Trigger Inheritance and Overriding in Active Object Database Systems

Elisa Bertino [1] *Giovanna Guerrini* [2] *Isabella Merlo* [2]

[1] Dipartimento di Scienze dell'Informazione
Università di Milano - Milano, Italy
Via Comelico 39/41 - I20135 Milano, Italy
`bertino@dsi.unimi.it`

[2] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova - Genova, Italy
Via Dodecaneso 35 - I16146 Genova, Italy
`{guerrini,merloisa}@disi.unige.it`

Abstract. An active database is a database in which some operations are automatically executed when specified events happen and particular conditions are met. Several systems which support active rules in an object-oriented data model have been proposed. However, many issues related to the integration of triggers with object-oriented modeling concepts have been devoted so far limited attention and still need investigation. In this paper, we discuss the problems related to trigger inheritance and refinement in the context of the Chimera active object-oriented data model.

1 Introduction

The need and the relevance of reactive capabilities as a unifying paradigm for handling a number of database features and applications are well-established. An *active* database system is a database system which automatically performs certain operations in response to certain events occurring or certain conditions being satisfied [8]. Active database systems are centered around the notion of *rule*. Rules are syntactic constructs by means of which the reactions of the system are specified. Active rules, often referred as triggers, are usually defined according to the *event-condition-action* paradigm. Events are monitored and their occurrences cause the rule to be triggered; a condition is a declarative formula that must be satisfied in order for the action to be executed, whereas the action specifies what must be done when the rule is triggered and its condition is true.

Most of the research and development efforts on active databases and commercial implementations have focused on active capabilities in the context of relational database systems. Recently, however, there have been several proposals incorporating active rules into object-oriented database systems. The paradigm shift from the relational model to the object-oriented one warrants a re-examination of the functionalities as well as the mechanisms by which reactive capabilities are incorporated into the object-oriented data model [2]. There are several factors not present in relational database systems that complicate the extension of object-oriented database systems to include active behavior. Among them, let us mention that in object-oriented data models, in contrast to a fixed number of predefined primitive events of the relational model, every method/message is a potential event. Moreover, issues related to scope, accessibility, and visibility of object states with respect to rules should be investigated. Other important, open issues concern trigger inheritance and overriding. Because inheritance is a central notion of the object-oriented data model, the definition of a proper approach for rule inheritance is crucial. However, those issues have been largely neglected by current implementations and research proposals, or only simplistic solutions are adopted. Those solutions are inadequate to model the large variety of rule semantics, that arise in practical applications. The approach taken by the majority of the systems for rule inheritance is to simply apply all rules, defined in a class, to the entire extent of the class, that is, to all the instances[1]. No rule overriding is supported by those systems. Only few systems [11, 13] support rule overriding, though in a completely uncontrolled way. In our opinion, there are several cases in which rule overriding is useful. However, we believe that rules should not be overridden in a uncontrolled way, rather they should only be *refined* in subclasses, that is, they can be overridden provided that (i) the redefined rule is triggered each time the overridden one would do; (ii) the redefined rule does at least what the overridden one would do.

In this paper, we examine the issues related to trigger inheritance in the context of the Chimera active object-oriented data model [7, 12]. Chimera is a database language integrating an object-oriented data model, a declarative query language based on deductive rules, and an active rule language for reactive processing, developed as part of the ESPRIT Project Idea P6333[2]. Though carried on with reference to Chimera, our discussion is highly independent from Chimera and can be applied to other active object-oriented systems. Moreover we think that Chimera action language with its declarative style in formulating triggers can be useful at a specification level to formulate triggers and to reason about them. In particular, we introduce a semantics for active rules in an object-oriented active system, taking into account rule inheritance and overriding and pointing out subtle issues such as method selection in trigger actions for inherited triggers. We introduce the notion of (semantic) trigger refinement and we establish some sufficient static conditions ensuring it.

The remainder of the paper is organized a follows. Section 2 presents the reference active rule language, extending it with the possibility of trigger overriding. Section 3 discusses trigger semantics, while Section 4 is deals with trigger refinement. Finally, Section 5 concludes the work.

---

[1] An object is a *proper instance* of a class if this class is the most specialized class in the inheritance hierarchy to which the object belongs. An object is an instance of a class if it is a proper instance of this class or a proper instance of any subclass of this class.

[2] 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.

2 Reference Active Rule Language

As reference rule language we consider a subset[3] of the Chimera active rule language [6]. Chimera supports *set-oriented* active rules [16]: rules 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 respect, 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 called *triggers*. Each trigger is targeted to a class and characterized by five components: a name, a class, a set of events[4], a condition and a reaction. Events are denoted by the name of the primitive operation and the schema elements to which the operation is applied. Primitive operations are object creations, deletions, modifications and object migrations in the inheritance hierarchy (ISA). Modifications refer to specific attributes. In addition, active rules may monitor operation calls (methods), although rule execution remains set-oriented as method execution itself is set-oriented.

In the following definitions we make use of a set of class names $\mathcal{CN}$, of a set of attribute names $\mathcal{AN}$ and of a set of operation names $\mathcal{MN}$.

Definition 1. Let $c \in \mathcal{CN}$ be a class name, $a \in \mathcal{AN}$ be an attribute name and $op \in \mathcal{MN}$ be an operation name. A Chimera *event* has one of the following forms: $create$; $delete$; $generalize(c)$; $specialize(c)$; $modify(a)$; op. □

The condition is a formula monitoring the execution of the reaction part. It is a conjunction of atomic formulas and it is interpreted as a predicate calculus expression over typed variables. Conditions may contain, in addition to conjunctions 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*. This predicate is used to inspect the events that have occurred during a transaction. It has two arguments: an event name and a variable ranging over the OIDs of the objects affected by the event. This variable is bound to OIDs of instances which are receivers of the event. References to *old 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 the state at transaction start[5].

Chimera atomic formulas can be of four types, being t_1, t_2 terms [12] and X a variable: *comparison formulas* (e.g. $t_1\ op\ t_2$, $op \in \{<, >, \geq, \leq, =, ==, ==_d\}$); *membership formulas* (e.g. t_1 *in* t_2 or t_1 *in* c where c is a class name); *class*

[3] In particular, we restrict ourselves to targeted, deferred, event-preserving rules without net effect composition.

[4] The set has a disjunction semantics (the trigger becomes active if *any* of its triggering events occurs).

[5] Since we restrict ourselves to event-preserving rules the old state always refers to the state at transaction start.

formulas (e.g. $c(X)$, where c is a class or type name); *event formulas* (e.g. $occurred(e, X)$, where e is an event according to Definition 1).

Complex formulas (or simply formulas) are obtained from atomic formulas and negated atomic formulas by means of conjunctions; moreover, atomic formulas may be evaluated on the state at transaction start. Formally, if F is an atomic comparison or membership formula[6], then $\neg F$ and $old(F)$ are (complex) formulas; if F_1 and F_2 are formulas, then F_1, F_2 is a (complex) formula, where the symbol "," denotes the *and* logical connective. Each formula is required to contain exactly one class formula for each variable, specifying the type of the variable. All variables are assumed to be implicitly quantified. In addition, formulas are required to be range restricted, to avoid formulas that are satisfied by an infinite set of instances.

The reaction is a sequence of database operations, including update 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.

Definition 2. Let $c, c' \in \mathcal{CN}$ be class names, $a \in \mathcal{AN}$ be an attribute name and O be an object-denoting variable. Moreover, let $op \in \mathcal{MN}$ be an operation name and $t, t_1, \ldots, t_n$ be terms. A Chimera *action* has one of the following forms: $create(c, t, O)$, $delete(c, O)$, $generalize(c, c', O)$, $specialize(c, c', O, t)$, $modify(c.a, O, t)$, $O.op(t_1, \ldots, t_n)$, $rollback$. □

We are now able to give the definition of trigger.

Definition 3. A Chimera *trigger* is a 5-tuple

$$(Name,\ Class,\ Events,\ Condition,\ Action)$$

where:

$Name$ is the trigger identifier;
$Class$ is the class to which the trigger is targeted;
$Events$ is the set of primitive operations monitored by the trigger, each event in the set is as specified by Definition 1;
$Condition$ is a Chimera formula;
$Action$ is a sequence of actions (cfr. Definition 2);

such that the following conditions are satisfied:

1. each variable occurring as input parameter of an operation in the $Action$ must appear in some positive literals of the $Condition$ (safety condition);
2. for each event formula $occurred(e, X)$ in the $Condition$, e must appear in $Events$. □

[6] Class and event formulas cannot be negated nor tested on past database states.

We remark that in Chimera rule events are not parametric and there is no parameter passing between the event and other rule components. Thus, events only cause rules to be triggered, but rules are then considered and executed with no reference to the triggering events. However, the triggering events can be explicitly bound to variables in rule conditions by means of event formulas.

Example 1. The following is a trigger, defined on a class `employee`, preventing an employee from being assigned a salary higher than the salary of his manager. If such a situation occurs, the employee salary is automatically overwritten by assigning it a salary equal to the manager salary.

Events: `create, modify(salary)`
Condition: `employee(X), X.salary > X.mgr.salary`
Action: `modify(employee.salary, X, X.mgr.salary)`

Another example of Chimera trigger is the following, always on the class `employee`, that has the effect of specializing each new employee earning more than `40000` by inserting the employee in the class `specialEmpl`.

Events: `create`
Condition: `employee(X), occurred(create,X), X.salary > 40000`
Action: `specialize(employee, specialEmpl, X)` $\triangle$

In order to provide a consistent behavior when multiple triggers are activated by the same events, it is important that a well-defined policy is established. In Chimera, triggers are ordered according to their priorities. A partial order $<_r$ is considered on the set of triggers to express trigger priorities. The meaning of the order is as follows: given two triggers r_1 and r_2, $r_1 <_r r_2$ means that when r_1 and r_2 are both triggered then r_1 is considered and executed before r_2. In our model the approach is to define the priority order on triggers by combining user-defined priorities among triggers in the same class c ($<_r^c$), with the order induced by inheritance relationships among classes. Thus, in Chimera local priorities, specified by the user for triggers in the same class, are combined by the system with the order induced by ISA relationships. To privilege the more specific behavior, the reverse ISA ordering is considered as a default for relating triggers defined on different classes, as stated by the following definition.

Definition 4. A trigger r_1 has *priority* over a trigger r_2 (denoted as $r_1 <_r r_2$) if either:

- $\exists\, c \in \mathcal{CN}$ such that $r_1 <_r^c r_2$, or
- $r_2.Class \leq_{ISA} r_1.Class$. $\square$

If for each class c the local priority ordering $<_r^c$ is an order, the priority ordering $<_r$ defined in Definition 4 is a (partial) order. Note that the acyclicity of the local trigger ordering is checked upon trigger definition. The default priority ordering of triggers obtained as in Definition 4 can be modified by the user in the subclass definition.

2.1 Trigger Overriding

The influence of inheritance on triggers has not been deeply investigated in existing object-oriented database systems. Under some proposals [2, 10, 14], triggers are always inherited and can never be overridden nor refined. Such an approach, that we refer to as *full trigger inheritance*, simply means that event types are propagated across the class inheritance hierarchy. Consider the event of a rule r, say $op(c)$, characterizing the operation op on a class c. If c has a subclass c', when an operation op occurs on a proper instance of c' rule r is triggered, as well as any other rule having as event $op(c')$. Thus, inheritance of triggers is accomplished by applying a trigger to all the instances of the class in which the trigger is defined, rather than only to proper instances of this class.

Full trigger inheritance is, however, not appropriate for all the situations. As we will see in what follows, there are situations in which trigger overriding is needed. Moreover, the meaning of the ISA hierarchy is just to define a class in terms of another class refining its attributes, methods and triggers. This way of reasoning is one of the aspects which has made of the object-oriented approach a powerful paradigm. Thus, in our opinion, the possibility of redefining triggers in subclasses instead of simply inheriting them is needed in an active object-oriented system.

In a system supporting trigger inheritance, but not trigger overriding, as in the full inheritance case, the only way to refine the behavior of a trigger in a subclass is to define in the subclass a trigger on the same events which performs the refined action. However, for this addition to be effective, the trigger in the superclass must have priority over the trigger in the subclass. Thus, upon occurrence of the common triggering event on an object belonging to the subclass, both triggers are activated, but, since the trigger defined in the superclass is executed first, the action in the trigger defined in the subclass "prevails". However, it is not always possible to refine the behavior of a trigger in a subclass by adding a new trigger, even by specifying that the subclass trigger has lower priority than (thus, is executed after) the superclass one. Consider the following example.

Example 2. Consider a class `person` with a subclass `employee`. Suppose, moreover, that a third class `department` is defined, with an attribute `nbr_of_emp` which maintains the number of employees of the department. Suppose that a trigger r_1 is defined on class `person` such that, whenever the age of a person is greater than 100, an object of a class `p_log` whose state refers to the deleted object is created (the class `p_log` is used for monitoring purposes) and then the object is deleted. Suppose that a corresponding trigger r_2 is defined on class `employee`, such that, whenever the age of an employee is greater than 100, an object of a class `e_log` is created, the value for attribute `nbr_of_emp` of the department in which the employee works is decremented and finally the employee is deleted. The two triggers can be expressed in Chimera as follows:

– trigger $\mathtt{r}_1$ with $\mathtt{r}_1.Class$ = `person`
    Events: `modify(age)`

```
Condition: person(X), occurred(modify(age),X), X.age > 100
Action:    create(p_log, (who:X, age:X.age), O);
           delete(person,X)
```

– trigger $\mathtt{r}_2$ with $\mathtt{r}_2.Class$ = `employee`

```
Events:    modify(age)
Condition: employee(X), occurred(modify(age),X), X.age > 100,
           department(Y), X.department = Y
Action:    create(e_log,(who:X,age:X.age,salary:X.salary),O);
           modify(department.nbr_of_emp,Y, Y.nbr_of_emp-1);
           delete(employee,X)
```

Whenever the age of an employee is set to 101, both $\mathtt{r}_1$ and $\mathtt{r}_2$ are triggered and, if $\mathtt{r}_1 <_r \mathtt{r}_2$ (as under the default ordering introduced in the previous subsection), $\mathtt{r}_1$ is executed first. The execution of $\mathtt{r}_1$ deletes the involved object, and then the execution of $\mathtt{r}_2$ does not have any effect. Indeed, when $\mathtt{r}_2$ is executed the object on which the age modification had occurred has already been deleted, thus it is not possible to access it. As a consequence, that object does not satisfies trigger $\mathtt{r}_2$ condition. Intuitively trigger $\mathtt{r}_2$ is just the refinement of $\mathtt{r}_1$ because it has a behavior similar to that of $\mathtt{r}_1$ but refined for subclass `employee`. In a system supporting inheritance and trigger overriding, trigger $\mathtt{r}_2$ would be the refinement of trigger $\mathtt{r}_1$, thus for the objects proper instances of `employee` only trigger $\mathtt{r}_2$ would be executed giving a correct result. △

We thus extend Chimera rule language with the possibility of overriding triggers in subclasses. Trigger overriding is accomplished by defining a new trigger in the subclass with the same name as the inherited trigger. When a trigger r_1 is overridden by a trigger r_2 such that $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$, the occurrence of an event $e \in r_1.Event$ on objects belonging to $r_2.Class$ does not trigger rule r_1.

3 Semantics

In this section we present a formal semantics for trigger refinement. The intuitive idea behind this semantics is the following. 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 consists of 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 of determining the triggered rules, that is, the ones for which any of the triggering events has occurred;
- *rule selection* consists of non deterministically choosing one of the triggered rules at highest priority;
- *rule consideration* consists of evaluating the condition, which is a declarative formula; at this point the selected rule is detriggered;

– *rule execution* occurs if the condition is true, that is, produces some bindings; the execution is performed by sequentially executing the operations in the reaction part of the rule.

Trigger execution consists of updates, which may in turn trigger other rules. The rule processing activity is iterated 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; techniques and tools for detecting the possible sources of non-termination in a rule set have been developed [6].

Remember that we consider a language supporting deferred, event-preserving rules without net effect composition. A transaction in our language is a sequence of data manipulation commands each of those may trigger some rules. Considering deferred rules means that for each command the corresponding event is added to the previous set of collected events, but no rule is executed. Only at the end of the transaction, corresponding to a commit command, rule processing is activated, that is, the set of rules, triggered by the set of events occurred during the transaction, is computed and on this set rule selection and the following phases are iterated until no rule is triggered.

3.1 Preliminaries

Before formalizing rule semantics, we need to introduce the preliminary definitions of *database state*, *set of bindings* and *reactive process*. In the following we consider: a set $\mathcal{V}$ of values; a set OID of all possible OIDs; a set C of classes; a set Var of variables. Moreover, the set $Rule$ denotes the set of rules defined for a database[7]. Finally, given a set S, 2^S denotes the powerset of S whereas, given a n-tuple $\langle el_1, \ldots, el_n \rangle$, Π_i, $i \in [1, n]$, denotes the projection of the i-th component el_i of the tuple.

Database state. We consider a simple and rather standard [1] definition of database state. Our model, like most object-oriented data models, distinguishes between the schema level, which is time-invariant, and the instance level, which is time-varying. Informally, the schema level consists of the class definitions, that is, for each class the attributes and the type of the attributes domains, the class hierarchy (represented through the ISA ordering $\leq_{ISA}$), a set of method signatures and the set $Rule$.

Definition 5. A *database state* (database for short) is a pair $S = (\pi, \nu)$ where:

– $\pi : C \rightarrow 2^{OID}$ is the function which associates with each class the set of OIDs of its proper instances[8];

[7] Notice that this set of rules is part of the database schema.

[8] We remark that $\pi(c)$ denotes the set of OIDs of those objects for which c is the most specific class in the inheritance hierarchy. In what follows $\pi^*(c)$ denotes the whole extent of a class c, that is, the set of all its instances.

– for each $i \in OID$, object identifier, $\nu(i)$ returns the state of the object, that is, the value of its attributes; if the attributes names are $A_1, \ldots, A_n$ and $v_1, \ldots, v_n$ the corresponding values, $\nu(i) = [A_1 : v_1, \ldots, A_n : v_n]$[9]. □

Set of bindings. As we said before, in our language the bindings obtained by the evaluation of the condition are passed to the action part of the rule. The set of bindings is the mean by which such variable passing is achieved. Condition and action parts share some variables, the action must be executed for every binding generated by the condition on the shared variables. We model a set of bindings as a set of ground substitutions.

Definition 6. A *substitution* ϑ is a partial function from Var to $\mathcal{V}$; $\vartheta : Var \rightarrow \mathcal{V}$. A set of bindings B is a set of substitutions $\{\vartheta_1, \ldots, \vartheta_m\}$. □

Given X, Y, Z variables and a set of values $\mathcal{V}$ including integers, the following is an example of substitution: $\vartheta_1 = \{X/5, Y/7, Z/8\}$.

Intuitively, the set of bindings $B = \{\vartheta_1, \ldots, \vartheta_m\}$ satisfying a condition C, is the set of substitutions such that the application of each ϑ_i ($i \in [1, m]$) to C, denoted as $C\vartheta_i$, is a ground formula which is true according to first order logic.

In what follows, given a substitution ϑ and a set of variables V, let $\vartheta_{|V}$ denote the restriction of substitution ϑ to variables in V. Moreover, given a set of substitutions S, let $S_{|V} = \{\vartheta_{|V} \mid \vartheta \in S\}$.

Reactive process. First we have to establish, given a set of events, which is the set of rules triggered by the occurrence of events in the set.

Definition 7. Let e be an event as in Definition 1, c be the class name this event is related to, and O be the set of OIDs of the objects affected by the event, then the triple $\langle e, c, O\rangle$ is called *event instance.* □

For example, the event corresponding to the action $create(c, t, O)$ is $create$ and a corresponding event instance is $\langle create, c, O\rangle$, where O denotes the set of the OIDs of the created objects. For the sake of simplicity, we will often use the word event to denote event instances, when the meaning is clear from the context.

Since our language supports trigger overriding, given an event instance, establishing which rules have to be triggered for each OID affected by the event is not trivial. Trigger overriding is accomplished by defining a new rule in the subclass with the same name as the inherited trigger. Thus, the set of rules triggered by an event is computed taking into account that for each object the most specific rule is triggered, as formalized by the following definition.

Definition 8. A rule $r = (N, c, Ev, C, A)$ is *triggered* by an event instance $\mathbf{e} = \langle e, c_e, O\rangle$ if:

[9] To denote the values of single attributes we use the following notation: $\nu(i).A_j = v_j$, $j \in [1, n]$.

1. $e \in Ev$, and
2. $\exists i \in O$ such that $c = min_{\leq_{ISA}}\{\bar{c} \mid i \in \pi^*(\bar{c})$ and $\exists \bar{r} = (N, \bar{c}, \bar{Ev}, \bar{C}, \bar{A}) \in Rule\}$. □

Definition 9. Let $\mathbf{e} = \langle e, c, O\rangle$ be an event instance, E be a set of event instances, then $react(\mathbf{e}) = \{r | r \in Rule$ and r is triggered by $\mathbf{e}\}$ and $react(E) = \bigcup_{\mathbf{e} \in E} react(\mathbf{e})$. □

As we said before, given a set of rules, in the rule selection phase we have to choose one of the triggered rules at highest priority. Since there can be more than one rule at highest priority, the choice is non deterministic. Function get_max performs a non deterministic choice among the rules at highest priority in a set of rules.

Definition 10. Let R be a set of rules, function $get\_max : 2^{Rule} \rightarrow Rule$ is such that, given $R \in 2^{Rule}$, if $get\_max(R) = r$ then $\nexists\, r' \in R : \quad r' <_r r$. □

3.2 Trigger Semantics

First of all we introduce semantic functions. Note that in the following we refer to semantic domains as the set of syntactically well-formed objects, that is, objects which meet the static constraints and for which the semantics can be defined.

Definition 11. Given a set $Bind$ of possible sets of bindings, a set $State$ of possible database states, a set $Event$ of possible sets of event instances, a set $Cond$ of possible condition parts of rules, a set $Update$ of possible action parts of rules[10], a set $Rule$ of possible rules of the language, as semantic domains, the *semantics* of the trigger language is a family of functions defined as follows:

$$\mathcal{C} : Cond \rightarrow ((State \times Event) \rightarrow Bind)$$
$$\mathcal{U} : Update \rightarrow ((Bind \times State \times Event) \rightarrow (Bind \times State \times Event))$$
$$\mathcal{R} : Rule \rightarrow ((State \times Event) \rightarrow (State \times Event))$$
$$\mathcal{P} : 2^{Rule} \rightarrow (State \rightarrow State).$$
□

Function $\mathcal{C}$ models the condition evaluation; function $\mathcal{U}$ models the action execution, whereas functions $\mathcal{P}$ and $\mathcal{R}$ model the reactive processing semantics. As we have said, we consider deferred rules. From a semantic point of view this means that during the transaction execution events (event instances) are collected; when the transaction ends two situations can arise:

- the transaction ends with a *rollback* command, in this case the resulting state is the state at the beginning of the transaction;
- the transaction ends with a *commit* command, in this case the reactive process is activated and all rules triggered by the events occurred during the transaction must be executed.

[10] $Update$ corresponds to the set of well-formed update sequences of the language.

Function $\mathcal{P}$ models the semantics of the reactive process associated with a transaction, by establishing a transformation from the state at the end of the transaction to the state at the end of the reactive process. Function $\mathcal{R}$ models the evaluation of a rule r which consists of evaluating the condition of r, getting a set of bindings as result, and executing the action of r on this set of bindings. The result of the evaluation of a rule is a new database state and a new set of events. No set of bindings is given as result because bindings in Chimera are local to rules.

In what follows we specify the semantic functions $\mathcal{R}$ and $\mathcal{P}$. Due to space limitations, we do not present here the formal definition of functions $\mathcal{C}$ and $\mathcal{U}$ but we only give an idea. We refer the interested reader to [5] for a complete definition of $\mathcal{C}$ and $\mathcal{U}$.

Let C be a rule condition (a formula), S be a database state and E be a set of event instances, then:

$$\mathcal{C}\,[\![\, C \,]\!]\, SE = B$$

where each substitution $\vartheta \in B$ is such that $C\vartheta$ evaluates to true in state S and with respect to events in E according to first order logic. Due to space limitations we do not analyze in depth the evaluations of event formulas (against the set of event instances E) nor the use of the *old* function in conditions. An analysis can be found in [5].

Function $\mathcal{U}$ performs the semantic evaluation of a sequence of updates, that is, an action part, u_1;...;u_n of a rule. The semantics of update concatenation is quite intuitive: at each step the first update of the sequence is evaluated, such evaluation gives as result a new set of binding, a new state and a new set of event instances with respect to which the remainder of the sequence is evaluated. Formally:

$$\mathcal{U}\,[\![\, u_1; \ldots ; u_n \,]\!]\, BSE = \mathcal{U}\,[\![\, u_2; \ldots ; u_n \,]\!]\, (\mathcal{U}\,[\![\, u_1 \,]\!]\, BSE)$$

For what concerns the evaluation of a single update, function $\mathcal{U}$ is specified for each different type of atomic update, that is, *create*, *delete*, and so on, and for method calls. Here we only specify, as an example, the semantics of the *delete* operation. Given a set of binding B and a variable $\bar{O}$, let $\bar{O}_B$ be the interpretation of $\bar{O}$ in B, that is, the set of OIDs to which $\bar{O}$ is bounded in B, then:

$$\mathcal{U}\,[\![\, \texttt{delete(}\bar{c},\, \bar{O}\texttt{)} \,]\!]\, BSE = \langle B, S', E \cup \{\langle delete, \bar{c}, \bar{O}_B \rangle\}\rangle$$

where $S' = \langle \pi', \nu' \rangle$, and:

$$\pi'(c) = \begin{cases} \pi(c) \setminus \bar{O}_B & \text{if } c = \bar{c} \\ \pi(c) & \text{if } c \neq \bar{c} \end{cases} \qquad \nu'(i) = \begin{cases} \nu(i) & \text{if } i \notin \bar{O}_B \\ \bot & \text{if } i \in \bar{O}_B \end{cases}$$

One of the problems arising in defining function $\mathcal{U}$ is method selection with respect to inherited triggers. Consider a trigger r defined in a class c and invoking in its action an operation op on the object affected by the event. Consider moreover a subclass c' of c and suppose that operation op is redefined in c'. Rule

r is triggered when the event monitored by r occurs both on objects proper instances of c and on objects proper instances of c'. Of course, for objects proper instances of c the method implementation in class c is selected, while for objects proper instances of c' two different options are possible: (i) choose the most specialized implementation of *op* (that is, the implementation in class c'); (ii) choose the implementation according to the class where the rule is defined (that is, the implementation in class c). We refer to the first and second approach as *object-specific method selection* and *rule-specific method selection*, respectively. Our choice is the first one, because it is consistent with the object-oriented approach, in that it respects the principle of exhibiting the most specific behavior. The rule specific method selection is not truly consistent with the object-oriented approach because it refers to the static nature of the objects, that is, the class in which the trigger is defined and not to their dynamic nature, that is, the class the object is proper instance of. Even though the rule-specific method selection is not coherent with the object-oriented approach, it is used in some active object-oriented database systems, like Ode [11].

By supporting trigger overriding and object-specific method selection, our execution model is purely object-oriented. For each object affected by an event which triggers a rule r, the most specific implementation of r defined for that object is triggered and, during trigger execution if a method is invoked in the trigger action, for each considered object the most specific method implementation for that object is chosen.

We now formally define the semantic functions $\mathcal{R}$ and $\mathcal{P}$.

Definition 12. Let $r = (N, c, Ev, C, A)$ be a rule, let E be a set of event instances, let S be a database state, and, finally, let $\mathcal{C}\,[\![\, C \,]\!]\, SE = B$, then:

$$\mathcal{R}\,[\![\, r \,]\!]\, SE = \langle S', E' \rangle$$

where $S' = \Pi_2(\mathcal{U}\,[\![\, A \,]\!]\, BSE)$ and $E' = \Pi_3(\mathcal{U}\,[\![\, A \,]\!]\, BSE)$. □

The semantics of the reactive process is given by the following definition.

Definition 13. Let R be a set of rules, E be a set of event instances, and S be a database state, $\mathcal{P}\,[\![\, R \,]\!]\, SE$ is defined as follows:

$$\mathcal{P}\,[\![\, R \,]\!]\, SE = \begin{cases} SE & \text{if } R = \emptyset \\ \mathcal{P}\,[\![\, R \setminus \{r\} \cup react(E') \,]\!]\, S'E \cup E' & \text{if } R \neq \emptyset, r = get\_max(R) \\ & \text{and } \mathcal{R}\,[\![\, r \,]\!]\, SE = \langle S', E' \rangle. \end{cases}$$

□

Note that the recursive definition of semantic function $\mathcal{P}$ corresponds to the idea that the reactive process is iterated till a quiescent state is reached. When there are no more triggered rules, that is $R = \emptyset$, the reactive process stops and the current state is returned. The reactive process semantics can then be seen as the least fixpoint of function $\mathcal{P}$.

To model reactive process activation, at transaction commit, the following semantics is specified for the *commit* command:

$$\mathcal{U}[\![\, commit \,]\!]\, BSE = \langle \emptyset, \mathcal{P}[\![\, react(E) \,]\!]\, SE, \emptyset \rangle.$$

Note that since the commit command is the last command of a transaction, the output values of B and E are set to $\emptyset$ because they are not meaningful.

4 Trigger Refinement

As we have discussed in Subsection 2.1, in some situations a class must be able to redefine a trigger of one of its superclasses, instead of simply inheriting it. Rule overriding is supported in some systems such as TriGS [13] and Ode [11], but no restrictions are imposed on rule overriding, thus a rule may override another rule on completely different events.

In our model, as in those systems, trigger overriding is directly supported. However, we believe that trigger redefinition must be carefully handled and is, therefore, subject to a number of restrictions. In particular, in order to preserve trigger semantics, it must be ensured that the trigger in the subclass is executed at least each time the trigger in the superclass would be executed, and that what would be executed by the trigger in the superclass is also executed by the refined trigger. In this case we say that the trigger in the subclass is a behavioral *refinement* of the trigger in the superclass.

More specifically, a trigger r_2 is a behavioral refinement of trigger r_1 if the portion of state manipulated by r_2 includes the portion of state manipulated by r_1 and if the portion of state modified by both is modified in the same way[11]. To formally define this notion, we must first model the changes made by a trigger execution[12]. Given a trigger r, let $\mu(r)$ be the set of classes manipulated by r. Given a trigger r and a class c, let $\delta_r(c)$ be the set of objects deleted from class c and $\iota_r(c)$ the set of objects inserted in class c as a consequence of the execution of trigger r; moreover, given an OID i and an attribute name A, let $\nu_r(i).A$ be defined if and only if the execution of trigger r has modified the value of attribute A of the object identified by i, and, if defined, let it contain the new value of the attribute.

Definition 14. Trigger r_2 is a *behavioral refinement* of trigger r_1, with $r_2.Name$ $= r_1.Name$ and $r_2.Class \leq_{ISA} r_1.Class$, if $\mu(r_1) \subseteq \mu(r_2)$ and if, for each database state, the execution of r_1 and r_2 restricted to the objects in $r_2.Class$ satisfies the following conditions:

1. $\forall \bar{c} \in \mu(r_1)$: $\delta_{r_1}(\bar{c}) \subseteq \delta_{r_2}(\bar{c})$ and $\iota_{r_1}(\bar{c}) \subseteq \iota_{r_2}(\bar{c})$;
2. $\forall \bar{c} \in \mu(r_1), \forall A$ attribute of $\bar{c}$, $\forall$ oid i instance of $\bar{c}$: if $\nu_{r_1}(i).A$ is defined, then $\nu_{r_2}(i).A$ is defined and $\nu_{r_1}(i).A = \nu_{r_2}(i).A$. □

[11] Note that a trigger executed on an object (set of objects) instance(s) of a class may manipulate objects of other classes.

[12] In [5] we show how these changes can be expressed in terms of trigger semantics.

Unfortunately, trigger refinement is undecidable. We have, however, devised some sufficient static conditions ensuring that a trigger r_2 is a refinement of a trigger r_1. These conditions can be checked at trigger definition time, so that the overriding of a trigger in a subclass can be disallowed if the overriding trigger is not a refinement of the overridden one. In what follows we illustrate those conditions, by first examining each trigger component separately.

4.1 Events

The refined trigger must be activated each time the inherited trigger would be activated. Thus, we impose the condition that for each event in the event component of the inherited trigger, a corresponding event is present in the event component of the refined trigger.

Definition 15. An event set Ev is a *refinement* of an event set Ev' iff $Ev' \subseteq Ev$. □

4.2 Condition

The basic idea is that the action of the refined trigger must be executed each time the inherited trigger action would be executed, and for each binding for which the inherited trigger action would be executed. Thus, the condition in the refined trigger must be less selective than the condition in the inherited one (that is, each binding returned by the condition of the inherited trigger must be returned by the condition in the refined one). In what follows, we formalize this notion. The bindings produced by the evaluation of the condition are represented as a set of substitutions, as seen in Section 3. Moreover, we introduce the following notations:

- given a trigger r, let $BV(r)$ denote the set of variables appearing in $r.Condition$ and in $r.Action$ (that is, the variables used for passing bindings);
- given a formula F, let F^{-E} denote the formula obtained by eliminating the event formulas appearing in F;
- given a formula F and two class identifiers c_1 and c_2, let $F[c_1/c_2]$ denote the formula obtained from F by substituting each class formula $c_1(X)$ with a class formula $c_2(X)$.

Definition 16. A condition $r_2.Condition$ is a *refinement* of a condition $r_1.Condition$ (denoted as $r_2.Condition \leq_c r_1.Condition$) iff the following conditions hold[13]:

1. $BV(r_1) \subseteq BV(r_2)$, and
2. for each event formula $occurred(e, X)$ in $r_1.Condition$ a corresponding event formula $occurred(e, X)$ is in $r_2.Condition$, and

[13] Due to space limitations, we do not consider here the possibility of renaming variables, which is considered in [5].

3. $\forall S$ database state,

$$\mathcal{C}\,[\![\, r_1.Condition^{-E}[r_1.Class/r_2.Class] \,]\!]\, S\emptyset \subseteq (\mathcal{C}\,[\![\, r_2.Condition^{-E} \,]\!]\, S\emptyset)_{|BV(r_1)}$$

□

Condition 3 of Definition 16 above is the subsumption property, that is $r_2.Condition^{-E}$ *subsumes* $r_1.Condition^{-E}$, restricted to be evaluated on the same class and to return the same variables. Query subsumption (also called query containment) has been widely investigated, and algorithms for deciding subsumption among object-oriented queries have been proposed [9]. Subsumption can be easily extended to handle also predicates on past database states, as long as the referred past state is the same state in both formulas[14]. Indeed, $old(F)$ subsumes $old(G)$ if and only if F subsumes G.

Example 3. Given trigger r_2 on class `employee` and trigger r_1 on class `person` such that

- $r_2.Condition$ = `employee(X),occurred(modify(age),X),X.age > 65,` `department(Y),X.department = Y`, and
- $r_1.Condition$ = `person(X),occurred(modify(age),X),X.age > 100`

$r_2.Condition$ is a refinement of $r_1.Condition$, indeed:

1. $BV(r_1) = \{$ `X` $\} \subseteq BV(r_2) = \{$ `X, Y` $\}$;
2. `occurred(modify(age),X)` is in $r_1.Condition$, and `occurred(modify(age),X)` is in $r_2.Condition$;
3. $r_1.Condition^{-E}$ $[r_1.Class/r_2.Class]$ = `employee(X), X.age > 100` subsumes $r_2.Condition^{-E}$ = `employee(X), X.age > 65, department(Y), X.department = Y` restricted to variable `X`. △

4.3 Action

The basic idea is to ensure behavior consistency, that is, the action of the refined trigger must do at least all what the action of the inherited trigger would be. This means that, for each action in the inherited trigger there must be a corresponding action in the refined one. However, since the action component of a rule can be a sequence, the corresponding action could be discarded by some complementary action executed after it in the sequence. Consider as an example the case of an inherited trigger creating an object in its action, overridden by a trigger whose action firstly creates a corresponding object and then deletes it. We consider, therefore, the net effect of the actions in the sequence and we state that for each action in the inherited trigger there must be a corresponding action in the net effect of the refined one. Note that the notion of net effect employed here is purely syntactical and relies only on complementary database operations. Net effect computation consists of composing the effects of those actions whose effect was compensated by a subsequent action on the same object. Classical compensations [10, 16] are performed as follows:

[14] Note that this is the case here, since we consider only event preserving rules, for which the referred past state is the state at transaction start.

- a sequence of *create* and *delete* operations on the same object, possibly with an arbitrary number of intermediate *modify* operations on that object, has a null net effect;
- a sequence of *create* and several *modify* operations on the same object has the net effect of a single create operation;
- a sequence of several *modify* and a *delete* operations on the same object has the net effect of a single delete operation on that object;
- a sequence of several *modify* operations on the same object has the net effect of a single modify operation on the old object which modifies it in the newest.

In addition to those classical compensations, we consider also compensations involving object migrations along the inheritance hierarchy. For the sake of brevity, we omit all rules for computing the net effect of a sequence of actions. Given a sequence of actions A, let $Net(A)$ denote the net effect of the sequence. The net effect of the sequence is performed at a syntactic level, by considering compensating actions on the same object-denoting term, contained in the sequence. Moreover, let $\leq_b$ denote the refinement on update operations (e.g. a create operation on a class c' is a refinement of a create operation on a class c, if c' is a subclass of c)[15].

Definition 17. A reaction $r_2.Action$ is a *refinement* of a reaction $r_1.Action$ (denoted as $r_2.Action \leq_a r_1.Action$) if the following conditions hold:

1. $Net(r_1.Action) = a_1; \ldots; a_n$, $Net(r_2.Action) = a'_1; \ldots; a'_m$ and $m \geq n$;
2. for each a_i, $i \in [1, n]$, in $Net(r_1.Action)$ a'_j, $j \in [1, m]$, exists in $Net(r_2.Action)$ such that $a'_j \leq_b a_i$, that is, a'_j is a refinement of a_i; let function $\xi : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$, such that $\xi(i) = j$, model this correspondence;
3. if a_i precedes a_k in $Net(r_1.Action)$, $a'_{\xi(i)}$ precedes $a'_{\xi(k)}$ in $Net(r_2.Action)$. □

We remark that, since both the determination of basic action refinement and the computation of net effect only rely on syntactical properties of the trigger action, action refinement is decidable.

Example 4. Suppose `e_log` $\leq_{ISA}$ `p_log` and `employee` $\leq_{ISA}$ `person`, then

- $r_2.Action =$ `create(e_log, (who:X, age:X.age, salary:X.salary), 0)` is a refinement of
  $r_1.Action =$ `create(p_log, (who:X, age:X.age), 0)`,
- $r_2.Action =$ `modify(department.nbr_of_emp,Y, Y.nbr_of_emp-1);`
  `delete(employee,X)`
  is a refinement of
  $r_1.Action =$ `delete(person,X)`. △

Note, moreover, that a notion of behavior refinement is imposed on Chimera operation overriding, known as *behavioral subtyping* [4]. Thus, by ensuring that whenever an operation *op* is invoked in the action of the inherited trigger, then

[15] Due to space limitations, we omit the definition of $\leq_b$, which can be found in [5].

operation *op* is invoked in the action of the refined one, we can guarantee that the action of the redefined trigger *refines* the action of the inherited one, because of behavioral subtyping.

4.4 Restrictions on Trigger Overriding

The following rule summarizes the restrictions on trigger redefinition.

Rule 1 A trigger $r_2 = (N, c, Ev, C, A)$ can be refined in a trigger $r_1 = (N, c', Ev', C', A')$, with $c' \leq_{ISA} c$ if the following conditions are satisfied:

1. $Ev \subseteq Ev'$, that is, Ev' is a refinement of Ev according to Definition 15;
2. $C' \leq_c C$, that is, C' is a refinement of C according to Definition 16;
3. $A' \leq_a A$, that is, A' is a refinement of A according to Definition 17. ◇

The following result holds.

Proposition 1 *Given two triggers r_1 and r_2:*

- *we can decide whether they meet Rule 1;*
- *if r_1 and r_2 meet Rule 1 then r_2 is a behavioral refinement of r_1 according to Definition 14.*

5 Conclusions and Future Work

Active object-oriented databases are being extensively researched. Though several research projects are being carried on and some prototype systems have been developed, a relevant issue in integrating triggers with object-oriented modeling capabilities has been so far neglected, namely trigger inheritance. In this paper, we have analyzed trigger inheritance and overriding in the context of the Chimera active object language, clarifying how trigger priority is influenced by inheritance, which different method selection policies can be exploited for method invocations in trigger actions, and under which restrictions triggers can be overridden in subclasses. In the current prototype implementation of Chimera, trigger overriding is not supported. In [5], we discuss how the existing architecture can be modified for supporting it.

Our work can be extended along a number of different dimensions. First of all, our conditions for trigger overriding can be extended to consuming rules, for which the old state referred by predicates on past database states depends on the last rule activation, and to triggers with composite events [15]. Moreover, the influence of multiple inheritance and multiple class direct membership [3] on triggers should be considered. For multiple inheritance, the main issue is how to order triggers (on the same events) inherited from different superclasses; this could be achieved by imposing a total order on classes, or by allowing a class to modify the relative priorities of triggers in its superclasses.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. C. Beeri and T. Milo. A Model for Active Object Oriented Database. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 337–349, 1991.
3. E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In W. Olthoff, editor, *Proc. Ninth European Conference on Object-Oriented Programming*, LNCS 952, pages 102–126, 1995.
4. E. Bertino, G. Guerrini, and I. Merlo. Reasoning about Set-Oriented Methods in Object Databases. Submitted for publication, 1996.
5. E. Bertino, G. Guerrini, and I. Merlo. Trigger Inheritance and Overriding in an Active Object Database System. Technical Report DISI-TR-97-4, Università di Genova, 1997. Extended version of this paper.
6. 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.
7. 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, 1994.
8. S. Ceri and J. Widom. *Active Database Systems - Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, 1996.
9. E. Chan. Containment and Minimization of Positive Conjunctive Queries in OODBs. In *Proc. of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 202–211, 1992.
10. 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.
11. N. Gehani and H. V. Jagadish. Active Database Facilities in Ode. In S. Ceri and J. Widom, editors, *Active Database Systems*. Morgan-Kaufmann, 1996.
12. G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. *Journal of Intelligent Information Systems*. To Appear.
13. 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*, LNCS 858, 1994.
14. C. Medeiros and P. Pfeffer. Object Integrity Using Rules. In P. America, editor, *Proc. Fifth European Conference on Object-Oriented Programming*, LNCS 512, pages 219–230, 1991.
15. R. Meo, G. Psaila, and S. Ceri. Composite Events in Chimera. In P. Apers, editor, *Proc. Fifth Int'l Conf. on Extending Database Technology*, LNCS 1057, pages 50–76, 1996.
16. 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.