Design and Implementation for the Active Rule Language of Chimera

S. Castangia[1] G. Guerrini[2] D. Montesi[1*] G. Rodriguez[1]

[1] Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza L. da Vinci 32 - 20133 Milano, Italy
montesi/german@ipmel2.elet.polimi.it

[2] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Viale Benedetto XV 3 - 16132 Genova, Italy
guerrini@disi.unige.it

Abstract

In this paper we describe how Chimera active rules are implemented in the prototype developed at Politecnico di Milano. Chimera active rule language is very rich, since it provides the integration of object-orientation with rules and the availability of alternative semantics for active rules. This paper shows how such a rich set of capabilities is designed and implemented, comparing Chimera active rule implementation with the implementations of well-known active database systems.

1 Introduction

An active database is a database in which some operations are automatically executed when specified events happen and particular conditions are met [9]. 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. An active database is able to autonomously reply to updates, when particular states or state transitions occur. An active rule is a syntactical construct to define the reaction of the system. Active rules are 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 for the action to be executed; an action is a sequence of database operations (possibly causing state changes). The introduction of active rules in a DBMS is important mainly for integrity constraint enforcement, for view materialization, for auditing and security issues and for different kinds of knowledge base processing. The 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 which support active rules in an object-oriented data model have indeed been proposed [6, 10].

Chimera is a novel database language designed with the aim of integrating active and deductive rules with an object-oriented data model[1] [5]. The reactive component constitutes the most innovative and challenging feature of Chimera. Chimera supports the usual *set-oriented* active rules computation [15]: they react to sets of changes to the database and may perform sets of changes. This approach is consistent with the rest of Chimera, which supports a set-oriented, declarative query and update language. In this aspect, Chimera is different from most other active object-oriented databases where rules are triggered by method activations, and are used to test pre and post conditions for method applications to individual objects. Active rules in Chimera are parametric with respect to several dimensions: (*i*) they support optional composition of event effects (called *net effect computation*), when the same object is the target of multiple operations, (*ii*) they support different *event consumption modes*, that is, different models for pro-

*Current address: Department of Computer Systems and Telematics, The University of Trondheim, 7034 Trondheim, Norway, danilo@idt.unit.no

[1]A Chimera is a monster of Greek mythology with a lion's head, a goat's body and a serpent's tail; each of them represents one of the three components of the language.

cessing events, (*iii*) they support different *processing modes*, that is, different activation times, and finally (*iv*) they support mechanisms for accessing intermediate states of affected objects during transaction execution. The above features are orthogonal to each other. Thus, the active rule language of Chimera is very rich and allows different semantics to be specified for different active rules. They yield a powerful language that supports a variety of applications, ranging from active integrity constraints, to derived data and view materialization, to monitoring and bookkeeping.

A prototype of Chimera focusing on the integration of object-orientation and event-driven behavior has been developed at Politecnico di Milano. The target DBMS used in the implementation is ALGRES [2], an extended relational DBMS characterized by a rich schema (supporting orthogonal type constructors such as sets, multisets and lists) and a powerful algebraic language (supporting operations over complex relations). The aim of this paper is mainly to describe the design and implementation of Chimera active rules.

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[2]. Rule selection and execution are described in Section 5. Section 6 provides a comparison with other active system implementations. Finally, Section 7 concludes the work.

2 Chimera active rule language

In this section we summarize the main characteristics of triggers in Chimera. For a complete specification of Chimera active rules and their execution model we refer to [4, 5]. Triggers are a mean of introducing specific reactions to particular events relevant to the database. Active rules are used in Chimera for expressing triggers. Triggers may be defined in the context of a single class, in which case they are called *targeted triggers*, or be defined in the context of multiple classes, in which case they are called *untargeted* triggers. 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.

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

Before introducing the active rule language we specify that in Chimera a transaction is a sequence of calls to the query and update primitives. Query primitives are used 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. A Chimera transaction is thus a sequence of transaction lines, appropriately delimited. Each transaction line contains itself queries and update primitives. Transaction lines are the basic execution units from the processing viewpoint.

A Chimera active rule consists of four components: events, condition, action and priority. Moreover each rule is characterized by a processing mode and an event consumption mode.

Events: Events indicate the primitive operations monitored by the rule; they are denoted by the name of the primitive operation and of the schema element to which the operation is applied (a class or an attribute of a class). Primitive operations are queries, object creations, deletions, modifications and object migrations in inheritance hierarchies. Queries and modifications may refer to specific attributes.

Condition: The condition is a formula monitoring the execution of the reaction part. It is a conjunction of atomic formulas and it is interpreted as an expression of predicate calculus over typed variables. Conditions may contain, in addition to usual conjunction of atoms, *event formulas* and references to *old state*. *Event formulas* are particular formulas supported by the declarative language of Chimera, built by means of the binary predicates *occurred* and *holds*. These predicates are used to inspect the events that have occurred during a transaction. Syntactically, these predicates have two arguments: an event name and a variable ranging over the OIDs of the objects affected by the event. This variable becomes bound to OIDs of instances 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 returned, 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:

- 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[3].

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 must be evaluated in a previous database state. The chosen state depends on the rule event consumption mode (see below). If the rule is event-preserving then the old state always refers to the state at transaction start. If the rule is event-consuming, then the old state is the one holding at the last consideration of the rule; for the first consideration of a rule the old state refers to the state at transaction start.

Action: 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.

Priority: An ordering among rules is defined to control their run-time selection. This ordering is defined through *before* and *after* declarations, which indicate the names of rules that have a lower or higher priority than the one being defined. Such specifications define a partial order on rules, acyclicity of the precedence relation between rules is checked when a new rule is defined.

Processing mode: The processing mode of an active rule may be either *immediate* or *deferred*. Immediate rules are considered for execution at the end of the transaction unit or reaction in which triggering occurs. Deferred rules are processed at the end of the transaction (after the *commit* command), or after the evaluation of the *savepoint* primitive. Default for processing mode is deferred.

Event Consumption Mode: Two distinct event consumption modes are possible for each active rule; this feature is relevant when a given rule 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 at its next execution, and then disregarded. Alternatively, events can be *preserved*, that is, all events since the transaction start are considered at each rule consideration. Default for event consumption mode is consuming.

[3] 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.

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 bookkeeping.

Example 1 *Consider a database containing a class* `employee`*, with attributes* `name`, `salary` *and* `mgr` *and a subclass* `specialEmpl` *of* `employee`*. Consider the following targeted trigger:*

```
define immediate  trigger adjSal 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:

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

The trigger is immediate and event consuming, it employs an event formula and a reference to old states in its con-

dition. The effect of this trigger is to invoke the execution of the procedure `recQueried(X)` *for each employee* `X` *which earned more than* `40000` *and whose salary has been accessed.*
Finally consider the following targeted trigger:

```
define trigger spEmp for employee
  events       create
  condition    holds(create,X),
               X.salary > 40000
  actions      specialize(employee,specialEmpl,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 `specialEmpl` *each employee inserted (and not deleted) during the transaction which earns more than* `40000`*. This specialization is performed at transaction commit.*

2.1 Execution model

The condition-action sequence of an active rule is essentially 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 four phases, called rule activation, selection, consideration and execution:

- *rule activation* consists in determining the triggered rules, that is the ones for which any of the triggering events has occurred;
- *rule selection* consists in choosing non deterministically one of the triggered rules at highest priority;
- *rule consideration* consists in the evaluation of the condition, which is a declarative formula; at this point the selected rule is detriggered;
- *rule execution* occurs if the condition is true i.e. 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 a state is reached where no 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 state is reached where no immediate rule is triggered (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 structure of the active rule processor in the Chimera prototype developed at Politecnico di Milano. Details of particular components are given in the remainder of the paper. A more detailed description of the active rule processor can be found in [1, 7], while an overall presentation of the prototype can be found in [3]. The prototype is built on the ALGRES environment, it is however highly independent from ALGRES, being thus portable to any other target system. ALGRES features are indeed encapsulated.

Figure 1 introduces the overall architecture for reactive processing, showing both data structures and functional components. Rectangles denote the structural components of the architecture and ovals denote the functional ones. Bold rectangles denote secondary memory structures. Plain arrows represent information flow, while dashed arrows represent control flow among modules. An incoming plain arrow to a data structure (from a process) represents writing of information into the data structure, while an outgoing plain arrow from a data structure (to a process) 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.

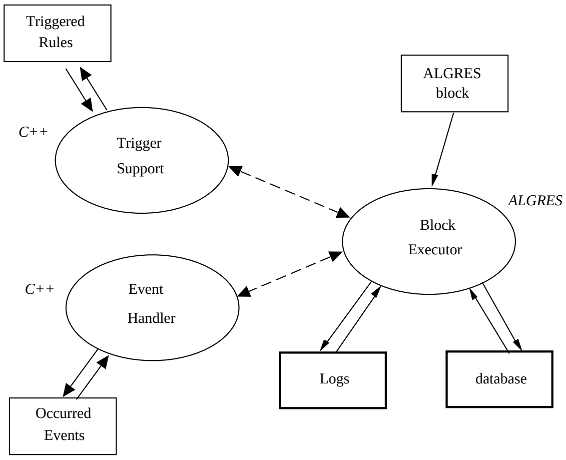

Figure 1: Reactive processing architecture

- *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[4].

The three main functional components are:

- *Block Executor:* It retrieves the correct ALGRES block and invokes an ALGRES 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[5].

- *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/final state has been reached).

For each transaction line in the transaction, the Block Executor executes it and then the Event Handler is activated, 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 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 and it fills the Triggered Rules table according to the triggered rules that are passed to it. Then, the Trigger Support selects from the Triggered Rules structure a triggered rule to be executed thus detriggering the rule. Now, the Block Executor is invoked to execute the ALGRES blocks corresponding to the selected rule condition and action in compiled form. During the evaluation of rule condition the Block Executor may moreover temporarily call the Event Handler to access the Occurred Events structure when evaluating event formulas (*occurred/holds* predicates); after this evaluation, the Event Handler returns the control to the Block Executor together with a set of bindings. A Chimera transaction is 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 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 returns control to the Block Executor which executes the next transaction line.

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 regards three entities: Logs, Occurred Events, 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

[4] 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.

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

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 satisfies 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 since the operation is set-oriented).
- Rule timestamps denote the last inflection point 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.

The semantic features of triggers that can be implemented by means of this timestamping mechanism include event consumption, determination of rules inflection point, 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 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.

For properly evaluating an old state of an object in rule conditions it is necessary, in addition to log the state, to include a timestamp to distinguish between the different instances of the object that may exist in the Log at a given time. In other words, the old state of an object is not determined in a unique way, rather it depends on the rule that evaluates it. 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[6].

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 scan the Occurred Events structure considering only events having a timestamp greater than that of the rule[7].

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, after having completed the block execution, calls the Event Handler to store the events that have occurred during block execution, 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 know 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[8]; all the rules triggered by the event are also extracted. The information on the last occurrence of an event with the specified event type is updated to take into account the

[6] 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.

[7] If the rule is event preserving the entire structure is scanned.

[8] 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.

```
update Triggered Rules;
while a rule exists in Triggered Rules to be considered
do
  select a rule r with max priority in Triggered Rules;
  update the entry for r in Triggered Rules setting
    that it has been considered and,
    if r is consuming, setting its timestamp
      to the next available timestamp;
  call the Block Executor to execute rule r condition;
  if the rule executor returns a non empty set of
    events (i.e., the condition evaluation succeeded)
  then
      call the Block Executor to execute rule r action;
      update Triggered Rules
  endif
enddo
```

Figure 2: Trigger Support activity

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.

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.

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).

5 Rule selection and execution

The overall activity of the Trigger Support is outlined by the cycle in Figure 2. The cycle in Figure 2 is applied in the case of reactive processing started at the end of a transaction or at a *savepoint*, when all the triggered rules have to be considered. In the case of reactive processing started at the end of a transaction line (when only immediate rules must be considered) the operations done by the Trigger Support are similar, but only rules whose processing mode is immediate are considered.

During rule selection, when looking for a rule with the greatest priority, if two or more rules with equal priority are found, the choice of which of them to execute is taken non deterministically according to [15]. 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.

Once a rule has been selected the Block Executor is activated to execute the selected rule condition. The rule condition evaluation 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. 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.

If the condition evaluation succeeds, that is, a non empty set of binding is returned, the Block Executor is activated again to execute rule action. We 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.

6 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 systems see [11]). The rule systems considered here are Starbust [14, 15], Postgres [13], Ariel [8] and RDL [12]. Note that the systems with which we compare are all based on a relational data model. So our comparison is "parametric" with respect to the data model, and takes into account the differences among the rule languages.

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. In Starbust implementation, the rule system maintains information about rules in a Rule Catalog (in secondary memory) and in a Global Rule Information structure (in main memory). The latter has the same role as the information on rules duplicated in the Chimera Triggered Rules table. Moreover, Starbust rule system maintains a Transition Log and a Rule Processing Information structure. The Transition Log 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 rule. This structure corresponds to Chimera Occurred Events and Logs. The Rule Processing Information structure has a role similar to that of Triggered Rules in Chimera architecture.

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 runtime, 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. It is quite difficult to compare these approaches with 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 (so 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. Thus in Ariel the Chimera distinct phases of verifying if a rule is triggered and evaluating a rule condition are merged in a single *matching* phase. Thus Chimera event handling and triggered rule handling mechanism has 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.

In RDL the event detection and rule triggering system is not integrated in a DBMS, rather it is tightly coupled with a DBMS. This approach is more flexible and portable, but it is not efficient. An executable program results from the compilation of a rule base; this program automatically activates and executes rule modules according to the operation performed by application programs. The program makes use of a Production Compilation Network to represent the relationships between relations, main memory variables, and rule conditions.

7 Conclusions

In this paper we have described an implementation of Chimera active rules. The prototype implementation of Chimera 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.

Preliminary measures of performance are encouraging: active rule selection is very fast and thus the overall execution times are dominated by execution time of the ALGRES code produced for transactions and for rules. As an additional observation, 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 [1].

The current Chimera prototype is being extended incorporating 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 are being developed as well.

Acknowledgments

This work has been supported by ESPRIT project P6333 IDEA. The work of Danilo Montesi has been partially supported by the ERCIM fellowship *Information and Knowledge Systems*.

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. A special acknowledgment is due to Pierpaolo Merli and Lanfranco Colella, who helped us in implementing the run-time support for Chimera active rules.

References

[1] S. Castangia, L. Colella, 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.

[2] S. Ceri, S. Crespi Reghizzi, et al. The ALGRES Project. In *Proc. Int'l Conference on Extending Database Technology.* Springer LNCS, 1988.

[3] 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.

[4] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active Rule Management in Chimera. In S. Ceri, U. Dayal, and J. Widom, editors, *Active Database Systems.* Morgan Kaufmann, 1995.

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

[6] N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 327–336, 1991.

[7] G. Guerrini, D. Montesi, and G. Rodriguez. Implementing Active Rules in an Object Database System. Technical Report TR 62-94, Politecnico di Milano - Dipartimento di Elettronica e Informazione, September 1994.

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

[9] E. Hanson and J. Widom. Rule Processing in Active Database Systems. *International Journal of Expert Systems*, 6(1):83–119, 1993.

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

[11] N. W. Paton et al. Dimensions of Active Behaviour. In *Proc 1st Int. Workshop on Rules In Database Systems*, Springer LNCS, pages 40-57, 1994.

[12] E. Simon, J. Kiernan, and C. de Maindreville. Implementing High Level Active Rules on top of a Relational DBMS. In *Proc. 18th Int'l Conf. on Very Large Data Bases*, pages 315–326, 1992.

[13] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In *Proc. Int'l ACM Conf. on Management of Data*, pages 281–290, 1990.

[14] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In *Proc. 17th Int'l Conf. on Very Large Data Bases*, pages 275–285, 1991.

[15] J. Widom and S. J. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proc. Int'l ACM Conf. on Management of Data*, pages 259–270, 1990.