Triggers in Java-based Databases

Elisa Bertino* — **Giovanna Guerrini**** — **Isabella Merlo***

** Dipartimento di Scienze dell'Informazione*
Università di Milano - Milano, Italy
Via Comelico 39/41
20135 Milano (Italy)
{bertino,merloisa}@dsi.unimi.it

*** Dipartimento di Informatica e Scienze dell'Informazione*
Università di Genova - Genova, Italy
Via Dodecaneso, 35
16146 Genova (Italy)
guerrini@disi.unige.it

ABSTRACT. *Java^{TM} is recently establishing itself as a very successful programming language, and it is more and more widely used also in applications requiring persistence support and database technology. It has not, however, been conceived as a database programming language. One of the main limitations of Java with respect to the mature relational database technology is the lack of integrity constraint support. An important functionality supported by many of the most recent relational and object-relational database systems is represented by triggers, that enhance the database with reactive capabilities and which can be used to support integrity constraints. In this paper, we discuss the problems entailed by the addition of active features to Java-based databases.*

RÉSUMÉ. *Java^{TM} s'est récemment imposé comme langage de programmation, il est de plus en plus utilisé dans des applications nécessitant des bases de données et un support pour la persistance. Toutefois il n'a pas été conçu comme un langage pour la programmation des bases de donnés. Une des principales limitations de Java en rapport avec la technologie des bases de donnés relationnelles est le manque de support pour les contraintes d'intégrité. Une fonctionnalité importante prévue dans la plupart des récents systèmes de base de données à objets ou relationnel est la notion de "déclencheur" qui enrichit la base de données avec des possibilités réactives et qui peuvent servir à assurer les contraintes d'intégrité. Dans cet article nous discutons des problèmes provenant de l'ajout de traits actifs dans les base de données basées sur Java.*

KEYWORDS: *Active databases, triggers, Java.*

MOTS-CLÉS : *Base de données active, déclencheur, Java.*

1. Introduction

Active features are critical for many advanced data management applications. An active database system is a system in which some operations are automatically executed when specified events happen and particular conditions are met. Examples of the use of active capabilities are integrity constraint enforcement, authorization and monitoring. Most recent relational and object-relational DBMSs and the forthcoming standard SQL:1999 [EIS 00b, EIS 00a] provide those capabilities. Moreover, several proposals for adding triggers to object-oriented database systems have been presented [CER 96, PAT 99].

In both the relational and 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. Even though current relational database systems provide stored procedures, and object-relational and object-oriented database systems provide methods, as a means to express behavior of data, specifying the semantics of data through rules has important advantages over coding it into methods (or stored procedures). For instance, the behavior represented by methods must be explicitly invoked by the user or by applications, whereas active rules are autonomously activated. Moreover, 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.

As remarked in [PAT 99], however, whereas relational database vendors have been quick to extend their products with active facilities, object-oriented database vendors have not yet incorporated active facilities into their products. This is also reflected in the fact that the object-oriented database standard ODMG [CAT 99] does not include triggers. In [BER 99] we have proposed an extension of the ODMG standard with triggers.

Java^{TM} [GOS 96] is recently establishing itself as a very successful programming language, and it is more and more widely used also in applications requiring persistence support and database technology. It has not, however, been conceived as a database programming language. Three major limitations of Java from the viewpoint of object-oriented database technology have been discussed in [ALA 98], namely lack of support for persistence, lack of parametric polymorphism, and lack of integrity constraint support. The first limitation is the most obvious. A considerable amount of work has however been devoted to that topic in the last few years [PJW98] and different proposals, such as PJama, JavaSPIN and ObjectStore PSE, have emerged. The second limitation is related to the heavy use of collections in database applications. The topic of extending Java with parametric classes has also been extensively investigated and several approaches have been proposed [OOP98]. The third aspect, by contrast, has received little attention. To the best of our knowledge, the only works in this direction are [ALA 98, COL 00]. In [ALA 98] a declarative assertional language to express pre and post conditions for methods, as well as some forms of integrity constraints, is proposed. In [COL 00] an approach that uses assertions to ensure consistency during the life cycle, starting from the design stage, of persistent applications is presented.

Triggers are a typical database mechanism, well suited to express integrity constraints. Besides allowing one to express the conditions that should not be violated, they allow one to specify the action to be taken upon constraint violation (*repairing* action) to restore a consistent state. Moreover, their use is absolutely not limited to integrity maintenance, since they allow one to specify any generic form of reactive behavior, such as consequential actions.

In this paper, we discuss the problems entailed by the addition of reactive features in object-oriented databases, focusing our attention to Java-based databases. We first briefly discuss trigger support in commercial “traditional" DBMSs and then discuss which issues have to be re-examined in an object-oriented context.

2. Triggers in Object Relational DBMSs

Most commercial DBMSs, such as Oracle, DB2, Informix, Sybase, supports triggers. Though their trigger languages present some differences, they are all quite close to the trigger language of SQL:1999 [SQL99]. In SQL:1999, triggers are expressed by means of event-condition-action (ECA) rules. The event specifies what causes the rule to be triggered, that is, the database operation monitored by the trigger. In the SQL:1999 standard proposal, each trigger reacts to a single event. Considered events are insert, delete, or update to a particular relation. For update events, the attribute (or set of attributes) target of the modification can also be specified. The condition (`WHEN` clause) 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 condition does not hold, nothing else associated with the trigger happens in response to the event. In SQL:1999, the condition is expressed as an arbitrary SQL predicate, potentially involving complex queries. The action is executed when the rule is triggered and its condition is true. The action may then prevent the event from taking place, or it could undo the event (e.g., delete the inserted tuple). The action can be any sequence of database operations, even operations not connected in any way to the triggering event. In particular, the action can include any SQL data manipulation statement, as well as invocations of user-defined functions and procedures.

The SQL:1999 trigger statement gives the user the possibility of specifying several different options. The main features are illustrated below.

– The trigger can be executed before or after the triggering operation.

– The possibility is given of specifying that the trigger be executed: (*i*) once for each modified tuple (*row-level trigger*), or (*ii*) once for all the tuples that are changed in a database operation (*statement-level trigger*).

– The trigger condition and action can refer to both old and new values of tuples that were inserted, deleted or updated by the operation that triggered the rule. In a statement-level trigger, similarly, the set of old tuples and the set of new tuples can be referred as two relations[1].

If an entire table is updated with an SQL update statement, a statement-level trigger would execute only once, while a row-level trigger would execute once for each tuple. In a row-level trigger, the condition is evaluated on each tuple affected by triggering operation, and if it holds, the trigger action is executed on the tuple. By contrast, in a statement-level trigger, the trigger condition is evaluated once on all the tuples affected by the triggering operation, and if it holds, the trigger action is executed in a set-oriented way.

The processing granularity is an orthogonal dimension with respect to the activation time, thus both before and after triggers can be either row-level or statement-level.

Triggers are all executed in the context of the same transaction to which the triggering operation belongs. Moreover, a single trigger activation time is considered. After triggers are indeed all activated (if their monitored event occurred) immediately after the execution of the triggering operation. This kind of triggers are usually referred to as *immediate* triggers. It could be possible to have after triggers whose activation is *deferred* at transaction commit [CER 96]. This is useful for rules that enforce integrity constraints, since a transaction may execute several operations that violate a constraint, but the transaction may restore the constraint before it reaches its commit point.

The interaction among triggers and inheritance have not been investigated in the object-relational context. Actually, some object-relational DBMSs (e.g. DB2 and Oracle) do not currently support inheritance; all of them, however, 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:1999 data model; however, no discussion on how inheritance interacts with triggers is included in the standard documentation. Note that in SQL:1999 triggers are not defined in the context of tables. However, an SQL:1999 trigger monitors a single event, thus it is implicitly associated with the table to which the monitored event refers.

3. Triggers in Java-based Databases

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. Although several approaches have been proposed in the past to extend object database systems with triggers and interesting results have been achieved, there is a lack of uniformity and standardization across those approaches, and most common commercial OODBMSs do not support triggers. An overview of the existing proposals of active object-oriented DBMSs can be found in [BER 00b].

1. Actually, those *transition tables* can also be referred in row-level triggers, for instance to apply aggregations over the whole set of tuples manipulated by the triggering operation.

```
<trig_dcl> :: = trigger <name>
                {before | after } <event> on <class>
                [referencing {old as <variable> | new as <variable> |
                        oldset as <variable> | newset as <variable>}]
                [when <condition>]
                <action>
                [for each {instance | statement}]
```

Figure 1. *Language extension for specifying triggers*

The paradigm shift from the relational model to the object-oriented one requires revising the functionalities as well as the mechanisms by which reactive capabilities are incorporated into the object-oriented data model. There are several factors not present in relational database systems that complicate the extension of object-oriented database systems to include active behavior.

In what follows, we discuss how to adapt the SQL:1999 trigger language to a Java-based context. This choice is mainly motivated by the convergence between object-oriented and object-relational approaches. Moreover, we are not interested in proposing a new trigger language, incorporating a large number of features. Rather, we simply would like to re-examine a trigger language, like the one proposed for SQL:1999, in a pure object-oriented data model, like the one supported by Java.

This shift entails addressing several issues. In the remainder of this section, we first briefly sketch the considered trigger definition language, then we focus on two of those issues that we believe are the most relevant. The first issue is related to the data manipulation primitives, with respect to which triggers are defined. The second issue is related to trigger inheritance and overriding.

3.1. *Trigger Definition Language*

The primitive for defining triggers we consider is presented in Figure 1. Each trigger is identified by a name and is targeted to a class. A trigger targeted to a class c monitors objects of class c.

As in SQL:1999, it is possible to specify whether a trigger must be executed before or after its triggering operation. As in SQL:1999, the possibility is given of referencing in the condition (as well as in the action) the objects affected by the operation that triggered the rule. This is accomplished through *transition variables* declared in the referencing clause of the trigger definition statement. Both the new and the past states of objects affected by the data manipulation statement (triggering event) can be queried. Affected objects can be seen individually (old and new) or jointly as a "temporary" extent (oldset and newset).

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. They can be expressed as OQL conditions (that is, any construct that can appear in an OQL query where clause) if the database is ODMG-compliant, otherwise they can be any side-effect free Java expression returning a boolean value. The when clause of the trigger definition statement is optional. If it is missing, the condition is supposed to be true and the trigger action is executed as soon as the trigger event occurs. The action is executed when the rule is triggered and its condition is true. Possible actions include database operations, that is, the data manipulation statements discussed in Section 3.2, and method invocations. A sequence of actions can be specified, so that the specified actions are sequentially executed, and other Java imperative constructs that can appear in method bodies can be used as well.

The set of events supported, as well as the trigger processing granularity, are strictly related to the approach adopted for data manipulation, and will thus be discussed in the following section.

3.2. *Data Manipulation Language*

Triggers usually react to data manipulation operations (such as `INSERT`, `DELETE` and `UPDATE` in SQL:1999). Data manipulation in Java-based databases is mainly performed through methods defined in the class to which the object to be manipulated belongs. A very limited set of manipulation primitives is predefined: the assignment statement, allowing one to set the value of an attribute of an object to a specified value, and the `new` operator, allowing one to create an instance of a class, even if no constructor method is specified for that class. No explicit deletion operation is provided, since objects are supposed to be removed from the database when no longer referenced, through a garbage collection mechanism. More important, data manipulation in SQL:1999 is set-oriented. In the database context, indeed, the common case is to execute a given update operation on a set of objects rather than on a single object. Data manipulation primitives in SQL have a set-oriented semantics, that is, they work on a set of objects at-a-time. Whereas DMLs support *sets* of instances as logical units of computation, conventional programming languages, such as Java, reason on a single instance (*record*) at-a-time. Note that SQL:1999 supports the two possibilities (instance-oriented and set-oriented computation) for triggers, whereas data manipulation is always set-oriented.

The possibility of extending Java with set-oriented data manipulation primitives can be considered. These primitives are employed for creating and updating objects. Rather than acting on a single object (instance), they work on a set of objects at-a-time, where this set is determined by the objects satisfying a given condition (query). This is exactly the approach of SQL data manipulation statements. Table 1 summarizes the different options for data manipulation. No set-oriented method invocations are considered.

OPERATION	EVENT	INSTANCE-ORIENTED	SET-ORIENTED
object creation	insert	new $c(p_1, \ldots, p_n)$	insert into c $expr$
attribute update	update, update of a	$o.a = expr$	update c set a = $expr$ where F
object deletion	delete	—	—
method invocation	m	$o.m(p_1, \ldots, p_n)$	—

Table 1. *Data manipulation*

If data manipulation is instance-oriented and trigger execution is immediate, as in SQL:1999, trigger execution will obviously be instance-oriented as well. By contrast, if deferred triggers are supported, then both set-oriented and instance-oriented triggers can be specified. In the NAOS system, for instance, trigger execution is instance-oriented for immediate trigger, whereas it is set-oriented for deferred ones. Thus, statement-level triggers make sense only if a set-oriented data manipulation is supported in the language, or if deferred triggers are included. The default processing granularity is instance-oriented.

Another important aspect to consider is that in Java-based databases data manipulation primitives handle in a uniform way persistent and ordinary data. It could be thus quite difficult to express triggers that only apply to persistent data, since the actual storage or deletion from disk (that is, the insertion or deletion of a persistent object) are handled by the underlying system and do not correspond to the execution of a user statement. In our proposal a trigger monitor both the persistent and the non-persistent instances of a class. Note that is in accordance with the *orthogonal persistence* [ATK 95] principle on which most Java-based databases are based.

3.3. *Trigger Inheritance and Overriding*

In adapting the SQL:1999 trigger definition language to an object-oriented context, however, the main issues to be investigated concern trigger inheritance and overriding. Such issues have neither been considered in the context of SQL:1999, nor been satisfactorily addressed by existing proposals of active object-oriented data models. They are however crucial for a proper integration of reactive capabilities with object-oriented modeling primitives.

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[2]. Such an approach, that we refer to as *full trigger inheritance*, simply means that event types are propagated across the class inheritance

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

hierarchy. Consider a trigger r, defined on a class c', monitoring an operation *op*. 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 defined in c having *op* as event. This means that, for example, given a class `Person` and a class `Employee`, extending class `Person`, a trigger monitoring the update of the `age` attribute of class `Person` would react also to updates to the `age` attribute of objects instances of class `Employee`. 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. In the remainder of this section we discuss some more subtle issues concerned with inheritance of triggers.

3.3.1. *Method Selection in Inherited Triggers.*

One of the problems arising in defining 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 opinion, the first approach should be adopted, 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. Rule-specific method selection can however, be useful in some cases. It can simply be realized in Java by inserting an explicit upward cast before the method invocation.

Example 1 *Consider the following class and trigger definitions.*

```
class Person{
  string name;
  int age;
  void display() {System.out.println(name);}
}
```

```
class Employee extends Person{
  int emp_number;
  void display() {System.out.println(emp_number);}
}

trigger test
  after update of age on Person
  when age > 120
    new.display();
```

If the statement `e.age = 150` *is executed, where variable* `e` *denotes an object instance of class* `Employee`*:*

- *under rule-specific method selection, the string* `e.name` *is printed;*
- *under object-specific method selection, the integer* `e.emp_number` *is printed.*

Under object-specific method selection, rule-specific behavior can be obtained by substituting the trigger action with the following casted method invocation: `((Person)new).display()`*.*

3.3.2. *Trigger Overriding.*

Another important issue to be investigated concerns rule overriding. Full trigger inheritance is not, indeed, always appropriate. There are situations in which trigger overriding is required. The lack of trigger overriding capabilities does not allow triggers to manage in different ways the proper and non-proper instances of a class. 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.

An active object language should thus provide the possibility of redefining triggers in subclasses, exactly as methods can be redefined. 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. Late binding is supported also for triggers, thus at execution time for each object affected by the execution of the specified trigger the most specific implementation will be chosen.

Example 2 *Referring to the classes and triggers of Example 1, if the following trigger definition is added, trigger* `test` *of class* `Person` *will not be executed on instances of class* `Employee`*. Thus, upon the execution of the statement* `e.age = 150`*, where variable* `e` *denotes an object instance of class* `Employee`*, no information will be displayed.*

```
trigger test
  after update of age on Employee
  when age > 70
    new.age = 70;
```

Note that trigger overriding is supported in very few active object systems. Rule overriding is supported in TriGS and, even with some limitations, in Ode[3]. In those systems no restrictions are imposed on rule overriding, thus a rule may also override another rule on completely different events. Some other active object systems (such as NAOS) suggest to program rule overriding "by hand". This requires a trigger language in which priorities among triggers can be explicitly defined (this is not the case in SQL:1999). Under this approach, to refine the behavior of a trigger in a subclass one can define in the subclass a trigger on the same event which performs the refined action, such that the trigger in the superclass has priority over the trigger in the subclass. Thus, upon occurrence of the common triggering event on an object belonging to the subclass, both triggers are activated, but, since the trigger defined in the superclass is executed first, the action in the trigger defined in the subclass "prevails". However, it is not always possible to refine the behavior of a trigger in a subclass by adding a new trigger, even by specifying that the subclass trigger has lower priority than (thus, is executed after) the superclass one [BER 00b].

3.3.3. *Preserving Trigger Semantics in Subclasses.*

Since trigger behavior is often quite complex and impredictable, because of mutual interactions among triggers, it is important to provide some mechanism for preserving trigger semantics in subclasses. This means, for example, that *conservative* trigger redefinitions are specified. We impose that in overriding a trigger only the condition and the action components can be redefined, that is, the monitored events must be the same. Moreover, to ensure that the trigger in the subclass is executed at least each time the trigger in the superclass would be executed, and that what would be executed by the trigger in the superclass is also executed by the refined trigger, the `super` mechanism provided by Java can be exploited. We allow the boolean expression corresponding to a trigger condition to contain the expression `super.condition()`, and the trigger action to contain the expression `super.action()`, to refer to the superclass trigger, currently being redefined, condition and action, respectively.

Example 3 *Referring to the classes and triggers of Example 1, the following trigger definition make use of the* `super` *mechanism to conservatively redefine the trigger in class* `Employee`.

```
trigger test
  after update of age on Employee
  when age > 70
    super.action();
    new.age = 70;
```

3. In Ode triggers must be explicitly activated on objects. 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.

4. Conclusions

Reactive capabilities are a very important component of current commercial database technology. We believe that though Java provide a notion of event and an exception handling mechanism, reactive capabilities similar to the ones that can be achieved through the language we propose cannot be provided relying on those mechanisms. An analysis of the differences between triggers and exceptions can be found in [BER 00a].

Thus, their support in Java-based databases is crucial. In this paper, we have discussed the main issues entailed by the introduction of these capabilities. The discussion applies both to persistent extensions of Java (like PJama) and to ODMG-compliant OODBMSs with a Java binding. The introduction of triggers in Java-based databases obviously entails addressing relevant issues also from the architectural point of view. Different architectural alternatives ranges from those based on a preprocessor to extensions of the Java Virtual Machine. The treatment of architectural issues is however beyond the scope of this work.

5. References

[ALA 98] ALAGIĆ S., SOLORZANO J., GITCHELL D., "Orthogonal to the Java Imperative", JUL E., Ed., *Proc. Twelfth European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, 1998, p. 212-233.

[ATK 95] ATKINSONS M., MORRISON R., "Orthogonally Persistent Object Systems", *VLDB Journal*, vol. 4, 1995, p. 319-401.

[BER 99] BERTINO E., GUERRINI G., MERLO I., "Extending the ODMG Object Model with Triggers", report , 1999, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova.

[BER 00a] BERTINO E., GUERRINI G., MERLO I., "Do Triggers Have Anything To Do With Exceptions?", ECOOP Workshop on Exception Handling in Object Oriented Systems. Available at http://www.cs.ncl.ac.uk/ people/alexander.romanovsky/home.formal/ehooslist.html, 2000, Cannes (France).

[BER 00b] BERTINO E., GUERRINI G., MERLO I., "Trigger Inheritance and Overriding in Active Object Database Systems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, num. 4, 2000, p. 588-608.

[CAT 99] CATTEL R., BARRY D., BERLER M., EASTMAN J., JORDAN D., RUSSEL C., SCHADOW O., STANIENDA T., VELEZ F., *The Object Database Standard: ODMG 3.0*, Morgan-Kaufmann, 1999.

[CER 96] CERI S., WIDOM J., *Active Database Systems - Triggers and Rules for Advanced Database Processing*, Morgan-Kaufmann, 1996.

[COL 00] COLLET P., VIGNOLA G., "Towards a Consistent Viewpoint on Consistency for Persistent Applications", *Proc. of the ECOOP 2000 Symposium on Objects and Databases*, Lecture Notes in Computer Science, 2000, To Appear.

[EIS 00a] EISENBERG A., MELTON J., "SQL Standardization: The Next Steps", *ACM SIGMOD Record*, vol. 29, num. 1, 2000, p. 63-67.

[EIS 00b] EISENBERG A., MELTON J., "SQL:1999, formerly known as SQL3", *ACM SIGMOD Record*, vol. 28, num. 1, 2000, p. 131-138.

[GOS 96] GOSLING J., JOY B., STEELE G., *The Java^{TM} Language Specification*, Addison-Wesley, 1996.

[OOP98] *Proc. of the Thirteenth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA98)*, Vancouver, Canada, October 1998, ACM SIGPLAN Notices 33(10).

[PAT 99] PATON N., *Active Rules in Database Systems*, Springer-Verlag, 1999.

[PJW98] *Proc. of the Third International Workshop on Persistence and Java^{TM} (PJW3)*, Tiburon, California, September 1998, Sun Microsystems Laboratories.

[SQL99] "ISO/IEC 9075-2:1999 Information technology - Database language - SQL - Part 2: Foundation (SQL/Foundation)", 1999.

***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, 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** is an assistant professor at the Department of Computer Sciences of the University of Milano. She 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 and Dr. Giovanna Guerrini, 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.*