# Trigger Inheritance and Overriding in an Active Object Database System

Elisa Bertino, Giovanna Guerrini, Isabella Merlo

*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 supporting active rules in an object-oriented data model have been proposed. However, several issues related to the integration of triggers with object-oriented modeling concepts have not been satisfactorily addressed. In this paper, we discuss issues related to trigger inheritance and refinement in the context of the Chimera active object-oriented data model. In particular, we introduce a semantics for an active object language that takes into account trigger inheritance and supports trigger overriding. Moreover, we state conditions on trigger overriding ensuring that trigger semantics is preserved in subclasses.**

*Keywords*— **Triggers, object-oriented data models, inheritance and refinement.**

## I. Introduction

The relevance of reactive capabilities as a unifying paradigm for handling a number of database features and applications is 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 [1]. Active databases enable important applications, such as alerting users that a given event has occurred, reacting to events by means of suitable actions, and controlling the invocation of operations and procedures. Examples of functions that can be effectively performed by active database systems are integrity constraint enforcement, monitoring, authorization, statistics gathering and view handling.

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 to as triggers, are usually defined according to the *event-condition-action* (ECA) 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. Several approaches have however been proposed to incorporate active rules into object-relational and object-oriented database systems. Both in the relational and in the object frameworks, active rules provide a comprehensive means to formally state the semantics of data, the high-level semantic operations on data and the integrity constraints. Though recently proposed relational database systems provide stored procedures, and object-relational and object-oriented database systems provide methods, as an alternative means to express behavior of data, specifying the semantics of data through rules has three important advantages over coding it into methods (stored procedures). First, the behavior represented by methods must be explicitly invoked by the user or by applications, while active rules are autonomously activated. Second, the semantics represented by a single rule often needs to be replicated in the code of several methods. Consider for instance integrity constraints: since a single constraint can be potentially violated by the execution of different methods, the code for enforcing the constraint should be replicated in all those methods. Finally, a specialized trigger subsystem, internal to the database system, supports a more efficient active behavior processing compared to the approach where the active behavior is coded into methods.

The paradigm shift from the relational model to the object-oriented one, requires revisiting 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 addressed. 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 to 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 of the class itself[1]. No rule overriding is

A preliminary version of a part of this paper has appeared in the Proc. of the Fifth Int'l Conference on Deductive and Object-Oriented Databases, Montreaux (Switzerland), December 1997.

Elisa Bertino is with the Dipartimento di Scienze dell'Informazione, Università di Milano, Via Comelico, 39/41 - 20135 Milano, Italy. E-mail: bertino@dsi.unimi.it.

Giovanna Guerrini and Isabella Merlo are with the Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Via Dodecaneso, 35 - 16146 Genova, Italy. E-mail: {guerrini,merloisa}@disi.unige.it.

[1] An object is a *proper instance* of a class if this class is the most

supported by those systems. Only few systems [3], [4] 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 be; ($ii$) the redefined rule does at least what the overridden one would do. A notion of *behavioral refinement* of triggers can be introduced, which is analogous to the notion of behavioral subtyping developed in the context of object-oriented programming languages [5]. This property aims at ensuring that, if a class $c$ is a subclass of a class $c'$, every object instance of $c$ behaves *like* some object of class $c'$. Such property leads to a notion of semantic method refinement: consider as an example an *init* method initializing the attributes of an object, the implementation of such method in each class initializes to the proper values the attributes of that class. An *init* implementation in class $c$ that only extends the *init* implementation of class $c'$ with a new piece of code to initialize the additional attributes of $c$ is an example of behavior refinement. Examples of mechanisms that could be exploited to obtain a correct behavioral subtyping are *super* calls or the *inner* mechanism.

We believe that behavioral refinement is even more crucial for triggers than for methods, since triggers codify object behavior that is autonomously activated, rather than exhibited upon request (invocation), as in the case of methods. This results in difficulties in predicting and managing the behavior of active systems. These difficulties are amplified in systems supporting uncontrolled trigger overriding, since the expected behavior of a trigger can be totally different from the one exhibited by one of its redefinitions, which is executed instead. We think that the possibility must be given to the database designer to limit and control trigger redefinition, for example by specifying at trigger definition time whether a trigger can be arbitrarily overridden in subclasses or if, instead, it can only be refined, ensuring a sort of behavior *preservation*.

In this paper, we address issues related to trigger inheritance in the context of the Chimera active object-oriented data model [6], [7] developed as part of the ESPRIT Project Idea P6333. In particular, we investigate some issues concerning trigger inheritance which have not been satisfactorily addressed so far, by revisiting notions such as trigger priority and method selection in trigger actions in the context of inherited triggers and specifying a formal semantics for active object rule languages. Moreover, we propose a trigger refinement policy, based on static restrictions, ensuring that the trigger semantics is preserved in subclasses. Though developed in the context of the Chimera language, our discussion is highly independent from Chimera, and applies to other object-oriented active languages as well. In particular, the semantics specified for Chimera models a generic set-oriented object active rule language. The static conditions devised for trigger refinement rely on the specific Chimera action specification language. We believe, however, that the Chimera action language is powerful enough, in that it provides all data manipulation statements supported by commercial database management systems. Moreover, its declarative style can be useful at least at the specification level to formulate triggers and to reason about them [8].

Notice, moreover, that the issues addressed in this paper also apply to object-relational databases with reactive capabilities, and in particular to the SQL-3 standard proposal [9]. We discuss in detail the application of our results to object-relational database systems at the end of the paper.

The contributions of the paper can then be summarized as follows:

*(i)* definition of a semantics for active object-oriented rule languages, modeling trigger inheritance and overriding;
*(ii)* investigation of issues related to rule inheritance, namely method selection in subclasses;
*(iii)* definition of semantic properties for trigger refinement;
*(iv)* identification of static conditions ensuring semantic trigger refinement;
*(vi)* discussion on the application of our approach for trigger inheritance and overriding to active object-relational databases.

This paper is organized a follows. Section II reviews the state-of-the-art in the field of active object-oriented databases. Section III motivates the need for trigger overriding and informally introduces the basic ideas behind our approach. Section IV presents the reference active rule language. In Section V trigger semantics is formalized. Section VI deals with trigger redefinition. The application of our approach to object-relational database systems is discussed in Section VII. A short overview of related work on trigger semantics and overriding in existing systems is given in Section VIII. Finally, Section IX concludes the work. Appendix I presents the definition of update semantics in our language.

## II. Active Object-Oriented Databases

In the last ten years, there has been a growing interest in extending object-oriented database systems with reactive capabilities. One of the earliest object-oriented active database projects is HiPAC [10], [11]. HiPAC provides an expressive active database rule language with flexible execution semantics, supporting a full complement of coupling modes through a nested transaction model. There are some projects on active object-oriented databases that are follow-on from HiPAC: among them, *Sentinel* [12] and REACH [13]. One of the best known active object-oriented database systems is Ode [3], [14], which extends the O++ database programming language with facilities for expressing rules in the form of constraints and triggers. Among other projects we would like to mention Adam [15], NAOS [16], SAMOS

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.

[17] and TriGS [4].

In Table I we compare these systems along a number of dimensions. Among the considered systems, NAOS and TriGS are extensions of commercial object-oriented database systems with active capabilities. All the proposals support ECA rules. However, in some systems, such as Ode, conditions are part of the event specification as a mask which qualifies the event and can refer to parameters of the method call defining the event.

In an active rule, the *event* specifies what causes the rule to be triggered. Relevant triggering events are internal events related to database operations, temporal events, external (that is, raised by the application), and user-defined events. Database operations are object accesses, attribute value updates, object creations and deletions, method executions and calls to transaction primitives (e.g. commit). In Chimera, also object migration between classes is included among events. In addition, in some systems it is possible to specify whether a rule must be triggered after or before its triggering event. Triggering a rule before its event is meaningful when methods are considered as events, and allows method preconditions to be tested. Triggering events may also be *composite*, that is, combinations of other events. Useful operators for combining events are logical operators, such as conjunction, disjunction and negation, and sequences. Some systems allow events to be parameterized; when a parameterized event occurs, values related to the event are bound to the event parameters, and these parameter values can be referenced in rule conditions or actions. Some systems only consider as an implicit parameter the identifier of the object receiver of the event.

In an active rule, the *condition* specifies an additional condition to be checked once the rule is triggered and before the action is executed. Conditions are predicates over the database state. If the rule language supports parameterized events, the condition language includes a mechanism for referencing the values bound to the event parameters. In some systems, values related to the condition can be passed to the action. In the most common approach for parameter passing, the condition is expressed as a query returning data, that are then passed to the rule action. Moreover, some systems allow triggers to refer to past database states.

In an active rule, the *action* is executed when the rule is triggered and its condition is true. Possible actions include database operations and calls to application procedures. Several active database rule languages allow sets of actions to be specified in rules, usually with an ordering, so that the actions in a set are sequentially executed.

The considered systems differ not only with respect to the supported rule language, but also in terms of rule execution semantics. First of all, active database rule execution can be either *instance-oriented* or *set-oriented*. With an instance-oriented execution, a rule is executed once for each database "instance" triggering the rule and satisfying the rule condition. By contrast, rule execution is set-oriented if a rule is executed once for all database instances triggering the rule and satisfying the rule condition. The issue of instance-oriented versus set-oriented execution is obviously related to the issue of rule processing granularity. The most straightforward approach is to evaluate a triggered rule condition and to execute its action within the same transaction in which the triggering event occurs, at the soonest rule processing point. However, for some applications it may be useful to delay the evaluation of a triggered rule condition or the execution of its action until the end of the transaction; or it may be useful to evaluate a triggered rule condition or execute its action in a separate transaction. These possibilities result in the notion of *coupling modes* [11]. One coupling mode can specify the transactional relationship between a rule triggering event and the evaluation of its condition, while another coupling mode can specify the transactional relationship between a rule condition evaluation and the execution of its action. Possible coupling modes are: *immediate* (immediately following, within the same transaction), *deferred* (at the commit point of the current transaction), *decoupled* (in a separate transaction).

Among the considered systems, all but Chimera support an instance-oriented rule execution. Chimera supports set-oriented rule execution, while NAOS rule execution is instance-oriented for immediate rule and set-oriented for deferred ones.

A further aspect in rule execution is to consider the *net effect* of a sequence of operations performed in the triggering transaction. For instance, if a rule is triggered by the creation of an object, but this object happens to be deleted before the actual execution of the rule, the rule should not be executed. The net effect of events is computed based on the classical composition of pairs of operations applied to the same object[2]. Another difference among existing systems is whether rule definitions are attached to classes. Attaching rule definitions to classes enhances modularization and supports an efficient detection of relevant events while there are sometimes useful rules triggered by events spanning sets of objects possibly from different classes (untargeted rules).

Finally we introduce the *priority* notion that is extremely relevant for active rule semantics. The execution semantics for active rules sometimes requires that one rule is selected from a set of eligible rules. For this reason, an active database rule language may include a mechanism for specifying rule priorities. Priorities might be specified by ordering the set of rules, by declaring relative priorities between pairs of rules, or by assigning a numeric priority value to each rule. Relative priorities are the most flexible approach, since they subsume the other two types of priority specification. Ordering the entire set of rules may not be necessary to achieve the correct behavior, while numeric priorities can be difficult to use since they may need to be adjusted as the set of rules evolves. In the following section we will discuss how priorities could be used to simulate trigger overriding and point out the disadvantages and the

[2]Note that net effect is supported with different semantics in NAOS and Chimera. Whereas in NAOS the elimination of events due to net effect composition results in the de-triggering of rules, this is not true for Chimera.

| | HiPAC | Ode | Adam | NAOS | TriGS | SAMOS | Chimera |
|---|---|---|---|---|---|---|---|
| Reference | [10] | [3], [14] | [15] | [16] | [4] | [17] | [18] |
| o-o data model | OODAPLEX | new | new | $O_2$ | GemStone | new | new |
| primitive events | messages<br>db ops<br>temporal<br>external | messages<br>db ops<br>temporal | messages | messages<br>db ops<br>user-def | messages | messages<br>db ops<br>temporal<br>user-def | messages<br>db ops<br>migrations |
| event composition | YES | YES | NO | YES | NO | YES | YES(1) |
| parametric events | YES | NO | YES | NO(2) | NO(3) | YES | NO |
| parameter passing | $E \rightarrow C, A$<br>$C \rightarrow A$ | $E \rightarrow A$ | $E \rightarrow C, A$ | $C \rightarrow A$ | $E \rightarrow C, A$(3) | NO | $C \rightarrow A$ |
| conditions on past states | NO | NO | NO | NO | NO | NO | YES |
| net effect | NO | NO | NO | YES | NO | NO | optional |
| evaluation mode | immediate<br>deferred<br>decoupled | immediate<br>deferred<br>decoupled(4) | immediate | immediate<br>deferred(5) | user-specified | immediate<br>deferred<br>decoupled | immediate<br>deferred(5) |
| untargeted/ targeted rules | untargeted | targeted | untargeted(6) | untargeted | both | both | both |
| overriding | NO | YES(7) | NO | NO | YES | NO | NO |

Legenda:
(1) In [18] only disjunction of events is considered; an extension of Chimera with other kinds of event composition is described in [19].
(2) The only parameter is the object receiver of the event (on which the rule is being executed).
(3) Delta tables are used for referring to objects affected by events.
(4) This coupling mode is between Event and Action; immediate and deferred coupling modes between Event and Condition can be supported by means of event composition.
(5) This coupling mode is between Event and Condition; coupling mode between Condition and Action is always immediate.
(6) Triggers are defined outside the scope of classes, however they are indexed on classes for efficient detection of relevant events.
(7) Rules can be overridden provided that they are not activated in the superclass.

TABLE I
**Comparison of active object-oriented data models**

problems of that approach.

## III. Trigger Overriding

In this section we first motivate the need for supporting trigger overriding and then provide an overview of the approach to trigger inheritance and overriding proposed in the paper.

### *A. Motivating Examples*

The influence of inheritance on triggers has not been extensively investigated in existing object-oriented database systems. Under some proposals [2], [16], [20], 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 the proper instances of this class.

Full trigger inheritance is, however, not always appropriate, as shown by the examples below. There are situations in which trigger overriding is required. Moreover the meaning of the ISA hierarchy is to define a class in terms of another class, possibly refining its attributes, methods and triggers. This modeling approach is one of the key features of the object-oriented paradigm. Thus the possibility of redefining triggers in subclasses, instead of simply inheriting them, should be provided.

In a system supporting trigger inheritance, but not trigger overriding[3], 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

[3] We remark that this is almost always the case for existing active object-oriented database systems.

after) the superclass one. Consider the following examples.

*Example 1:* Suppose that a class `employee` and a subclass `manager` are defined. Moreover suppose that in the former class an attribute `rank` is defined. Consider a trigger $r_1$ that increases the salary of an employee each time the rank of that employee is increased. Moreover consider a corresponding trigger $r_2$ for the class `manager` that increases the salary of a manager of an amount greater than that of a simple employee. Notice that the trigger $r_2$ is the refinement of $r_1$ for the class `manager`. If trigger overriding is not supported we have to define two different triggers as follows[4]:

- trigger $r_1$ on class `employee`
```
Events:    modify(rank)
Condition: employee(X),occurred(modify(rank),X),
           Y=X.rank,old(Y>X.rank)
Action:    modify(employee.salary, X,
                  X.salary + 50)
```
- trigger $r_2$ on class `manager`
```
Events:    modify(rank)
Condition: manager(X),occurred(modify(rank),X),
           Y=X.rank, old(Y>X.rank)
Action:    modify(manager.salary, X,
                  X.salary + 100)
```

In the above triggers whenever the rank of an employee is increased, both $r_1$ and $r_2$ are triggered and if $r_1$ has priority over $r_2$ (as under the default ordering we mentioned above) $r_1$ is executed first. The execution of $r_1$ increases the manager salary by `50` and the execution of $r_2$ increases again the salary by `100`, thus at the end the manager salary has been increased by `150`, instead of only by `100`, which is not the desired result. This problem can be overcome by increasing the salary of a manager by the right amount in $r_2$ keeping into account that the salary has already been increased by $r_1$, but this is not intuitive and not object-oriented, since it does not model a refinement. In the case of complex class hierarchies, these calculi become very complex and not always possible. △

*Example 2:*
Consider a class `person` with a subclass `employee`. Suppose, moreover, that a third class `department` is defined, with an attribute `nbr_of_employees` 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, creates an object of a class `person_log` whose state refers to the deleted object (the class `person_log` is used for monitoring purposes) and then deletes the object. Suppose that a corresponding trigger $r_2$ is defined on class `employee`, such that, whenever the age of an employee is greater than 100, creates an object of a class `employee_log`, decrements the value for attribute `nbr_of_employees` of the department in which the employee works, and finally deletes the employee. The triggers are expressed in Chimera as follows:

- trigger $r_1$ on class `person`
```
Events:    modify(age)
Condition: person(X),occurred(modify(age),X),
           X.age > 100
Action:    create(person_log,(who:X,age:X.age),O);
           delete(person,X)
```
- trigger $r_2$ on class `employee`
```
Events:    modify(age)
Condition: employee(X),occurred(modify(age),X),
           X.age > 100, department(Y),
           X.department = Y
Action:    create(employee_log,(who:X,age:X.age,
                  salary:X.salary), O);
           modify(department.nbr_of_employees,Y,
                  Y.nbr_of_employees-1);
           delete(employee,X)
```

Whenever the age of an employee is set to `101`, both $r_1$ and $r_2$ are triggered and, if $r_1$ has priority over $r_2$ (as under the default ordering we mentioned above), $r_1$ is executed first. The execution of $r_1$ deletes the involved object, and then the execution of $r_2$ does not have any effect. Indeed, when $r_2$ is executed the object whose `age` attribute has been modified, has already been deleted. Therefore, it cannot be accessed any longer. As a consequence, the object does not satisfy trigger $r_2$ condition and therefore the value of attribute `nbr_of_employees` is not decremented. Intuitively trigger $r_2$ is just the refinement of $r_1$ because it has a behavior similar to that of $r_1$ but refined for subclass `employee`. In a system supporting inheritance and trigger overriding, trigger $r_2$ would be the refinement of trigger $r_1$, thus for the objects proper instances of `employee` only trigger $r_2$ would be executed giving a correct result. △

As shown by the previous examples, the lack of trigger overriding capabilities does not allow triggers to manage in different ways the proper and non-proper instances of a class.

Note that in Example 2 above trigger $r_2$ performs a behavioral refinement of trigger $r_1$. This means that after the modification of the age of an employee the execution of $r_2$ ensures the expected effect of $r_1$ (that is, the creation of the appropriate `person_log` object and the deletion of the employee, if the age is greater than 100). This ensures that the intended semantics is inherited by the subclass.

### *B. Overview of the Proposed Approach*

Our approach extends the Chimera rule language with the possibility of overriding triggers in subclasses. In the current version of Chimera trigger overriding is not allowed and only full trigger inheritance is supported. The way in which trigger overriding is accomplished in our approach is simple. Let $r$ be a trigger defined in a class $c'$, $r$ can be overridden by the definition of a new trigger in class $c$, subclass of $c'$, with the same name of $r$. Chimera supports late binding, thus at execution time for each object affected by the execution of the specified trigger the most specific implementation will be chosen.

In order to fully investigate our approach to trigger inheritance and overriding, we develop a formal definition of our reference language and its semantics to have a strong, well-defined formalism. One of the problems arising in de-

[4] The syntax used in these examples is Chimera syntax which will be formally introduced in Section IV.

fining the semantics of an active object language supporting trigger inheritance 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 objects 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 $c$. For objects proper instances of $c'$ the method implementation in class $c'$ is selected, where the trigger itself is defined. By contrast, for objects proper instances of $c$ two different options are possible: ($i$) choosing the most specialized implementation of $op$ (that is, the implementation in class $c$); ($ii$) choosing 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. In our model, we have adopted the first approach, because it is consistent with the object-oriented approach, in that it conforms to the principle of exhibiting the most specific behavior. The rule-specific method selection is not consistent with the object-oriented approach because it refers to the static nature of objects, that is, the class in which the trigger is defined, and not to their dynamic nature, that is, the classes the objects are proper instances 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 [14].

After having defined the semantics of the language, we investigate semantic properties of trigger refinement. Because our model directly supports trigger overriding, it is important to give the possibility of specifying that trigger semantics must be preserved in subclasses. In particular, in order to preserve trigger semantics, it must be ensured that: ($i$) the trigger in the subclass is executed at least each time the trigger in the superclass would be executed; ($ii$) 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. Trigger refinement is formalized in the paper in terms of the defined semantics and, unfortunately, will be stated as undecidable (see Proposition 1).

The following step is thus that of devising some sufficient static conditions ensuring trigger refinement that can be checked at trigger definition time. These properties are based on query containment for what concerns the condition part, and on a static net effect computation for what concerns the action part. As we will show in the remainder of the paper, these condition can also be exploited to detect *redundant* triggers in an active object database schema.

## IV. Active Rule Language: Syntax

As reference rule language we consider a subset[5] of the Chimera active rule language [18]. Chimera supports *set-oriented* active rules [21]: rules react to sets of changes to the database and may perform sets of changes. This approach is consistent with the remainder 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 have several innovative features: ($i$) 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 processing events; ($iii$) they support different *processing modes*, that is, different activation times; and finally ($iv$) they provide mechanisms for accessing intermediate states of affected objects during transaction execution. In Chimera, the processing mode of an active rule (which is the coupling mode between event and condition) 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). In Chimera, two distinct event consumption modes are possible for each active rule; this feature is relevant when a given rule is considered multiple times within 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. However, we do not consider all these features here, since they are not relevant with respect to the problems of trigger inheritance and redefinition. Thus, we restrict ourselves to a simpler rule execution model, allowing a clear understanding of rule overriding and supporting only deferred, preserving rules without net effect composition. For the sake of simplicity in presentation, moreover, we consider here only targeted rules, disregarding untargeted ones. The restriction to targeted rules does not affect the generality of the problem.

Active rules in Chimera are called *triggers*. Each trigger is characterized by five components: a name, a class, a set of events[6], a condition and a reaction. Events are denoted by the name of the primitive operation and the schema element to which the operation is applied. Primitive operations are object creations, deletions, modifications and object migrations in the inheritance hierarchy. Modifications refer to specific attributes. In addition, active rules

[5] This restriction, as we will discuss in what follows, allows us to focus on problems related to trigger inheritance and overriding. The proposed approach can however be generalized to the full Chimera language.

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

may monitor operation calls (methods), although rule execution remains set-oriented. Method execution itself is indeed set-oriented.

In the remainder of this section we formally state the trigger definition language. In the 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 method names $\mathcal{MN}$.

*Definition 1:* (*Event*). Let $c \in \mathcal{CN}$ be a class name, $a \in \mathcal{AN}$ be an attribute name and $op \in \mathcal{MN}$ be a method name. A Chimera event has one of the following forms:

- $create$;
- $delete$;
- $generalize(c)$;
- $specialize(c)$;
- $modify(a)$;
- $op$. □

The condition, that is, the formula monitoring the execution of the reaction part, 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 predicate *occurred*. This predicate is used to inspect the events that have occurred during a transaction. Syntactically, this predicate has two arguments: an event name and a variable ranging over the *oids* of the objects affected by the event, which becomes bound to *oids* of instances which are receivers of the event. *References to past database states* are allowed in active rule conditions through the use of function *old*. Such a function, applied to an atomic formula, indicates that the formula must be evaluated in a previous database state. Since we restrict ourselves to event preserving rules the old state always refers to the state at transaction start.

Chimera terms are defined as follows. Constants (except *oids*) and variables are terms; structured terms can be built by applying set, list and record constructors. Path expressions (built by making use of the dot notation) are terms, too. In addition Chimera supports several standard predefined operators that can be used to build terms. These predefined operators include arithmetic operators, set operators and list operators.

Chimera atomic formulas can be of four types, in which $t_1, t_2$ are terms:

- COMPARISON FORMULAS: $t_1\ op\ t_2$ where $op \in \{<, >, \geq, \leq, =, ==, ==_d\}$[7];
- MEMBERSHIP FORMULAS: $t_1\ in\ t_2$ or $t_1\ in\ c$ where $c$ is a class name;
- CLASS FORMULAS: $c(X)$, where $X$ is a variable and $c$ is a class (or type) name;
- EVENT FORMULAS: $occurred(e, X)$, where $X$ is a variable and $e$ is an event according to Definition 1.

[7] Notice that Chimera provides three different types of equality: $=$ denotes equality by identity, $==$ denotes equality by value on the direct attributes, while $==_d$ denotes equality by value on all the attributes, even the ones recursively reached by means of *oid* based references.

Atomic formulas may be applied to the state at transaction start through the *old* function. All variables are assumed to be implicitly quantified as in Datalog [22].

Complex formulas (or simply formulas) are obtained from atomic formulas and negated atomic formulas by means of conjunctions. Formally, if $F$ is an atomic comparison or membership formula[8], 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.

We require that each formula contains exactly one class formula for each variable, specifying the type of the variable. In addition, we require formulas 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 statements. A condition and an 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 sequence, because each of them may have side effects.

*Definition 2:* (*Action*). 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:* (*Trigger*). A Chimera trigger is a 5-tuple

$$(Name,\ Class,\ Events,\ Condition,\ Action)^9$$

where:

$Name$ is the trigger identifier;

$Class$ is the class the trigger is targeted to;

$Events$ is the set of operations monitored by the trigger, each event in the set is as in 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 atomic formulas of the $Condition$ (safety condition);
2. for each event formula $occurred(e, X)$ in the $Condition$, $e$ must appear in $Events$. □

We remark that in Chimera events are not parametric and there is no parameter passing between the event and

[8] Class and event formulas cannot be neither negated nor tested on past database states.

[9] In the following, given a trigger $r$, $r.Name$, $r.Class$, $r.Events$, $r.Condition$, $r.Action$ denote the respective components of trigger $r$.

other rule components. Thus, events only cause rules to be triggered; 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 3:* The following is a Chimera trigger, defined on a class `employee`, whose effect is to prevent an employee from earning more than his manager. If an employee is assigned a salary higher than the salary of his manager, 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, also defined on the class `employee`, that has the effect of specializing each new employee earning more than `40000` by inserting the employee in the class `specialEmp`[10]).

```
Events:    create
Condition: employee(X), occurred(create,X),
           X.salary > 40000
Action:    specialize(employee,specialEmp,X,()
```

△

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 an object-oriented system, an important question is whether the triggers inherited by the superclasses of a class should have higher priority than triggers defined in the class, especially when dealing with action refinement in triggers.

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 belonging to the same class $c$ (denoted by $<_r^c$), with the order induced by inheritance relationships among classes. Thus, local priorities, specified by the user for triggers in the same class, are combined by the system with the order induced by ISA relationships [11]. To privilege the most specific behavior, the reverse ISA ordering is considered as a default for relating triggers defined in different classes. That is, given two classes $c$ and $c'$, such that $c$ is a subclass of $c'$, each trigger $r'$ defined in $c'$ has priority over $r$ defined in $c$ ($r' <_r r$), that is, it is executed first. Note that this policy is in accordance with the intuition discussed in Section III for simulating trigger overriding when it is not directly supported. Our policy for trigger priority is formally established by the following definition.

*Definition 4:* (*Priority Order on Triggers*). 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
- $r_2.Class \leq_{ISA} r_1.Class$. □

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 could be modified by the user in the subclass definition.

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 class $r_2.Class$ does not trigger $r_1$. Notice that a trigger can be overridden in a subclass only by a trigger with the same name defined in the subclass and that, viceversa, a trigger in a subclass that has the same name as a trigger $r$ in a superclass overrides $r$.

## V. Active Rule Language: Semantics

In this section we present a formal semantics of Chimera triggers. We start by providing an intuitive idea of the active rule execution model, then we formalize the notions of database state, set of bindings, and reactive process, and finally in Subsection V-C we present the semantics. The semantics defined in this section will be the basis for investigating trigger semantic refinement, that will be discussed in Section VI.

### A. Intuitive Idea

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 triggering each other exists in Chimera; techniques and tools for detecting the possible sources of nontermination in a rule set have been developed [18].

A transaction in Chimera is a sequence of data manipulation statements, each of which may trigger some rules. Remember, however, that we consider a language supporting

[10] In the trigger symbol () denotes the empty record value. Class `specialEmp`, indeed, has no proper attributes in addition to those inherited by class `employee`.

[11] Given two classes $c$ and $c'$, $c \leq_{ISA} c'$ denotes that $c$ is a subclass of $c'$.

deferred, preserving rules without net effect composition. 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. The set of rules, triggered by the set of events associated with the transaction, is computed when the transaction ends. Therefore, the end of the transaction corresponds to reactive processing activation. On this set of triggered rules, rule selection and the other phases of rule processing are then iterated until no rule is triggered any longer.

In order to formalize the semantics we have, first of all, to model the following preliminary notions.

### *B. Preliminaries*

Before formalizing the rule language semantics, we need to introduce some preliminary definitions. In particular we introduce the notions 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 $\mathcal{CN}$ of classes; a set $Var$ of variables. In the following the set $Rule$ denotes the set of rules defined for a database. Moreover: let $S$ be a set, then $2^S$ denotes the powerset of $S$; let $\langle el_1, \ldots, el_n \rangle$ be a $n$-tuple, then $\Pi_i$, $i \in [1, n]$, denotes the projection on the $i$-th component of the tuple.

#### B.1 Database State

In the literature different definitions, some of which quite complex, of object-oriented database state have been proposed. The notion of state we propose here is complete (according to our needs) but simple and quite similar to the ones that can be found in literature [23].

Our model, like most object-oriented data models, distinguishes between the schema level, which represents the database structure definition and is the time-invariant component, and the instance level, which represents the database content and is the time-varying component. Informally, a database schema is a set of class definitions, related by inheritance relationships. A class definition consists of a class name, a list of attribute definitions (name and domain of each attribute), a list of method definitions (name and signature of each method), and a list of triggers.

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

- $\pi : \mathcal{CN} \rightarrow 2^{OID}$ is a function associating with a class the set of its *oids*, that is, for each $c \in \mathcal{CN}$, $\pi(c) = \{oid | oid \in OID \textit{ and is the oid of an object belonging to class } c\}$. $\pi(c)$ is called the *extent* of class $c$.
- for each $oid \in OID$, $\nu(oid)$ returns the state of the object, that is, the value of its attributes; let the attribute names be $a_1, \ldots, a_n$, and $v_1, \ldots, v_n$ the corresponding values, then $\nu(oid) = [a_1 : v_1, \ldots, a_n : v_n]$[12]. □

We remark that $\pi(c)$ denotes the proper instances of class $c$, that is, 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 class $c$, that is, the set of all its instances. $S.\pi$, $S.\nu$ denote the first and the second component of the database state $S$, respectively.

#### B.2 Set of Bindings

Informally we can state that a set of bindings $B$ is a set of substitutions. 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 means by which such variable passing is achieved. Because condition and action parts share some variables, the action must be executed for every binding generated by the condition on the shared variables. We are interested in defining the set of bindings that satisfy a given condition, that is, the set of values which, substituted to the variables in the condition, makes the condition true. We model a set of bindings as a set of ground substitutions.

*Definition 6:* (*Substitution and Set of Bindings*). A ground substitution $\theta$[13] is a partial function from $Var$ to $\mathcal{V}$, $\theta : Var \rightarrow \mathcal{V}$. A set of bindings $B$ is a set of ground substitutions $\{\theta_1, \ldots, \theta_m\}$[14]. □

Intuitively, the set of bindings $B = \{\theta_1, \ldots, \theta_m\}$ satisfying a condition $C$ is the set of ground substitutions such that the application of each $\theta_i$ $(i \in [1, m])$ to $C$, denoted as $C\theta_i$, is a ground formula which is true according to first order logic. Given $X$, $Y$, $Z$, $W$ variables and a set of values $\mathcal{V}$ including integers, the following are examples of substitutions:

$$\theta_1 = \{X/5, Y/7, Z/8\} \quad \theta_2 = \{X/7, W/10\} \quad \theta_3 = \{W/8\}.$$

In what follows, given a substitution $\theta$ and a set of variables $V$, $\theta_{|V}$ denotes the restriction of substitution $\theta$ to variables in $V$. Moreover, given a set of substitutions $S$, $S_{|V}$ denotes the set of substitutions $\{\theta_{|V} \mid \theta \in S\}$.

#### B.3 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:* (*Event Instance*). Let $e$ be an event as in Definition 1, $c$ the class name such 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 an 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[15]. For the sake of simplicity, we will often use the word event to denote event instances, when the meaning is clear from the context.

[12] To denote the value of an attribute $a_j, j \in [1, n]$, we use the following notation: $\nu(oid).a_j = v_j$. Note that $\nu$ is a function, because, given an object, its state is unique. Moreover, $\nu$ is not injective: given two distinct objects, $\nu$ can return the same state, although they are distinct entities, and $\nu$ is partial since $OID$ is the set of all possible *oids* that can be allocated by the system.

[13] In general a substitution is a partial function $\theta : Var \rightarrow \mathcal{T}$ where $\mathcal{T}$ is the set of terms of the language. $\mathcal{V} \subseteq \mathcal{T}$ are the simple ground terms, that is, terms with no variables and no operators.

[14] In the following, we often refer to substitutions as a subset of the cartesian product $Var \times \mathcal{V}$ which includes only pairs for which $\theta$ is defined.

[15] Note that $O$ in $create(c,t,O)$ denotes a variable while in $\langle create, c, O \rangle$ denotes a set of *oids*.

Given an event instance a relevant issue is how to establish which rule has to be triggered for each *oid* in the set of objects affected by the event. Consider two classes $c$ and $c'$, where $c$ is a subclass of $c'$ and the action $delete(c, O)$. Since each object instance of $c$ is an instance of $c'$ as well, objects of class $c$ have been deleted, but objects of class $c'$ have been deleted as well. This means that the delete event is propagated along the ISA hierarchy. Triggers defined in class $c'$, and not redefined in class $c$, must be activated, as class $c$ inherits them from class $c'$. Moreover, since our language supports trigger overriding, given an event instance, establishing which rules have to be triggered for each object 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. Trigger semantics, however, is complicated by the fact that usually an event instance does not trigger a single rule but a set of rules, possibly defined in different classes in the ISA hierarchy. The basic idea is to partition the set $O$ of the objects affected by the event into disjoint subsets of *oids*, say $O_1, \ldots, O_n$, according to the most specific class of each object. Then, for each class $c_i$, $i \in [1, n]$, the set of the triggered rules is determined through a lookup mechanism. An ascending visit of the ISA hierarchy is performed, collecting in the set of triggered rules all triggers whose event part contains the occurred event and for which a trigger with the same name has not yet been included in the set[16]. Thus, the set of rules triggered by an event is computed by taking into account that, for each object, the most specific rules are triggered, as formalized by the following definition.

*Definition 8:* (*Rule Triggered by an Event Instance*). A rule $r = (N, c, Ev, C, A)$ is triggered by an event instance $\mathbf{e} = \langle e, c_e, O\rangle$ if the following conditions are verified:

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

We make use of the following notation. Let $oid \in O$, then $trig(oid, \langle e, c_e, O\rangle) = \{r \mid r = (N, c, Ev, C, A)$ and $e \in Ev$ and $c = min_{\leq_{ISA}}\{\bar{c} \mid oid \in \pi^*(\bar{c})$ and $\exists\ \bar{r} = (N, \bar{c}, \bar{Ev}, \bar{C}, \bar{A}) \in Rule\}\}$.

*Definition 9:* (*React*). 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 $react(E) = \bigcup_{\mathbf{e} \in E} react(\mathbf{e})$. □

*Definition 10:* (*Reactive Process*). Given a transaction $T$, let $E$ be the set of event instances associated with $T$, $react(E)$ is said to be the reactive process of transaction $T$. □

Given a reactive process, that is, 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 several rules at highest priority, the choice is non deterministic. We therefore introduce function *get_max*, which nondeterministically chooses a rule among the rules with the highest priority in a set of rules. In this context we do not deal with issues related to non determinism in rule selection. We refer the reader to [24] for an analysis of such issues.

*Definition 11:* (*Function get_max*). Let $R$ be a reactive process, then function $get\_max : 2^{Rule} \rightarrow Rule$ returns one of the rules at highest priority belonging to $R$. Formally: given $R \in 2^{Rule}$ if $get\_max(R) = r$ then $\nexists\ r' \in R : r' <_r r$. □

## C. Trigger Semantics

First of all we introduce semantic domains and semantic functions. In defining semantic domains we refer to well-formed triggers defined according to Definition 3.

*Definition 12:* (*Semantic Domains*). The semantic domains we consider in giving the semantics are the following:

- $Bind$ = set of possible sets of bindings.
- $State$ = set of possible database states.
- $Event$ = set of possible sets of event instances.
- $Cond$ = set of possible condition parts of rules.
- $Update$ = set of possible action parts of rules[17].
- $Rule$ = set of possible rules of the language. □

*Definition 13:* (*Semantics*). The semantics of the trigger language is a family of functions defined as follows:

$\mathcal{C} : (Cond \times State) \rightarrow ((State \times Event) \rightarrow Bind)$

$\mathcal{U} : (Update \times State) \rightarrow ((Bind \times State \times Event) \rightarrow (Bind \times State \times Event))$

$\mathcal{R} : (Rule \times State) \rightarrow ((State \times Event) \rightarrow (State \times Event))$

$\mathcal{P} : (2^{Rule} \times State) \rightarrow ((State \times Event) \rightarrow State)$. □

Function $\mathcal{C}$ models condition evaluation; function $\mathcal{U}$ models action execution, whereas functions $\mathcal{R}$ and $\mathcal{P}$ model reactive processing semantics. Notice that the arguments of the semantics functions are pairs of the form $\langle$*construct to be evaluated, state*$\rangle$, where the state, according to our assumption of event preserving rules, is the one at the beginning of the transaction. We need the state as argument for two reasons: ($i$) evaluation of *old* formulas; ($ii$) rollback upon transaction errors.

In the first case to evaluate formulas of the form $old(F)$ we need to evaluate the formula $F$ in the state at the beginning of the transaction. In the case of errors in evaluating a rule, such as a division by zero in a condition or other errors, according to the transaction *all or nothing* philosophy, the transaction must be aborted and the initial state must be restored.

In what follows, to simplify the notation, we will often omit the argument $S_i$, the initial state, if it is not strictly necessary.

As we have said, we consider deferred event preserving rules. From a semantic point of view this means that during the execution of the transaction, events (event instances) are collected. When the transaction terminates two situations can arise:

[16] This most specific trigger overrides the one which is currently under examination.

[17] *Update* corresponds to the set of well-formed update sequences of the language.

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

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. Thus, function $\mathcal{P}$, at each step: $(i)$ selects from the set of rules constituting the reactive process the rule at highest priority through function $get\_max$, let $r$ be this rule; $(ii)$ evaluates $r$ through function $\mathcal{R}$; $(iii)$ deletes $r$ from the reactive process; $(iv)$ adds the rules triggered by the action part of $r$ to the reactive process; $(v)$ evaluates the new set of rules by function $\mathcal{P}$ on the state resulting from the evaluation of $r$ through function $\mathcal{R}$ and so on. The evaluation of a rule $r$ is performed by function $\mathcal{R}$ and consists of evaluating the condition of $r$ using function $\mathcal{C}$, obtaining a set of bindings and evaluating through function $\mathcal{U}$ 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, in Chimera triggers, bindings are local to rules.

### C.1 Condition Semantics

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 $\theta \in B$ is such that the instantiation of $C$ with respect to it, that is $C\theta$, evaluates to true in state $S$ and with respect to events in $E$, according to first order logic.

As several approaches to semantic evaluation of first order logic formulas can be found in the literature [25], we do not discuss them in this context. Rather, we focus on event and *old* formulas which are typical of trigger conditions.

First we analyze the semantics of *occurred* formulas. We have to evaluate the semantics of a formula of form $occurred(e, X)$ appearing in the condition part of a trigger $r$, given a database state $S$ and a set of event instances $E$. Notice that, during a transaction, a trigger $r$ can be activated by several update operations. This fact corresponds to the insertion of $r$ in the reactive process of transaction $T$ and to the insertion in $E$ of event instances activating $r$. Thus, when we evaluate $occurred(e, X)$ in $r$ we have to take into account all event instances that have activated $r$. Moreover for each event instance $\mathbf{e} = \langle e, c_e, O\rangle$ that has activated $r$, we have to choose only the *oids* belonging to $O$ which actually have activated $r$, that is, those *oids* for which $r$ is the most specific trigger. Formally, the evaluation of the formula $occurred(e, X)$ in trigger $r$ is:

$$\mathcal{C}\,[\![\, occurred(e, X) \,]\!]\, SE = B$$

where $B = \{\{X/oid\} \mid \exists\ \langle e, c, O\rangle \in E \text{ and } r \in trig(oid, \langle e, c, O\rangle)\}$.

For what concerns *old* formulas, as we consider event preserving triggers, the formula $F$ in $old(F)$ must be evaluated in the state at the beginning of the transaction, that is, $S_i$. Formally the semantics is:

$$\mathcal{C}\,[\![\, old(F)_{S_i} \,]\!]\, SE = \mathcal{C}\,[\![\, F_{S_i} \,]\!]\, S_i E.$$

### C.2 Action Semantics

Function $\mathcal{U}$ models the semantic evaluation of a sequence of updates, $u_1$;...;$u_n$, that is, an action part of a rule $r$. 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 bindings, 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. A complete definition of function $\mathcal{U}$ can be found in Appendix I.

Note that in defining function $\mathcal{U}$ we consider an object-specific method selection, that is, 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.

Our strategy is based on the assumption that methods redefined in subclasses are a behavioral refinement of corresponding methods in superclasses, according to the behavior refinement constraints imposed on Chimera method redefinitions in [26].

To model reactive process activation, at transaction commit, the following semantics is specified for the *rollback* and *commit* transactional statements:

$$\mathcal{U}\,[\![\, rollback_{S_i} \,]\!]\, BSE = \langle \emptyset, S_i, \emptyset \rangle$$

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

Notice that since *rollback* (*commit*) is the last command of a transaction, the output values of $B$ and $E$ are set to $\emptyset$ because they are not meaningful. The only interesting value is the resulting state, the second component of the result.

### C.3 Reactive Process Semantics

Now we can formally define the semantic functions $\mathcal{R}$ and $\mathcal{P}$.

*Definition 14:*
(*Rule Semantics*). 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 $E' = \Pi_3(\mathcal{U}\,[\![\, A \,]\!]\, BSE)$. □

Notice that the set of bindings, resulting from the evaluation of the sequence $u_1; \ldots; u_n$, that is, $\Pi_1(\mathcal{U}\,[\![\, A \,]\!]\, BSE)$, does not appear in the previous definition because in Chimera triggers there is no variable passing among rules in the reactive process. The semantics of a reactive process is given by the following definition.

*Definition 15:* (*Reactive Process Semantics*). 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:

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

□

Notice 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.

## VI. Trigger Refinement

As we have already discussed, in our approach a class can redefine a trigger of one of its superclasses, instead of simply inheriting it. Rule overriding is supported in some systems such as TriGS [4] and Ode [27], but no restrictions are imposed on rule overriding, thus a rule may override another rule on completely different events and performing completely different actions. In our model, as in those systems, trigger overriding is directly supported. However, we support the possibility to impose that the overriding trigger in the subclass is a *behavioral refinement* of the trigger in the superclass, that is, ($i$) the trigger in the subclass is executed at least each time the trigger in the superclass would be executed; ($ii$) what would be executed by the trigger in the superclass is also executed by the overriding trigger.

More specifically, a trigger $r_2$ is a behavioral refinement of a trigger $r_1$ if the portion of state manipulated by $r_2$ includes the portion of state manipulated by $r_1$ and the portion of state modified by both is modified in the same way[18]. To formally define this notion, we must first model the changes made by a trigger execution. Given a trigger $r$, let $\mu(r)$ be the set of classes manipulated by $r$ (either through a *create*, a *delete*, or a *modify* operation)[19]. 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 object $oid$ and an attribute name $a$, let $\nu_r(oid).a$ be defined if and only if the execution of trigger $r$ has modified the value of attribute $a$ of the object identified by $oid$, and, if defined, let it contain the new value of the attribute. Those notions are formally defined as follows.

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

[19] They can be deduced syntactically.

*Definition 16:* (*Portion of State Manipulated by a Trigger*). Given a trigger $r$ such that $\mu(r)$ is the set of classes manipulated by $r$ and a database state $S$, such that $\mathcal{R}\,[\![\, r \,]\!]\, SE = \langle S', E' \rangle$, consider the following functions:

- $\forall\ c \in \mu(r)$ then $\delta_r(c) = S.\pi(c) \setminus S'.\pi(c)$
- $\forall\ c \in \mu(r)$ then $\iota_r(c) = S'.\pi(c) \setminus S.\pi(c)$
- $\forall\ c \in \mu(r)$, $\forall\ oid \in S'.\pi(c)$ then let $\nu_r(oid).a = v$ if and only if the execution of trigger $r$ has modified the value of the attribute $a$ of object $oid$ setting it equal to $v$.

The triple $\langle \delta_r, \iota_r, \nu_r \rangle$ models the portion of state manipulated by trigger $r$. □

*Definition 17:* (*Behavioral Trigger Refinement*). 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:

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

Unfortunately, the following proposition (proved in [28]) holds.

*Proposition 1:* 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. These conditions are referred to as *refinement conditions*. In what follows we illustrate those conditions, by first analyzing each trigger component separately.

### A. 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 18:* (*Event Refinement*). An event set $Ev$ is a refinement of an event set $Ev'$ if and only if $Ev' \subseteq Ev$. □

*Example 4:* Consider a class `person`, with an attribute `income`, and a trigger that if the value of income is less than 0, assigns 0 to the income. The event component of that trigger is the set { `modify(income)` }. Consider moreover a class `employee`, subclass of `person`, with two attributes `income` and `fees` and suppose to refine the trigger so that whenever the amount of income minus the amount of fees is less than 0, the attribute `income` is set to the value of attribute `fees`. The event component of the refined trigger is { `modify(income), modify(fees)` }. △

### B. Condition

The basic idea behind condition refinement is that, with respect to the instances of the subclass, the condition in the refined trigger must be less selective than the condition in the inherited trigger. This ensures that the action of the refined trigger is executed each time the action of the inherited trigger would have been executed on an instance of the subclass.

*Example 5:* Consider the classes `person` and `employee`, and the constraint that the age of a person must be greater than 0, while the age of an employee must be greater than 15. Suppose this constraint is enforced in `person` by the following trigger:

Events: `modify(age)`
Condition: `person(X), occurred(modify(age),X),`
`X.age < 0`
Action: `rollback`

whereas the constraint is enforced in `employee` by the following refined trigger:

Events: `modify(age)`
Condition: `employee(X), occurred(modify(age),X),`
`X.age < 15`
Action: `rollback`

△

To formalize this notion we first need to introduce a new concept, that is, the one of specialized condition. Since conditions are formulas we give the general concept of specialized formula in the following definition.

*Definition 19:* (*Specialized Formula*). Let $c_1$ and $c_2$ be two classes such that $c_2$ is subclass of $c_1$. Moreover, let $F$ be a formula, then $F[c_1/c_2]$, the specialized formula of $F$ with respect to class $c_2$, denotes the formula obtained from $F$ by substituting each class formula $c_1(X)$ with a class formula $c_2(X)$. □

*Example 6:* Let $F$ be the first condition of Example 5, then $F[\texttt{person/employee}] = \texttt{employee(X)}$, $\texttt{occurred(modify(age),X)}, \texttt{X.age} < \texttt{0}$. △

The bindings produced by the evaluation of the condition are represented as a set of ground substitutions, as seen in Section V-B.2. A substitution $\rho$ is a *renaming* of variables if $\rho = \{X_1/t_1, \ldots, X_n/t_n\}$, and, for each $i \in [1, n]$, $t_i$ is a distinct variable. Moreover, we introduce the following notations.

**Notations**

- Given a trigger $r$, let $BVar(r)$ denote the set of variables appearing in $r.Condition$ and in $r.Action$ (that is, the variables employed for passing bindings).
- Given a renaming of variables $\rho$, let $\precsim^\rho$ denote subset (non strict) inclusion modulo renaming $\rho$. That is, let $B$ and $B'$ be sets of substitutions (sets of bindings) or set of variables, $B \precsim^\rho B'$ means $B \subseteq B'\rho$; let $\simeq^\rho$ denote equality modulo renaming $\rho$, $B \simeq^\rho B'$ means $B = B'\rho$. Equality modulo renaming $\rho$ also holds between terms. Let $t$ and $t'$ be terms, $t \simeq^\rho t'$ means $t = t'\rho$ where $t'\rho$ is the term $t'$ where each variable has been substituted according to renaming $\rho$.
- Given a formula $F$, let $F^{-E}$ denote the formula obtained by eliminating the event formulas appearing in $F$.

We can now formalize the condition refinement notion.

*Definition 20:* (*Condition Refinement*). A condition $r_2.Condition$ is a refinement of a condition $r_1.Condition$ (denoted as $r_2.Condition \leq_c^\rho r_1.Condition$) if and only if the following conditions hold:

1. $BVar(r_1) \precsim^\rho BVar(r_2)$, and
2. for each event formulas $occurred(e, X)$ in $r_1.Condition$, a corresponding event formulas $occurred(e, X')$ is in $r_2.Condition$, such that $X \simeq^\rho X'$, and
3. $\forall S$ database state, $\mathcal{C}\,[\![\, r_1.Condition^{-E}[r_1.Class/r_2.Class] \,]\!]\, S\emptyset \precsim^\rho (\mathcal{C}\,[\![\, r_2.Condition^{-E} \,]\!]\, S\emptyset)_{|BVar(r_1)}$

for a renaming of variables $\rho$. □

Condition 3 of Definition 20 above is that the specialized condition corresponding to $r_1.Condition^{-E}$ is *subsumed* by $r_2.Condition^{-E}$. Query subsumption (also called query containment) has been widely investigated, and algorithms for deciding subsumption among object-oriented queries have been proposed [29], [30]. Algorithms for testing subsumption among Chimera formulas have been also developed [31]. 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[20]. Indeed, $old(F)$ subsumes $old(G)$ if and only if $F$ subsumes $G$.

We remark that, though the test for subsumption has a cost exponential in the formulas that can appear in rule conditions, this is not a problem. First of all, this complexity is exponential in the dimension of the formula and the schema, not in the dimension of the database. Second, the analysis of whether a rule is a correct refinement of another one is executed once at rule definition time. More efficient subsumption tests for rule conditions could be used at the expense of reducing the expressiveness of the language for condition specification [29].

*Example 7:* 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 $r_1.Condition$ under the empty renaming of variables $\epsilon$, indeed:

1. $BVar(r_1) = \{\texttt{X}\} \precsim^\epsilon BVar(r_2) = \{\texttt{X}, \texttt{Y}\}$;
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` is subsumed by $r_2.Condition^{-E}$ = `employee(X), X.age > 65, department(Y), X.department = Y` restricted to variable `X`. △

### *C. Action*

The basic approach to achieve behavior consistency is based on ensuring that, for each action in the inherited

[20] 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.

trigger, there is a corresponding action in the refined trigger. However, since rule actions are sequences, 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 first of all creates a corresponding object and then deletes it. We consider, therefore, the net effect of the actions in the sequence. We state that for each action in the net effect of 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.

We now formalize these notions. Given a record term $t$ and a class $c$, let $t_{|c}$ denote the restriction of the term to the field whose labels are attributes of $c$.

*Definition 21:* (*Basic Action Refinement*). An action $u$ is a refinement of an action $u'$ under the renaming of variables $\rho$ (denoted as $u \leq_b^\rho u'$) if and only if one of the following conditions holds:

- $u = create(c, t, O)$, $u' = create(c', t', O')$ and $c \leq_{ISA} c'$, $O \simeq^\rho O'$, $t_{|c'} \simeq^\rho t'$;
- $u = delete(c, O)$, $u' = delete(c', O')$ and $c \leq_{ISA} c'$, $O \simeq^\rho O'$;
- $u = generalize(c, c_1, O)$, $u' = generalize(c', c'_1, O')$ and $c \leq_{ISA} c'$, $c'_1 \leq_{ISA} c_1$, $O \simeq^\rho O'$;
- $u = specialize(c, c_1, O, t)$, $u' = specialize(c', c'_1, O', t')$ and $c \leq_{ISA} c'$, $c_1 \leq_{ISA} c'_1$, $O \simeq^\rho O'$, $t_{|c'_1} \simeq^\rho t'$;
- $u = modify(c.attr, O, t)$, $u' = modify(c'.attr, O', t')$ and $c \leq_{ISA} c'$, $t \simeq^\rho t'$, $O \simeq^\rho O'$;
- $u = O.op(t_1, \ldots, t_n)$, $u' = O'.op(t'_1, \ldots, t'_n)$ and for each $i, i \in [1, n]$, $t_i \simeq^\rho t'_i$, $O \simeq^\rho O'$;
- $u = rollback$ and $u' = rollback$. □

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 [16], [21] 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;
- a sequence of several *modify* primitives 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 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.

*Definition 22:* (*Action Refinement*). A reaction $r_2.Action$ is a refinement of a reaction $r_1.Action$ (denoted as $r_2.Action \leq_a^\rho r_1.Action$) if the following conditions hold:

- $Net(r_1.Action) = u'_1; \ldots; u'_n$, $Net(r_2.Action) = u_1; \ldots; u_m$ and $m \geq n$;
- for each $u'_i$, $i \in [1, n]$, in $Net(r_1.Action)$, $u_j$, $j \in [1, m]$, in $Net(r_2.Action)$ exists, such that $u_j \leq_b^\rho u'_i$, that is, $u_j$ is a refinement of $u'_i$ according to Definition 21; let function $\xi : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$, such that $\xi(i) = j$, model this correspondence;
- if $u'_i$ precedes $u'_k$ in $Net(r_1.Action)$ $u_{\xi(i)}$ precedes $u_{\xi(k)}$ in $Net(r_2.Action)$. □

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

*Example 8:* Suppose `employee_log` $\leq_{ISA}$ `person_log` and `employee` $\leq_{ISA}$ `person`, then

- $r_2.Action$ = `create(employee_log, (who:X, age:X.age, salary:X.salary), O)` is a refinement of $r_1.Action$ = `create(person_log, (who:X, age:X.age), O)` under renaming $\epsilon$, and
- $r_2.Action$ = `modify(department.nbr_of_employees, Y, Y.nbr_of_employees-1); delete(employee,X)` is a refinement of $r_1.Action$ = `delete(person,X)` under renaming $\epsilon$. △

Note, that our notion of corresponding action is correct if and only if a notion of behavior refinement is imposed on Chimera operations. Indeed, we can ensure that if an operation *op* is invoked in the action of the inherited trigger, then operation *op* is invoked in the action of the refined one. Conditions on behavioral subtyping in Chimera are presented in [26].

### D. Restrictions on Trigger Overriding

The following subsection summarizes the restrictions on trigger redefinition.

*Definition 23:* (*Static Trigger Refinement*). A trigger $r_2 = (N, c, Ev, C, A)$ is a *static refinement* of a trigger $r_1 = (N', c', Ev', C', A')$, if $\rho$, renaming of variables, exists such that the following conditions are satisfied:

- $Ev' \subseteq Ev$, that is, $Ev$ is a refinement of $Ev'$ according to Definition 18;
- $C \leq_c^\rho C'$, that is, $C$ is a refinement of $C'$ according to Definition 20;
- $A \leq_a^\rho A'$, that is, $A$ is a refinement of $A'$ according to Definition 22. □

The following properties (proved in [28]) hold for redefined triggers.

*Proposition 2:* Given two triggers $r_1$ and $r_2$:

1. we can decide whether $r_2$ is a static refinement of $r_1$;
2. if $r_2$ is a static refinement of $r_1$, $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$, each event on proper instances of $r_2.Class$, that triggers $r_1$ also triggers $r_2$;

3. if $r_2$ is a static refinement of $r_1$, $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$, $r_2$ is a behavioral refinement of $r_1$ according to Definition 17.

We can state the following rule.

*Rule 1:* Given the set of triggers $R$ of a database schema, for each $r_1$, $r_2$ belonging to $R$ if $r_1.Name = r_2.Name$ and $r_2.Class \leq_{ISA} r_1.Class$ then $r_2$ must be a static refinement of $r_1$ according to Definition 23. $\diamond$

Note moreover that, according to Proposition 2, $r_2$ is a behavioral refinement of $r_1$.

Finally, let us mention that whenever the conditions of Definition 23 hold among two triggers, then, under certain conditions, the more general of the two triggers is redundant, as stated by the following definition.

*Definition 24:* (*Redundant Trigger*). A trigger $r_2 = (N, c, Ev, C, A)$ is redundant with respect to a class $c$ if a trigger $r_1 = (N', c', Ev', C', A')$ exists, with $c \leq_{ISA} c'$ and $N \neq N'$, such that:

- $Ev \subseteq Ev'$, that is, $Ev'$ is a refinement of $Ev$ according to Definition 18;
- $C' \leq^{\rho}_{c} C$, that is, $C'$ is a refinement of $C$ according to Definition 20;
- $A' \leq^{\rho}_{a} A$, that is, $A'$ is a refinement of $A$ according to Definition 22. $\Box$

*Example 9:* Consider the classes `person` and `employee`, with `employee` subclass of `person`, and the following trigger $r_1$ on class `person`:

```
Events:    modify(income)
Condition: person(X), occurred(modify(income),X),
           X.income < 0
Action:    rollback
```

Suppose that the following trigger $r_2$ is defined on class `employee`:

```
Events:    modify(income)
Condition: employee(X), occurred(modify(income),X),
           X.income < 100
Action:    rollback
```

Then, trigger $r_1$, inherited in class `employee`, is redundant with respect to this class. $\triangle$

Detecting redundant triggers allows one to save useless rule executions during rule processing, thus restricting the set of rules to be executed to a minimal one realizing the intended effect on the database.

## VII. Trigger Inheritance and Overriding in Object-Relational DBMS

Object-relational DBMS represent one of the most interesting extensions of relational DBMS and many of them support active capabilities. In particular, the forthcoming database standard SQL-3 [9] relies on an object-relational data model and provides active capabilities. In our opinion, the issues discussed in this paper apply to active object-relational data models as well, and our approach can help in clarifying how active capabilities can be integrated in object-relational DBMS. Thus, the contribution of the paper has value beyond the specific data model, Chimera, in the context of which it has been developed.

Though some object-relational data models do not currently support inheritance (e.g. DB2 [32] and Oracle [33]), all of them mention inheritance as one of the most relevant planned extensions to the model. Inheritance, both at the type (ADT) and at the table level, is part of the SQL-3 data model; however, no discussion on how inheritance interacts with triggers is included in the standard documentation. Note that in SQL-3 triggers are not defined in the context of tables. However, an SQL-3 trigger monitors a single event, thus it is implicitly associated with the table to which the monitored event refers.

*Example 10:* By transposing our approach in the SQL-3 context, given a table `employees` and a table `managers`, created as `UNDER employees`, a trigger defined as:

```
CREATE TRIGGER upd_rank
AFTER UPDATE OF rank ON employees
...
```

would react also to updates to the `rank` attribute of tuples of the `managers` table, unless the following redefinition is included, overriding the trigger above:

```
CREATE TRIGGER upd_rank
AFTER UPDATE OF rank ON managers
...
```

$\triangle$

The semantic framework we have developed can be easily extended for specifying the semantics of trigger on object-relational databases. In particular, Chimera immediate triggers corresponds to SQL-3 statement-level after triggers. Dealing with immediate rules implies that the reactive process is started after the execution of any update operation, rather than only after the execution of the `commit` statement. The semantics we have presented in the paper can easily model this execution mode. A semantics modeling the execution of other kinds of SQL-3 triggers, that is, row-level and before triggers, could be defined as well, relying on the same semantics framework.

A (syntactic) difference between SQL-3 (and DB2) triggers and Chimera ones is that in SQL-3 there is no parameter passing between condition and action parts, and that actions contain SQL DML statements, that is, statements of the form `UPDATE-SET-WHERE`, `DELETE-WHERE`, that both specify the action to be performed and select the objects on which the action has to be performed. In Chimera triggers, instead, these two tasks are decoupled in the action and condition components. We remark that, however, this is a syntactic difference, since we can easily establish a correspondence between each SQL DML statement and a (set of) Chimera condition-action pair(s) [8].

*Example 11:* The following SQL-3 trigger reacts to insertions of new employees, whose rank is manager. It sets the salary of all employees in the inserted manager department whose salary excedes the salary of the newly inserted manager, to the salary of that manager.

```
CREATE TRIGGER upd_sal
  AFTER INSERT ON employees
  REFERENCING NEW AS nemp
  FOR EACH STATEMENT
  WHEN nemp.rank = 'manager'
  UPDATE employees
```

```
    SET salary = nemp.salary
    WHERE employees.dept = nemp.dept AND
          employees.salary > nemp.salary
```

It corresponds to the Chimera trigger:

```
Events:    create
Condition: employee(X), employee(Y),
           occurred(create,Y), Y.rank ='manager',
           X.salary > Y.salary
Action:    modify(employee.salary, X, Y.salary)
```

△

Finally, also the proposed conditions, specifying when trigger refinement is allowed, could be applied to object-relational active databases. The Chimera action language actually allows one to specify sequences of SQL DML statements (modulo the correspondences sketched above). Sequences of SQL DML statements are exactly the actions that can be specified in DB2 triggers, and a meaningful subset of the actions that can be specified in SQL-3 triggers. In particular, SQL-3 trigger actions are sequences containing SQL DML statements and SQL control statements (SQL/PSM). While our static conditions do not cover SQL control statements such as loop statements, trigger refinement can be tested by exploiting our techniques as long as trigger actions contain SQL DML statements only. The following example illustrate the discussion.

*Example 12:* Let `employees`, `employee_log` be tables inheriting from tables `persons`, `person_log`, respectively. Consider the following SQL-3 trigger, corresponding to the trigger $r_2$ of Example 2, whose condition has been modified to select all employees whose age exceeds 65:

```
CREATE TRIGGER del_old
  AFTER UPDATE age ON employees
  REFERENCING NEW AS nemp
  FOR EACH STATEMENT
    BEGIN ATOMIC
      UPDATE department
        SET nbr_empl = nbr_empl - 1
        WHERE nemp.age > 65 AND
              nemp.dept = department.name;
      INSERT INTO employee_log
        SELECT name, age, salary
        FROM nemp
        WHERE nemp.age > 65;
      DELETE FROM employees
        WHERE nemp.name = employees.name AND
              nemp.age > 65;
    END;
```

According to the conditions stated in Section VI it is a refinement of the following SQL-3 trigger, corresponding to the trigger $r_1$ of Example 2:

```
CREATE TRIGGER del_old
  AFTER UPDATE age ON persons
  REFERENCING NEW AS npers
  FOR EACH STATEMENT
    BEGIN ATOMIC
      INSERT INTO person_log
        SELECT name, age
        FROM npers
        WHERE npers.age > 100;
      DELETE FROM persons
        WHERE npers.name = persons.name AND
              npers.age > 100;
    END:
```

△

## VIII. Related Work

In this section we compare our work with existing approaches dealing with active rule semantics and trigger overriding in existing systems.

### A. Active Rule Semantics

The growing interest in active database systems has led to the development of formal techniques to analyze the main characteristics of these systems as the rule definition language and the execution model. In the following we present a brief overview of how the problem of defining a semantics for active database systems has been addressed in the literature. We refer the interested reader to [34] for a more detailed overview.

An approach to the definition of a semantics which is the most similar to ours is described in [35] and proposes a denotational semantics for the Starbust system. As in our approach, the semantics is seen as a function which maps a transaction and the current database state into a resulting state. The main limitation of these semantics is that it is tightly bound to the Starbust execution model, devoting little attention to the condition and action language.

Another important work is [36] where a generic framework for formally specifying the semantics of different systems is proposed, relying on the Object-Z formalism [37]. The formalism used is object-oriented but totally abstract. The main drawback of this approach, like the previous one, is that no aspect concerning the active rule language is analyzed.

Another line of thinking is the one which has adapted the semantics of deductive databases to active systems. The main works in this direction are [38], [39], and [40]. In [38], the database version of the *Event Calculus* is exploited. Through the Event Calculus, the set of logical consequences derived by the event history creates a sequence of sets of facts, each of those can be seen as the extensional part (EDB) of a deductive database (DDB). The main contribution of this approach is the definition of specification languages for active rule languages with a logical semantics. In [39], an integration between the classical operational semantics of the deductive databases and the semantics of active databases is described. The approach is based on the syntactical notion of *XY-stratification* and underlines the declarative nature of database updates. Such an approach, however, does not support the analysis of some important features of active rules, concerning the execution model, such as coupling modes.

A simple specification language of the form condition-action is introduced in [41]. The aim of that approach is to develop a rule execution formalism for reasoning about the basic properties of rules. A drawback is the restrictive assumptions characterizing the approach, for example a single coupling mode is supported.

The last approach [42] we mention here is particularly relevant as it gives a semantics for the Chimera active rule language. Such an approach, applicable to a generic active database system, tries to provide a formal model onto which the various aspects of such a system, such as execution of a set of rules, coupling modes, consumption modes and so on, can be mapped. Three steps are distinguished: the first step consists of a translation of the rules into an intermediate syntax to make explicit the characteristics of the rules of a particular system; the second step consists of a translation of the set of rules, obtained in the previous step, in an inner format (*core format*); the third step consists of an execution model, specified through an algorithm, for the rules expressed in the core format. The main drawback of this approach is that rule semantics is not specified directly, rather through the translation of rules in an auxiliary format, and thus it is less intuitive.

This short overview highlights the fact that most proposals formalizing active database system semantics deal with relational systems rather than with object-oriented ones and the few that follow the object-oriented approach do not address issues related to inheritance and overriding.

### B. Overriding in Active Object Systems

Under most proposals [2], [16], [20], triggers are always inherited and can never be overridden nor refined. Such an approach, that we have referred to as *full trigger inheritance*, simply means that events are propagated across the class inheritance hierarchy. 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. The approach proposed by Shyy and Su [43] achieves the same result, but it is complicated by the fact that events are not propagated along the class hierarchy.

Rule overriding is supported in TriGS [4]. A lookup mechanism is used that starts from the class of the object for which the event was signaled, and ends at the class where the method corresponding to the event[21] is defined (in the worst case at the root of the class hierarchy). To recognize rule overriding, a mechanism based on the equality of rule names is used. At implementation level, a transient rule dictionary is used to filter overridden rules. In TriGS, however, this process is simplified since only single inheritance is considered. No restrictions are imposed on rule overriding, thus a rule may also override another rule on completely different events.

In Ode [27] a subclass may contain a different definition for a trigger defined in a superclass. Then, if the trigger is activated on a subclass object, only the most specific trigger applies to it, thus trigger overriding is supported. Note, however, that if the trigger activation is part of the superclass constructor, than both triggers apply to a subclass object, and there is no way to override the trigger. Moreover, also in Ode no controls are performed on trigger redefinition; thus a rule may override another rule on completely different events.

In [16] the possibility is suggested to program rule overriding "by hand" as follows. Suppose that a trigger $r' = (N, c', Ev', C', A')$ is defined and suppose, moreover, that it needs to be overridden by a rule with the same events $Ev'$, but different conditions and actions, $C$ and $A$, in a class $c$, with $c$ subclass of $c'$. Then, in class $c$ a trigger $r$ can be declared such that $r$ has priority over $r'$, and $r.Action$ executes $A$ and deactivates (that is, disables) trigger $r'$. With this approach, only $r$ will be executed, but it is responsibility of the rule programmer to enforce the overriding. This approach, however, does not account for true overloading and overriding, as the new trigger $r$ cannot have the same name as trigger $r'$. Other drawbacks of this approach are that only few systems allow a rule to be disabled in the action of another rule and that most systems require rules to be explicitly re-activated once deactivated; therefore, it is not clear when and by whom rule $r'$ is re-activated.

Finally, trigger overriding has been recently addressed in [44]. Their approach, however, is very different from ours, since they consider rules over multiple classes with parameterized events and address the issues of trigger signature redefinition and trigger (multiple) dispatching. They do not discuss at all trigger refinement.

## IX. Conclusions and Future Work

Active object-oriented databases are being extensively investigated. 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, by formally specifying a semantics and by investigating under which restrictions triggers can be overridden in subclasses. In [28] we discuss how the existing architecture of the Chimera prototype [45] can be modified to support trigger overriding.

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 [19]. Our notion of event containment can be extended to more complex event languages, since it is possible to establish static conditions ensuring that a composite event will occur each time another composite event occurs. Moreover, the influence of multiple inheritance and multiple class direct membership [46] 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.

[21] We recall that in TriGS events are only method calls.

## References


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

[2] C. Beeri and T. Milo, "A Model for Active Object Oriented Database," in *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, G. M. Lohman, A. Sernadas, and R. Camps, Eds., 1991, pp. 337–349.

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

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

[5] G. Leavens and W. Weihl, "Reasoning about Object-Oriented Programs that use Subtypes," in *Proc. Fifth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications joint with Fourth European Conference on Object-Oriented Programming*, 1990, pp. 212–223.

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

[7] G. Guerrini, E. Bertino, and R. Bal, "A Formal Definition of the Chimera Object-Oriented Data Model," *Journal of Intelligent Information Systems*, vol. 11, no. 1, pp. 5–40, 1998.

[8] S. Ceri and P. Fraternali, *Designing Database Applications with Objects and Rules - The IDEA Methodology*, Addison-Wesley, 1997.

[9] N. Mattos, "An Overview of the SQL3 Standard," Database Technology Institute, IBM Santa Teresa Lab., San Jose - California, July 1996.

[10] U. Dayal, A. Buchmann, and S. Chakravarthy, "The HiPAC project," in *Active Database Systems*, S. Ceri and J. Widom, Eds. Morgan-Kaufmann, 1996.

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

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

[13] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmerman, "Rules in an Open System: the REACH Rule System," in *Proc. First International Workshop on Rules in Database Systems*, N. Paton and M. Williams, Eds., 1993, Workshops in Computer Science, pp. 111–126.

[14] N. Gehani and H. Jagadish, "Active Database Facilities in Ode," in *Active Database Systems*, S. Ceri and J. Widom, Eds. Morgan-Kaufmann, 1996.

[15] O. Diaz, N. Paton, and P. Gray, "Rule Management in Object Oriented Databases: A Uniform Approach," in *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, G. M. Lohman, A. Sernadas, and R. Camps, Eds., 1991, pp. 317–326.

[16] 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*, 1994, pp. 132–143.

[17] S. Gatziu and K. Dittrich, "SAMOS: an Active Object-Oriented Database System," *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, vol. 15, no. 4, pp. 23–26, December 1992.

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

[19] R. Meo, G. Psaila, and S. Ceri, "Composite Events in Chimera," in *Proc. Fifth Int'l Conf. on Extending Database Technology*, P. Apers, Ed., 1996, number 1057 in Lecture Notes in Computer Science, pp. 56–76.

[20] C. Medeiros and P. Pfeffer, "Object Integrity Using Rules," in *Proc. Fifth European Conference on Object-Oriented Programming*, P. America, Ed., 1991, number 512 in Lecture Notes in Computer Science, pp. 219–230.

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

[22] S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases*, Springer-Verlag, 1990.

[23] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, 1995.

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

[25] J.H. Gallier, *Logic for Computer Science*, Harper and Row, 1986.

[26] E. Bertino, G. Guerrini, and I. Merlo, "A Set-Oriented Method Definition Language for Object Databases," Tech. Rep. DISI-TR-97-11, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1997, Submitted for publication.

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

[28] E. Bertino, G. Guerrini, and I. Merlo, "Trigger Inheritance and Overriding in an Active Object Database System," Tech. Rep. DISI-TR-97-4, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1997.

[29] M. Buchheit, M. Jeusfeld, W. Nutt, and M. Staudt, "Subsumption Between Queries to Object-Oriented Databases," in *Proc. Fourth Int'l Conf. on Extending Database Technology*, M. Jarke, J. Bubenko, and K. Jeffery, Eds., 1994, number 779 in Lecture Notes in Computer Science, pp. 15–22, Extended version in *Information Systems*, 19(1).

[30] 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*, 1992, pp. 202–211.

[31] E. Bertino, R. Ferrari, and G.Guerrini, "Containment among Chimera Positive Conjunctive Queries," Technical Report, 1997.

[32] D. Chamberlin, *Using the New DB2 - IBM's Object-Relational Database System*, Morgan-Kaufmann, 1996.

[33] Oracle Corporation, *Oracle8 Server SQL Reference, Release 8.0*, 1997.

[34] N. Paton, J. Campin, A.A.A. Fernandes, and M. Howard Williams, "Formal Specification of Active Database Functionality: A Survey," in *Rules in Database Systems, Proc. of the 2nd International Workshop RIDS'95*, T . Sellis, Ed., 1995, number 985 in Lecture Notes in Computer Science, pp. 21–35.

[35] J. Widom, "A Denotational Semantics for the Starbust Production Rule Language," *SIGMOD Record*, vol. 21, no. 3, pp. 4–9, September 1992.

[36] J. Campin, N. Paton, and H. Williams, "Specifying Active Databases Systems in an Object-Oriented Framework," *Software Engineering and Knowledge Engineering*, vol. 7, no. 1, pp. 101–123, 1997.

[37] G. A. Rose, "Object-Z," in *Object Orientation in Z*, S. Stepheney, R. Barden, and D. Cooper, Eds., pp. 59–77. Springer-Verlag, 1992.

[38] R. Kowalsky, "Database Updates in Event Calculus," *Journal of Logic Programming*, vol. 12, pp. 121–146, 1992.

[39] C. Zaniolo, "A Unified Semantics for Active and Deductive Databases," in *Rules in Database Systems*, N. W. Paton and M. H. Williams, Eds. Springer-Verlag, 1994.

[40] A. Fernandes, H. Williams, and N. Paton, "A Logic-Based Integration of Active and Deductive Databases," *New Generation Computing*, vol. 15, no. 2, pp. 205–244, 1997.

[41] Y. Zhou and M. Hsu, "A Theory of Rule Triggering Systems," in *Proc. Second Int'l Conf. on Extending Database Technology*, 1990, pp. 407–421.

[42] P. Fraternali and L. Tanca, "A Structured Approach for the Definition of the Semantics of Active Databases," *ACM Transactions on Database Systems*, vol. 20, no. 4, pp. 414–471, 1995.

[43] Y. Shyy and S. Su, "Refinement Preservation for Rule Selection in Active Object-Oriented Database Systems," in *Proc. Fourth International IEEE Workshop on Research Issues in Data Engineering - Active Database Systems*, J. Widom and S. Chakravarthy, Eds., 1994, pp. 115–123.

[44] N. A. Chaudhry, J. R. Moyne, and E. A. Rundensteiner, "A Formal Model for Rule Inheritance and Overriding in Active Object-Oriented Databases," in *Proc. of the Third Int'l Conf. on Integrated Design and Process Technology, IDPT - Volume*

*2, International Workshop on Issues and Applications of Database Technology (IADT'98)*, M. T. Özsu and A. Dogac and Ö. Ulusoy, Ed., 1998, pp. 128–135.

[45] G. Guerrini and D. Montesi, "Design and Implementation of Chimera Active Rule Language," *Data and Knowledge Engineering*, vol. 24, pp. 39–67, 1997.

[46] E. Bertino and G. Guerrini, "Objects with Multiple Most Specific Classes," in *Proc. Ninth European Conference on Object-Oriented Programming*, W. Olthoff, Ed., 1995, number 952 in Lecture Notes in Computer Science, pp. 102–126.

# Appendix

## I. Update Semantics

In this appendix we sketch the semantics of Chimera updates. We refer the reader to [26] for an extensive presentation.

In giving the semantics two kinds of updates are distinguished: atomic and non-atomic. An update is atomic when it cannot be decomposed into simpler updates. The atomic updates of our language are $create$, $delete$, $modify$, $specialize$, $generalize$, while the non-atomic updates are method invocations. Atomic operations transform the database from one state to another without intermediate states, whereas non-atomic updates require several intermediate states.

The semantics of atomic update statements is presented in Table II. In that definition, besides the ones in Definition 12, the semantics domains are:

- $Term$ = set of the well-formed terms of the language.
- $2^{\mathcal{V}}$ = powerset of $\mathcal{V}$, set of possible values of the language.

Note moreover that the semantic domains $Cond$ and $Update$ denote condition and action parts both of rules and of methods. Actually, trigger conditions slightly differ from method conditions. For example, in trigger conditions event formulas can appear while in method conditions they cannot. Analogously, in rule action parts rollback operations can appear whereas in method action parts they cannot. We consider these constraints as static constraints.

The following semantic functions are used:

$\mathcal{E} : (Term \times State) \rightarrow ((Bind \times State) \rightarrow 2^{\mathcal{V}})$

$\mathcal{CM} : (Cond \times State) \rightarrow ((Bind \times State) \rightarrow Bind)$

$\mathcal{U} : (Update \times State) \rightarrow ((Bind \times State \times Event) \rightarrow (Bind \times State \times Event))$

Function $\mathcal{E}$ models the semantics of Chimera terms. We do not present here the formal definition of such semantics since it is the usual interpretation of terms constructed by the standard arithmetical set, record and list operators of a general programming language. Function $\mathcal{CM}$ corresponds to function $\mathcal{C}$ (Definition 13) for method conditions. Note that the functionality of function $\mathcal{CM}$ is different from the one of $\mathcal{C}$. Both of them take as input a condition, the construct to be evaluated, and a database state which is the initial state. Conditions for methods must be evaluated in the current database state and with respect to the current set of bindings, as in Chimera methods there is parameter passing and no event set is needed, because event formulas may not appear in method conditions. By contrast, in function $\mathcal{C}$ a rule's condition is evaluated with respect to the current database state and the current event set, necessary for *occurred* formula evaluation, and no set of bindings is considered, since rules have no parameters.

In what follows, given a set of bindings $B$ 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 bound in $B$.

We do not elaborate further on atomic update operations in this context. In particular we do not analyze issues related to dynamic errors. We refer the reader to [26] for an extensive discussion.

Method calls are the only non-atomic updates of our language. A method invocation has the form $O.op(t_1, \ldots, t_n)$ where $O$ is the object receiver of the event, $op$ the method name, and $t_1,\ldots,t_n$, the actual parameters, are terms. A method call is implemented by several rules, where each rule has the form: $condition \rightarrow u_1; \ldots; u_n$.

The semantics of condition evaluation is specified through function $\mathcal{CM}$, which takes a condition $C$ and a state $S$ and returns a set of bindings $\{\theta_1, \ldots \theta_n\}$ such that condition $C$ where each variable has been substituted with the corresponding value in $\theta_i$, $i \in [1, n]$, is a ground formula which evaluates to true in state $S$.

In specifying the semantics of method calls, four phases can be distinguished:

1. Selection of method implementation.
2. Generation of the set of bindings in which the rule conditions are evaluated.
3. Evaluation of the method implementation.
4. Generation of the set of bindings to return as result.

Selection of method implementation is the most interesting phase for what concerns method inheritance and overriding. In the definition of the semantics of method calls a fundamental problem is to establish which method has to be invoked for each *oid* bound to the variable, on which the method is invoked. This way of proceeding is in accordance with the one adopted for establishing the rule triggered by an event instance in Definition 8. Our language supports *late binding*, that is, at run-time, for each object the method related to its most specific class is invoked.

We do not consider these phases further in this context, all the details can however be found in [26].

**Elisa Bertino** received the doctor degree in Computer Sciences from the University of Pisa, Italy, in 1980. She is currently professor of database systems in the Department of Computer Science of the University of Milan where she heads the Database Systems Group. Since October 1997, she is also the chair of the Computer Science School of the University of Milano. She has also been on the faculty in the Department of Computer and Information Science of the University of Genova, Italy. Until 1990, she was a researcher for the Italian National Research Council in Pisa, Italy, where she headed the Object-Oriented Systems Group. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, at the Microelectronics and Computer Technology Corporation in Austin, Texas, at George Mason University in Fairfax, Virginia, and at Rutgers University in Newark, New Jersey.

Her main research interests include object-oriented databases, distributed databases, deductive databases, multimedia databases, interoperability of heterogeneous systems, integration of artificial intelligence and database techniques, database security. In those areas,

$\mathcal{U}[\![\ create(\bar{c},\bar{t},\bar{O})]\!]BSE = \langle B', S', E \cup \{\langle create, \bar{c}, OIDNEW\rangle\}\rangle$

$B' = \{\theta_1 \cup \langle \bar{O}/oid_{f(1)}\rangle, \ldots, \theta_m \cup \langle \bar{O}/oid_{f(m)}\rangle\}$, where:

$f : \{1,\ldots,m\} \rightarrow \{1,\ldots,k\}$, $k \leq m$, is a function such that given $V$ the set of variables in term $\bar{t}$, $f(i) = j$ if $\theta_{i_{|V}} = \sigma_j$, where $\{\sigma_1,\ldots,\sigma_k\} = \{\theta_{i_{|V}} \mid i = 1,\ldots,m\}$. Let $\{oid_1,\ldots,oid_k\} = OIDNEW$.

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

$$\pi'(c) = \begin{cases} \pi(c) \cup OIDNEW & \text{if } c = \bar{c} \\ \pi(c) & \text{if } c \neq \bar{c} \end{cases} \qquad \nu'(oid_i) = \begin{cases} \nu(oid_i) & \text{if } oid_i \notin OIDNEW \\ \mathcal{E}[\![\bar{t}]\!]\sigma_i S & \text{if } oid_i \in OIDNEW,\ 1 \leq i \leq k \end{cases}$$

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

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

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

$\mathcal{U}[\![modify(\bar{c}.A,\bar{O},\bar{t})]\!]BSE = \langle B, S', E \cup \{\langle modify, \bar{c}, \bar{O}_B\rangle\}\rangle$

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

$$\pi'(c) = \pi(c) \qquad \nu'(oid.A_k) = \begin{cases} \nu(oid.A_k) & \text{if } oid \notin \bar{O}_B \text{ or if } oid \in \bar{O}_B \text{ and } A_k \neq A \\ v & \text{if } oid \in \bar{O}_B, A_k = A \text{ and if } \theta_i(\bar{O}) = oid \\ & \text{then } \mathcal{E}[\![\bar{t}]\!]\theta_i S = \{v\} \end{cases}$$

$\mathcal{U}[\![specialize(c_1,c_2,\bar{O},\bar{t})]\!]BSE = \langle B, S', E \cup \{\langle specialize(c_2), c_1, \bar{O}_B\rangle\}\rangle$

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

$$\pi'(c) = \begin{cases} \pi(c) & \text{if } c \neq c_1 \text{ and } c \neq c_2 \\ \pi(c) \setminus \bar{O}_B & \text{if } c = c_1 \\ \pi(c) \cup \bar{O}_B & \text{if } c = c_2 \end{cases} \qquad \nu'(oid) = \begin{cases} \nu(oid) & \text{if } oid \notin \bar{O}_B \\ \nu(oid)\bar{\cup}^{(1)}\mathcal{E}[\![\bar{t}]\!]\theta_i S & \text{if } oid \in \bar{O}_B,\ \theta_i(\bar{O}) = oid \end{cases}$$

$\mathcal{U}[\![generalize(c_1,c_2,\bar{O})]\!]BSE = \langle B, S', E \cup \{\langle generalize(c_2), c_1, \bar{O}_B\rangle\}\rangle$

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

$$\pi'(c) = \begin{cases} \pi(c) & \text{if } c \neq c_1 \text{ and } c \neq c_2 \\ \pi(c) \setminus \bar{O}_B & \text{if } c = c_1 \\ \pi(c) \cup \bar{O}_B & \text{if } c = c_2 \end{cases} \qquad \nu'(oid) = \begin{cases} \nu(oid) & \text{if } oid \notin \bar{O}_B \\ \nu(oid)^{(2)}_{|c_2} & \text{if } oid \in \bar{O}_B \text{ and } \theta_i(\bar{O}) = oid \end{cases}$$

Legenda:
(1) The symbol $\bar{\cup}$ denotes the record concatenation.
(2) $\nu(oid)_{|c_2}$ denotes the restriction of $\nu(oid)$ to the attribute of $c_2$.

TABLE II

**Semantics of atomic update operations**

Prof. Bertino has published several papers in all major refereed journals, and in proceedings of international conferences and symposia. She is a co-author of the books "Object-Oriented Database Systems - Concepts and Architectures" 1993 (Addison-Wesley International Publ.), and "Indexing Techniques for Advanced Database Systems" 1997 (Kluwer Academic Publishers). She is member of the advisory board of the IEEE Transactions on Knowledge and Data Engineering and a member of the editorial boards of the following scientific journals: the International Journal of Theory and Practice of Object Systems, the Very Large Database Systems (VLDB) Journal, the Parallel and Distributed Database Journal, the Journal of Computer Security, Data & Knowledge Engineering, the International Journal of Information Technology, the International Journal of Cooperative Information Systems. She has been consultant to several italian companies on data management systems and applications and has given several courses to industries. She has been also involved in several European Projects sponsored by the EEC under the ESPRIT programme.

Elisa Bertino is a senior member of IEEE and a member of ACM and AICA and has been been named a Golden Core Member for her service to the IEEE Computer Society. She has served as Program Committee members of several international conferences, such as ACM SIGMOD and VLDB, as Program Chair of the 1996 European Symposium on Research in Computer Security (ESORICS'96), as General Chair of the 1997 International Workshop on Multimedia Information Systems, and as Program Co-Chair of the 1998 IEEE International Conference on Data Engineering (ICDE).

**Giovanna Guerrini** is an assistant professor at the Department of Computer and Information Sciences of the University of Genova. She received the MS and PhD degrees in Computer Science from the University of Genova, Italy, in 1993 and 1998, respectively. Her research interests include object-oriented, active, deductive and temporal databases, semi-structured data.

**Isabella Merlo** received a MS Degree in Computer Science (with honours) at the University of Genova in 1996. Since November 1996, she is enrolled in a PhD program, under the supervision of Prof. Elisa Bertino, in the Department of Computer and Information Sciences of the University of Genova as a member of the Database and Information System Group. Her current research interests include object-oriented, active and temporal databases, data models for management of semi-structured data.