tree structure. The evaluation of event formulas in rule condition is very fast, too. Event formulas may in some cases be produced automatically from events and class formulas in order to optimize the evaluation of conditions, the early evaluation of event formulas indeed restrict the number of instances on which the condition must be evaluated. Moreover, thanks to the set-oriented nature of active rules, optimization techniques for the condition part of rules are identical to those used for the *select* query primitive.

Coming to active object-oriented database systems implementations, the emphasis in most of them is placed on efficient event composition. In particular, in Ode [0] composite events are detected by means of extended finite state machines, in Sentinel [0] event graphs are exploited, and in SAMOS [0] they make use of Petri nets, while REACH [0] detects composite events in parallel with the normal application flow.

Ode [0,0] is an active object-oriented database system developed at AT&T Bell Labs. Ode supports instance-oriented triggers based in the E-A paradigm, in which the condition can be (optionally) specified as a *mask* in the composite event. Moreover, it supports a powerful language for composing events [0]. In Ode implementation [0] event expressions are compiled into finite state machines (FSMs). FMSs are extended to handle masks by using mask states which evaluate predicates to produce the pseudo-events `true` and `false`. Event posting is achieved by rewriting invocations of methods that have associated events (through the use of wrapper functions). The run-time trigger information are stored in a persistent data structure (events from different transactions can be composed). For each trigger and for each object the trigger was activated for[6] the current state of the trigger FSM is stored. Moreover, an index is kept that maps an object to all the triggers active on that object (that index is used when posting events).

Sentinel [0] is an active OODBMS being developed at University of Florida in a follow-on project to HiPAC. Sentinel supports an expressive event specification language (called *Snoop* [0,0]). Sentinel has been implemented [0] by extending the Open OODB object-oriented DBMS. The implementation of an efficient mechanism for composite event detection is the main concern of that project. In Sentinel, primitive events are signaled by adding a notify procedure call in the wrapper method; event parameters are also collected at that stage. Each application has a local composite event detector, to which all primitive events are signaled. Composite events are detected by making use of an event graph [0]. Events are composed only within transactions. The overall architecture of Ode and Sentinel is quite similar. Both use a pre-processor to modify the user code to post events, and both support a similar set of composite events. However, Sentinel currently supports only local composite events, while Ode supports global composite events (composite events whose constituent basic events may span more than one application and more than one transaction).

[6] Note that there is a substantial difference between Ode rule language and Chimera rule language: in Chimera, once a rule has been triggered, it is executed once, no matter how many triggering events have occurred. This is due to the set-oriented nature of Chimera.

Thus, in Ode they store the trigger state in the database, while in Sentinel they store the corresponding structures in transient program memory.

Like Sentinel, the REACH project [0] is a follow-on project to HiPAC. The goal of that project is to develop an active object system that provides a mediation framework for heterogeneous data repositories. The REACH system is implemented [0] on top of the Open OODB commercial object-oriented DBMS. The goal of that implementation is to detect composite events in parallel with the normal application flow, while in the previously described approaches event posting is combined with composite event detection. However, they have been forced to some restrictions: composite events cannot be used in triggers that must be executed immediately after the composite event occurs. All the systems examined till now are focused on efficient composite event detection and are thus not easily comparable to Chimera, which, in its current version, only supports primitive events. An extension of Chimera to support composite events is presented in [0]. In that work, it is shown how the architecture described in this paper can be easily extended to support composite event detection.

TriGS [0] is an active object system proposing an event specification mechanism not only for defining the points in time for rule triggering, but also the points in time for condition evaluation and action execution. TriGS has been implemented on top of the commercial OODBMS GemStone. Referring to the classification of alternative architectures for active database systems proposed by [0] TriGS is based on a layered architecture, while all the other active object-oriented systems we consider here are based on an integrated architecture. In the TriGS architecture there are four main components: the event detector (which detects and signals events), the rule scheduler (which determines and schedules for execution triggered rules), the condition evaluator and the action executor. Events are generated (as in Ode and in Sentinel) by using method wrappers. In TriGS, they store a rule base indexed on the triggering event. This allows a fast determination of the triggered rules, comparable to the one obtained in Chimera by using the event tree structure. Finally, in TriGS an auxiliary structure is supported to allow for method overriding, with a trigger lookup mechanism. We remark, indeed, that TriGS is the only active object-oriented system supporting rule overriding. Currently, rule overriding is not supported in Chimera. However, a proposal for adding rule overriding to Chimera is described in [0]. That work also sketches how the architecture described in this paper should be modified if trigger overriding were supported.

NAOS [0] integrates active rules in the O_2 object-oriented database system. Two kinds of rules are considered: immediate rules which have an instance-oriented semantics and deferred rules which have a set-oriented semantics. Immediate and deferred rules have different execution cycles. Each rule execution is associated with a delta structure containing data related to the triggering operation. For deferred rules, both in determining which rules have to be executed and in building the corresponding delta structures, the net effect of the sequence of operations constituting the triggering transaction is considered. The net effect in NAOS is thus analogous to Starbust one, that, unlike Chimera one, may result in detriggering of rules. NAOS has been implemented by extending the O_2 system. Rule compilation results in O_2C methods for rule condition and action and in an O_2 object which stores the static features of rules. These objects are stored in persistent O_2 lists, ordered by priority of the corresponding rules. Thus, with the use of clusters and indexes, rules can be efficiently selected. Moreover, to minimize the accesses to the object manager, a C++ snapshot of rule definitions is maintained, too. That snapshot has the same role of the partial duplication of rule static information we have in Chimera Triggered Rules structure. A subscription mechanism is employed to detect only events relevant for the rules in the schema. As discussed in Section 4 that approach would be much more complicated in Chimera, since Chimera permits the use of event formulas on any arbitrary event, without restricting event formulas to triggering events. In NAOS, to get an efficient selection of the triggered rules a hierarchy of event types is maintained, such that in each object representing an event type there is an ordered list of rules which can be triggered by this event type (that is, rules are indexed on the triggering event). That structure has the same role of Chimera event tree.

8 Conclusions

In this paper we have described an implementation of Chimera active rules. The Chimera prototype based on the described architecture has been completed. The implementation is quite complex because of the richness of the Chimera active rule language which supports the specification of different rule semantics with respect to rule processing mode, consumption and composition of events. However, we have shown that the proposed implementation handles in a very simple way such a complex language. This simplicity mainly comes from the introduction of an explicit event base, synchronized with other structures by means of the timestamping mechanism. Moreover, the proposed architecture has a number of advantages, including its easy extensibility for concurrent transaction processing, the possibility of explanation support, parametricity with respect to different rule semantics. In particular, concurrent transactions can be handled by our architecture provided that a global timestamping mechanism can be supported [0]. Explanation support is obtained thanks to the presence of an explicit event base and to the timestamping mechanism which relates event occurrence and consequent rule activations. Finally, parametricity has been obtained because of the very rich nature of the Chimera language, that forced us to design an architecture suitable to different semantics.

Preliminary measures of performance are encouraging: active rule selection is very fast and thus the overall execution times are dominated by the execution time of the ALGRES code produced for transactions and for rules. As an additional remark, early evaluation of event predicates (*occurred* and *holds*) has proved to be useful for immediately suspending the evaluation of a rule condition if the relevant predicates are empty. Moreover, event formulas may in some cases be produced automatically from events and class formulas in order to optimize the evaluation of conditions; this optimization is important because event instances are normally few with respect to class instances.

The Chimera prototype moreover includes a sophisticated debugger for development of active rule applications. The debugger provides several functionalities, such as the inspection of the set of rules currently triggered, information on occurred events and on condition bindings, rule activation and deactivation, dynamic modification of rule priorities [0]. The current Chimera prototype incorporate moreover design techniques and tools for mapping views and constraints defined in Chimera into suitable rules. Tools performing static analysis of the set of defined rules to detect possible sources of nonterminating computations have been developed as well [0].

Acknowledgement

We wish to thank Stefano Ceri, who supervised our work on implementation of active rule processing, Elisa Bertino, who provided us several useful suggestions and carefully read a first version of this paper, and all the IDEA group at Politecnico di Milano. German Rodriguez helped us in designing the active rule support of Chimera. Finally, a special acknowledgement is due to Stefano Castangia, Lanfranco Colella and Pierpaolo Merli, who implemented the run-time support for Chimera active rules as part of their thesis.

References

[1] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In P. Buneman and S. Jajodia, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 99–108, 1993.

[2] E. Bertino and G. Guerrini. Trigger Inheritance and Overriding in an Active Object Database System. Submitted for publication, 1996.

[3] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmerman. Rules in an Open System: the REACH Rule System. In N. Paton and M. Williams, editors,

Proc. First International Workshop on Rules in Database Systems, Workshops in Computer Science, pages 111–126, 1993.

[4] A. Buchmann, J. Zimmermann, J. Blakeley, and D. Wells. Building an Integrated Active OODBMS:Requirements, Architecture, and Design Decisions. In *Proc. Eleventh IEEE Int'l Conf. on Data Engineering*, pages 117–128, 1995.

[5] S. Castangia, L. Colella, and P.Merli. Architecture, Design and Implementation of a Run Time Support for Chimera. Master's Thesis in Electronic Engineering, Politecnico di Milano, October 1994. In Italian.

[6] S. Castangia, G. Guerrini, D. Montesi, and G. Rodriguez. Design and Implementation for the Active Rule Language of Chimera. In N. Revell and A M. Tjoa, editors, *Proc. Sixth International Conference and Workshop on Database and Expert Systems Applications*, pages 45–54, London (UK), September 1995.

[7] S. Ceri, P. Fraternali, S. Paraboschi, and G. Psaila. The Alres Testbed of Chimera: An Active Object-Oriented Database System. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, page 473, 1995.

[8] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active Rule Management in Chimera. In S. Ceri and J. Widom, editors, *Active Database Systems*. Morgan-Kaufmann, 1996.

[9] S. Ceri and R. Manthey. Chimera: A Model and Language for active DOOD Systems. In J. Eder and L. Kalinichenko, editors, *Extending Information System Technology, Proc. Second International East/West Database Workshop*, pages 9–21, Klagenfurt, 1994.

[10] S. Ceri and R. Manthey. Consolidated Specification of Chimera. Technical Report IDEA.DE.2P.006.01, ESPRIT Project 6333, November 1993.

[11] S. Ceri, S. Crespi Reghizzi, et al. The ALGRES Project. In *Proc. First Int'l Conf. on Extending Database Technology*, Lecture Notes in Computer Science, Venice, 1988. springer.

[12] S. Ceri, S. Crespi Reghizzi, P. Fraternali, G. Guerrini, G. Lamperti, S. Paraboschi, and G. Psaila. Implementation of the ALGRES Testbed (Year-2 Version). Technical Report IDEA.DE.3P.009.01, ESPRIT Project 6333, May 1994.

[13] S. Ceri and J. Widom. *Active Database Systems - Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, 1996.

[14] S. Chakravarthy. Architectures and Monitoring Techniques for Active Databases: An Evaluation. *Data and Knowledge Engineering*, 16:1–26, 1995.

[15] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proc. Twentieth Int'l Conf. on Very Large Data Bases*, pages 606–617, 1994.

[16] S. Chakravarthy, V. Krishnaprasad, Z. Tamizzudin, and R. Badani. Eca Rule Integration into an OODBMS: Architecture and Implementation. In *Proc. Eleventh IEEE Int'l Conf. on Data Engineering*, pages 341–348, 1995.

[17] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Secification Language for Active Databases. *Data and Knowledge Engineering*, 13(3), October 1994.

[18] C. Collet, T. Coupaye, and T. Svensen. Naos: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. In *Proc. Twentieth Int'l Conf. on Very Large Data Bases*, pages 132–143, 1994.

[19] S. Gatziu and K. Dittrich. SAMOS: an Active Object-Oriented Database System. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):23–26, December 1992.

[20] N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 327–336, 1991.

[21] N. Gehani, H. Jagadish, and O. Shmueli. Event Specification in Active Object-Oriented Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 81–90, 1992.

[22] N. Gehani and H. V. Jagadish. Active Database Facilities in Ode. In S. Ceri and J. Widom, editors, *Active Database Systems.* Morgan-Kaufmann, 1996.

[23] E. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 49–58, 1992.

[24] G. Kappel, S. Rausch-Schott, and W. Retschitzegger. Beyond Coupling Modes: Implementing Active Concepts on Top of a Commercial ooDBMS. In E. Bertino and S. Urban, editors, *Proc. Int'l Symp. on Object-Oriented Methodologies and Systems*, number 858 in Lecture Notes in Computer Science, 1994.

[25] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–569, 1978.

[26] D. Lieuwen, N. Gehani, and R. Arlein. The Ode Active Database: Trigger Semantics and Implementation. In *Proc. Twelfth IEEE Int'l Conf. on Data Engineering*, 1996.

[27] D. McCarthy and U. Dayal. The Architecture of an Active Data Base Management System. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 215–223, 1989.

[28] R. Meo, G. Psaila, and S. Ceri. Composite Events in Chimera. In P. Apers, editor, *Proc. Fifth Int'l Conf. on Extending Database Technology*, 1996.

[29] N. W. Paton et al. Dimensions of Active Behaviour. In *Proc. First International Workshop on Rules in Database Systems*, Workshops in Computer Science, pages 40–57. Springer-Verlag, Berlin, 1993.

[30] 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 Int'l Conf. on Management of Data*, pages 281–290, 1990.

[31] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 275–285, 1991.

[32] J. Widom and S. J. Finkelstein. Set-Oriented Production Rule in Relational Database Systems. In H. Garcia-Molina and H.V. Jagadish, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 259–270, 1990.