Extending the Object-Oriented Database Management System VODAK with Active Capabilities

Diplomarbeit
zur Erlangung des akademischen Grades eines Magisters der Sozial- und Wirtschaftswissenschaften

ausgeführt am
Institut für Wirtschaftsinformatik
Forschungsbereich Data & Knowledge Engineering
Johannes Kepler Universität Linz

unter der Anleitung von

Univ. Ass. Dipl.-Ing. Brigitte Röck

o. Univ.-Prof. Dipl.-Ing. Dr. Michael Schrefl

und am
Institut für Integrierte Publikations- und Informationssysteme
Gesellschaft für Mathematik und Datenverarbeitung (GMD-IPSI)
Darmstadt, Deutschland[1]

unter der Anleitung von

Dr. Wolfgang Klas

eingereicht durch

Werner Obermair

Matr.-Nr.: 176/9056393
Studienadresse: Weingartshofstraße 34, A-4020 Linz

Linz, Jänner 1995

[1] Der Forschungsaufenthalt am GMD-IPSI wurde durch ein Stipendium für kurzfristige wissenschaftliche Aufenthalte im Ausland des Bundesministeriums für Wissenschaft und Forschung ermöglicht.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die Diplomarbeit mit dem Titel "Extending the Object-Oriented Database Management System VODAK with Active Capabilities" selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Werner Obermair

Linz, Jänner 1995

Kurzfassung

Die Entwicklung objektorientierter Datenbanken trägt wesentlich dazu bei, die semantische Lücke zwischen der Repräsentation von Objekten in der realen Welt und der Repräsentation derselben Objekte in Datenbanken zu schließen. Objektorientierte Datenbanken sind passiv, d.h., sie beinhalten Objekte, die nicht von sich aus auf bestimmte Situationen reagieren. Diese Schwachstelle objektorientierter Datenbanken wird vor allem in Situationen deutlich, in denen der Status einer Datenbank überwacht werden muß. Im Rahmen dieser Diplomarbeit wird das am GMD-IPSI als Forschungs-Prototyp entwickelte verteilte objektorientierte Datenbanksystem VODAK um aktive Fähigkeiten erweitert. ECA-Regeln werden eingesetzt, um Überwachungssituationen zu modellieren. Neben konventionellen Anwendungsklassen und Metaklassen beinhaltet eine ActiveVODAK-Datenbank Regeln. Diese Regeln definieren Situationen, die überwacht, und Aktionen, die gegebenenfalls ausgeführt werden sollen. Regeln und Regel-Komponenten (Ereignisse, Bedingungen und Aktionen) werden als Objekte modelliert; sie stellen eine homogene Erweiterung des VODAK Datenmodells dar. Neben Fragen der Integration von Regeln in das VODAK Datenmodell werden verschiedene Aspekte der Regelausführung betrachtet und Fragen der Architektur diskutiert.

Abstract

The development of object-oriented database systems means a great advance in narrowing the semantic gap between the real-world representation and the database representation of objects. But, object-oriented database systems are passive, they contain objects that do not act autonomously on predefined situations. This weakness of passive object-oriented database systems becomes apparent in situations, where database states have to be monitored. Within this thesis, the distributed object-oriented database system VODAK, developed as a research prototype at GMD-IPSI, is extended with active capabilities using the concept of ECA rules. Besides conventional application classes and metaclasses, an ActiveVODAK database contains rules which define situations to be observed and actions to be initiated. Rules and rule components (events, conditions, and actions) are treated as first-class objects; they represent a homogenous extension of the VODAK data model. Besides questions of integrating rules into the VODAK data model, diverse aspects of rule execution are considered and architectural issues are discussed.

Acknowledgements

The present work has been carried out as a master thesis on basis of a cooperation between the Department for Data & Knowledge Engineering (DKE) at the University of Linz and the Integrated Publication and Information Systems Institute (IPSI) at GMD, Darmstadt. Among other topics, active databases represent a research focus at DKE. Research at GMD-IPSI concentrates on the development of a new generation publication environment, within which the distributed object-oriented database management system VODAK plays the role of an integrating platform. Knowledge, experience, and requirements stemming from both institutes proved to be a solid basis for my work.

A series of persons have considerably contributed to my work: Despite of their packed time schedules, Prof. Michael Schrefl and Brigitte Röck of DKE and Wolfgang Klas of GMD-IPSI granted excellent support. Their valuable comments have been essential to the outcome of my work. Achim Kraiss and Thomas Tesch of GMD-IPSI adapted the VODAK transaction model to meet the requirements imposed by my active extensions. Hannelore Eisner of GMD-IPSI assisted me in becoming acquainted with the VODAK system and in installing and configuring VODAK in Linz.

Contents

Chapter 1

Introduction

The *distributed object-oriented database management system VODAK* is developed as a research prototype at GMD-IPSI, the German "Gesellschaft für Mathematik und Datenverarbeitung"-Integrated Publication and Information Systems Institute. Research at GMD-IPSI focuses on a new generation of publication and information systems dealing with multimedia documents, which combine text, graphics, voice, audio, image, animation, and video data. All tasks involved in publishing—informaton retrieval, editing, and publication—are assumed to be supported in the distributed system, within which VODAK plays the role of an integrating platform.

This new generation publication environment represents an advanced database application exceeding the capabilities of conventional (especially relational) database systems: Complex multimedia objects have to be managed, time-crucial operations have to be performed on these objects, and long-duration design transactions must be supported. Therefore, VODAK realizes concepts of object-oriented database systems, which capture the complex structure and behaviour of advanced (especially multimedia) applications.

VODAK has been designed as a *passive* system: The objects contained in VODAK databases perform only operations initiated on them. Since objects do not act autonomously, they are not *active*. Active database systems provide the ability to monitor the database state and to react to predefined situations without explicit user action or program request.

As a passive system, VODAK encounters problems in modelling monitoring situations. Concerning the publication environment developed at GMD-IPSI, two examples for monitoring situations are:

1. *Multimedia synchronization:* Multimedia documents combine data of different kinds. For example, a multimedia document may contain a film, which combines a series of advertisements (images), the moving pictures (movie) and the sound (audio). When a "reader" of the document starts the film, different situations have to be monitored. E.g., on termination of the advertisement presentation the moving pictures and the sound have to be started simultaneously.

2. *Workflow management:* Information retrieval, editing, and publication represent cooperative tasks (workflows) that have to be managed by the integrating platform of a publication environment. Time schedules, cost budgets, and working plans define situations that have to be monitored.

Databases with active capabilities provide the ability to monitor such situations and to act autonomously on them. The aim of this thesis is to extend VODAK with active capabilities, which increase the system's potential to cope with monitoring situations. ActiveVODAK should be designed to provide basic active capabilities, it should not be tailored to the needs of specific fields of application.

The remainder of the thesis is organized as follows:

Chapter 2 gives a short introduction to object-oriented database management systems (OODBMS). It motivates and summarizes the concepts of object-orientation. Chapter 3 motivates the development of active object-oriented database systems and presents their concepts. In particular, event-condition-action rules are introduced for modelling situations of the universe of discourse that require active capabilities. Chapter 4 describes VODAK. It presents the data model and the transaction model of VODAK and classifies the characteristics of VODAK by referring to the requirements for OODBMSs. Further, chapter 4 describes the architecture of VODAK. Chapter 5 represents the thesis's central part—it describes the extension of VODAK with active capabilities: It introduces the fundamental concepts adopted to integrate rules into the VODAK data model, it describes the knowledge model of ActiveVODAK, it discusses diverse aspects of rule execution, and it explains the necessary extensions of a VODAK process to provide active capabilities. Based on these conceptual descriptions, chapter 6 presents the implementation of ActiveVODAK. Chapter 7 concludes with an outlook on possible future extensions of ActiveVODAK and on the role of ActiveVODAK in an integrated publication environment.

Throughout the thesis, a bookseller's repertoire of tasks serves as illustration. The bookseller's tasks considered are:

1. *Inventory control:* The bookseller has to keep track of the number of a book's copies in stock. If the amount in stock falls below a given threshold, a reorder process has to be initiated.

2. *Keeping track of orders placed:* If the number of a book's copies in stock falls below the threshold, the bookseller orders the book with his supplier. Two weeks after placing an order, the bookseller checks whether the book has arrived. If not, the supplier is urged on delivery.

   The bookseller has established a special agreement with his suppliers: Every time an order has been shipped, the bookseller informs his supplier about the successful shipment of the order.

3. *Reminding policy:* If a bookseller's customer is in delay for payment, every three weeks the bookseller sends a reminder to his customer until payment is made.

4. *Price reduction:* On 1 January 1995, the bookseller reduces the price of books published more than five years ago by 20 per cent.

5. *Book announcement:* If a new book becomes part of the bookseller's assortment, the book is set on a list of new books which is sent to customers periodically.

Chapter 5 will show, how these tasks can be supported in ActiveVODAK. The bookseller's basic data model and the necessary definitions of rules, events, conditions, and actions are summarized in appendix A.

Chapter 2

Object-Oriented Database Management Systems

This chapter gives a short introduction to object-oriented database management systems (OODBMS). It motivates and summarizes the concepts of object-orientation.

2.1 Motivation for OODBMSs

This section motivates the concepts of OODBMSs. It enumerates shortcomings of conventional database systems in coping with the requirements of advanced database applications.

Advanced database applications include computer-aided design, engineering, software engineering, and manufacturing (CAD, CAE, CASE, and CAM) systems; knowledge-based systems; multimedia systems which deal with images, voice, and textual documents; and programming language systems. A detailed description of advanced database applications can be found in [Cat91, pages 15–45].

Kim [Kim90] summarizes shortcomings of conventional, especially relational, database technology:

- Conventional data models, especially the relational model, are too simple for modelling complex nested entities, such as design and engineering objects, and complex documents. Additionally, conventional database systems do not provide mechanisms, such as configuration management, to represent and manage such entities.
- Conventional database systems support only a limited set of atomic data types, such as integer or string; they do not support general data types found in programming languages. In particular, they do not even allow the storage and retrieval of long unstructured data (long data) such as images, audio, and textual documents.

- Conventional data models do not provide frequently used semantic concepts, such as generalization and aggregation relationships; this means that application programmers must explicitly represent and manage these relationships in their programs.

- The performance of conventional database systems, primarily relational ones, is unacceptable for various types of compute-intensive applications, such as simulations in computer-aided design and programming language environments.

- Database languages are very different from programming languages, in both data model and data structures. This impedance-mismatch is more severe if high-level object-oriented programming languages are combined with conventional database languages.

- The model of transactions supported in conventional database systems is inappropriate for long-duration transactions necessary in interactive and/or cooperative design environments. Conventional database systems do not support facilities for representing and managing the temporal dimension in databases, including the notion of time and versions of objects and schemata.

For a more detailed discussion of shortcomings and disadvantages of relational database systems see Unland/Schlageter [US90] and Heuer [Heu92, pages 83–143]

The new-generation publication environment developed at GMD-IPSI [NT92] is an exponent of an advanced database application exceeding conventional database systems' capabilities. Complex multimedia objects have to be managed, time-crucial operations have to be performed on these objects, and long-duration design transactions must be supported in an appropriate way. To cope with these requirements, an advanced database system has to be employed. Concepts of object-oriented database systems proved to be a solid basis for advanced applications in general and multimedia applications in special [AK92, Mey91]. Object-oriented database systems capture the complex structure and behaviour of advanced applications.

2.2 Concepts of OODBMSs

This section represents the most prominent concepts of object-oriented database systems. Since there has been considerable debate about the definition of what constitutes an OODBMSs, this section characterizes OODBMSs with reference to [ABD+89].

Atkinson et. al. [ABD+89] classify the features of OODBMSs into three groups: *mandatory* features, which a system must satisfy in order to be referred to as an object-oriented database system; *optional* features, which can be added to enhance the system; and *open* features, where the designer can make a number of choices.

The mandatory features ensure that an OODBMS is both a complete database management system and an object-oriented system. They are:

- *Complex objects:* Complex objects are built from simpler ones by applying constructors to them. The minimal set of constructors that the system should support are set, list, and tuple. The object constructors must be orthogonal—any constructor should be applicable to any object (e.g., also lists of tuples should be supported).

- *Object identity:* In a model with object identity, an object has an existence that is independent of its value. Objects are not identified by their values but by unique object identifiers (OIDs). Thus, two notions of equivalence can be distinguished: Two objects are value-equal, if they have the same value; they are identity-equal, if they represent the same object, i.e., they have the same OID.

- *Encapsulation:* The idea of encapsulation stems from abstract data types. An object has an interface and an implementation part. The interface part, the object's only visible part, specifies the set of operations that can be performed on the object. The implementation part consists of the data part, representing the state of an object, and the procedural part, implementing each operation.

- *Types or classes:* The differences between the terms *type* and *class* are very subtile; moreover they are not always used uniformly. A *type*, in an object-oriented system, summarizes the common features of a set of objects with the same characteristics. It corresponds to the notion of an abstract data type. Also a *class* specifies the common features of a set of objects, but it contains two more aspects: A class acts as an object factory and as an object warehouse. By performing an operation *new* on the class, a new representative of the class is created and attached to the class's extension, i.e., the set of objects that are instances of the class.

  Notions of type and class that are different from the definitions above are summarized in [BM91, page 37]: In some systems, for example Gemstone [Heu92, page 476], the class has only the meaning of specification. A collection constructor groups objects of the same class. In other systems, for example Orion [Kim90, pages 16–18], the class has a meaning of both specification and extension. Finally, other systems, for example O2 [Ban92, pages 59–96], provide the notions of type and class. In O2, instances of types are values and, therefore, do not have OIDs, while instances of classes are objects. Another notion of type and class can be found in subsection 4.1.1, when the VODAK data model is introduced.

  Regardless of the concept chosen in a concrete approach, a system is referred to as object-oriented, if it supports some form of data structuring, be it classes or types.

- *Class or type hierarchies:* Class or type hierarchies form lines of inheritance among classes or types[1]. Inheritance is a powerful modelling tool, because it gives a concise and precise description of the world, and it helps in factoring out shared specifications and implementations of an application domain. Inheritance lets a class, called a subclass, be defined starting from the definition of another class, called the superclass. The subclass inherits the attributes and the operations defined within its superclass, unless it overrides them.

[1] No distinction is made between classes and types in this context.

- *Overriding, overloading* and *late binding:* A redefinition of an inherited operation's implementation is called *overriding*. The redefinition results in operations that share a common name in the type or class hierarchy. An operation's name used in either different type hierarchies or along a type hierarchy with different parameters is *overloaded*. In order to provide polymorphism, a system cannot bind operation names to programs at compile time, but at run-time (*late binding*), since an operation might be overriden in a subtype. Polymorphism means that instances of subtypes can be freely used instead of supertype instances (e.g., in assignment, in formal/actual parameter substitution), but not vice versa.

- *Computational completeness:* This criterion simply means that an application developer should be able to express any computable function, using the data manipulation language of the database system.

- *Extensibility:* The set of a database system's predefined types must be extensible in the following sense: there is a means to define new types and there is no distinction in usage between system-defined and user-defined types. Of course, there might be differences between the system-internal treatment of built-in datatypes and user-defined datatypes, but system internals must be transparent to the user.

- *Persistence:* Persistence means that data survive the execution of an application program. Persistence should be orthogonal (i.e., independent of its type, each object is allowed to become persistent) and implicit (i.e., it should not be necessary to explicitly move data from transient to persistent memory).

- *Secondary storage management:* Performance features such as index management, data clustering, data buffering, access path selection, and query optimization must be supported invisibly to the user.

- *Concurrency:* At least serializability of operations must be supported, less strict alternatives may be offered.

- *Recovery:* In case of hardware or software failures, the system must be able to recover, i.e., bring itself back to a coherent state of the data.

- *Ad hoc query facility:* A query facility allows the user to ask *simple* queries to the database *simply*. An ad hoc query facility is not necessarily a query language, a graphical browser could be sufficient to fulfill this functionality. A query facility should be high level, efficient, and application independent.

Optional features, which clearly improve a system, but which are not mandatory for an object-oriented database system, are targeted primarily to serve advanced applications. *Multiple inheritance*, a high degree of *type checking*, the possibility of *distribution*, the efficient support of long-duration *design transactions*, and the administration of *versions* improve a system's capabilities for supporting advanced applications [ABD+89].

Besides the mandatory features, which widely represent a consensus among the scientific community, and the optional features, enriching the system, a lot of design choices have to be made. Open choices comprise the *programming paradigm* adopted, the *representation system* (i.e., the set of atomic types and the set of constructors), the type system, and the degree of uniformity (e.g.: Is a type an object? Is a method an object?) [ABD+89].

Chapter 3

Active OODBMSs

Object-oriented database systems, as described in chapter 1, are passive. The objects contained in the database perform only operations initiated on them. As objects do not act autonomously, they are not active. Active database systems provide the ability to monitor the database state and to react to predefined situations without explicit user action or program request. Situations to be observed and actions to be initiated are defined in the database schema. Active object-oriented database systems combine the capabilities of object-oriented database systems and active database systems. This chapter motivates the development of active object-oriented database systems and presents their concepts. In particular, event-condition-action rules are introduced for modelling situations of the universe of discourse that require active capabilities.

3.1 Motivation for Active OODBMSs

This section motivates the extension of an OODBMS to an *active* OODBMS. This is done by demonstrating the shortcomings of pure object-oriented database systems to model object behaviour. Alternatives to active capabilities are worked out and evaluated.

Look at the bookseller's task of inventory control (cf. chapter 1): If a book's number of copies on stock falls below a threshold, a reorder process has to be initiated. In the absence of capabilities to monitor situations of this kind, there are two ad-hoc solutions for this problem [SR94]:

- *Immediate checking:* Each program decreasing the number of copies on stock has to check, whether the quantity on stock has fallen below the threshold and, consequently, an order has to be placed. The shortcomings of this approach are obvious: The entire complexity of the application environment is hidden in programs. The principle of modularity is violated, as the same situation might be monitored in different programs. A description of disadvantages of the immediate-checking strategy can be found in [SR94].

- *Periodic polling:* A special program is periodically performed on the database, and a reorder is placed for every book for which the amount on stock has fallen below the threshold. Also periodic polling is an unsatisfactory solution: To determine the appropriate polling period is difficult, if the frequency of withdrawals changes over time and differs for various items. Polling too seldom, may cause delayed reorders; polling too often, may burden the system with unnecessary queries.

Databases with active capabilities provide the ability to monitor the database state and to act autonomously on predefined situations. The modelling concept to support this ability is the concept of *Rules*. Situations, as described in the above example, are of the following form: On an event ocurrence (the amount on stock falls) under a certain condition (the amount on stock is below threshold) an action (reorder) has to be executed. Since a rule is a combination of *event*, *condition*, and *action*, it is called an *event-condition-action rule* (ECA rule). The term *ECA rule* goes back to [Day88, DBB+88]. In active database systems, ECA rules are part of the database schema. Thus, they are defined independently of application programs. A comprehensive comparison of the three alternatives—immediate checking, periodic polling, and active capabilities—can be found in [SR94].

Active capabilities have a long tradition in computer science. An overview on early proposals for their use is given in [Day88, SR94]. Also recent advances in commercial database systems are partially basing on simple forms of triggers[1]. E.g., triggers are used to propagate modifications among replications of relations in a distributed ORACLE environment [Ora93].

Object-oriented database systems aim at representing real-world semantics (cf. section 2.1). Active capabilities complement to this aim increasing the system's potential to capture behavioural aspects of objects in the universe of discourse.

3.2 Concepts of Active OODBMSs

This section establishs a common understanding of terms used in active object-oriented database systems and motivates the decision to adopt the ECA rule formalism. It explains the components and characteristics of ECA rules. This section abstracts from architectural issues; design decisions made within different prototypes are not evaluated. Prototypes of active object-oriented database management systems, which provide a basis for the explanations in this section, are HiPAC [Day88, MD89, DBB+88, CN90, HLM88], SAMOS [GGD91, Gat92], Sentinel [CAM93, CM93], Ode [GJS92a, GJ91, GJS92b], ADAM [DPG91], TriGS [KRR+94, KRR+93], and NAOS [CCS94].

[1] A trigger is the description of an action, together with its triggering condition. Triggers and other kinds of active capabilities will be explained in section 3.2.1

3.2.1 ECA Rules

An intuitive definition of ECA rules has already been given in subsection 3.1. At this point, alternative approaches to cope with the problem of the bookseller—to keep track of the books on stock—have been introduced and compared with the ECA rule approach. Proposals for active capabilities in database systems, mainly used to achieve consistency within the database, are summarized in [Kot89]. The concept of triggers and the event-trigger mechanism are briefly presented here:

- *Trigger:* Kotz [Kot89, page 42] defines a trigger as a "predefined sequence of operations, consisting of a body and a condition". The body is executed, if the condition, representing a predicate over the database, is satisfied. A trigger could be defined as follows:

```
DEFINE TRIGGER t1
ON DELETION OF person p
    (DELETE projectMembership
     WHERE memberName = p.name)
```

- *Event-Trigger Mechanism (ETM):* Within the ETM approach, Kotz [Kot89] defines a trigger as a pair of *event* and *action*. The action is executed every time the associated event raises. By defining an optional *condition*, it is possible to restrict action execution to particular *event contexts*. An event context is made up of parameters of an event occurrence. Thus, a *condition* does not represent a predicate over the database, but over the actual parameters of an event. An example for the event-trigger mechanism looks as follows:

```
EVENT endOfDesignPhase (designObject)

ACTION verification (verificationObject)
       = <verification of verificationObject>

TRIGGER t1 = ON endOfDesignPhase
         DO verification (designObject);
```

  Using the statement *"RAISE endOfDesignPhase (chapter 1)"* an occurrence of the event *endOfDesignPhase* is created with the actual parameter *"chapter 1"*. A condition (missing in the example) could, for example, check the type of the actual parameter.

ECA rules are recommended by Dayal [Day88] to generalize special-purpose mechanisms such as triggers or event-triggers. The ECA rule concept integrates the notions of event, condition, and action. Moreover, the expressive power of ECA rules allows to specify additional features, such as the execution semantics of ECA rules. Since any number of action firings may be caused by a single situation, a decision has to be made in which order these actions are executed. Further, the need may arise to postpone the evaluation of a condition or the execution of an action to a later point in time. Together with the components of rules, these aspects of ECA rules are described in the following subsections. The explanations are accompanied by examples based on the bookseller's repertoire of tasks.

3.2.2 Events

Literature characterizes events as follows:

- An event ocurrence represents a point in time, at which some transformation in the database or in the database's environment takes place [GJS92a, GJS92b, SR94].
- An event is an atomic occurrence [CM93]. It happens completely or not at all.
- Formal arguments may be defined for an event; these are bound to actual arguments when the event raises (parameterization) [Day88, MD89]. The arguments bound may be accessed from within condition evaluation and/or action execution.

Before enumerating types of events supported in different active object-oriented database systems, the term *event* has to be looked at more closely. Gatziu et. al. [GGD91, Gat92] distinguish between *event class* and *event instance*. An event class corresponds to the definition of an event, whereas each event occurrence creates a new event instance[2]. The difference between event class and event instance becomes paramount when events are parameterized. An event class specifies name and type of parameters to be bound at instantiation time. An event instance contains actual values for these parameters, thus representing the specific context of a particular instantiation.

Example: The execution of a predefined transaction—named *turnoverPreview*—is modelled as an event. Therefore, an event class *turnoverPreviewExecution* is defined, which specifies name and type of the parameters bound:

```
DEFINE EVENT turnoverPreviewExecution
   AS BEGIN OF TRANSACTION turnoverPreview(month: INTEGER);
```

The argument *month*, the month for which the turnover preview has to be made, is bound to an actual value any time the transaction *turnoverPreview* is started. Besides *month*, there are implicitly defined arguments—the name of the user executing the transaction, and the identifier of the transaction.

[2]Within the scope of this thesis, a distinction between *event class* and *event instance* is made only, if the meaning is not clear from context.

Literature on active object-oriented databases differentiates between two categories of events—primitive and complex events [SR94, GJS92b, GGD91, CM93, MD89, CAM93]. Primitive events correspond to elementary occurrencies in the database system or in its environment; complex events are used to handle more complex requirements of the application domain and represent compositions of primitive and complex events.

3.2.2.1 Primitive Events

All activities in a database which examine or modify database objects are candidates for primitive events. Also external signals, e.g., points in time, or signals originating from external system components, potentially act as primitive events. Database events, time events, and abstract events are subsumed under primitve events.

Database Events. Database events represent primitive events which are generated and detected automatically by the database system. Database events comprise method events, value events, and transaction events.

- *Method events:* In an object-oriented system the only way to change the state of an object is to perform a method on it (cf. section 2.2). A method event is raised, whenever the method for which the method event is defined is executed. Different granularities of method event definition can be distinguished:

  1. **Method events at object level:**
     - *Specific method:* The execution of a specific method over a specific instance raises a method event.
     - *Arbitrary method:* The execution of an arbitrary method over a specific instance raises a method event.
  2. **Method events at class level:**
     - *Arbitrary class:* The execution of a specific method over instances of an arbitrary class raises a method event.
     - *Specific method:* The execution of a specific method over an instance of a specific class raises a method event.
     - *Arbitrary method:* The execution of an arbitrary method over an instance of a specifc class raises a method event.

  There are two potentially suited points in time to raise a method event—before or after method execution. The application developer must be free to fix the appropriate point in time according to application needs. Method events can be utilized to solve the bookseller's inventory control problem:

  Example: The bookseller has to keep track of the number of a book's copies in stock. If the amount on stock falls below a threshold, a reorder process has to be initiated.

Every time, a book is sold, a method named *Sell* is performed on an instance of class *Book*. To detect sales, it is necessary to define a method event for a specific method (*Sell*) at class level (*Book*).

- *Value events:* Besides method events, a series of active object-oriented database prototypes support value events [GJS92b, GGD91]. Like method events, value events can be raised *before* or *after* execution of a value operation. Access primitives such as *READ*, *WRITE*, or *UPDATE*, which represent potential events, abstract from the semantics of the methods calling them. Since object-oriented data models aim at capturing real-world semantics to a high extent, value operations play a role of subordinate importance in object-oriented systems.

- *Transaction events:* If a transaction is executed, there are four possible points in time to raise an event: at *start*, *before commit*, *after commit*, or at *abort*. *Before commit* denotes the point in time, when execution of the transaction code is completed, but the transaction has not yet attempted to commit [GJS92b]. An example for a transaction event has been given in section 3.2.2. The example illustrates that, in order to support transaction events, it must be possible to identify transactions uniquely.

Time Events Time events define specific points in time. Unlike database events, time events cannot be detected autonomously by the database system. Generally, an additional system component, using the system clock, acts as a pulse generator, sending time pulses to the database system. Time events are classified into three groups: absolute, relative, and periodic time events [Gat92].

- *Absolute time events:* An absolute time event represents an absolute point in time. There is only one instance per event class as an absolute point in time occurs exactly once.

  Example: On 1 January 1995, the bookseller intends to reduce the price of books published more than five years ago by 20 per cent.

  1 January 1995 is modelled as an absolute time event. In the systems considered, the finest granularity in specifying an absolute point in time is one second.

- *Relative time events:* A relative time event specifies a point in time relative to the occurrence of another event (of arbitrary type).

  Example: If the number of a book's copies in stock falls below the threshold, the bookseller orders the respective book at his supplier. Two weeks after a book has been ordered, the bookseller checks whether the book has arrived. If not, the supplier is urged on delivery.

  The order for a book represents the basis for the definition of the relative time event that takes place two weeks after the order has been placed.

- *Periodic time events:* Periodic time events are raised periodically after the start event (of arbitrary type) occurred.

  Example: If a bookseller's customer is in delay for payment, every three weeks the bookseller will send a reminder to his customer until payment is made.

  "To be in delay for payment" represents the start event for the periodic sending of reminders. There are two alternatives to realize this start event. It can either be a relative time event (e.g., two weeks after the book has been sold) or an abstract event raised explicitly by the bookseller. An abstract event allows to raise the start event depending on the individual customer's solvency. Abstract events are described in the following paragraph.

Abstract Events Up to now, several types of events have been introduced, all together conveying semantics known by the database management system. Schema designers and application developers, however, may demand for events with specific semantics. Such events are referred to as *abstract* or *explicit* events [CM93, GGD91]. Abstract events are not detected by the database management system, but users, applications, or external system components have to notify the system about their occurrence by issuing an explicit *raise event* operation. Depending on its origin, an abstract event can be internal or external:

- *Internal events* are raised by applications or users. An example for an internal abstract event has already been given in the paragraph describing periodic time events. If the "To-be-in-delay-for-payment" situation is modelled by means of an abstract event, the program or user is responsible to raise the start event for periodic reminders.
- *External events* originate from system external components, such as a printer device, a process running in the background, or an independent database system, e.g., a background process that dumped a database's contents to streamer signals completion of the job.

3.2.2.2 Complex Events

The expressive power of primitive events is insufficient for monitoring complex situations. For this reason, a series of active object-oriented database systems offer higher-level event specification mechanisms, providing a set of *event operators*. Event operators allow the specification of a complex event E_c as a combination of arbitrary primitive and complex events, referred to as *basic events* of E_c. Event hierarchies represent *directed acyclic graphs*. The following operators are supported in the diverse literature [CM93, Gat92, GD94, GGD91, GJS92a, GJS92b]:

- *Conjunction:* A conjunction of two or more basic events represents a complex event that occurs, when all basic events have occurred, irrespective of their chronological order.

  Example: Besides other alternatives, a new publication is made part of the bookseller's assortment, if the new book has been requested by a customer, and its publication has been announced by the publishers. If the new publication matches with the current assortment (regarding price, topic etc.), a specimen copy is ordered.

  Independent of the chronological order of the two basic events (customer's request, announcement of the publishers) the complex event, *interest for a new book*, raises.

- *Sequence:* A sequence of two or more basic events represents a complex event that occurs, when all basic events have occurred, respecting a specified chronological order.

  Example: The bookseller has established a special agreement with his suppliers: Every time an order has been shipped, the bookseller informs the supplier about successful execution.

  The complex event, *order has arrived*, is signalled, if the two basic events, *order placement* and *order shipment*, have been signalled in the proper chronological order.

- *Negation:* A negation, consisting of a basic event and a monitoring interval, represents a complex event that occurs, if the basic event has not occurred during the monitoring interval. Monitoring intervals are introduced in subsection 3.2.2.3.

  Example: A negation has already been introduced when describing relative time events. If an order is not shipped within two weeks after placement, the supplier is urged on delivery.

  The monitoring interval starts with order placement and ends two weeks later. If no *order shipped* event occurs within this interval, the negation event raises, i.e., the supplier is urged on delivery.

- Other event operators explained in the diverse literature are, e.g., *disjunction*, *star*, *history*, etc. A detailed description of these operators can be found in [SR94].

3.2.2.3 Monitoring Intervals

Monitoring intervals restrict event monitoring to one or more period(s). Two events of arbitrary type—a *start event* and an *end event*—delimit a monitoring interval. Monitoring intervals are associated to monitored events. An occurrence of a monitored event is registered only, if the monitoring interval is *on*, i.e., the start event but no corresponding end event has occurred. Since the delimiting events may be instantiated more than once (e.g. method events, periodic time events), an associated monitoring interval may be *on* for several times.

One application of monitoring intervals has already been anticipated in the paragraph describing the *negation operator* (cf. subsection 3.2.2.2). A method event (order placement) constitutes the monitoring interval's start event; a relative time event (two weeks after order placement) terminates the interval. Apart from negations, a monitoring interval can be associated with any event type, then representing a *monitored event* (cf. subsection 3.2.2.4).

3.2.2.4 Events with Associated Monitoring Intervals

An event with an associated monitoring interval (also referred to as *monitored event*) is signalled, when the basic event is signalled while the monitoring interval is on (cf. subsection 3.2.2.3).

Example: The example illustrating periodic time events already introduced a monitored event. Reminders are sent periodically until payment is made. "To be in delay for payment" and "payment is made" represent the delimiting events of the monitoring interval within which reminders are sent periodically.

3.2.3 Conditions and Actions

After an event has been signalled, an active system determines the rules that have to be fired. Rule firing comprises condition evaluation and—if the condition is satisified—action execution. In the diverse literature, different kinds of conditions and actions can be found [SR94]:

A *condition* represents a predicate over the state of the object for which the event occurred, the parameters bound at the time the event occurred, and any database object reachable from the former objects and from database entry points[3]. A condition can return a boolean value (*simple condition*) or a set of objects satisfying the condition predicate (*result condition*). In the latter case, the condition is satisfied, if the result set is not empty. Conditions are parameterized and may bind additional parameters available at action execution.

If the condition is satisfied, the rule's *action* is executed. An action can refer to any executable method of the application environment or to an arbitrary set of them. Furthermore, within action execution it is possible to abort the current transaction and also to raise abstract events explicitly by sending a *raise*-message. Actions are parameterized, too. The database context reachable at action execution corresponds to the context of condition evaluation extended with additional parameters bound during condition evaluation.

[3]Depending on the data model employed, database entry points may be user-named collections of objects or, if the data model's notion of class has a meaning of extension, class names (cf. section 2.2).

3.2.4 Additional Features of ECA Rules

Requirements to active object-oriented database management systems are more diversified than those to traditional systems. ECA rules have to cope with the needs of a wide variety of applications. *Coupling modes* and *priorities on rules* increase the ECA rule concept's flexibility.

3.2.4.1 Coupling Modes

Coupling modes specify the temporal and transactional relationship between event occurrence on the one hand, and condition evaluation/action execution on the other [GGD91][4]. For each ECA rule two coupling modes have to be defined: the coupling between event occurrence and condition evaluation (*event-condition coupling*) and the coupling between event occurrence and action execution (*event-action coupling*).

For applications which have to meet strict time requirements it is necessary to evaluate the condition and to execute the action *immediately* after event occurrence. In this case, condition evaluation and action execution are part of the triggering transaction; the execution of the triggering transaction is suspended until rule execution terminates. Since rule execution may cause cascading rule firings, suspension of the original transaction may last for a long time.

Other applications allow postponed reactions. Condition evaluation and/or action execution are *deferred* to a later point in time, but are still part of the triggering transaction. If the triggering transaction reaches a *ready-to-commit status*, condition evaluation and/or action execution are performed.

A higher degree of parallelism can be achieved by decoupling condition evaluation and/or action execution from the triggering transaction. In this case, condition evaluation and/or action execution are processed in separate transactions, totally *independent* or *commit dependent*[5] on the triggering transactions.

3.2.4.2 Multiple Rule Firings

A single event occurrence may fire an arbitrary number of rules simultaneously. Unless a mechanism is established to determine the execution order of simultaneously fired rules, the execution order is unpredictable. If rules should not be executed in parallel independent transactions, conflict resolution has to be employed. Two major approaches for conflict resolution employing priorities are surveyed in [SR94]:

[4] Another alternative—not adopted in this thesis—would have been to relate event occurrence to condition evaluation and condition evaluation to action execution.

[5] If transaction t_2 is commit dependent on transaction t_1, t_2 is not allowed to commit until t_1 has committed. If t_1 aborts, t_2 has to abort, too.

- *Total ordering:* A unique system priority is associated to each rule at rule definition time. During runtime, users can override priorities in a way such that the ordering of the whole set of rules remains total.
- *Partial ordering:* The user can specify priorities between pairs of rules. Rules for which no priority is defined are executed by the database system in non-deterministic order.

Chapter 4

VODAK

This chapter describes the *distributed object-oriented database management system VODAK*. VODAK is being developed as an OODBMS prototype at GMD–IPSI, the German "Gesellschaft für Mathematik und Datenverarbeitung"–Integrated Publication and Information Systems Institute. Research at GMD–IPSI focuses on a new generation of publication and information systems dealing with multimedia documents, which combine text, graphics, voice, audio, image, animation, and video data. All tasks involved in publishing—information retrieval, editing, and publication—are assumed to be supported in the distributed system, within which VODAK plays the role of an integrating platform [NT92].

Section 4.1 presents the data model and the transaction model of VODAK. It classifies the characteristics of VODAK by referring to the requirements for OODBMSs (cf. section 2.2). Section 4.2 describes the architecture of VODAK, it explains the different modules constituting a VODAK process.

4.1 Modelling Concepts of VODAK

Current research on VODAK deals with scientific themes such as extensible object-oriented data modelling, transaction management for cooperative applications, query processing and optimization, and advanced application domains (e.g., multimedia document databases or biomolecular databases). VODAK serves as the platform for testing and evaluating scientific concepts. Realizations of these concepts which have become part of the VODAK system are presented in this section. The VODAK data model and the VODAK transaction model are discussed in detail. The section is concluded by considering VODAK as an OODBMS.

4.1.1 Data Model

The general requirement for an object-oriented data model is to capture structure and behaviour of an application domain. Object-oriented data models engage different notions of classes to represent real-world objects (cf. section 2.2). In some systems (e.g., C++ [Str87]) a class specifies structure and behaviour of a set of objects—the class's instances. If the system is a metaclass system (e.g., SMALLTALK [GR83]), classes are treated as objects, too. In this case, structure and behaviour of classes are determined by means of classes, too; i.e., classes are instances of metaclasses.

VODAK employes metaclasses which do not only determine structure and behaviour of their instances (which represent classes or other metaclasses) but also structure and behaviour of the instances of their instances. Additionally, VODAK supports the notion of object types. Object type definitions serve as templates for specifying the different types associated with classes [KNA+93]. In the sequel, the definition of object types, classes, and metaclasses is explained in more detail. A more comprehensive discussion on history and development of the VODAK data model can be found in [DKT88, KNS88, KNS89, Kla90].

4.1.1.1 Object Types, Classes, and Metaclasses

Within VODAK, metaclasses, classes, and terminal instances are treated as objects. Objects in VML[1] obey the principle of object identity. Object identity is realized by object identifiers which are attached to objects. Each object has its own distinct object identifier, which is different from that of any other object and which does not change throughout its lifetime. Object identity is the basis for sharing and updating objects.

Object types play a central role in the VODAK data model. Object types are templates defining structure and behaviour of objects. An object type definition consists of a public and a private part, thereby separating the interface from the implementation. The interface definition of an object type embraces a set of property (i.e. attribute) definitions and a set of method definitions. The exclusive way objects communicate with each other is calling methods that are part of the interface definition. The implementation of these methods as well as additional local (hidden) properties and method implementations are defined in the private part. An object type can be defined as a subtype of one or several other object types. A subtype inherits all the properties and methods defined by its supertypes. It is not possible to override definitions inherited from supertypes. To avoid ambiguities, the sets of properties and methods defined in multiple supertypes must be disjunct.

Example: Besides publications (books and journals), the bookseller's assortment also comprises maps and gifts. Publications, maps, and gifts represent stock items. An object type *StockItem* may look as follows:

[1] VML = **V**odak **M**odel **L**anguage

```
OBJECTTYPE StockItem;
   INTERFACE
       PROPERTIES
           price: REAL;
           supplier: STRING;
           threshold: INT;
       METHODS
           Sell(noOfItems: INT);
           Order(noOfItems: INT);
           RecordItemArrival(noOfItems: INT);
           Available(no: INT):BOOL;

   IMPLEMENTATION
       PROPERTIES
           itemsOnStock: INT;
           itemsOrdered: INT;
       METHODS
           Sell(noOfItems: INT);
           { ... };

           ReturnToSupplier(noOfItems: INT);
           { ... };
           ...
END;
```

An object type *Publication_InstType* derived from *StockItem* inherits all properties and methods defined for its supertype. Neither the definition of properties and methods, nor the implementation of methods must be changed in the subtype.

```
OBJECTTYPE Publication_InstType SUBTYPEOF StockItem;
    INTERFACE
        PROPERTIES
            ISBN: STRING;
            title: STRING;
            publishers: STRING;

    IMPLEMENTATION
END;
```

Each publication, irrespective of its actual kind (book or journal), is described by the three properties *ISBN* (International Standard Book Number), *title* (the title of the book or journal), and *publishers* (the book's or journal's publishing company). The implementation part is empty, since *Publication_InstType* does not define additional methods and privat properties.

Objects represent real-world entities. Every object has a structure and a behaviour. Objects which have identical structure and behaviour are organized into *classes*. In VML, every object is an *instance* of exactly one class. VML distinguishes between *application classes* and *metaclasses*:

1. *Application classes:* Application classes organize and classify the objects dealt with by applications (e.g., publications, books, journals, publishers). A definition of the application class *Publication* may look as follows:

   ```
   CLASS Publication METACLASS Generalization
       OWNTYPE  Publication_OwnType
       INSTTYPE Publication_InstType
   END;
   ```

   The INSTTYPE clause of the class definition determines structure and behaviour of the instances of class *Publication*. The optional OWNTYPE clause specifies structure and behaviour of the class object (i.e., *Publication*) itself; for example, the object type *Publication_OwnType* may define an instance creation method.

2. *Metaclasses:* Metaclasses are defined to organize application classes and to tailor the database model to the specific needs of an application domain (cf. subsection 4.1.1.2). Every application class is associated to exactly one metaclass. Metaclasses are used to describe the common structure and behaviour of their instances (other metaclasses or application classes) and of the instances of their instances. Besides the INSTTYPE/OWNTYPE clauses the definition of a metaclass additionally comprises an INSTINSTTYPE clause defining structure and behaviour of the instances of its instances. The definition of metaclass *Generalization*, which is explained in subsection 4.1.1.2 and illustrated in Figure 4.2, looks as follows:

   ```
   CLASS Generalization METACLASS Metaclass
       INSTTYPE     Gen_InstType
       INSTINSTTYPE Gen_InstInstType
   END;
   ```

Figure 4.1 shows the general scheme of determining an object's structure and behaviour through its class and the class's metaclass (cf. [KNA+93]). Summarizing, the interface defined for any object (be it a metaclass object, a class object, or a terminal instance) consists of

- the methods and properties specified through the *own-type* (and with its supertypes) of the object (exception: there is no own-type definition for terminal instances),
- the methods and properties specified through the *instance-type* (and with its supertypes) of the object's class, and
- the methods and properties specified through the *instance-instance-type* (and with its supertypes) of the metaclass of the object's class.

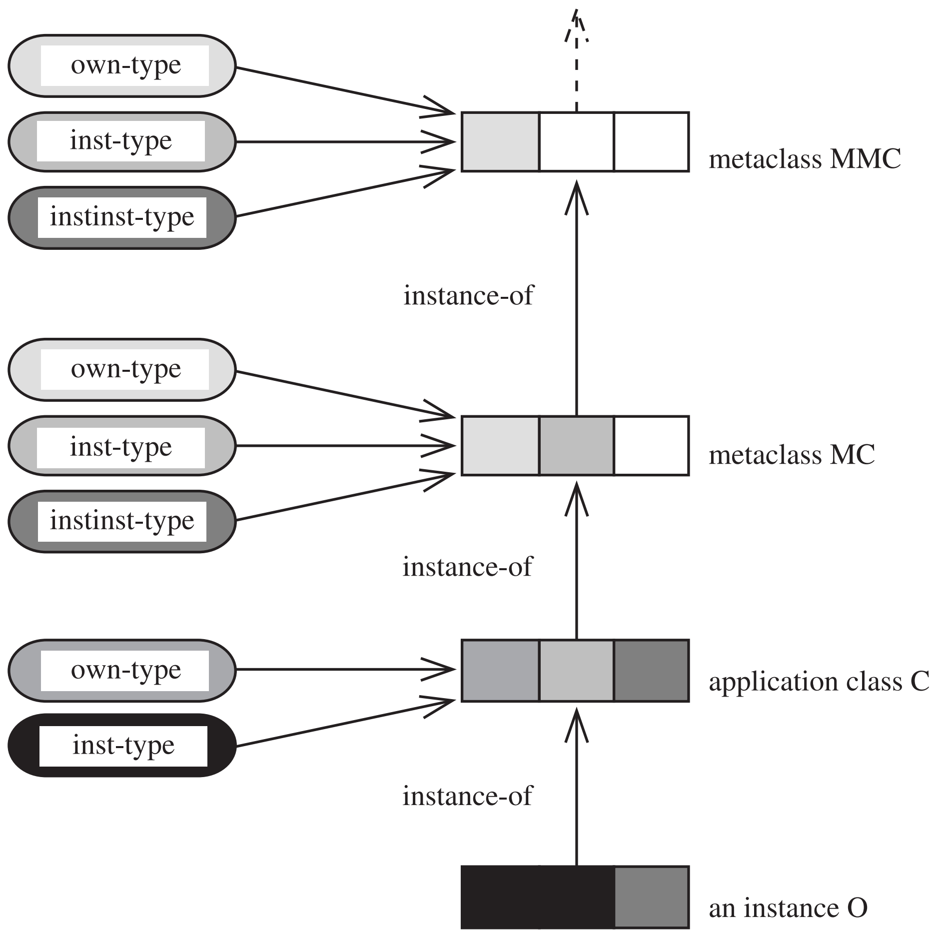


Figure4.1: The general scheme of an object's structure and behaviour

4.1.1.2 Semantic Relationships

Modelling means to define an appropriate representation of real-world objects and of the relationships between them. In the real world, the same object may appear in *different contexts*. In each of these contexts the object may show different structures and behaviours. The general mechanism to handle this problem in the VML data model is to represent the real-world object by instances of different classes, where one of the classes represents an abstraction for the real-world object in general (*generalization*), and every other class represents an abstraction for the real-world object in one of its specific contexts (*specializations*).

Example: If a customer of the bookseller searches for a publication of which the customer only knows the title, the specific context, whether the publication is a book or a journal, is irrelevant, i.e., the bookseller cannot restrict his search to books or journals. On the other side, the inventory control policy for books and journals will be different; a distinction between books and journals is necessary. Therefore, it seems appropriate to refer to a book once as a publication and once as what it actually is—a book. The class *Publication* abstracts from the individual properties of books and journals. The classes *Book* and *Journal* contain specializations of the instances collected in class *Publication*.

Such a generalization/specialization relationship represents a *semantic relationship*. Semantic relationships reflect real-world semantics. Referring to the above example, the semantics of the relationship can be described as follows: For each instance of class *Publication* there exists a specialized instance of class *Book* or of class *Journal*[2]. On the other side, for each instance of the classes *Book* and *Journal* there exists one generalized instance of class *Publication*. If a method defined for instances of class *Publication* is invoked for an instance O_{spec} of class *Book* or *Journal* (for which it is not defined) it is delegated to O_{gen}, the corresponding instance of class *Publication* representing the generalization of O_{spec}.

In VML, metaclasses are engaged to tailor the modelling capabilities to the specific needs of an application domain, i.e., to realize semantic relationships. Metaclasses are used to control dependencies between objects and to coordinate the behaviour of the instances according to the semantics of the relationship. A semantic relationship is realized by means of a specific *inheritance strategy*; a method call that fails for a specific object is delegated to the object with which the original object is semantically related. The VML mechanism for realizing inheritance strategies is the *NOMETHOD clause* which implements a method realizing the inheritance strategy. The *NOMETHOD clause* is executed whenever a method invocation fails, e.g., because the method called is not defined for the receiver. A *NOMETHOD clause* usually is specified with the instance-instance-type of a metaclass, thus defining a method inheritance strategy for terminal instances.

Figure 4.2 illustrates the VML schema realizing the semantic generalization/specialization relationship between publications, books, and journals. Metaclass

[2]As a publication is either a book or a publication, but never both, the relationship is disjunct.

Generalization defines a property *spec* for the instances of its instances; *spec* represents a reference to the specialized instance (of class *Book* or *Journal*) that is semantically related with the generalized instance of class *Publication*. The property *specCls*, which *Generalization* defines for its instances, represents a set of references to specialized classes. Metaclass *Specialization* defines a property for its instances (*genCls*) and for the instances of its instances (*gen*) which refer back to the respective generalized counterparts. The *NOMETHOD*-implementation, which *Specialization* defines for the instances of its instances, realizes the propagation of an unknown method (together with its arguments) to the generalized instance. The symbols used to represent the VML schema are explained in Table 4.1. The notation introduced here is adhered to throughout the thesis.

4.1.2 Transaction Model

A high number of both, conventional (especially relational) and object-oriented database systems, employ conventional concurrency control protocols that operate on pages or on the storage atoms (i.e., flat records) onto which the components of complex objects are mapped. The only access modes that are distinguished for concurrency control are *read access* and *write access*.

Weikum ([Wei88, pages 72 – 74]) summarizes the shortcomings of the classical serializability theory: The term *serializability* is of pure syntactic nature. A *read-write schedule* abstracts from transaction semantics. The only information available on a transaction's step is its type (read or write) and the database object that is accessed. A scheduler which only knows the syntax of a transaction cannot realize any other correctness criterion but serializablility. Conflict and commutativity definitions entirely base on primitive operations—read and write. Serializability theory refers to all possible operations to be positioned on the same level of a database system. Different levels of abstraction are not supported.

In an object-oriented database system, which provides the capabilities for modelling, storing, and manipulating *encapsulated complex objects*, concurrency control should generally operate on objects rather than pages or storage atoms, thus making the additional data modelling power available to the transaction management. The semantics of operations (methods) on encapsulated objects should be exploited to enhance concurrency (cf. [MRW+93]).

To cope with these requirements, VODAK treats the dynamic method invocation hierarchy of a transaction as an *open nested transaction* (cf. [VG93, page 254]). Open nested transactions make the changes of a subtransaction visible at the end of the subtransaction, not at the end of the whole transaction. But the changes must not be visible to every other transaction. To avoid inconsistent use of the results of committed subtransactions, only those subtransactions which commute with the committed one are allowed to use the results (cf. [MRW+93, page 237]). The two principles utilized by open nested transactions are summarized in [MRK+94, page 174]:

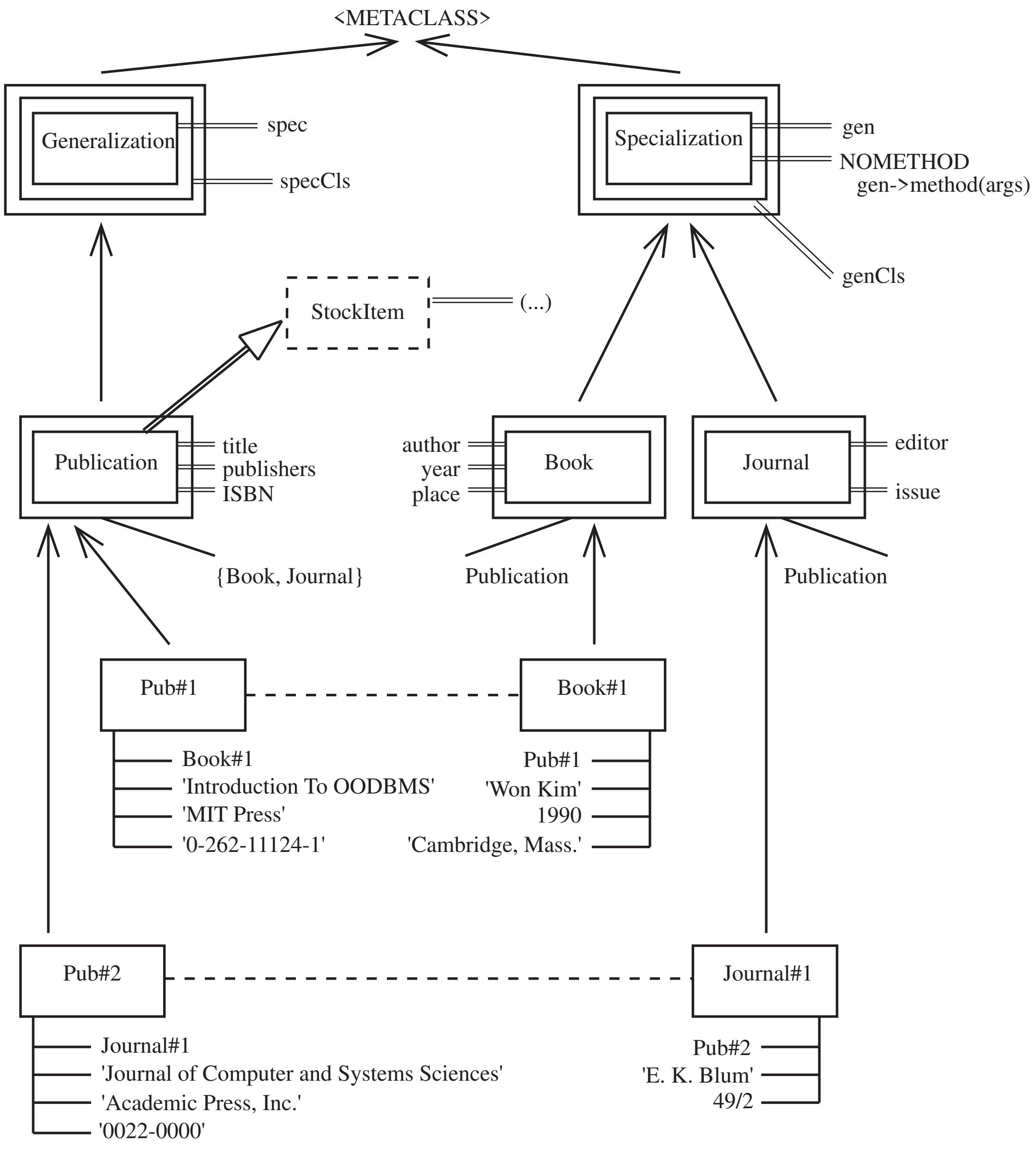


Figure4.2: Generalization/specialization relationship

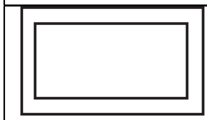
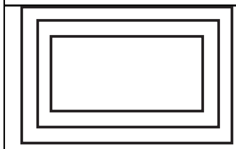
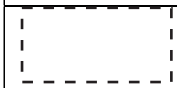
	A single rectangle symbolizes a *terminal instance*.
	A double rectangle symbolizes a *class*. Definitions attached to the inner rectangle represent the class's instance-type, those of the outer rectangle stand for the class's own-type. Properties of the class object are attached to the outer rectangle.
	A triple rectangle symbolizes a *metaclass*. The definitions attached to the innermost rectangle represent the metaclass's instance-instance-type, properties and methods connected with the middle recangle stand for the metaclass's instance-type, and definitions attached to the outermost rectangle represent the metaclass's own-type. Properties of the metaclass object are connected with the outermost rectangle.
	A dashed rectangle symbolizes an *object type* which is not directly associated with any class.
	A double arrow symbolizes a *subtype-of relationship* between two object types.
	A double line symbolizes a *method* or *property definition* of an object type.
	A plain line symbolizes an *actual property value* of a metaclass object, a class object, or a terminal instance.
	A dashed line symbolizes a *semantic relationship* between terminal instances.
	A plain arrow symbolizes an *instance-of relationship* between two objects.

Table4.1: Symbols used to represent VML schemata

1. *Commutativity is semantically defined.* In conventional concurrency control, only read operations among the operations accessing an object do not conflict, i.e., they commute. A semantic definition of commutativity also allows update operations of an object to be defined as commuting[3]. Compatibility matrices for an order-entry example can be found in [MRW+93].

2. *The levels of abstraction are considered.* Operations on a higher level of abstraction are implemented by a sequence of operations on a lower level. Low-level conflicts need not be considered at higher levels of abstraction.

   Example: It is assumed that the increment of a counter at one level is implemented as a read and write operation of a disk page on the next lower level. An uncontrolled interleaving of the page operations of different transactions may cause parallelism anomalies as lost update and inconsistent read. Hence, an increment must be executed in isolation. If the property of isolated execution is guaranteed, the execution order of conflicting page operations on the lower level does not matter on the higher level because the increment operations commute. Low-level conflicts can be "forgotten" at higher levels of abstraction.

Open nested transactions also require a new concept for *recovery* (cf. [MRK+94, pages 190–200]). Since the enhanced concurrency control allows updates on a single object to occur in parallel, transactions are no longer aborted by simply restoring before-values. Instead, operations are executed which compensate all previous changes semantically. An operation m that is executed during normal transaction execution is called *primary action*. An operation m^{-1} which compensates the changes of the corresponding primary action m is called *inverse action*.

To support open nested transactions, two extensions of a conventional VODAK object type definition are necessary (cf. [KNA+93, pages 126–136]):

- For each updating subtransaction an *inverse action* has to be defined. From within the inverse action, the arguments of the primary action and values stored explicitly before and after the primary action's execution are accessible.
- For each object type a *commutativity table* has to be defined. Commuting methods are specified as pairs of method calls together with a boolean expression indicating under which condition the two subtransactions commute.

4.1.3 VODAK as an OODBMS

The conceptual description of VODAK concludes by considering VODAK as an object-oriented database management system. Thereby, the mandatory features of an OODBMS

[3]Two operations m_1 and m_2 on object o commute, if the results of the two sequences $(m_1;m_2)$ and $(m_2;m_1)$ are the same and there does not exist any sequence of operations (s) such that $((m_1;m_2);s)$ and $((m_2;m_1);s)$ have different results. If two operations do not commute they conflict [MRK+94].

worked out in section 2.2 provide a basis for classification. The aim of this subsection is not to qualify or disqualify VODAK as an OODBMS, but to summarize and look at VODAK characteristics more closely. In the following, the mandatory features are itemized and their realization in VODAK is described:

- *Complex objects:* The type universe of VML is divided into *data types* and *object types*. Data types comprise primitive data types (Boolean, Integer, Real, String, Bytestring, and Audio) and structured data types. Constructors for records, sets, lists, arrays, and dictionaries are available. Object types are employed to define structure and behaviour of complex objects that are stored in the database (cf. subsection 4.1.1.1). Orthogonality of the VML type concept is secured: Data type constructors can be applied to arbitrary types (data types and object types); also properties of object types can be of any type.

- *Object identity:* Each object, be it a metaclass, class, or terminal instance, has an object identifier associated.

- *Encapsulation:* In VML, a strict separation between the interface and the implementation part of an object is made (cf. subsection 4.1.1.1).

- *Types or classes:* In VML, an object type constitutes an intensional description of an object's structure and behaviour, whereas a class has a meaning of extension (cf. subsection 4.1.1.1).

- *Class or type hierarchies:* Lines of inheritance only exist among object types, there is no implicit inheritance relationship among classes, but it can be realized by means of semantic relationships (cf. subsection 4.1.1.2).

- *Overriding, overloading and late binding:* VODAK does not support polymorphism. Overriding of method or property definitions in derived object types is not possible (cf. subsection 4.1.1.1).

- *Computational completeness:* VML integrates programming language, data definition language, and data manipulation language. The language available for application developers (application code) and schema designers (method implementation) is a C++-like typed programming language with the possibility to include C++-code directly.

- *Extensibility:* By means of object type definitions the set of types can be extended dynamically.

- *Persistence* and *secondary storage management*: The underlying storage system—ObjectStore^{TM}—provides for persistence and secondary storage management.

- *Concurrency* and *recovery*: The VODAK transaction system realizes open nested transactions, recovery is secured in case of hardware or software failures (cf. subsection 4.1.2).

- *Ad hoc query facility:* The *Vodak Query Language* (VQL) provides a set-oriented, declarative access to VODAK databases [AF93]. VQL queries are functional expressions and operate on the extensions of classes. This means that a VQL query takes VML classes as arguments and returns a value of any VML type as its result. The only access to objects within VQL queries is via the public interfaces, i.e., via method calls. The syntax of VQL is SQL like; it is similar to POSTQUEL (the query language of POSTGRES [SRH90]) and the query language of ORION [Kim90, pages 71–89].

4.2 Architecture of VODAK

In this section, the architecture of VODAK is described. The different components of a VODAK process are explained in depth. Further, embedding a VODAK process into a full VODAK environment is addressed.

VODAK represents a *distributed* object-oriented database system. Aspects of distributing a VODAK environment on different nodes have not been considered yet. Since transaction management for distributed VODAK environments is not completely implemented at the moment, the possibility to distribute VODAK on different nodes is disregarded within this thesis.

Figure 4.3 shows a VODAK environment. The VODAK processes X, Y, and Z (where process X is made apparent) are located at one node together with the databases on which their applications work.

Each VODAK process that opened a database is connected to the underlying storage system *ObjectStore* [Obj93]. A separate ObjectStore client is expected for each VODAK process. All VODAK processes located at the same node share common memory segments where, among other information, common objects and lock tables are maintained.

The most relevant components of a VODAK process are:

- *Application:* Application code is linked dynamically to the modules of a VODAK process. From within an application, databases can be created, opened, closed, or deleted; methods can be performed on database objects; and transactions can be started, committed, or aborted explicitly.

- *Message Handler:* The Message Handler is the central module of the runtime environment. Every execution of a VML method is performed by the Message Handler which uses the services provided by the Object Manager and the Communication Manager in order to guarantee the correct execution of the message. Thereby, the Message Handler performs the following steps:

    1. It requests the address of the receiver object making use of the Object Manager. This may cause the loading of the object from the database into the

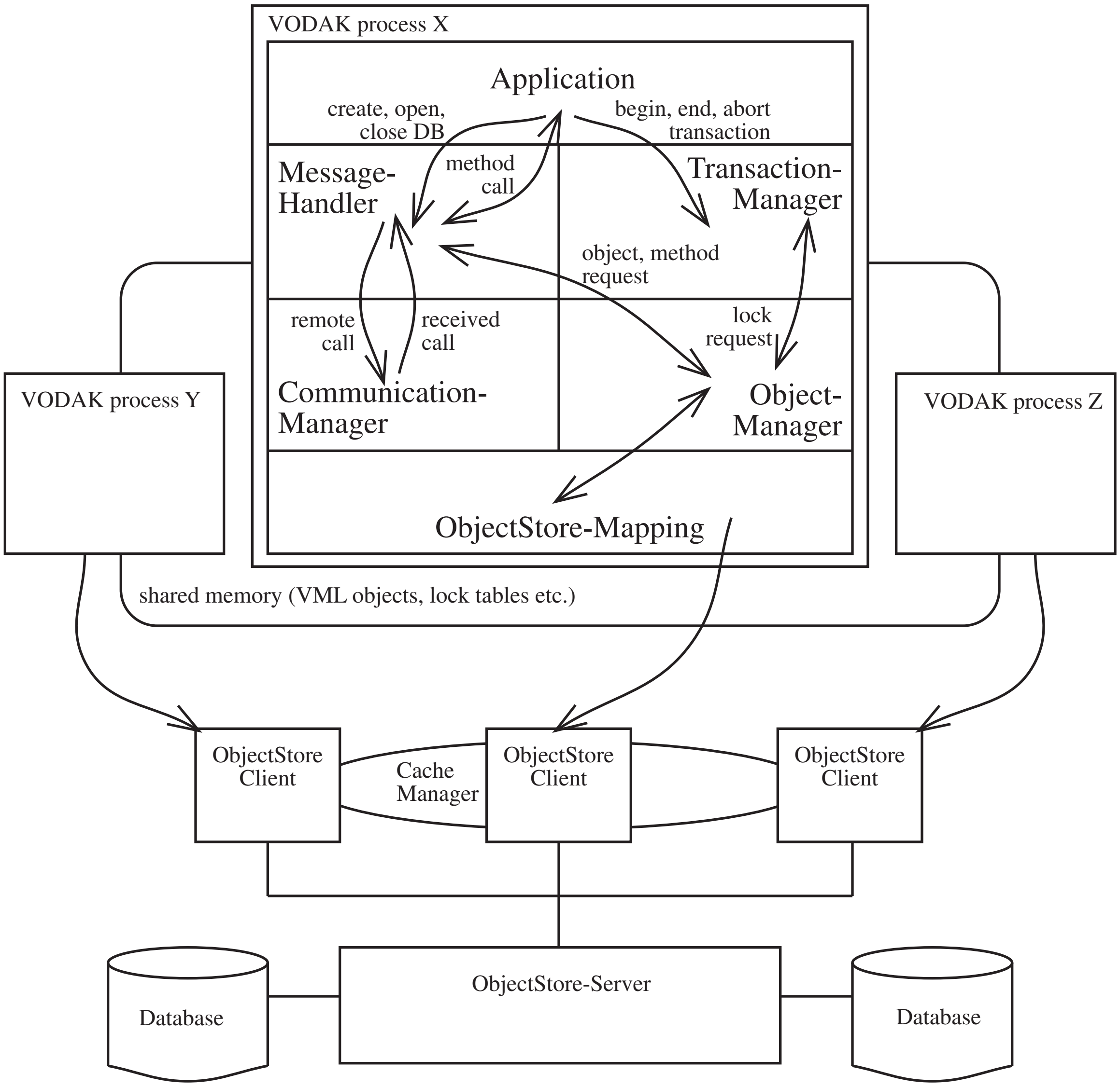


Figure4.3: VODAK process

object buffer. If the receiver object is at the same time accessed from within a non-compatible transaction, the requesting process is forced to wait.

2. It checks whether the method to be executed belongs to the interface of the object's class. Unless the method is defined, the inheritance behaviour implemented in the NOMETHOD-clause is called (cf. subsection 4.1.1.2).

3. It determines the address of the function which implements the method utilizing the services of the Object Manager.

4. It executes the function which implements the method.

5. Unless an error occurred during function execution, it returns the result of the function execution.

- *Transaction Manager:* The Transaction Manager implements VODAK open nested transactions (cf. subsection 4.1.2). It embraces the submodules for lock, log, and recovery management. The Transaction Manager's interface provides the functionality to begin, end, and abort (sub-)transactions, to request locks for objects, and to write the log.

- *Object Manager:* The Object Manager provides methods for retrieving objects and methods, services primarily utilized by the Message Handler. Moreover the Object Manager implements the basic functionality to read and update a property's value.

- *Communication Manager:* The Communication Manager maintains communication channels to VODAK processes located on remote nodes. A method call that cannot be executed locally by the Message Handler is passed to the Communication Manager for further handling. Only non-distributed VODAK environments, in which the Communication Manager is "umemployed", are subject of this thesis.

- *ObjectStore Mapping Manager:* The Mapping Manager is responsible for the communication with ObjectStore Database Server. Thus, it plays the role of an intermediary between the Object Manager and persistent storage.

Message Handler and Transaction Manager will prove to be the central modules in extending VODAK with active capabilities. Like the modelling concepts of VODAK introduced in this chapter, Message Handler and Transaction Manager represent both, basis and limitations for active extensions.

Chapter 5

ActiveVODAK

This chapter represents the thesis's central part; it describes the extension of VODAK with active capabilities. It focuses on conceptual and architectural issues, whereas chapter 6 deals with implementational details. Section 5.1 introduces the fundamental concepts adopted to integrate rules into the VODAK data model. Section 5.2 describes the knowledge model of ActiveVODAK: how are rules, events, conditions, and actions modelled from the schema designer's point of view? Section 5.3 discusses diverse aspects of rule execution, and section 5.4 explains the necessary extensions of a VODAK process to provide active capabilities.

5.1 Design Rationale

Extending VODAK with active capabilities requires the integration of rules and rule components into an originally "passive" data model. Conceptual decisions for the following questions had to be made: Should rules be local to classes, or should rules be defined globally? How should rules be represented in the database? This section motivates the choices made by comparing the pros and cons of alternatives.

5.1.1 Local vs. Global Rules

A point of discussion in the diverse literature on active object-oriented database systems is, whether rules should be *local* or *global*. A rule is referred to as local if it is associated with a single class, otherwise it is said to be global ([GJ91]).

Since local rules are defined within class definitions, they are also called *internal* rules. Local rules obey the principles of encapsulation and locality—two principles of object orientation. Local rules are well suited to monitor situations in which only a single object is involved and in which the rule's underlying event is object-related (i.e., the event represents either a method invocation or a value operation). If instances of different classes should be monitored by a single rule, the problem of splitting the rule and of assigning it to different classes arises. Similarly, a rule that bases on a non object-related event (abstract event, time event) cannot clearly be associated to a class.

Definitions of global rules are detached from class definitions, therefore, global rules are also called *external* rules. If global rules are employed, rules monitoring a single object and rules monitoring objects of different classes can be defined in a uniform way; no splitting of rules is necessary. The problem of associating object-independent rules to classes does not arise.

Rules are global in ActiveVODAK for two reasons. First, global rules allow to fully exploit the modelling semantics of ECA rules in a uniform way. Second, global rules do not necessitate to extend class definitions and, thus, the VODAK kernel, since rule definitions are separate from class definitions.

5.1.2 Representation of Rules

How should global rules and their components—events, conditions, and actions—be represented in the data model of ActiveVODAK? Rule representation is influenced by two requirements:

1. *Homogeneity:* An ActiveVODAK database contains rules and rule components in addition to other objects. To provide for homogeneity throughout an ActiveVODAK database, it is obvious to treat rules and rule components as *first class objects*, too. Alternatives such as to treat rules as declarations only inside classes, or to treat rules as data members (cf. [CAM93]) are not compatible with the concept of global rules adopted in ActiveVODAK.

2. *Rule definition within database schemata:* Besides metaclasses and application classes, rules are utilized by schema designers to model application domains. To provide for uniformity, it must be possible to define rules within database schemata (at schema definition time), too. In VODAK, the only objects defined at schema definition time are metaclasses and application classes (cf. subsection 4.1.1.1). Consequently, it is obvious to represent rules and rule components by means of classes in ActiveVODAK. Rules and their components are treated like other application classes.

Figure 5.1 illustrates the representation of a rule R in an ActiveVODAK database. Aggregation relationships between R and its components (event E, condition C, and action A) are symbolized by means of dashed lines. Since rules, events, conditions, and actions are altogether represented by classes, they have in common that they can be identified by unique names. To treat events, conditions, and actions as classes and to detach their definition from rule definition, allow a schema designer to associate events, conditions, and actions with several rules. Rules, events, conditions, and actions are instances of predefined metaclasses, which must become part of an "active" database

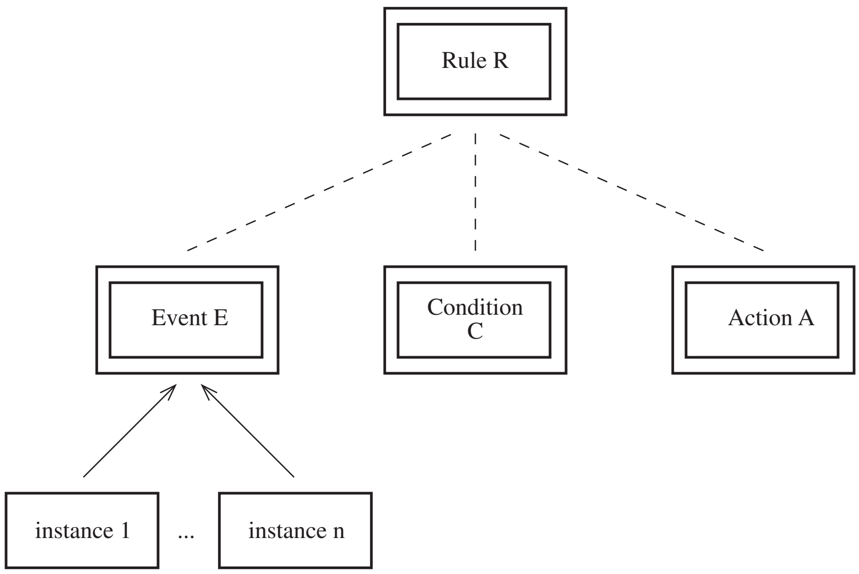


Figure5.1: Representation of rules

schema. This is done by explicitly including schema *Rules* which contains all necessary metaclass definitions (cf. section 5.2).

- *Events:* Following the explanations in subsection 3.2.2, a strict distinction between *event classes* and *event instances* is made. Besides a meaning of specification, in the ActiveVODAK data model event classes also have an extensional meaning, i.e., they collect their instances.
- *Conditions:* A condition class specifies a single method implementing the condition. No instances of condition classes are created.
- *Actions:* An action class specifies a single method implementing the action. No instances of action classes are created.
- *Rules:* A rule class aggregates an event class, a condition class, and an action class. No instances of rule classes are created.

Rules, conditions, and actions are represented by classes for which no terminal instances are created. Therefore, the question may arise, why rules, conditions, and actions are not treated as terminal instances. Currently, VODAK and, thus, ActiveVODAK are lacking schema modification capabilities. It is not possible to add method definitions once a database schema has been compiled. The specification of conditions and actions requires the definition of methods—consequently, conditions and actions have to be defined at schema definition time. The requirement to define rules and their components within database schemata forces the decision to model rules, conditions, and actions as classes. Terminal instances cannot be created at schema definition time!

timeStamp	point in time the event was raised

Table5.1: General parameter: timestamp

5.2 Event-Condition-Action Rules

ECA rules provide the basic semantics for extending VODAK with active capabilities (cf. subsection 3.2.1). This section discusses both, semantic and syntactic aspects of *ECA rule modelling* as it is supported in ActiveVODAK. It describes the definition of rules from a schema designer's point of view. The following subsections deal with the definition of events, conditions, actions, and rules.

5.2.1 Events

The event types supported in ActiveVODAK are method events, time events (absolute, relative, and periodic time events), abstract events, conjunctions, sequences, negations, and events with associated monitoring intervals. In the sequel, the following topics are treated for each event type:

- Event class definitions are specified using a formal notation (EBNF[1]).
- Event class definitions are illustrated continuing the examples introduced in subsection 3.2.2.
- The event classes' embedding in schema *Rules* is described conceptually.
- Event type specific parameter bindings are enumerated. Different parameters are needed to reflect the context of instantiating different event types. The only parameter bound by every event is *timeStamp* indicating the point in time the event was raised (cf. Table 5.1).
- Event class parameterization is discussed. To provide a mechanism to adopt an event class to environment changes, event classes are parameterized, i.e., specific property values of event classes can be changed at runtime.

5.2.1.1 Primitive Events

As mentioned in subsection 3.2.2.1, literature subsumes database events (method events, value events, transaction events), time events, and abstract events under primitive events. Two kinds of database events, value events and transaction events, are not part of the knowledge model of ActiveVODAK.

[1]The notation used is explained in appendix B

Value events are disregarded in ActiveVODAK since they are of subordinate importance in object-oriented systems (cf. subsection 3.2.2.1). Consequently, if value events are needed, they must be mapped to method events. Intuitively, two mapping strategies are applicable:

1. A value event is mapped to *several* method events: For each method that potentially modifies the value of the monitored property a method event must be defined. Since a disjunction operator is not supported in ActiveVODAK (cf. subsection 5.2.1.3), a separate rule has to be defined for each method event. The same condition and the same action have to be associated with all the rules defined.
2. A value event is mapped to *one* method event: This alternative is applicable only if the agreement, to modify the monitored property's value exclusively via a primitive access method, is strictly adhered to. For the primitive access method a method event must be defined.

For performance reasons a direct support of value events might be desirable. Considerations for possible future extensions of ActiveVODAK are made in chapter 7.

Transaction event definition requires the possibility to identify transactions uniquely, for example via their name. VODAK does not allow to assign a name or another identifying attribute to a transaction. Therefore, transaction events cannot be defined by schema designers. In ActiveVODAK, transaction events are only used internally to realize different coupling modes (cf. subsection 5.3.4).

Method Events. In the VODAK data model, an object's interface is defined on three levels—the object's *OWNTYPE* (if the object represents a class or metaclass), the *INSTTYPE* associated with the object's class, and the *INSTINSTTYPE* associated with the object's metaclass (cf. subsection 4.1.1.1). Therefore, three different scopes for applying methods exist. The first scope represents a *single object* (i.e., class or metaclass), the second scope represents the *extension* of a class or metaclass (i.e., all instances of a class or metaclass), and the third scope represents the *scope of a metaclass* (i.e., all the instances of the instances of a metaclass). The way in which method events are defined reflects the three different scopes for applying methods. A method event can be defined on a single object, the extension of a class or metaclass, or the scope of a metaclass:

1. *Single object:* The method event is defined on a single object. Since event definitions are part of the database schema, single-object method events cannot be based on terminal instances. The only objects available at schema definition time are classes and metaclasses. User-defined method events that should be valid for a single object are based on the predefined metaclass *MethEvntOnObject*.

```
'CLASS' <methEvnt> 'METACLASS MethEvntOnObject'
    'OWNTYPE OBJECTTYPE'
        'INTERFACE METHODS'
```

```
                'Raise(newEvnt: OID'{',' <arg>':' <argType>}');'

            'IMPLEMENTATION METHODS'
                'Raise(newEvnt: OID'{',' <arg>':' <argType>}');'
                '{'
                    {'newEvnt->SYS_set_'<argBound>'('<arg>');'}
                '};'
        'END;'

        'INSTTYPE OBJECTTYPE'
            'INTERFACE PROPERTIES'
                {<argBound>':' <argBoundType>';'}

            'IMPLEMENTATION'
        'END;'

    'INIT' <methEvnt>'->Init('<actvCls>','<actvMethod>','<modifier>')'
    'END;'
```

The non-terminal symbols denote:

$< methEvnt >$	unique name of the method event
$< actvCls >$	name of a class or metaclass for a method of which a method event is being defined
$< actvMethod >$	name of the method for which a method event is being defined; the method must be defined for $< actvCls >$
$< modifier >$	flag, indicating whether the method event is to be raised before or after method execution
$< arg >$	name of an argument of $< actvMethod >$; name, order, and type of the arguments have to be identical with the signature of $< actvMethod >$
$< argType >$	type of $< arg >$
$< argBound >$	name, under which an argument bound can be accessed
$< argBoundType >$	type of $< argBound >$; it has to be identical with the type of the argument assigned in the implementation of *Raise*

Implementing the method *Raise* of $< methEvnt >$, a schema designer can determine the context of instantiation which is represented by instances of $< methEvnt >$. Besides *newEvnt* (representing the new event instance), the signature of *Raise* corresponds to that of $< actvMethod >$. All actual parameters of *Raise*, apart from *newEvnt*, are potential candidates to be stored in event instances. A schema designer has to define a property in the instance-type of $< methEvnt >$ for each parameter that should be bound at event instantiation. Within the implementation of *Raise*, actual values have to be assigned to the defined properties[2]. In

[2] To assign a value *val* to a property *prop* defined in the inface part of an object, VODAK provides the predefined method $SYS\_set\_prop(val)$. To access the value of *prop*, method $SYS\_prop()$ can be engaged.

addition to these method-specific parameter bindings, every method event instance binds the identifier of the object, on which the method is executed (o), the identifer of the top-level transaction embracing the method ($transId$), and the point in time of instantiation ($timestamp$). A schema designer does not have to provide for binding these default parameters (cf. subsection 6.1.1.1).

Example: If a new book becomes part of the bookseller's assortment, the book is set on a list of new books which is sent to customers periodically. A new book is created by performing a method *NewBook* on class *Book*. The signature of *NewBook* is defined as follows:

```
NewBook(price: REAL, supplier: STRING, title: STRING,
        publishers: STRING, ISBN: STRING, author: STRING,
        year: INT, place: STRING);
```

The method event *NewBookEvent* is raised every time a new book has been created, i.e., *after* each invocation of method *NewBook*.

```
CLASS NewBookEvent METACLASS MethEvntOnObject
    OWNTYPE OBJECTTYPE
        INTERFACE METHODS
            Raise(newEvnt: OID, price: REAL,
                  supplier: STRING, title: STRING,
                  publishers: STRING, ISBN: STRING,
                  author: STRING, year: INT,
                  place: STRING);

        IMPLEMENTATION METHODS
            Raise(newEvnt: OID, price: REAL,
                  supplier: STRING, title: STRING,
                  publishers: STRING, ISBN: STRING,
                  author: STRING, year: INT,
                  place: STRING);
            {
             newEvnt->SYS_set_priceBnd(price);
             newEvnt->SYS_set_supplierBnd(supplier)
             newEvnt->SYS_set_titleBnd(title);
             newEvnt->SYS_set_publishersBnd(publishers);
             newEvnt->SYS_set_ISBNBnd(ISBN);
             newEvnt->SYS_set_authorBnd(author);
             newEvnt->SYS_set_yearBnd(year);
             newEvnt->SYS_set_placeBnd(place);
            };
    END;

    INSTTYPE OBJECTTYPE
        INTERFACE PROPERTIES
            priceBnd: REAL;
            supplierBnd: STRING;
            titleBnd: STRING;
            publishersBnd: STRING;
            ISBNBnd: STRING;
            authorBnd: STRING;
```

```
                    yearBnd: INT;
                    placeBnd: STRING;

                IMPLEMENTATION
            END;

        INIT NewBookEvent->Init(Book,'NewBook',AFTER)
        END;
```

2. *Extension of a class or metaclass:* Execution of a specific method on an arbitrary instance of a class or metaclass causes instantiation of a method event. User-defined method events that should be valid for all instances of a class or metaclass base on the predefined metaclass *MethEvntOnInst*.

```
'CLASS' <methEvnt> 'METACLASS MethEvntOnInst'
    'OWNTYPE OBJECTTYPE'
        'INTERFACE METHODS'
            'Raise(newEvnt: OID'{',' <arg>':' <argType>}');'

        'IMPLEMENTATION METHODS'
            'Raise(newEvnt: OID'{',' <arg>':' <argType>}');'
            '{'
                {'newEvnt->SYS_set_'<argBound>'('<arg>');'}
            '};'
    'END;'

    'INSTTYPE OBJECTTYPE'
        'INTERFACE PROPERTIES'
            {<argBound>':' <argBoundType>';'}

        'IMPLEMENTATION'
    'END;'

'INIT' <methEvnt>'->Init('<actvCls>','<actvMethod>','<modifier>')'
'END;'
```

Apart from two exceptions, the non-terminal symbols denote the same as described above:

$< actvCls >$	name of a class or metaclass for whose instances a method event is being defined
$< actvMethod >$	name of the method for which a method event is being defined; $< actvMethod >$ must be defined for instances of $< actvCls >$

Example: The bookseller's problem of inventory control has already been introduced in subsection 3.2.2.1. For instances of class *Publication* a method *Sell* is defined, which records a sale. It decreases the instance's property *itemsOnStock* by the number of copies sold. Assumed that *Sell* is the only method decreasing the value of *itemsOnStock* an inventory control policy can be based on a single method event. Each time a publication (a book or a journal) is sold, an instance of *SellEvent* is created. *SellEvent* is raised after method execution. Since the reorder policy is different for books and journals, the actual kind of the publication has to be checked in the condition and/or action part of the rule basing on *SellEvent*.

```
CLASS SellEvent METACLASS MethEvntOnInst
    OWNTYPE OBJECTTYPE
        INTERFACE METHODS
            Raise(newEvnt: OID, noOfItems: INT);

        IMPLEMENTATION METHODS
            Raise(newEvnt: OID, noOfItems: INT);
            {
             newEvnt->SYS_set_noOfItemsBnd(noOfItems);
            };
    END;

    INSTTYPE OBJECTTYPE
        INTERFACE PROPERTIES
            noOfItemsBnd: INT;

        IMPLEMENTATION
    END;

INIT SellEvent->Init(Publication,'Sell',AFTER)
END;
```

3. *Scope of a metaclass:* Execution of a specific method on an arbitrary instance of an instance of a metaclass causes instantiation of a method event. The predefined metaclass *MethEvntOnInstInst* provides a basis for the definition of method events with metaclass-scope.

```
'CLASS' <methEvnt> 'METACLASS MethEvntOnInstInst'
    'OWNTYPE OBJECTTYPE'
        'INTERFACE METHODS'
            'Raise(newEvnt: OID' {',' <arg>':' <argType>}');'

        'IMPLEMENTATION METHODS'
            'Raise(newEvnt: OID' {',' <arg>':' <argType>}');'
            '{'
                {'newEvnt->SYS_set_'<argBound>'('<arg>');'}
            '};'
    'END;'

    'INSTTYPE OBJECTTYPE'
        'INTERFACE PROPERTIES'
            {<argBound>':' <argBoundType>';'}

        'IMPLEMENTATION'
    'END;'

'INIT' <methEvnt>'->Init('<actvCls>','<actvMethod>','<modifier>')'
```

```
'END;'
```

Two definitions have to be reviewed:

$< actvCls >$	name of a metaclass for whose instance-instance scope a method event is being defined
$< actvMethod >$	name of the method for which a method event is being defined; $< actvMethod >$ must be defined for instances of the instances of $< actvCls >$

Example: VML metaclasses specify structure and behaviour of their instances and the instances of their instances on an abstract level. Therefore, methods defined by metaclasses are not necessarily suited to base application domain-specific event definitions on them. For a schema designer it might be useful to raise a *SetSpecEvent* whenever a specialization is assigned to a generalized instance. Class *Generalization* defines a method *SetSpec*, which takes the specialized instance as an argument, for the instances of its instances:

```
CLASS SetSpecEvent METACLASS MethEvntOnInstInst
    OWNTYPE OBJECTTYPE
        INTERFACE METHODS
            Raise(newEvnt: OID, specObj: OID);

        IMPLEMENTATION METHODS
            Raise(newEvnt: OID, specObj: OID);
            {
             newEvnt->SYS_set_specObjBnd(specObj);
            };
    END;

    INSTTYPE OBJECTTYPE
        INTERFACE PROPERTIES
            specObjBnd: OID;

        IMPLEMENTATION
    END;

INIT SetSpecEvent->Init(Generalization,'SetSpec',AFTER)
END;
```

Embedding of the metaclasses *MethEvntOnObj*, *MethEvntOnInst*, and *MethEvntOnInstInst* in schema *Rules* is illustrated in Figure 5.2. The symbols employed correspond to the notation introduced in Table 4.1. Metaclass *MethEvnt* determines the common structure and behaviour of the user-defined method event classes *NewBookEvent*, *SellEvent*, and *SetSpecEvent*.

Disregarding the three different extensional scopes, the common parameter bindings of method events are summarized in Table 5.2.

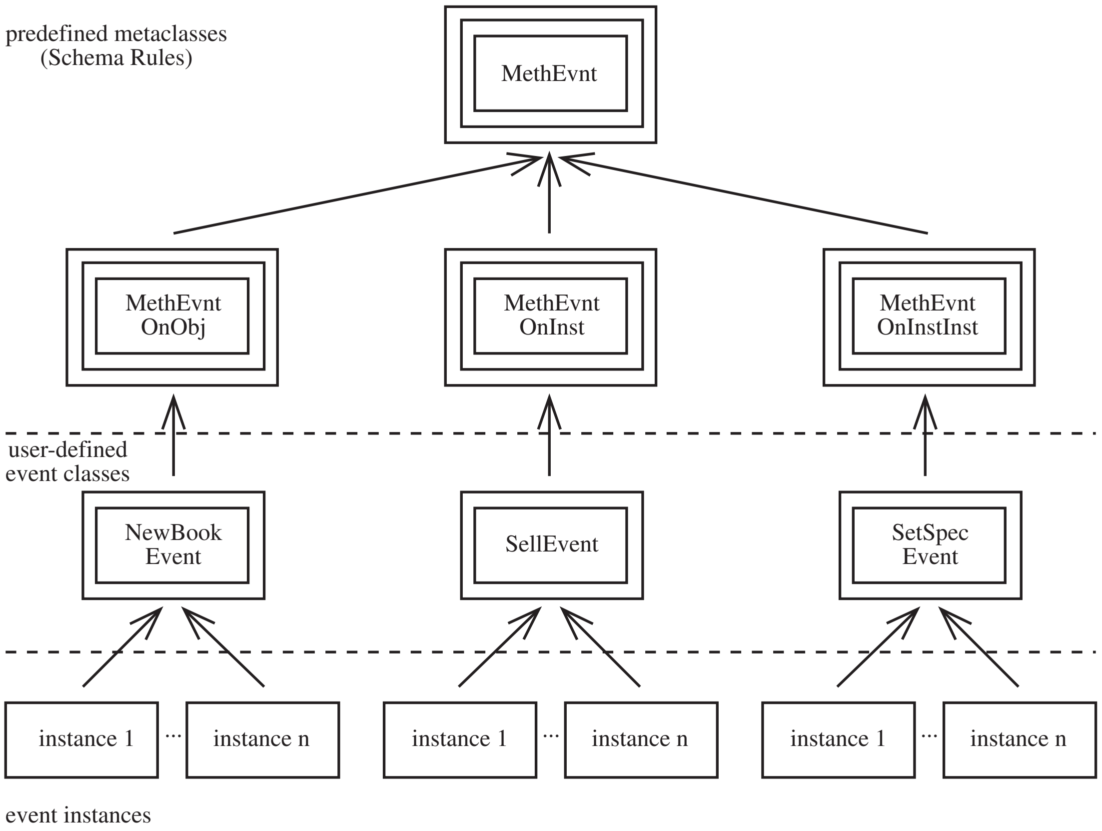


Figure5.2: Embedding of method event metaclasses

o	OID of the object, on which the method, is executed
$transId$	identifier of the top-level transaction embracing the method call
$\{arg\}$	set of method arguments (equivalent to the set of $< argBound >$, specified in the INSTTYPE of the event class)

Table5.2: Method event parameters

component	lower bound	upper bound
year	00	99
month	01	12
day	01	days of month
hour	00	23
minute	00	59
second	00	59

Table5.3: Components of absolute points in time

Parameterization of a method event class allows to modify the flag indicating whether the method event is to be raised before or after method execution. By sending a $SetModifier(< modifier >)$-message to a method event class, the flag can be set to $< modifier >$—i.e. to before or after. Modifications are effective only for future instantiations of the method event.

Time Events. Time events define specific points in time. In ActiveVODAK a specific point in time is either an absolute point in time or a point in time defined relatively to another. The finest granularity in specifying points in time is one second. Absolute time events, relative time events, and periodic time events are supported:

Absolute Time Events. Absolute time events represent an absolute point in time. An absolute point in time is of the form: 'year-month-day hour:minute:second'. The domains of the components of an absolute point in time are summarized in Table 5.3. Unless all components of an absolute point in time are explicitly indicated, the missing components are assumed to take the default value—the lower bound of the respective domain. The specification of an absolute point in time starts at the low-granularity end, i.e., at least the year-component has to be indicated.

Examples: Abbreviated absolute point-in-time specifications and their complete equivalences are:
$93 - 11 - 06\ 08 : 30 \equiv 93 - 11 - 06\ 08 : 30 : 00$
$93 - 11 - 06 \equiv 93 - 11 - 06\ 00 : 00 : 00$
$93 \equiv 93 - 01 - 01\ 00 : 00 : 00$

A VODAK class definition requires the specification of an INSTTYPE. Since a user-defined absolute time-event class does not define additional methods or properties for its instances, the INSTTYPE clause is empty. Note that an absolute time event is raised exactly once, i.e., there is only one instance per event class! The definition of an absolute time event class looks as follows:

```
'CLASS' <absTimeEvnt> 'METACLASS AbsTimeEvnt'
'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <absTimeEvnt>'->Init('<absPointInTime>')'
'END;'
```

The non-terminal symbols denote:

$< absTimeEvnt >$	name of the absolute time event
$< absPointInTime >$	an absolute point in time obeying the conventions introduced above

Example: On 1 January 1995 the bookseller intends to reduce the prices of books older than five years. The event $On1Jan95$ will be raised on 1 January 1995 at midnight.

```
CLASS On1Jan95 METACLASS AbsTimeEvnt
    INSTTYPE OBJECTTYPE
        INTERFACE
        IMPLEMENTATION
    END;
INIT On1Jan95->Init('95-01-01')
END;
```

The sole instance of an absolute time event class does not bind additional parameters. Absolute time event classes cannot be parameterized.

Relative Time Events. A relative time event specifies a point in time relative to another event ocurrence. Like absolute points in time, also a relative point in time is an expression of the form: 'year-month-day hour:minute:second'. Contrary to absolute points in time, relative points in time are defined starting at the high-granularity end, i.e., at least the second-component has to be indicated. Omitted components are assumed to be zero.

Examples: Abbreviated relative point in time specifications and their complete equivalences are:
one day: $01\ 00:00:00 \equiv 00-00-01\ 00:00:00$
five minutes: $05:00 \equiv 00-00-00\ 00:05:00$
ten seconds: $10 \equiv 00-00-00\ 00:00:10$

Relative time events are defined as follows:

```
'CLASS' <rltvTimeEvnt> 'METACLASS RltvTimeEvnt'
'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <rltvTimeEvnt>'->Init('<evnt>','<delay>')'
'END;'
```

The non-terminal symbols denote:

$< rltvTimeEvnt >$	name of the relative time event
$< evnt >$	name of an arbitrary event class the time event is relative to
$< delay >$	delay factor; relative point in time

Example: Two weeks after the bookseller has placed an order with his supplier, the bookseller checks whether the order has been shipped. If not, the supplier is urged on delivery. "Two weeks after an order has been placed" represents a point in time relative to the order's placement. *PlaceOrderEvent* represents a method event defined for all instances of class *Order*. Every time an order is placed, the event *PlaceOrderEvent* is raised, and the current system time is bound in *timeStamp*. A fortnight after *timeStamp* the relative time event *TwoWeeksAfterOrder* is instantiated.

```
CLASS PlaceOrderEvent METACLASS MethEvntOnInst
OWNTYPE OBJECTTYPE
    INTERFACE METHODS
        Raise(newEvnt: OID);

    IMPLEMENTATION METHODS
        Raise(newEvnt: OID);
        {
        };
END;

INSTTYPE OBJECTTYPE
    INTERFACE
    IMPLEMENTATION
END;
INIT PlaceOrderEvent->Init(Order,'PlaceOrder',AFTER)
END;


CLASS TwoWeeksAfterOrder METACLASS RltvTimeEvnt
    INSTTYPE OBJECTTYPE
        INTERFACE
        IMPLEMENTATION
    END;
INIT TwoWeeksAfterOrder->Init(PlaceOrderEvent,'14 00:00:00')
END;
```

On instantiating a relative time event no additional parameters are bound. The parameters bound by a relative time event include the parameter bindings of the event on which the relative event is based. The parameterization of relative time-event classes allows to modify the delay factor. By performing a $SetDelay(< delayValue >)$-message on a relative time-event class, the delay factor can be set to $< delayValue >$. The new delay factor is effective only for future instantiations of the basic event. Pending event instances, i.e., event instances for which a basic event instance is potentially available but for which the delay time has not yet elapsed, are not affected by a new delay factor.

Periodic Time Events. A periodic time event is raised periodically after each instantiation of the start event. There might be multiple instantiations of the start event, each of them inducing a sequence of periodic event instantiations. Since applications may require not to induce a new sequence of instantiations in every case, but possibly to abort an existing sequence and start a new one, the notion of a *guard-function* is introduced. Every time, the start event is raised, a guard-function determines, whether either a new (additional) sequence of periodic instantiations should be initiated, or an existing sequence should be aborted and replaced by a new one.

To be able to define a guard-function correctly, some implementational details must be explained: At a given point in time, n sequences of periodic instantiations exist. Always, when the start event is raised, the guard-function is performed on pairs ($evntToCheck$,$firingToCheck_i$); $evntToCheck$ represents the new start-event instance, $firingToCheck_i$ sequentially takes the start-event instances of all existing sequences of instantiation ($1 < i < n$). If the (boolean) guard-function evaluates to true for a pair ($evntToCheck$,$firingToCheck_i$), the existing sequence of instantiation that started with $firingToCheck_i$ is aborted. A new sequence of instantiations is started with $evntToCheck$. Within the implementation of the guard-function $evntToCheck$ and $firingToCheck$ have to be compared; the comparison criterion entirely depends on application domain semantics.

Two predefined guard-functions are available: *AnyObject* and *SameObject*. *AnyObject* evaluates to true in any case, i.e., the existing periodic firing is replaced. *SameObject* only evaluates to true for those start-event instances that have been bound to the same o (cf. Table 5.2) as the new start event. Since o is only bound by method events, the *SameObject*-guard-function is only applicable to start events that are method events.

Periodic time events are defined as follows:

```
'CLASS' <prdcTimeEvnt> 'METACLASS PrdcTimeEvnt'
['OWNTYPE OBJECTTYPE'
    'INTERFACE METHODS'
        <guardFunc>'():BOOL;'

    'IMPLEMENTATION'
        <guardFunc>'():BOOL;'
        '{'
            <method>
```

```
        '};'
'END;']
'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <prdcTimeEvnt>'->Init('<startEvnt>','<period>','<guardFunc>')'
'END;'
```

The non-terminal symbols denote:

$< prdcTimeEvnt >$	name of the periodic time event
$< guardFunc >$	an optional user-defined guard-function; if the OWNTYPE is missing, only a predefined guard-function can be indicated
$< startEvnt >$	arbitrary start-event class
$< period >$	time interval between two instantiations; relative point in time (as introduced above)
$< method >$	method body implementing the guard-function; a boolean value has to be returned

Example: If the bookseller's customer is in delay for payment, the bookseller will send a reminder to his customer every three weeks until payment is made. "To be in delay for payment" is modelled by means of the abstract event $InDelayForPayment$ (see next paragraph). Instances of $InDelayForPayment$ bind the sale number and the amount of money owed. A periodic sending of reminders should only be replaced by a new one, if the $InDelayForPayment$-event is raised for the same sale number once again. E.g., after two reminders sent for a specific sale, it might be customary to raise the $InDelayForPayment$-event for the same sale, but with an increased amount, once again.

```
CLASS ReminderEvent METACLASS PrdcTimeEvnt
OWNTYPE OBJECTTYPE
    INTERFACE METHODS
        SameSale():BOOL;

    IMPLEMENTATION
        SameSale():BOOL;
        {
            RETURN evntToCheck->(INT)SYS_saleNo() ==
                   firingToCheck->(INT)SYS_saleNo();
        };
END;
INSTTYPE OBJECTTYPE
    INTERFACE
    IMPLEMENTATION
END;
INIT ReminderEvent->Init(InDelayForPayment,
                         '21 00:00:00','SameSale')
END;
```

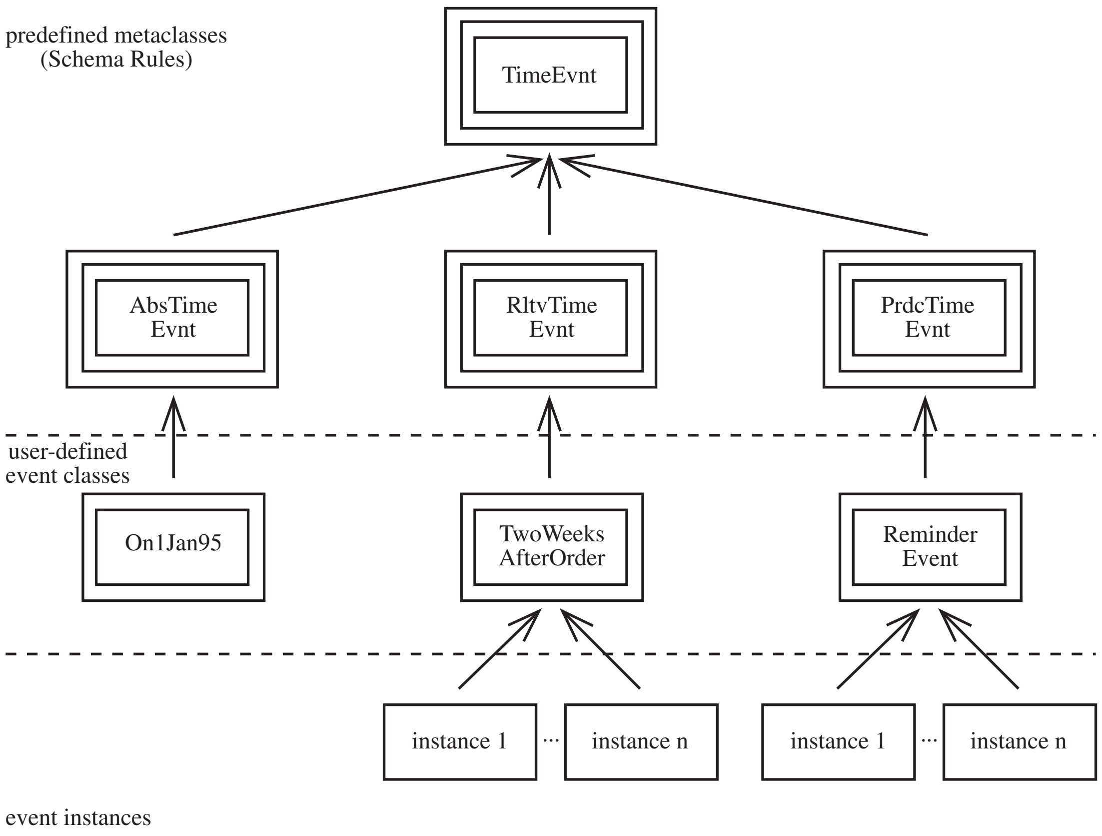


Figure5.3: Embedding of time event metaclasses

On instantiating a periodic time event no additional parameters are bound. The parameters bound by a periodic time event also include the parameter bindings of the start event. The start-event instance is part of each periodic time-event instance. The parameterization of relative time-event classes allows to modify the time interval between two instantiations. By performing a $SetPeriod(< periodValue >)$-method on a periodic time-event class, the time interval between two instantiations can be set to $< periodValue >$. The new value effects existing sequences of instantiation from the point in time when the next firing takes place.

Embedding of the metaclasses *AbsTimeEvnt*, *RltvTimeEvnt*, and *PrdcTimeEvnt* in schema *Rules* is illustrated in Figure 5.3. Metaclass *TimeEvnt* determines the common structure and behaviour of the user-defined time-event classes *On1Jan95*, *TwoWeeksAfterOrder*, and *ReminderEvent*.

Abstract Events. All event types introduced up to now are detected autonomously by the database system or by system components attached to it (e.g., system clock). The specific semantics of abstract events are unknown to the database system. Since the database system is not aware of the point in time an abstract event occurs, the database system cannot take measures before and after event instantiation. Before event instantiation a new instance has to be allocated and the default parameters (see below) have to be bound. After event instantiation, event classes, monitoring intervals, and rules that are based on the abstract event class have to be informed (cf. 5.3.1) about instantiation. Instance allocation and parameter bindings are performed by method (*NewListElem*). Method (*InformSubscribers*) informs subscribers of an event class. *NewListElem* and *InformSubscribers* have to be explicitly invoked in method *Raise*. User-defined abstract events are based on the predefined metaclass $AbstrctEvnt$.

```
'CLASS' <abstrctEvnt> 'METACLASS AbstrctEvnt'
'OWNTYPE OBJECTTYPE'
    'INTERFACE METHODS'
        'Raise('[<arg>':' <argType>]{',' <arg>':' <argType>}');'

    'IMPLEMENTATION METHODS'
        'Raise('[<arg>':' <argType>]{',' <arg>':' <argType>}');'
        '{'
            'VAR newEvnt: OID;'

            'newEvnt:=SELF->NewListElem();'
            {'newEvnt->SYS_set_'<argBound>'('<arg>');'}
            'SELF->InformSubscribers(newEvnt);'
        '};'
    'END;'

'INSTTYPE OBJECTTYPE'
    'INTERFACE PROPERTIES'
        {<argBound>':' <argBoundType>';'}
    'END;'

'INIT' <abstrctEvnt>'->Init()'
'END;'
```

The non-terminal symbols denote:

$< abstrctEvnt >$	unique name of the abstract event
$< arg >$	name of an argument that is bound at event occurrence; the actual value has to be provided explicity by the user, program, or system component raising $< abstrctEvnt >$
$< argType >$	type of $< arg >$
$< argBound >$	name, under which an argument bound can be accessed
$< argBoundType >$	type of $< argBound >$; it has to be identical with the type of the argument assigned in the implementation of $Raise$

Example: An example for an abstract event has already been given in the above paragraph describing periodic time events. "To be in delay for payment" is modelled by means of the abstract event $InDelayForPayment$. Every time, a user explicity raises the $InDelayForPayment$-event, the new instance binds the two arguments indicated by the user: the sale number and the amount of money owed.

transId	identifier of the top-level transaction that embraces the raising of the abstract event
$\{arg\}$	set of user-defined arguments (equivalent to the set of $< argBound >$, specified in the INSTTYPE of the abstract event class)

Table5.4: Abstract event parameters

```
CLASS InDelayForPayment METACLASS AbstrctEvnt
OWNTYPE OBJECTTYPE
    INTERFACE METHODS
        Raise(sale: INT, amount: REAL);

    IMPLEMENTATION METHODS
        Raise(sale: INT, amount: REAL);
        {
            VAR newEvnt: OID;

            newEvnt:=SELF->NewListElem();
            newEvnt->SYS_set_saleNo(sale);
            newEvnt->SYS_set_openAmount(amount);
            SELF->InformSubscribers(newEvnt);
        };
END;

INSTTYPE OBJECTTYPE
    INTERFACE PROPERTIES
        saleNo: INT;
        openAmount: REAL;

    IMPLEMENTATION
END;

INIT InDelayForPayment->Init()
END;
```

Embedding of metaclass *AbstrctEvnt* in schema *Rules* is trivial. *AbstrctEvnt* is an instance of the standard metaclass *Metaclass*, which does not specify any active capabilities.

Table 5.4 summarizes the parameters bound by an abstract event in addition to *timeStamp*. Database operations can be performed within transactions exclusively. Raising an abstract event represents a database operation, too. Hence, each abstract event (irrespective of its kind— internal or external) binds the identifier of the transaction from within which it has been raised.

Abstract event classes cannot be parameterized.

5.2.1.2 Monitoring Intervals

A monitoring interval does not represent an event type. Monitoring intervals are engaged to restrict event observation to the period between a begin-event instantiation and an end-event instantiation. A monitoring interval typically is "on", if a begin event but no end event occurred; otherwise it is "off".

In ActiveVODAK, the transitions from "on" to "off" and from "off" to "on" are more flexible. At the same time, a monitoring interval may be "on" for the state[3] of one begin-event instance, and "off" for another one. Every time the begin event raises, the monitoring interval switches to "on" for the state of the begin-event instance newly created. At instantiation of the end event, compatibility between the new end-event instance and existing begin-event intances is checked. If the end-event instance's state proves to be compatible with the state of a begin-event instance, the monitoring interval switches to "off" for the state of the respective begin-event instance. Compatibility of begin and end-event instances is application domain specific and is guaranteed by means of a *guard-function*.

Example: From the point in time, the bookseller places an order with his supplier, execution of the order placed has to be monitored for two weeks. If the order does not arrive within two weeks, the supplier is urged on delivery. Monitoring has to be order-specific, i.e., the monitoring interval must be on for an order that was placed less than two weeks ago; it must be off for an order placed more than two weeks ago. A guard-function has to guarantee that an end-event instance (two weeks after order placement) is compatible with a begin-event instance (order placement) only, if the same order is concerned.

Two predefined guard-functions are available: *AnyObject*, which considers any pair of begin event instance and end event instance to be compatible; and *SameObject*, which considers a pair of begin event instance and end event instance to be compatible, if the event instances bound the same o. Consequently, *SameObject* is applicable only to interval boundaries representing method events. If $begEvntToCheck$ (begin event instance) and $endEvntToCheck$ (end event instance) are compatible, a checking guard-function evaluates to true.

Definition of a monitoring interval looks as follows:

```
'CLASS' <MonitInt> 'METACLASS MonitInt'
['OWNTYPE OBJECTTYPE'
    'INTERFACE METHODS'
        <guardFunc>'():BOOL;'

    'IMPLEMENTATION METHODS'
        <guardFunc>'():BOOL;'
        '{'
```

[3]The state of an event instance is constituted by the values of the event instance's parameter bindings.

```
            <method>
        '};'
'END;']

'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <MonitInt> '->Init(' <beginEvnt> ',' <endEvnt> ',' <guardFunc> ')'
'END;'
```

The non-terminal symbols denote:

$< MonitInt >$	unique name of the monitoring interval
$< guardFunc >$	an optional user-defined guard-function; if the OWNTYPE is missing, only a predefined guard-function can be indicated
$< method >$	method body implementing the guard-function; a boolean value has to be returned
$< beginEvnt >$	arbitrary begin event
$< endEvnt >$	arbitrary end event

Example: Continuing the above example, the interval between order placement and the point in time two weeks after order placement is modelled as the monitoring interval *OrderExecInt*. *PlaceOrderEvent* and *TwoWeeksAfterOrder* have already been defined in the example describing relative time events. The guard-function *SameOrder* evaluates to true, if the event instance to which the end-event instance is relative corresponds to the start-event instance.

```
CLASS OrderExecInt METACLASS MonitInt
INSTTYPE OBJECTTYPE
    INTERFACE
        SameOrder():BOOL;

    IMPLEMENTATION
        SameOrder():BOOL;
        {
            RETURN endEvntToCheck->(OID)SYS_evntInst() ==
                   begEvntToCheck;
        };
END;
INIT OrderExecInt->Init(PlaceOrderEvent,TwoWeeksAfterOrder,
                        'SameOrder')
END;
```

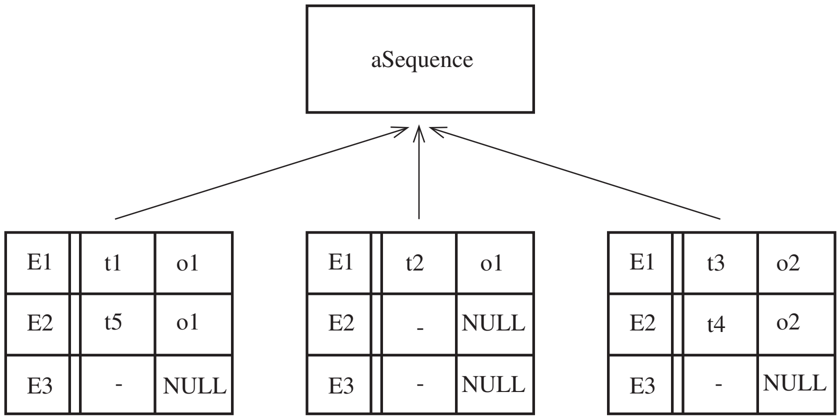


Figure5.4: Incomplete instances

5.2.1.3 Complex Events

In the sequel, the three[4] complex event operators supported in ActiveVODAK—*conjunction*, *sequence*, and *negation*—are described.

Conjunctions and Sequences. Conjunctions and sequences of events are identical in their definitions. Unless an explicit distinction is made, the following explanations are valid for both, conjunctions and sequences. Conjunctions and sequences define a list of basic event classes[5] and, optionally, a guard-function. Every time a basic event is instantiated, the guard-function is employed to check, whether the complex event has to be raised. The guard-function guarantees that the states of the basic event instances forming a complex event instance are compatible. Here, the notion of a *incomplete instance* has to be introduced: Since several instantiations of the basic events of a single complex event are not necessarily compatible, several *incomplete instances* of the complex event may exist at the same time.

Example: Imagine a sequence event $aSequence$ based on the three events $E1$, $E2$, and $E3$, altogether representing method events. The predefined guard-function *Same-Object* ensures that $aSequence$ is raised only, if the three basic event instances have bound the same o. Figure 5.4 shows three incomplete instances of class $aSequence$. Each complex event instance consists of basic event instances represented by a timestamp and the object bound in o. Missing basic event instances are indicated by "NULL".

Every time a basic event is instantiated, the attempt is made, to add the new basic event instance to an existing incomplete instance. The associated guard-function deci-

[4]Extension of ActiveVODAK with additional event operators may be subject of future work (cf. chapter reftheOutlook).

[5]A single event class must not be included in the list of basic event classes more than once.

des, whether the basic event instance can be added to an existing incomplete instance. If the guard-function evaluates to false (i.e., the basic event instance is not compatible with an incomplete instance), a new incomplete instance is created. From within the guard-function, the incomplete instance can be referred to as *instToCheck*. Two predefined guard-functions are available: *AnyObject*, which evaluates to true in any case, and *SameObject*, which has already been explained in the above example. If the complex event represents a sequence, the chronological order of basic event instantiation has to be considered, too. Note that one instance of a basic event can only contribute to one instance of a given complex event! Of course, a single instantiation can contribute to several instances of *different* complex events.

Apart from metaclasses, the definition of conjunctions and sequences of events is identical; conjunctions are based on the predefined metaclass *Conjunction*, sequences are instances of *Sequence*:

```
'CLASS' <ConjOrSequ> 'METACLASS' 'Conjunction' | 'Sequence'
['OWNTYPE OBJECTTYPE'
    'INTERFACE METHODS'
        <guardFunc>'():BOOL;'

    'IMPLEMENTATION METHODS'
        <guardFunc>'():BOOL;'
        '{'
            <method>
        '};'
'END;']

'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <ConjOrSequ>'->Init('<basicEvents>','<guardFunc>')'
'END;'
```

The non-terminal symbols denote:

$< ConjOrSequ >$	unique name of the sequence or conjunction
$< guardFunc >$	an optional user-defined guard-function; if the OWNTYPE is missing, only a predefined guard-function can be indicated
$< method >$	method body implementing the guard-function; a boolean value has to be returned
$< basicEvents >$	list of arbitrary basic event classes

Example: The bookseller informs his suppliers about successful shipments of orders. The sequence of order placement (*PlaceOrderEvent*) and order shipment (*ShipOrderEvent*) is modelled by means of the complex event *PlacedOrderShipped*. *PlaceOrderEvent* and *ShipOrderEvent* represent method events, they are both defined for instances of class *Order*. The predefined guard-function *SameObject* ensures that *PlacedOrderShipped* is raised only, if the shipped order is identical with the placed one. *PlaceOrderEvent* has already been introduced to illustrate relative time events.

```
CLASS ShipOrderEvent METACLASS MethEvntOnInst
    OWNTYPE OBJECTTYPE
        INTERFACE METHODS
            Raise(newEvnt: OID);

        IMPLEMENTATION METHODS
            Raise(newEvnt: OID);
            {
            };
    END;

    INSTTYPE OBJECTTYPE
        INTERFACE
        IMPLEMENTATION
    END;
INIT ShipOrderEvent->Init(Order,'RecordShipment',AFTER)
END;

CLASS PlacedOrderShipped METACLASS Sequence
INSTTYPE OBJECTTYPE
    INTERFACE
    IMPLEMENTATION
END;
INIT PlacedOrderShipped->
     Init([<PlaceOrderEvent,ShipOrderEvent>],'SameObject')
END;
```

Embedding of the metaclasses *Conjunction* and *Sequence* in schema *Rules* is conceptually described in Figure 5.5. Metaclass *ComplxEvnt* determines common active capabilities of user-defined sequences and conjunctions.

Sequences and conjunctions bind a *timeStamp*, too. The *timeStamp* of the complex event instance is equivalent to that of the completing basic event instance. If the completing basic event instance binds *transId*, sequences and conjunctions additionally bind the identifier of the transaction within which they occurred. Whether an instance of a conjunction/sequence possesses a transaction binding can be determined only at runtime. Sequence event classes and conjunction event classes are not parameterized.

Negations A negation consists of a basic event, a monitoring interval, and a guard-function. A negation is instantiated, if the basic event does not occur during the monitoring interval. Negations are based on a monitoring interval which is defined according to the semantics introduced in subsection 5.2.1.2. The guard-function associated with the negation itself checks compatibility between basic event instances and an ended monitoring interval. If the guard-function does not evaluate to true for any basic event instance occurred within the monitoring interval's boundaries, the negation event is raised.

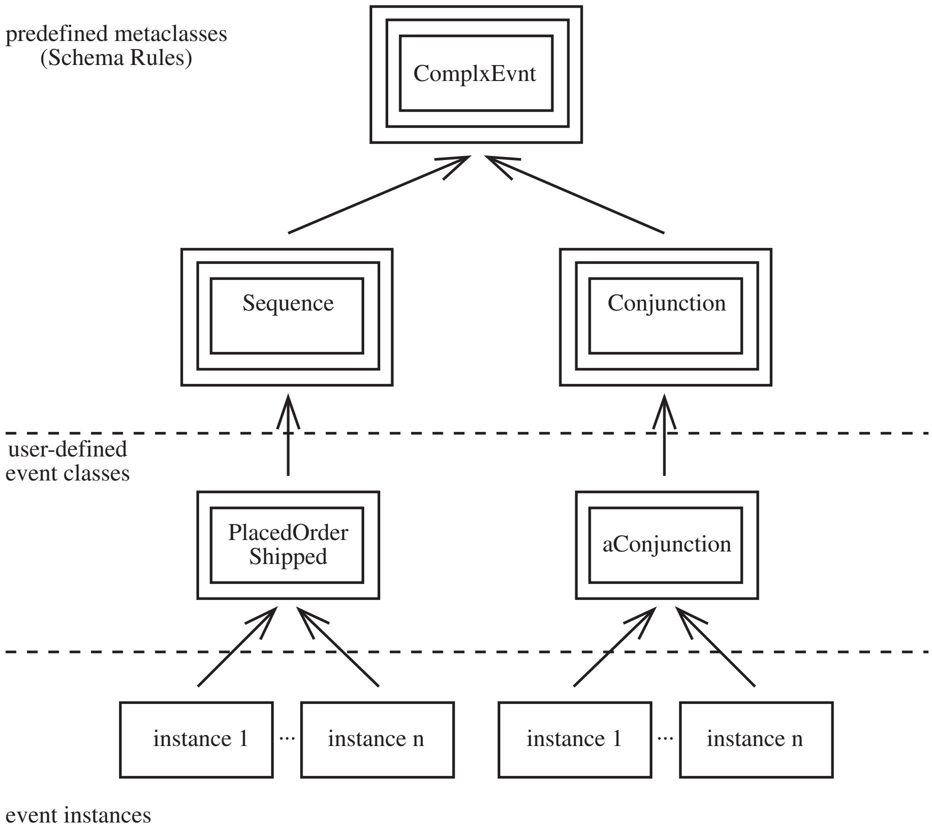


Figure5.5: Embedding of sequences and conjunctions

Example: The monitoring interval $OrderExecInt$ (cf. subsection 5.2.1.2) starts at order placement and ends two weeks later. If the interval ends for an order $order_i$, it has to be checked whether $order_i$ has already been shipped. Unless a compatible instance of $ShipOrderEvent$ (described above) has been created, the negation event $DelayOfDelivery$ will be raised. The negation's guard-function has to ensure that an instance of $ShipOrderEvent$ and an ended monitoring interval are considered to be compatible only, if the same order is concerned.

Two predefined guard-functions are available: $AnyObject$, which evaluates to true in any case, and $SameObject$, which evaluates to true only, if the basic event instance ($evntToCheck$) and the begin event instance of the ended monitoring interval ($monitIntToCheck$) have been bound to the same o. Thus, $SameObject$ is only applicable in cases, in which the basic event and the begin event of the monitoring interval represent method events.

Negation events base on the predefined metaclass $Negation$:

```
'CLASS' <Negation> 'METACLASS Negation'
['OWNTYPE OBJECTTYPE'
    'INTERFACE METHODS'
        <guardFunc>'():BOOL;'

    'IMPLEMENTATION METHODS'
        <guardFunc>'():BOOL;'
        '{'
            <method>
        '};'
'END;']

'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <Negation>'->Init('<basicEvent>','<monitInt>','<guardFunc>')'
'END;'
```

The non-terminal symbols denote:

$< Negation >$	unique name of the negation event
$< guardFunc >$	an optional user-defined guard-function; if the OWNTYPE is missing, only a predefined guard-function can be indicated
$< method >$	method body implementing the guard-function; a boolean value has to be returned
$< basicEvent >$	arbitrary event that represents the negation's basic event
$< monitInt >$	monitoring interval on which the negation is basing

Example: Continuing the above example, the bookseller's task of keeping track of orders is supported by means of the negation event $DelayOfDelivery$:

```
CLASS DelayOfDelivery METACLASS Negation
INSTTYPE OBJECTTYPE
    INTERFACE

    IMPLEMENTATION
END;
INIT DelayOfDelivery->Init(ShipOrderEvent,OrderExecInt,'SameObject')
END;
```

Embedding of metaclass *Negation* in schema *Rules* is trivial, since *Negation* is an instance of the standard metaclass *Metaclass*, which does not specify any active capabilities. Negation event instances do not bind parameters additional to *timeStamp*. Negation event classes are not parameterized.

5.2.1.4 Events with Associated Monitoring Intervals

An event with an associated monitoring interval consists of a basic event, a monitoring interval and a guard-function. A monitored event is raised, when the basic event occurs during the monitoring interval. The guard-function associated with the monitored event guarantees compatibility between a basic event instance and a monitoring interval on.

Example: The bookseller sends reminders periodically from the point in time, a customer is in delay for payment for a sale $sale_i$, until the point in time, payment for $sale_i$ is made. The periodic sending of reminders represents a monitored event basing on a sale-specific monitoring interval; i.e., the underlying monitoring interval may be on for one sale, it may be off for another one. The guard-function defined with the monitored event has to ensure, that a reminder is sent only for an open sale; i.e., for a sale for which the monitoring interval is on.

Two predefined guard-functions are available: *AnyObject*, which evaluates to true in any case, and *SameObject*, which evaluates to true only, if the basic event instance (*evntToCheck*) and the begin-event instance of a monitoring interval on (*monitIntToCheck*) have bound the same *o*. Thus, *SameObject* is only applicable in cases, in which the basic event and the begin event of the monitoring interval represent method events.

Monitored events base on the predefined metaclass *MonitEvnt*:

```
'CLASS' <MonitEvent> 'METACLASS MonitEvnt'
['OWNTYPE OBJECTTYPE'
    'INTERFACE METHODS'
        <guardFunc>'():BOOL;'

    'IMPLEMENTATION METHODS'
        <guardFunc>'():BOOL;'
```

```
        '{'
            <method>
        '};'
'END;']

'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <MonitEvent>'->Init('<basicEvent>','<monitInt>','<guardFunc> ')'
'END;'
```

The non-terminal symbols denote:

$< MonitEvent >$	unique name of the monitored event
$< guardFunc >$	an optional user-defined guard-function; if the OWNTYPE is missing, only a predefined guard-function can be indicated
$< method >$	method body implementing the guard-function; a boolean value has to be returned
$< basicEvent >$	arbitrary event that represents the monitored event's basic event
$< monitInt >$	monitoring interval on which the monitored event is basing

Example: Continuing the above example, the monitored event *ReminderToSendEvent* is raised, if an instance of the basic event *ReminderEvent* (cf. example utilized to describe relative time events) is considered to be compatible with the monitoring interval *OpenSaleInt*. The delimiters of *OpenSaleInt*, which is sale-specific, are *InDelayForPayment* (cf. example used to illustrate abstract events) and *SalePayed*. The guard-function *SameSale* guarantees that an instantiation of *ReminderEvent* is compatible with the monitoring interval *OpenSaleInt*, if the same sale is concerned.

```
CLASS SalePayed METACLASS MethEvntOnInst
    OWNTYPE OBJECTTYPE
        INTERFACE METHODS
            Raise(newEvnt: OID);

        IMPLEMENTATION METHODS
            Raise(newEvnt: OID);
            {
            };
    END;

    INSTTYPE OBJECTTYPE
        INTERFACE
        IMPLEMENTATION
    END;
INIT SalePayed->Init(Sale,'RecordPayment',AFTER)
END;
```

```
CLASS OpenSaleInt METACLASS MonitInt
INSTTYPE OBJECTTYPE
    INTERFACE
        SameSale():BOOL;

    IMPLEMENTATION
        SameSale():BOOL;
        {
            VAR saleNo: INT;
            VAR openSale: OID;

            saleNo:=begEvntToCheck->(INT)SYS_saleNo();
            openSale:=Sale->GetSaleWithId(saleNo);

            RETURN endEvntToCheck->(OID)SYS_o() == openSale;
        };
END;
INIT OpenSaleInt->Init(InDelayForPayment,SalePayed,'SameSale')
END;


CLASS ReminderToSendEvent METACLASS MonitEvnt
INSTTYPE OBJECTTYPE
    INTERFACE
        SameSale():BOOL;

    IMPLEMENTATION
        SameSale():BOOL;
        {
            VAR saleNo: INT;
            VAR startEvnt: OID;

            startEvnt:=evntToCheck->(OID)SYS_evntInst();
            saleNo:=startEvnt->(INT)SYS_saleNo();

            RETURN monitIntToCheck->(OID)SYS_beginEvnt()->
                   (INT)SYS_saleNo() == saleNo;
        };
END;
INIT ReminderToSendEvent->Init(ReminderEvent,OpenSaleInt,'SameSale')
END;
```

Embedding of metaclass *MonitEvnt* in schema *Rules* is trivial, since *MonitEvnt* is instance of the standard metaclass *Metaclass*, which does not specify any active capabilities. A monitored event instance binds the same *timeStamp* as the triggering basic event instance. If the basic event instance has bound the embracing transaction's identifier (*transId*), the transaction identifier is passed along to the monitored event instance. Monitored event classes are not parameterized.

5.2.2 Conditions and Actions

In ActiveVODAK, condition evaluation and action execution are realized by means of parameterless methods. A condition class specifies the method that implements the condition to be evaluated; an action class implements the method for action execution. Neither condition classes nor action classes have instances.

The database context reachable from methods that implement conditions and actions comprises:

- *database entry points*, i.e., class names, and all objects reachable by starting navigation at these class objects; and
- *parameters* bound by the event instance triggering condition evaluation and action execution. Since object identifiers may be among the parameters bound, all objects reachable by starting navigation at these OID-parameters are accessible, too. By performing a method named *Event()* on the condition/action class, the triggering event instance can be determined.

  Event instances of relative time events, of periodic time events, and of complex events are structured (i.e., they include instances of other event classes). Mechanisms must exist to access the components of a structured event instance: A property *evntInst* is defined for instances of relative and periodic time event classes, which represents the identifier of the event instance, to which a relative time event instance is relative, or to the start event instance of a periodic time event instance. *EvntInst* can be accessed like a plain event parameter. For instances of conjunctions and sequences a method *GetSubEvntOfCls(className)* is defined, which returns the basic event instance of class *className*. Since a single event class (*className*) must not be included in a sequence's or conjunction's list of basic event classes more than once, the basic event instnance returned by *GetSubEvntOfCls(className)* is unique.

5.2.2.1 Conditions

Conditions in ActiveVODAK are *simple conditions*, i.e., they return a boolean value (cf. section 3.2.3). Further, conditions do not bind additional parameters. User-defined conditions are instances of the predefined metaclass *Condtn*:

```
'CLASS' <Condition> 'METACLASS Condtn'
'OWNTYPE OBJECTTYPE'
    'INTERFACE METHODS'
        <condMeth>'():BOOL READONLY;'

    'IMPLEMENTATION METHODS'
        <condMeth>'():BOOL READONLY;'
        '{'
            <method>
```

```
        '};'
'END;'

'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <Condition>'->Init('<condMeth>')'
'END;'
```

The non-terminal symbols denote:

$< Condition >$	unique name of the condition
$< condMeth >$	name of the method implementing the condition
$< method >$	method body implementing the condition; a boolean value has to be returned; the database context accessible must not be modified (READONLY!)

Example: The bookseller's inventory control policy is to initialize a reorder process, if the number of a book's copies in stock falls below a threshold after a sale. Condition $ThresholdCheck$ evaluates to true, if the amount in stock is below the given threshold and if no reorder process has yet been initiated. The reorder policy is only valid for books, i.e., the actual kind of a publication which is subject of a sale has to be determined: Performing method *AsSpec* on an instance of class *Publication* returns the instance's specialized representation, an instance of class *Book* or an instance of class *Journal*.

```
CLASS ThresholdCheck METACLASS Condtn
OWNTYPE OBJECTTYPE
    INTERFACE METHODS
        BelowThreshold():BOOL READONLY;

    IMPLEMENTATION METHODS
        BelowThreshold():BOOL READONLY;
        {
            VAR publication: OID;
            VAR book: OID;
            VAR threshold: INT;
            VAR itemsOnStock: INT;
            VAR itemsOrdered: INT;

            publication:=SELF->(OID)Event()->(OID)SYS_o();
            book:=publication->(OID)AsSpec();
            IF (book->(OID)class()==Book) {
                threshold:=publication->(INT)SYS_threshold();
                itemsOnStock:=publication->(INT)SYS_itemsOnStock();
                itemsOrdered:=publication->(INT)SYS_itemsOrdered();
                RETURN itemsOnStock<threshold AND
                       itemsOrdered==0; };
            RETURN FALSE;
        };
```

```
END;

INSTTYPE OBJECTTYPE
    INTERFACE
    IMPLEMENTATION
END;
INIT ThresholdCheck->Init('BelowThreshold')
END;
```

5.2.2.2 Actions

Actions in ActiveVODAK can refer to an arbitrary set of methods. Furthermore, within action execution it is possible to abort the current transaction and also to raise abstract events explicitly. User-defined actions are instances of the predefined metaclass *Actn*:

```
'CLASS' <Action> 'METACLASS Actn'
'OWNTYPE OBJECTTYPE'
    'INTERFACE METHODS'
        <actnMeth>'();'

    'IMPLEMENTATION METHODS'
        <actnMeth>'();'
        '{'
            <method>
        '};'
'END;'

'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <Action>'->Init('<actnMeth>')'
'END;'
```

The non-terminal symbols denote:

$< Action >$	unique name of the action
$< actnMeth >$	name of the method implementing the action
$< method >$	method body implementing the action

Example: If condition $ThresholdCheck$ evaluates to true, a reorder process is initiated, i.e., a new order is composed and placed with the publication's supplier:

```
CLASS Reorder METACLASS Actn
OWNTYPE OBJECTTYPE
    INTERFACE METHODS
        InitiateReorder();
```

```
        IMPLEMENTATION METHODS
            InitiateReorder();
            {
                VAR publication: OID;
                VAR newOrderNo: INT;
                VAR newOrder: OID;
                VAR supplier: STRING;

                publication:=SELF->(OID)Event()->(OID)SYS_o();
                supplier:=publication->(STRING)SYS_supplier();
                newOrderNo:=Order->GetNewOrderId();
                newOrder:=Order->NewOrder(newOrderNo,supplier,
                                          publication,REORDERQUNTY);
                newOrder->Place();
            };
    END;

    INSTTYPE OBJECTTYPE
        INTERFACE
        IMPLEMENTATION
    END;
    INIT Reorder->Init('InitiateReorder')
    END;
```

5.2.3 Rules

A rule is an aggregation of three components: an event, a condition, and an action, where each component is defined independently. Additionally, a rule specifies the coupling between event instantiation and condition evaluation on the one hand, and between event instantiation and action execution on the other. The three different coupling modes supported in ActiveVODAK—immediate, deferred, and independent—are discussed in detail in subsection 5.3.4.

User-defined rules are instances of the predefined metaclass *Rule*:

```
'CLASS' <Rule> 'METACLASS Rule'
'INSTTYPE OBJECTTYPE'
    'INTERFACE'
    'IMPLEMENTATION'
'END;'
'INIT' <Rule>'->Init('<Event>','<Condition>','<ECcoupl>','
                      <Action>','<EAcoupl>')'
'END;'
```

The non-terminal symbols denote:

$< Rule >$	unique name of the rule
$< Event >$	event, on which $< Rule >$ is basing
$< Condition >$	condition, on which $< Rule >$ is basing
$< ECcoupl >$	coupling between event instantiation and condition evaluation; supported coupling modes: 'imm', 'def', and 'ind'
$< Action >$	action, on which $< Rule >$ is basing
$< EAcoupl >$	coupling between event instantiation and action execution; supported coupling modes: 'imm', 'def', and 'ind'

Example: The bookseller's inventory control policy can be realized by means of rule *InventoryControl*. The event component *SellEvent*, the condition component *ThresholdCheck*, and the action component *Reorder* of *InventoryControl* have already been introduced in this section. Condition evaluation is performed immediately after event instantiation. If the condition evaluates to true, the action is executed immediately, too.

```
CLASS InventoryControl METACLASS Rule
INSTTYPE OBJECTTYPE
    INTERFACE
    IMPLEMENTATION
END;
INIT InventoryControl->Init(SellEvent,ThresholdCheck,'imm',
                            Reorder,'imm')
END;
```

To provide for flexibility, rules can be deactivated (and reactivated) by applications. A deactivated rule does not react on instantiations of the event associated, i.e., event instantiations are ignored. By invoking method *DeActivate()* on class *InventoryControl*, the bookseller's inventory control policy is suspended until deactivation is revoked by means of method *Activate()*. Deactivation (and reactivation) of rules is discussed in more detail in subsection 5.3.2.

5.3 Execution Model

This section discusses diverse aspects of rule execution in ActiveVODAK. Explanations start with a description of the event instantiation mechanism. Since event instances potentially contribute to several instantiations of a single higher-level complex event, this section presents the consumption strategy for event instances. Further explanations concern the temporal and transactional scope of rules. The possibility to deactivate (and reactivate) rules causes situations that have to be looked at more closely. The multi-user characteristics of ActiveVODAK deserve special attention, too. Descriptions of the coupling modes supported and of the strategy adopted to cope with multiple rule firings conclude the section.

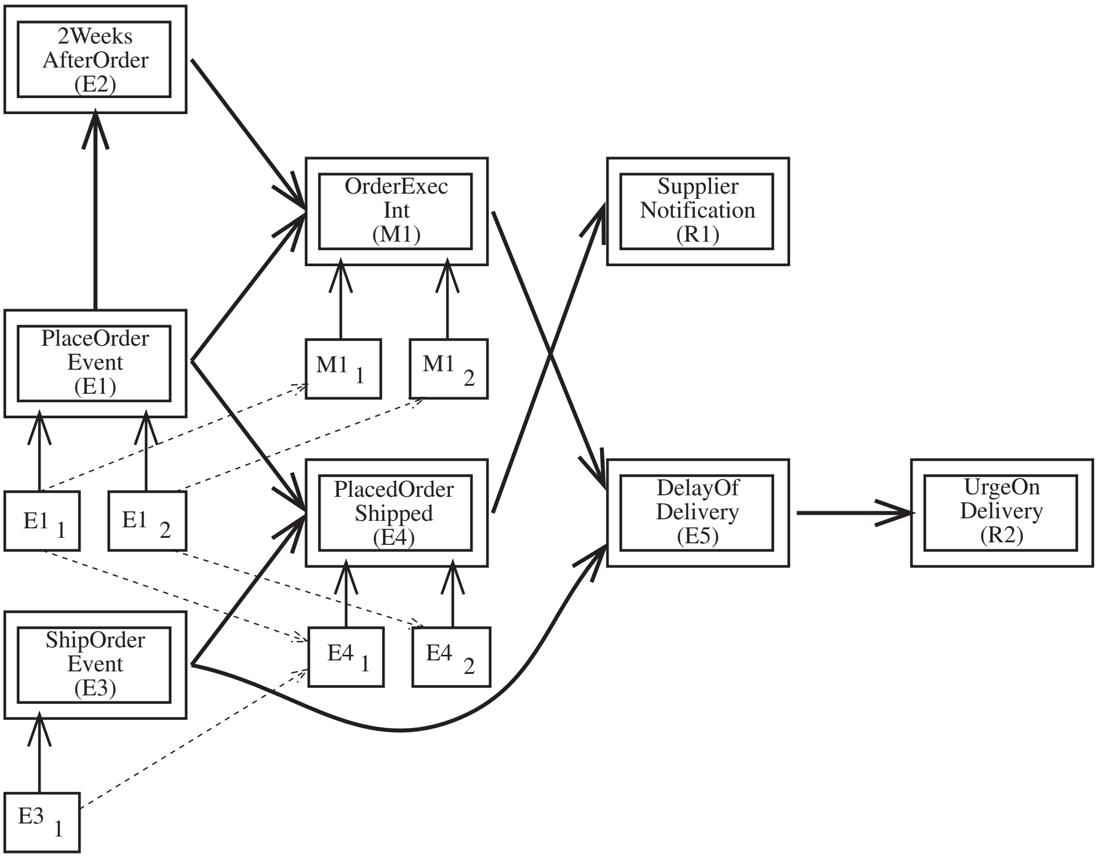


Figure5.6: Subscription relationship

5.3.1 Event Instantiation and Event Consumption

Upon event instantiation, a series of classes realizing active capabilities has to be informed about instantiation. These classes—relative time events, periodic time events, complex events, monitored events, monitoring intervals, and rules—are referred to as *subscribers*. Figure 5.6 shows subscription relationships existing between the classes defined to support the bookseller's task of keeping track of orders (cf. section 5.2).

In Figure 5.6, thick arrows symbolize subscription relationships, dashed arrows express aggregation relationships between event instances[6], and plain arrows indicate an instance-of relationship. $E1$, $E2$, and $E3$ represent primitive events; $E4$ and $E5$ are complex events; $M1$ represents a monitoring interval; and $R1$ and $R2$ are rules. Condition classes and action classes are disregarded. In the following, a single event instance is referred to as Ex_i.

Event detection in ActiveVODAK proceeds by an *incremental* strategy, i.e., complex events are detected step by step. Complex event instances that have not yet occurred completely are called *incomplete instances*. In Figure 5.6, $E4_2$ represents an incom-

[6]Both, subscription relationships and aggregation relationships represent directed acyclic graphs.

plete instance; no instance of class $E3$ is available to make it complete. When $E3$ is instantiated, the event detector checks, whether an existing incomplete instance of $E4$ is compatible with the new event instance. If this is the case, this incomplete instance is completed[7]; otherwise, a new incomplete instance is created. Thus, each time a primitive event occurs, the incremental event detector makes a "step forward" in the detection of complex events. A comparable event detection mechanism based on Petri nets is described in [GD94][8].

Another characteristic of the adopted event detection strategy is that a single event instance can contribute to only *one* instantiation of a specific complex event. Event instances are *consumed*.

Example: Irrespective of the guard-function defined for the sequence $E4$, a single instantiation of $E3$ or $E1$ can contribute to only one instance of $E4$. Imagine the following history H of primitive event instantiations:

$$H = (E1_1, E1_2, E3_1, E3_2)$$

History H results in two instantiations of $E4$: $E1_1$ and $E3_1$ are consumed by $E4_1$, whereas $E1_2$ and $E3_2$ are consumed by $E4_2$.

Another alternative—not adopted in ActiveVODAK—would have been to consider all possible pairs of primitive event instances. In this case, four instances of $E4$ would have been created: $E4_1 = (E1_1, E3_1)$, $E4_2 = (E1_2, E3_1)$, $E4_3 = (E1_1, E3_2)$, $E4_4 = (E1_1, E3_2)$. A prerequisite for this approach is that event instances are never consumed.

5.3.2 Temporal Scope of Rules

Rules and their components are part of a database schema. They are defined at schema definition time and remain valid throughout the database's lifetime. Thus, in general, the temporal scope of rules and rule components corresponds to the lifetime of the database for which they are defined. The possibility to deactivate and reactive rules (cf. section 5.2.3) may cause two special situations, for which conceptual decisions had to be made:

1. *Deactivation of a rule between rule firing and condition evaluation/action execution:* Since condition evaluation and/or action execution can be deferred to a later point in time (cf. section 5.3.4), a rule may be deactivated between rule firing and rule execution. Figure 5.7 shows rule $R1$, which is fired at t_1 and which should be evaluated/executed at t_3. At t_2 rule $R1$ is deactivated.

   Intuitive solutions for this problem are: rule execution is performed or rule execution is cancelled. Since a rule represents an integrated whole, a rule fired should be executed, too. Therefore, the first alternative—rule execution despite rule deactivation—has been adopted in ActiveVODAK.

[7] If several incomplete instances reveal compatible, the *oldest* instance is completed.

[8] An introduction to Petri nets, for example, is given in [Bau90].

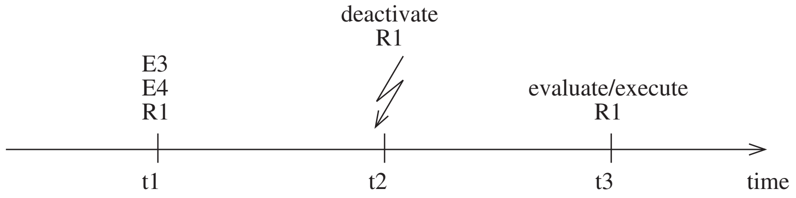


Figure5.7: Deactivation of a rule fired

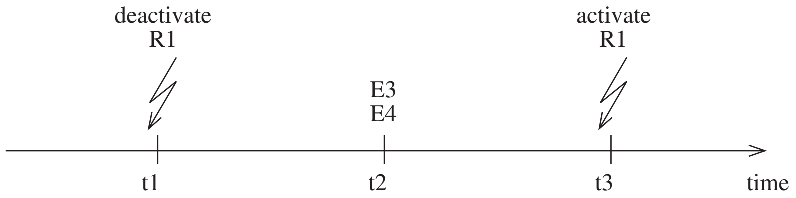


Figure5.8: Activation of a rule

2. *Reactivation of a rule (temporarily deactivated), for which the triggering event has been instantiated:* Since the triggering event of a temporarily deactivated rule may be instantiated, the question arises, whether these event instances should be considered at rule reactivation. Figure 5.8 shows an instantiation of $E4$ at t_2; at t_3 rule $R1$ is reactivated.

   Since event instantiation and rule firing represent a logical unit, they should coincide, too. Events that occurred during a period of deactivation are not considered in ActiveVODAK, they are ignored. Activation of a rule affects future instantiations of the associated event only. In the case of rules that are based on complex events, only the instantiation of the last contributing basic event has to occur after reactivation of the rule.

5.3.3 Transactional Scope of Rules

Programs working on a database engage transactions to manipulate database objects. ActiveVODAK, as a database management system, allows multiple transactions to work on a database concurrently. Consequently, basic event instantiations that contribute to a single complex event instantiation may stem from different transactions. Figure 5.9 illustrates the potential problem arising from executing concurrent transactions. To some extent, the engaged example is lacking sense: Placement of an order and shipment of the same order will seldom be effected by interleaved transactions. But, the example is well-suited to demonstrate the fundamental problem.

The two top-level transactions $T1$ and $T2$ are executed concurrently. Among other actions, $T1$ places a new order (cf. Figure 5.6) within subtransaction $T1_i$. *PlaceOrderE-*

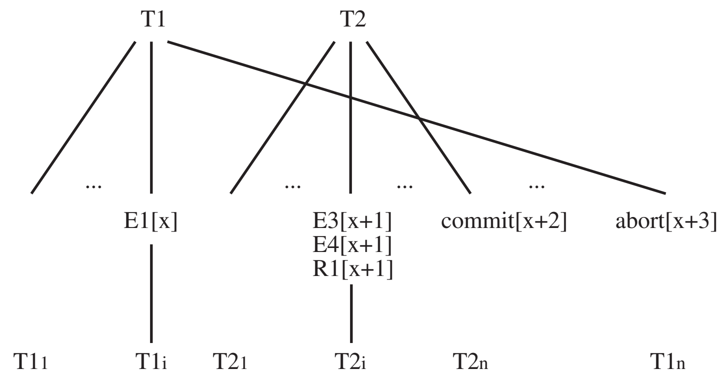


Figure5.9: Parallel top-level transactions

vent (E1) is instantiated at point in time $[x]$. Within subtransaction $T2_i$ shipment of the order is recorded. *ShipOrderEvent (E3)* and, consequently, the complex event *PlacedOrderShipped (E4)* are instantiated at point in time $[x + 1]$; rule $R1$ is fired. At point in time $[x + 2]$, the top-level transaction $T2$ is committed, i.e., all database modifications effected by $T2$—including the instantiation of $E4$ and the firing of $R1$—become irrevocable. Aborting transaction $T1$ at point in time $[x + 3]$ means, that all actions performed by $T1$ are undone. Order placement and instantiation of $E1$ are cancelled.

Since the instantiation of $E1$ has been a basic requirement for the instantiation of $E4$, instantiation of $E4$ would have to be revoked, too. But, revocation of $E4$ contradicts the semantics of the commit operation which terminated transaction $T2$. The instantiation of a complex event, whose basic events have been raised within different transactions, results in a dilemma! Alternatives, to avoid such a dilemma in advance, are:

1. ActiveVODAK is restricted to a single-user system, which does not allow multiple transactions to be executed concurrently: Since concurrency control is a mandatory feature of database management systems, this alternative does not represent a real solution.

2. Complex events are required to be entirely instantiated within one transaction: A complex event is instantiated only, if all basic events have been instantiated within the same transaction. This alternative considerably restricts the modelling capabilities of rules. Inter-transaction rules cannot be defined.

3. The transaction model of ActiveVODAK is extended with the concept of a "delayed" commit: A transaction T_i commits only, if all *contributing transactions* (i.e., transactions that directly or indirectly have contributed to rule firings within T_i) have already committed. If a contributing transaction aborts, T_i must abort, too. Referring to the example illustrated in Figure 5.9, this means that transaction $T2$ is allowd to commit only, if the contributing transaction $T1$ has committed. Since $T1$ aborts, $T2$ has to abort, too. A "delayed" commit strategy often results in cascading transaction aborts.

4. A transaction is prevented from raising a complex event that has partially been raised by another transaction not yet committed: This alternative can be realized by means of the object locking mechanism of ActiveVODAK. Complex events are raised by signalling instantiation of a basic event to the complex event class, i.e., by performing a method on the complex event class. Before a method can be executed on an object, a lock on the object has to be requested. The lock is granted only, if no other transaction already holds a lock. Referring to the example illustrated in Figure 5.9, $T1$ requests a lock on the complex event class $E4$ when $E1$ is instantiated. The lock is granted, since no other transaction holds a lock on $E4$. When $E3$ is instantiated in $T2$, the lock request on $E4$ is denied, since $T1$ already holds a lock. $T2$ has to await termination of $T1$. When transaction $T1$ aborts, instantiation of $E1$ is undone and the lock on $E4$ is released. The following instantiation of $E3$ does not raise $E4$.

Alternative 4 is realized in ActiveVODAK. Alternative 4 retains the multi-user capabilities of ActiveVODAK, it does not necessitate conceptual extensions, and it does not restrict modelling capabilities.

5.3.4 Coupling Modes

ActiveVODAK supports three different temporal and transactional relationships between event instantiation and condition evaluation (action execution): *immediate*, *deferred*, and *independent*[9]. Each coupling mode can be defined between event occurrence and condition evaluation and between event occurrence and action execution.

- *Immediate coupling:* The condition evaluation (action execution) is initiated immediately after the event occurrence. The method implementing the condition evaluation (action execution) is performed within the triggering transaction. The prerequisite for immediate coupling is that the triggering event possesses transaction binding.
- *Deferred coupling:* The condition evaluation (action execution) is initiated when the triggering transaction signals its ready-to-commit status. The method implementing condition evaluation (action execution) is performed within the triggering transaction. Like for immediate coupling, the prerequisite for deferred coupling is that the triggering event possesses transaction binding.
- *Independent coupling:* The condition evaluation (action execution) is initiated immediately after the event occurrence. In contrast to immediate coupling, the condition evaluation (action execution) is performed within a separate, entirely independent top-level transaction.

[9] To support a *commit dependent* relationship between event instantiation and condition evaluation (action execution), it would have been necessary to extend the transaction model of VODAK.

C/A	immediate	deferred	independent
immediate	(I)	(II)	(III)
deferred	/	(IV)	(V)
independent	/	/	(VI)

Table5.5: Meaningful pairs of coupling modes

Since an action must always be executed after the condition evaluation has finished, only a subset of all possible pairs of coupling modes are meaningful. Table 5.5 presents all meaningful pairs of event-condition coupling and event-action coupling. A slash denotes a not meaningful combination.

1. *Pair I (immediate,immediate):* The condition is evaluated immediately after event occurrence. If the condition is satisfied, the action is executed immediately after condition evaluation. The triggering transaction is suspended until action execution terminates. Condition evaluation as well as action execution represent method invocations within the triggering transaction.

2. *Pair II (immediate,deferred):* The condition is evaluated immediately after event occurrence. If the condition is satisfied, the action is executed when the triggering transaction signals its ready-to-commit status. Condition evaluation as well as action execution represent method invocations within the triggering transaction.

3. *Pair III (immediate,independent):* The condition is evaluated immediately after event occurrence. Condition evaluation represents a method invocation within the triggering transaction. If the condition is satisfied, action execution is performed as an independent top-level transaction.

4. *Pair IV (deferred,deferred):* The condition is evaluated when the triggering transaction signals its ready-to-commit status. If the condition is satisfied, the action execution is initiated immediately after condition evaluation. Both, condition evaluation and action execution represent method invocations within the triggering transaction.

5. *Pair V (deferred,independent):* The condition is evaluated when the triggering transaction signals its ready-to-commit status. Condition evaluation represents a method invocation within the triggering transaction. If the condition is satisfied, action execution is performed as an independent top-level transaction.

6. *Pair VI (independent,independent):* Immediately after the event occurred, an independent top-level transaction T_i is initiated for condition evaluation. If the condition is satisfied, action execution is performed within the same top-level transaction T_i.

5.3.5 Multiple Rule Firings

On instantiation of an event multiple rules may be fired. The set of rules fired may comprise both, rules that are fired *directly* (since they are based on the event class instantiated), and rules that are fired *indirectly* (since they are based on complex event classes, to which the event class instantiated contributes). In rule firing, there may be different levels of indirection. The acyclic character of event hierarchies (cf. subsection 5.3.1) ensures that the number of indirections always is finite. The conflict resolution mechanism adopted by ActiveVODAK ensures that rules of a lower level of indirection are executed before rules of a higher level of indirection. Thus, rule execution starts with directly fired rules. Rules of the same level of indirection are executed in a non-deterministic order.

5.4 Architecture

This section explains the extensions of a VODAK process to provide active capabilities. The additional tasks that have to be dealt with by ActiveVODAK are *rule management* and *event detection*. Multiple concepts to cope with these new tasks have been considered. This section motivates the choices made by comparing the pros and cons of different approaches.

Figure 5.10 illustrates a non-distributed ActiveVODAK environment. Contrary to Figure 4.3, describing a passive VODAK environment, Figure 5.10 abstracts from mapping VODAK objects to objects of the underlying storage system. The underlying storage system is transparent for the extension of VODAK with active capabilities. Communication among the different modules constituting an ActiveVODAK process is disregarded, too. The components that have been attached in ActiveVODAK (emphasized by means of thickly edged boxes in Figure 5.10) are:

1. *Schema "Rules":* Each active database has to include schema "Rules". Within schema "Rules" all metaclasses necessary to define rules and their components are defined. Besides factoring out common structure and behaviour of their instances and of the instances of their instances, the metaclasses *MethEvnt* and *TimeEvnt* detect primitive (method and time) events. Every time, the method execution notifier signals method execution, class *MethEvnt* determines whether a method event is defined for the method execution. If so, the method event is raised. Similarly, every time the clock process sends a time pulse to the database system, class *TimeEvnt* determines whether a time event is defined for the point in time that occurred.

2. *Method execution notifier:* The method execution notifier, an extension of the *MessageHandler*, informs an active database about execution of a method.

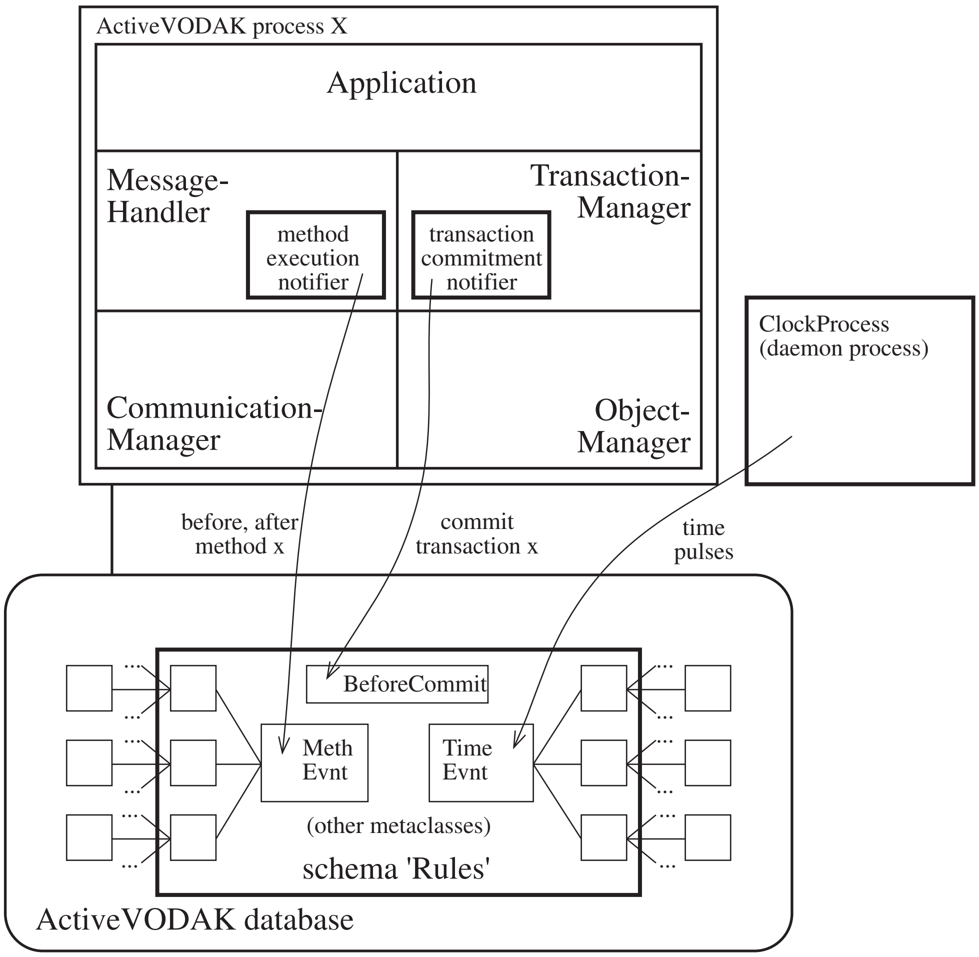


Figure5.10: ActiveVODAK process

3. *Transaction commitment notifier:* The transaction commitment notifier, an extension of the *TransactionManager*, informs an active database about a transaction that has achieved a ready-to-commit status.

4. *Clock process:* The clock process, a full VODAK process, periodically sends time pulses to an active database.

The following subsections describe rule management and event detection in detail.

5.4.1 Rule Management

The task of rule management can be divided into three subtasks:

- To maintain rules and rule components in the database system.
- To incrementally detect composite events (cf. subsection 5.3.1).
- To execute rules fired (cf. section 5.3)

Incremental detection of composite events and rule execution have already been described in section 5.3. Here, rule maintenance in general and mapping from events to rules in special are considered. Should rule-event relations be managed by a central system component, or should each rule know the event on which it is based?

Every time an event is instantiated, the set of rules fired has to be determined. Therefore, a mapping of events to rules has to be maintained. Two alternative approaches to maintain the relations between events and rules have been considered during the design process of ActiveVODAK:

1. *Centralized rule management:* Mapping of events to rules is realized by means of an additional system component—a *rule manager*. Instantiation of any event is signalled to the rule manager which determines the rules to be executed. In the HiPAC system (cf. [Day88, MD89, DBB+88, CN90, HLM88]), where this central approach has been adopted, the rule manager fulfills the following tasks:

   - It determines the rules fired in consequence of an event instantiation (mapping of events to rules).
   - It groups the rules fired according to coupling modes (immediate, deferred, detached but causally dependent, detached and causally independent).
   - It provides for rule execution respecting priorities.

Employing a central rule manager represents an efficient way to deal with inter-rule dependencies. I.e., a rule manager can act as a scheduler in the case, multiple rules with different priorities are fired by a single event instantiation. On the other hand, a global mapping of events to rules may be inefficient if the number of events and rules is high.

2. *Decentralized rule management:* In a decentralized approach each event "knows" which rules to notify in the case of instantiation. In Sentinel (cf. [CAM93]), for example, rules are event subscribers (i.e., rules subscribe events to be informed about event instantiation). A rule fired executes autonomously.

   Decentralized rule management is efficient, since no mapping tables have to be looked up on event instantiation. Handling of multiple rule firings proves to be more complicated, because a coordinating facility is missing.

Rule management in ActiveVODAK is decentralized. A rule is informed about event instantiation by its underlying event. Upon firing, a rule executes autonomously. Inter-rule dependencies cannot be resolved after rule firing. The metaclasses defined in schema "Rules" provide all the rule management capabilities of ActiveVODAK. Implementational details of rule management are described in chapter 6.

5.4.2 Event Detection

Mechanisms had to be established to detect method events, time events, and transaction events—the primitive events supported by ActiveVODAK. Since abstract events are raised by programs or external system components explicitly, no arrangements had to be made for their detection.

5.4.2.1 Detection of Method Events

Like rule management, method event detection can follow a centralized or a decentralized strategy:

1. *Decentralized method event detection:* A method event is raised directly by the method for which it is defined. Raising of the method event represents either the first (i.e., the modifier flag is *before*) or the last (i.e., the modifier flag is *after*) action within the method body. This strategy has been adopted in the design of Ode (cf. [GJ91, GJS92b]) or Sentinel (cf. [CAM93]). Decentralized method event detection is efficient since no administrative overhead is necessary. On the other hand, it proves to be inflexible since it requires recompilation of method code if method events are modified (e.g., from before to after), added, or deleted.

2. *Centralized method event detection:* Upon execution of any method, a central system component determines whether a method event has to be raised (cf. HiPAC

[Day88]). Centralized method event detection necessitates additional administrative overhead; only a small number of method executions actually results in method events. On the other hand, a centralized detection is more flexible, no recompilation of method code is necessary.

Method event detection in ActiveVODAK is centralized. The metaclass *MethEvnt* (cf. Figure 5.2) plays the role of a central "system component". Start and completion of any method execution are signalled to *MethEvnt* by the *method execution notifier*, an extension of the message handler. *MethEvnt* determines whether a method event has to be raised.

5.4.2.2 Detection of Time Events

Time events cannot be detected autonomously by the database system. A separate component—basing on the system clock—has to send time pulses periodically to the database system. In ActiveVODAK this task is performed by the *clock process*. The clock process not only serves a single active database, but it periodically sends time pulses to all active databases located on the clock process' node. Since the clock process has to perform methods on database objects, it has to represent a complete VODAK process with transaction management, object management, and message handling.

Time pulses are sent to the metaclass *TimeEvnt* (cf. Figure 5.3). *TimeEvnt* determines, whether user-defined absolute time events, relative time events, or periodic time events have to be raised.

5.4.2.3 Detection of Transaction Events

Transaction events are exclusively used to realize deferred coupling modes (cf. subsection 5.3.4). Every time a top-level transaction reaches its ready-to-commit status, the *transaction commitment notifier*—an extension of the transaction manager—sends a signal to the *BeforeCommit* class of schema *Rules*. Class *BeforeCommit* informs all rules, for which deferred event-condition coupling or deferred event-action coupling has been specified, about transaction T, which is ready to commit. Then, all the rules awaiting commitment of transaction T can initiate condition evaluation and/or action execution.

Chapter 6

Implementation

Based on the conceptual descriptions of chapter 5, this chapter presents the implementation of ActiveVODAK. The explanations of this chapter concern the extensions of VODAK to provide active capabilities. The extensions—identified in section 5.4—are: the schema "Rules", the method execution notifier, the transaction commitment notifier, and the clock process.

This chapter does not represent the entire implementation code. Rather, it concentrates on parts crucial for understanding the functionality of ActiveVODAK.

6.1 Schema "Rules"

Metaclasses are defined for the different event classes (primitive and complex events, events with associated monitoring intervals), for monitoring intervals, for conditions and actions, and for rules. Embedding of rules and rule components into schema "Rules" has already been described in section 5.2. This section describes in detail, how the metaclasses in schema "Rules" provide active capabilities by specifying structure and behaviour of their instances and of the instances of their instances.

6.1.1 Events

The predefined metaclasses of event classes specify capabilities for the administration of event instances, for the detection of events, for the maintenance of subscription relationships (cf. subsection 5.3.1), and for the release of event instances. Common properties and methods of event classes and event instances are factored out into separate object types. So, separate object types are defined for the administration of event instances, for the maintenance of subscription relationships, and for the release of event instances:

- *Event instance administration:* VODAK classes (application classes and metaclasses) administer their instances as set elements. Every time a new instance is created (by performing method *new* on the class), the new instance is inserted in the set of instances. If an instance is deleted (by performing method *delete* on the class), the instance is removed from the set of instances. Since instance creation is very time consuming, this default instance administration mechanism is not adopted in ActiveVODAK. Rather, the new object type *EvntInstAdminType* provides for a more efficient event instance administration, in which released event instances are not actually deleted. Released event instances are made only "invisible" temporally.

```
OBJECTTYPE EvntInstAdminType;
  INTERFACE
    METHODS
        InitInstList();

        // A predefined number (DEF_NUM_OF_INST) of invisible
        // instances is created.

        NewListElem():OID;

        // A new instance is made visible.

        DelListElem(elem: OID);

        // The instance <elem> is made invisible.

  IMPLEMENTATION
    ...

END; // EvntInstAdminType
```

Method *InitInstList* must be called once at class initialization. It creates the number of invisible instances defined by the global constant *DEF_NUM_OF_INST*. The default instance creation method *new* is replaced by *NewListElem*. If an invisible instance is available, this instance is made visible. If the number of invisible instances is exhausted, before making an instance visible, the number of invisible instances defined by the global constant *DEF_INC_OF_INST* is created. The default instance deletion method *delete* is substituted by *DelListElem*, which makes an instance invisible.

- *Maintenance of subscription relationships:* As introduced in section 5.3.1, relative time events, periodic time events, complex events, monitored events, monitoring intervals, and rules are referred to as event subscribers. Event subscribers are informed about event instantiation. Subscription relationships are maintained bidirectionally: each event class holds references to its subscribers, and each subscriber holds reference(s) to its event class(es). Object type *EvntClsType* implements subscription relationship maintenance for event classes:

```
OBJECTTYPE EvntClsType;
  INTERFACE
    METHODS
        AddRuleSubscr(rule:OID);
```

```
            // <rule> is inserted into the set of subscribing rule
            // classes.

            AddCompEvntSubscr(compEvnt:OID);

            // <compEvnt> is insterted into the set of subscribing
            // complex event classes.

            AddTimeEvntSubscr(timeEvnt:OID);

            // <timeEvnt> is inserted into the set of subscribing
            // relative and periodic time event classes.

            AddMonitIntSubscr(monitInt:OID);

            // <monitInt> is inserted into the set of subscribing
            // monitoring interval classes.

            AddMonitEvntSubscr(monitEvnt:OID);

            // <monitEvnt> is inserted into the set of subscribing
            // monitored event classes.

            AllSubscribers():{OID} READONLY;

            // A set containing all subscribers is returned.

            InformSubscribers(evntInst:OID);

            // All subscribers are informed about the instantiation
            // (evntInst) of the event class. Method SignalEvnt(evntInst)
            // is performed on all subscribing classes. Order: rule
            // classes, complex event classes, time event classes,
            // monitoring intervals, monitored event classes

  IMPLEMENTATION
    ...

END; // EvntClsType
```

Event subscribers can be grouped into three categories: event subscribers that are based on *one* event class (rule classes, relative and periodic time event classes, monitored event classes, negation event classes), event subscribers that are based on *two* event classes (monitoring intervals), and event subscribers that are based on an arbitrary number of event classes (conjunction event classes, sequence event classes).

Object type *UnarySubscrClsType* implements subscription relationship maintenance for unary subscriber classes:

```
OBJECTTYPE UnarySubscrClsType;
  INTERFACE
```

```
      PROPERTIES
          evnt: OID;
            // event class on which the subscriber is based

    IMPLEMENTATION

  END; // UnarySubscrClsType
```

Object type *BinarySubscrClsType* implements subscription relationship maintenance for binary subscriber classes (i.e., for monitoring intervals):

```
  OBJECTTYPE BinarySubscrClsType;
    INTERFACE
      PROPERTIES
          beginEvnt: OID;
            // event class representing the begin event class of
            // the (monitoring-interval) subscriber

          endEvnt: OID;
            // event class representing the end event class of
            // the (monitoring-interval) subscriber

    IMPLEMENTATION

  END; // BinarySubscrClsType
```

Object type *N_arySubscrClsType* implements subscription relationship maintenance for n-ary subscriber classes. The property *noOfEvnts* represents the size of the list of event classes the subscriber is based on (*evnts*):

```
  OBJECTTYPE N_arySubscrClsType;
    INTERFACE
      PROPERTIES
          evnts: LIST OF OID;
            // list of event classes the subscriber is basing on

          noOfEvnts: INT;
            // number of event classes subscribed

    IMPLEMENTATION

  END; // N_arySubscrClsType
```

- *Release of event instances:* If an event instance is released by all subscribers of its class (i.e., the information bound by the event instance is not needed any more), an event instance can be deleted. Object type *EvntInstType* keeps track of the subscribers which have/have not released an event instance and defines the parameter *timeStamp*, which is bound by every event instance:

```
OBJECTTYPE EvntInstType;
  INTERFACE
    PROPERTIES
        timeStamp: AbsTime;
          // point in time of instantiation

    METHODS
        SetAllSubscribers(subscribers: {OID});

        // <toBeReleasedBy> is set to the conjunction of
        // all subscribers (<subscribers>) of the instance's class.

        ReleasedBy(subscriber: OID);

        // The subscriber (<subscriber>) releasing the event instance
        // is removed from <toBeReleasedBy>.

        IsReleased():BOOL READONLY;

        // If <toBeReleasedBy> is empty, TRUE is returned.

  IMPLEMENTATION
    PROPERTIES
        toBeReleasedBy: {OID};
          // set of subscribers that have not released the event
          // instance
  ...

END; // EvntInstType
```

The event instance property *timeStamp* is of type *AbsTime* representing an absolute point in time:

```
DATATYPE AbsTime = [year:INT, month:INT, day:INT,
                    hour:INT, minute:INT, second:INT];
```

An instance of an event subscriber maintains references to underlying event instances. Before an instance of an event subscriber can be deleted, the underlying event instances have to be released.

Object type *UnarySubscrInstType* implements the maintenance of references and the release of underlying event instances for unary subscriber instances:

```
OBJECTTYPE UnarySubscrInstType;
  INTERFACE
    PROPERTIES
        evntInst: OID;
          // event instance underlying the instance
          // of the unary subscriber class

    METHODS
```

```
            ReleaseInst(subscriber: OID);

            // <evntInst> is released by sending
            // Release(<evntInst>,<subscriber>) to the class of
            // <evntInst>. The implementation of Release is
            // event type specific.

      IMPLEMENTATION
        METHODS
            ReleaseInst(subscriber: OID);
            {
                evntInst->(OID)class()->Release(evntInst,subscriber);
            };

    END; // UnarySubscrInstType
```

Object type *BinarySubscrInstType* implements the maintenance of references and the release of underlying event instances for binary subscriber instances:

```
    OBJECTTYPE BinarySubscrInstType;
      INTERFACE
        PROPERTIES
            beginEvntInst: OID;
              // begin event instance underlying the instance
              // of the binary subscriber class

            endEvntInst: OID;
              // end event instance underlying the instance
              // of the binary subscriber class

        METHODS
            ReleaseInst(subscriber: OID);

            // <beginEvntInst> and <endEvntInst> are released by sending
            // Release to their respective classes; implementation of
            // Release is event type specific.

      IMPLEMENTATION
        METHODS
            ReleaseInst(subscriber: OID);
            {
                beginEvntInst->(OID)class()->
                               Release(beginEvntInst,subscriber);
                endEvntInst->(OID)class()->
                             Release(endEvntInst,subscriber);
            };

    END; // BinarySubscrInstType
```

Object type *N_arySubscrInstType* implements the maintenance of references and the release of underlying event instances for n-ary subscriber instances:

```
OBJECTTYPE N_arySubscrInstType;
  INTERFACE
    METHODS
        IsComplete():BOOL READONLY;

        // Checks, whether an event instance is complete (all
        // elements of the list of basic event instances are not NULL).

        IsCompleteUntil(n: INT):BOOL READONLY;

        // Checks, whether an event instance is complete until index <n>
        // (all elements of the list of basic event instances are not
        // NULL until index n).

        SetComplete(status: BOOL);

        // Sets the flag, indicating whether the instance is complete
        // or not, to <status>.

        GetSubEvntNo(no: INT): OID READONLY;

        // Returns the element of the list of basic event instances
        // with index <no>.

        SetSubEvntNo(no: INT, subEvnt: OID);

        // Sets the element of the list of basic event instances at
        // index <no> to <subEvnt>.

        ResetSubEvntNo(no: INT);

        // Resets the element of the list of basic event instances
        // with index <no> to NULL

        GetSubEvntOfCls(evntCls: OID): OID READONLY;

        // Returns the element of the list of basic event instances
        // which is of class <evntCls>

        ReleaseInst(subscriber: OID);

        // Releases all elements of the list of basic event instances
        // sending Release to their respective classes; implementation
        // of Release is event type specific

  IMPLEMENTATION
    ...

END; // N_arySubscrInstType
```

In the sequel, the following topics are considered for each event type:

1. *Object type hierarchy and class hierarchy:* The object type hierarchy and the class hierarchy implemented are illustrated. The illustration concentrates on property

and method *definitions*. Actual property values are disregarded. Property names are characterized by small, method names by capitalized initial letters. The default subtyping relationships[1] are disregarded, too.

2. *Object type interfaces:* The interfaces of the object types engaged to implement event detection, event instantiation, subscriber information, and releasing event instances are described. Method implementations are considered only, if they contribute to a better understanding. The naming conventions for object types are: The OWNTYPE of class X is denoted as *X_OwnType*, the INSTTYPE is referred to as *X_InstType*, and the INSTINSTTYPE is named *X_InstInstType*.

3. *Event detection and event instantiation:* The process of event detection and event instantiation is explained.

4. *Release of event instances:* The process of releasing event instances is explained.

6.1.1.1 Primitive Events

In the sequel, the implementation of method events, of different types of time events, and of abstract events are described.

Method Events. Depending on the extensional scope of a method event, a user-defined method event class is an instance of metaclass *MethEvntOnObject*, of metaclass *MethEvntOnInst*, or of metaclass *MethEvntOnInstInst* (cf. subsection 5.2.1.1). These metaclasses are instances of metaclass *MethEvnt*. Figure 6.1 illustrates the object type hierarchy and the class hierarchy established to support method events.

Besides determining common structure and behaviour of the instances of its instances, metaclass *MethEvnt* is utilized for method event detection. Object type *MethEvnt_OwnType* defines a single method for the metaclass object *MethEvnt*:

```
OBJECTTYPE MethEvnt_OwnType;
  INTERFACE
    METHODS
        IsEvnt(object:OID, meth:STRING):ActvMeths READONLY;

        // This method is called by the method execution notifier. All
        // event classes which have to be notified either before
        // or after method execution are returned.

  IMPLEMENTATION
    METHODS
        IsEvnt(object:OID, meth:STRING):ActvMeths READONLY;
        {
            VAR actvs:ActvMeths;
```

[1]Each instance (instance-instance) type of a metaclass represents a direct or indirect subtype of the instance (instance-instance) type associated with the system internally defined metaclass *Metaclass*.

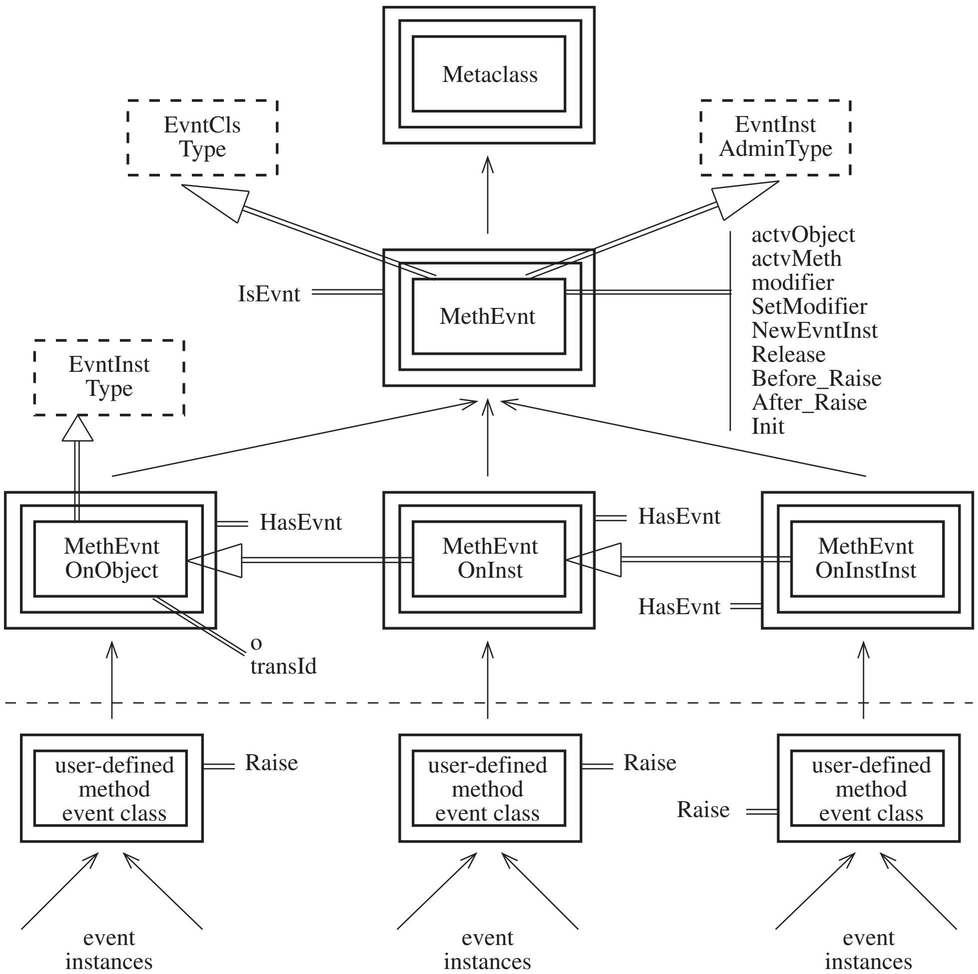


Figure6.1: Implementation of method events

```
            VAR allActvs:ActvMeths;
            VAR inst:OID;

            allActvs.before:={};
            allActvs.after:={};
            FORALL (inst IN SELF->allInstances()) {
                actvs:=inst->(ActvMeths)HasEvnt(object,meth);
                allActvs.before:=allActvs.before UNION actvs.before;
                allActvs.after:=allActvs.after UNION actvs.after; };

            RETURN allActvs;
        };

END; // MethEvnt_OwnType
```

The INSTTYPE of metaclass *MethEvnt* represents an empty type, since no common structure and behaviour of the instances of *MethEvnt* can be factored out.

Object type *MethEvnt_InstInstType*, a subtype of *EvntInstAdminType* and *EvntClsType* (see above), determines common structure and behaviour of user-defined method event classes.

```
OBJECTTYPE MethEvnt_InstInstType SUBTYPEOF Metaclass_InstInstType,
           EvntClsType, EvntInstAdminType;
  INTERFACE
    PROPERTIES
        actvObject: OID;
          // the class object or metaclass object specifying actvMeth
          // for itself, its instances or the instances of its
          // instances

        actvMeth: STRING;
          // the method for which a method event is defined

        modifier: INT;
          // should the event be raised BEFORE or AFTER method
          // execution? BEFORE = 0, AFTER = 1

    METHODS
        SetModifier(status: INT);

        // <modifier> is set to <status>

        NewEvntInst(obj: OID):OID;

        //  - new event instance is generated;
        //  - <o> is set to <obj>;
        //  - <transId> and <timeStamp> of instance are set;
        //  - <toBeReleasedBy> is set to the conjunction of
        //    all subscribers

        Release(evntInst: OID, subscriber: OID);
```

```
        // <evntInst> is released by <subscriber>; <subscriber> is removed
        // from the <toBeReleasedBy>-list of <evntInst>; if <evntInst>
        // has been released by all subscribers, <evntInst> is
        // deleted

        Before_Raise(obj: OID):OID;

        // This method is called by the method execution notifier
        // before method <Raise> is called. The only purpose is to call
        // <NewEvntInst> of the same class. The new method event
        // instance is returned to the method execution notifier.

        After_Raise(newEvntInst: OID);

        // This method is called by the method execution notifier
        // after method <Raise> is called. The only purpose is to call
        // <InformSubscribers> of the same class.

        Init(o: OID, m: STRING, b: INT);

        // The method event class is initialized:
        // <o> is assigned to <actvObject>;
        // <m> is assigned to <actvMeth>;
        // <b> is assigned to <modifier>

  IMPLEMENTATION
    ...

END; // MethEvnt_InstInstType
```

The object types *MethEvntOnObject_OwnType*, *MethEvntOnInst_OwnType*, and *MethEvntOnInstInst_OwnType* specify a single method named *HasEvnt* for the metaclasses of user-defined method event classes. The implementation of *HasEvnt* differs from on object type to the other, whereas the interfaces are equal:

```
OBJECTTYPE MethEvntOnObject_OwnType;
  INTERFACE
    METHODS
        HasEvnt(object:OID, meth:STRING):ActvMeths;

        // All instances (user-defined method event classes) which
        // are defined for the single object <object> and for method
        // <meth> are returned.

  IMPLEMENTATION
    ...

END; // MethEvntOnObject_OwnType


OBJECTTYPE MethEvntOnInst_OwnType;
  INTERFACE
    METHODS
```

```
        HasEvnt(object:OID, meth:STRING):ActvMeths;

        // All instances (user-defined method event classes) which
        // are defined for the class of <object> and for method
        // <meth> are returned.

  IMPLEMENTATION
    ...

END; // MethEvntOnInst_OwnType


OBJECTTYPE MethEvntOnInstInst_OwnType;
  INTERFACE
    METHODS
        HasEvnt(object:OID, meth:STRING):ActvMeths;

        // All instances (user-defined method event classes) which
        // are defined for the
        // metaclass of <object> and for method <meth> are returned.

  IMPLEMENTATION
    ...

END; // MethEvntOnInstInst_OwnType
```

The object types *MethEvntOnObject_InstType*, *MethEvntOnInst_InstType*, and *MethEvntOnInstInst_InstType* do not define any properties or methods. Object type *MethEvntOnObject_InstInstType*, a subtype of *EvntInstType* (see above) specifies the two parameters bound by each method event instance: the object, on which the method representing the basis for the method event was executed, and the identifier of the transaction embracing the method call. The object types *MethEvntOnInst_InstInstType* and *MethEvntOnInstInst_InstInstType*, which do not specify any additional properties or methods, are subtypes of *MethEvntOnObject_InstInstType*:

```
OBJECTTYPE MethEvntOnObject_InstInstType SUBTYPEOF Metaclass_InstInstType,
           EvntInstType;
  INTERFACE
    PROPERTIES
        o: OID;
          // object on which method <actvMeth> was executed

        transId: INT;
          // transaction embracing the execution of <actvMeth>

  IMPLEMENTATION

END; // MethEvntOnObject_InstInstType
```

In the following, the process of method event detection and instantiation and the process of releasing method event instances are explained:

1. *Detection and instantiation of method events:* Before a method is executed, the method execution notifier performs method *IsEvnt* on the metaclass *MethEvnt*. Within *IsEvnt*, all instances of *MethEvnt* (i.e., *MethEvntOnObject*, *MethEvntOnInst*, and *MethEvntOnInstInst*) are consulted (by performing method *HasEvnt* on them) to determine, whether method events are defined for method *meth* on object *object*. *IsEvnt* and *HasEvnt* return values of the record type *ActvMeths*, which is defined as follows:

   ```
   DATATYPE ActvMeths = [before: {OID}, after: {OID}];
   ```

   The record components *before* and *after* represent sets of method event classes, which have to be instantiated before/after method execution. Before actually executing a method, the method execution notifier raises each method event contained in *before*. This is done by sequentially performing the methods *Before_Raise*, *Raise*, and *After_Raise* on each element of *before*. Then, the method is actually executed. After method execution, *Before_Raise*, *Raise*, and *After_Raise* are sequentially performed on each method-event class contained in *after*. A detailed discussion on the implementation of the method execution notifier will follow in section 6.2.

2. *Release of method event instances:* When a method event class is instantiated, the new event instance's set of subscribers (which have not yet released the event instance) is set to the conjunction of all subscribers of the method event class. This is done by performing method *SetAllSubscribers* on the new event instance. Every time a subscriber releases the event instance, the subscriber performs method *Release* on the event instance's class. The event class removes the subscriber from the event instance's set of subscribers, i.e., it performs method *ReleasedBy* on the event instance. If the event instance is released by all subscribers (method *IsReleased* evaluates to true), the event instance is "deleted" (i.e., temporarily made invisible by performing method *DelListElem*).

Time Events. User-defined time event classes are instances of metaclass *AbsTimeEvnt*, of metaclass *RltvTimeEvnt*, or of metaclass *PrdcTimeEvnt* (cf. subsection 5.2.1.1). These metaclasses are altogether instances of metaclass *TimeEvnt*. Figure 6.2 illustrates the object-type hierarchy and the class hierarchy established to support time events.

Before describing the object types in detail, two central datatypes—*AbsTime* and *RelTime*—must be introduced. The datatypes *AbsTime/RelTime* are used to represent absolute/relative points in time, as they have been specified in subsection 5.2.1.1. Two record structures are defined:

```
DATATYPE AbsTime = [year:INT, month:INT, day:INT,
                    hour:INT, minute:INT, second:INT];

DATATYPE RelTime = [dYear:INT, dMonth:INT, dDay:INT,
                    dHour:INT, dMinute:INT, dSecond:INT];
```

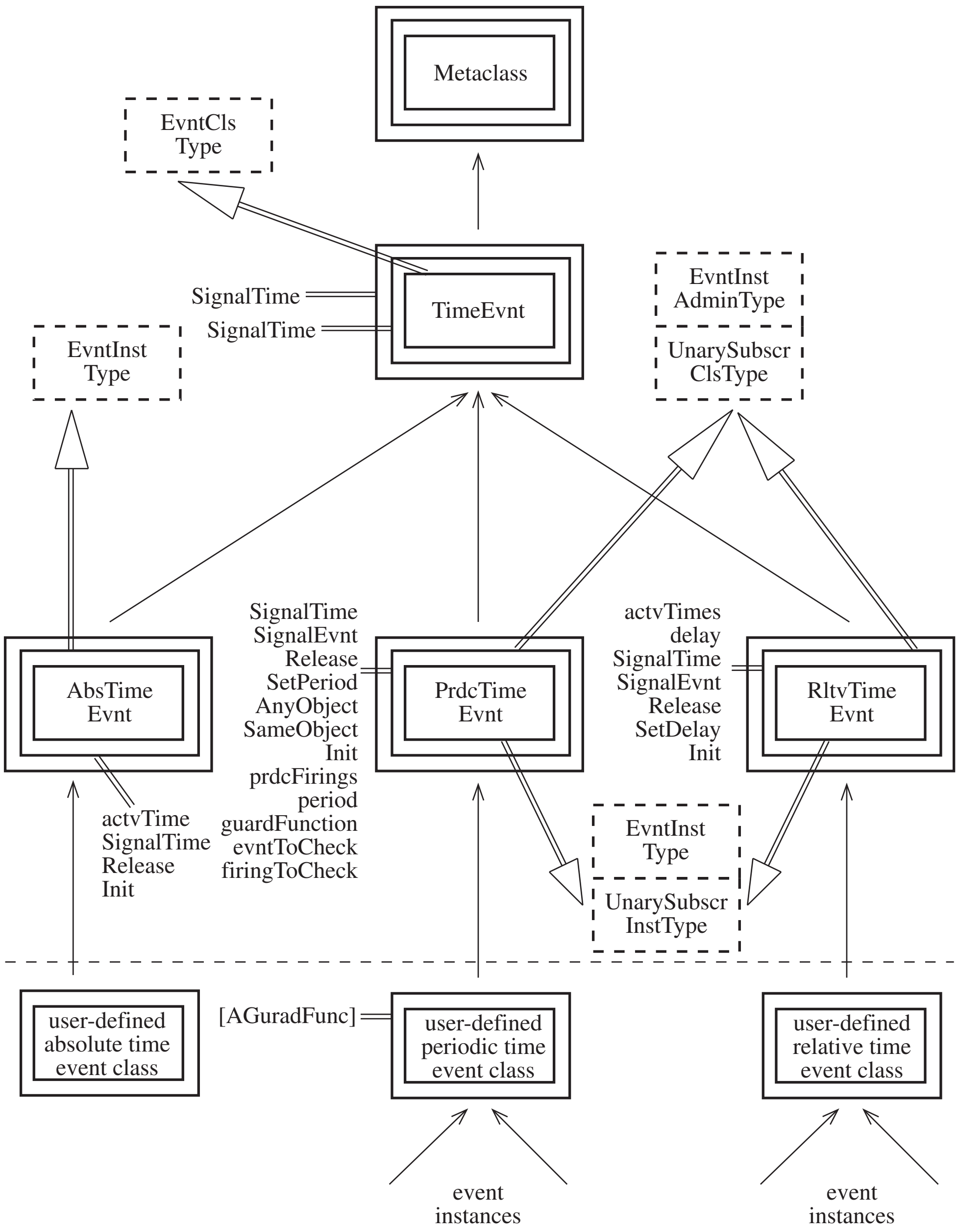


Figure6.2: Implementation of time events

When a schema designer defines time event classes, points in time are specified as strings, since strings allow a flexible way of definition. Within the UNIX operating system, which provides the platform for ActiveVODAK, points in time are not treated as *AbsTime/RelTime*-structures. The system clock, for example, treats points in time as integer values. Consequently, a series of methods are necessary to convert points in time from one kind of representation to another. In ActiveVODAK, all methods necessary for manipulating points in time are collected in class *Time*. Besides conversion between different kinds of representation, class *Time* provides the functionality for reading the current system time and for adding, comparing, and displaying points in time. Class *Time* serves only to collect methods; it does not have instances! Object type *Time_OwnType* is defined as follows:

```
OBJECTTYPE Time_OwnType;
  INTERFACE
    METHODS
        CurrentSysTime():AbsTime READONLY;

        // The current system time is returned.

        IntToAbsTime(intTime: INT):AbsTime READONLY;

        // A point in time represented as an integer value
        // (<intTime>) is converted to AbsTime.

        AsAbsTime(timeAsStr: STRING):AbsTime READONLY;

        // A time string (<timeAsStr>) is converted to AbsTime.
        // The time-string must be of the form:
        // 'JJ-MM-DD HH:MM:SS'
        // Apart from the year specification, components can
        // be omitted (defaults are assumed!).

        AsRelTime(timeAsStr: STRING):RelTime READONLY;

        // A time string (<timeAsStr>) is converted to RelTime.
        // The time-string must be of the form:
        // 'JJ-MM-DD HH:MM:SS'
        // Apart from the second specification, components can
        // be omitted (0-values are assumed!).

        Add(time: AbsTime, delay: RelTime):AbsTime READONLY;

        // A relative point in time is added to an absolute
        // point in time.

        Compare(time1: AbsTime, time2: AbsTime):INT READONLY;

        // Compare returns
        //  1: <time1> is later than <time2>
        //  2: <time2> is later than <time1>
        //  0: <time1> and <time2> are identical

        DisplayAbsTime(time: AbsTime) READONLY;
```

```
        // An absolute point in time is displayed.

        DisplayRelTime(time: RelTime) READONLY;

        // A relative point in time is displayed.

  IMPLEMENTATION
    ...

END; // Time_OwnType
```

Metaclass *TimeEvnt* is utilized primarily for time-event detection. To detect time events, a "hierarchy" of *SignalTime*-methods has been established. A *SignalTime*-method is defined for metaclass *TimeEvnt*, for the metaclasses *AbsTimeEvnt*, *Reltv-TimeEvnt*, and *PrdcTimeEvnt*, and for user-defined time event classes:

```
OBJECTTYPE TimeEvnt_OwnType;
  INTERFACE
    METHODS
        SignalTime(time: INT);

        // <time> is converted from INT to AbsTime. Then, <time> is
        // passed to the instances of TimeEvnt, i.e., to AbsTimeEvnt,
        // to RltvTimeEvnt, and to PrdcTimeEvnt.

  IMPLEMENTATION
    METHODS
        SignalTime(time: INT);
        {
            VAR timeAsAbsTime: AbsTime;
            VAR i:OID;

            timeAsAbsTime:=Time->IntToAbsTime(time);
            FORALL (i IN SELF->allInstances()) {
                i->SignalTime(timeAsAbsTime); };
        };

END; // TimeEvnt_OwnType
```

Object type *TimeEvnt_InstType* defines a method *SignalTime*, too. At this level, *SignalTime* propagates a time pulse to the user-defined time-event classes. The INSTINSTTYPE of metaclass *TimeEvnt*—a subtype of *EvntClsType*—is empty, since no common structure and behaviour of user-defined time-event classes can be factored out.

Within its INSTTYPE, metaclass *AbsTimeEvnt* provides the functionality for user-defined absolute time-event classes. Absolute time event classes differ from other event classes: An absolute time event is raised exactly once, therefore, an absolute time-event class simultaneously represents its own single instance. Thus, object type *AbsTimeEvnt_InstType* is a subtype of *EvntInstType*:

```
OBJECTTYPE AbsTimeEvnt_InstType SUBTYPEOF Metaclass_InstType,
           EvntInstType;

  INTERFACE
    METHODS
        SignalTime(time: AbsTime);

        // If <time> is equal to or earlier than <actvTime>
        // the absolute time event is raised.

        Release(evntInst: OID, subscriber: OID);

        // As class objects cannot be deleted, Release is an
        // empty method.

        Init(absTimeAsStr: STRING);

        // <absTimeAsStr> is converted to AbsTime and assigned
        // to <actvTime>.

  IMPLEMENTATION
    PROPERTIES
        actvTime: AbsTime;
            // point in time the absolute time event is raised

    METHODS
        SignalTime(time: AbsTime);
        {
            VAR compResult: INT;

            compResult:=Time->Compare(time,actvTime);
            IF (compResult==0 OR compResult==1) {
                timeStamp:=time;
                SELF->InformSubscribers(SELF); };
        };

        ...

END; // AbsTimeEvnt_InstType
```

Since no instances of user-defined absolute-time event classes are created, the object-type *AbsTimeEvnt_InstInstType* is empty. Contrary to an absolute time event class, a relative time-event class administers instances. Further, relative time-event classes represent unary event subscribers:

```
OBJECTTYPE RltvTimeEvnt_InstType SUBTYPEOF Metaclass_InstType,
           EvntInstAdminType, UnarySubscrClsType;
  INTERFACE
    METHODS
        SignalTime(time: AbsTime);

        // If <actvTimes> contains keys earlier or equal <time>, a new
        // relative time event instance is created with each underlying
```

```
        // event instance. There might be multiple instantiations of
        // the relative time event at the same point in time!

        SignalEvnt(evntInst: OID);

        // Instantiation of <evnt> is signalled to the relative time
        // event class. The relative time event will be instantiated
        // after <delay> has elapsed. <delay> is added to the timestamp
        // bound to <evntInst>, the result is inserted as a key into
        // <actvTimes>. <evntInst> represents the key's associated value.
        // If the key is already included, <evntInst> is inserted into
        // the key's associated set of OIDs.

        Release(evntInst: OID, subscriber: OID);

        // <evntInst> is released by <subscriber>. If <evntInst> is
        // released by all subscribers, the event instance to which
        // <evntInst> is relative, is released and <evntInst> is
        // deleted.

        SetDelay(delayTimeAsStr: STRING);

        // <delayTimeAsStr> is converted to RelTime and assigned to
        // <delay>.

        Init(evntCls:OID, delayTimeAsStr: STRING);

        // <evntCls> is subscribed (by means of method
        // <AddTimeEvntSubscr>) and assigned to <evnt>.
        // <delayTimeAsStr> is converted to RelTime and assigned to
        // <delay>.

  IMPLEMENTATION
    PROPERTIES
        actvTimes: ||AbsTime --> {OID}||;
            // mapping from absolute points in time to sets of instances
            // of <evnt>; absolute points in time represent the keys,
            // sets of instances represent the values of this dictionary

        delay: RelTime;
            // temporal delay between instantiation of <evnt> and
            // instantiation of the relative time event


    METHODS
        ...

        SignalTime(time: AbsTime);
        {
            VAR concurrEvntInsts:{OID};
            VAR newRelTimeInst:OID;
            VAR i:OID;

            concurrEvntInsts:=SELF->GetConcurrentInsts(time);
            FORALL (i IN concurrEvntInsts) {
                newRelTimeInst:=SELF->NewListElem();
```

```
                newRelTimeInst->SYS_set_timeStamp(time);
                newRelTimeInst->SYS_set_evntInst(i);
                newRelTimeInst->SetAllSubscribers(SELF->AllSubscribers());
                SELF->InformSubscribers(newRelTimeInst); };
        };

        ...

        Release(evntInst: OID, subscriber: OID);
        {
            evntInst->ReleasedBy(subscriber);
            IF (evntInst->(BOOL)IsReleased()) {
                evntInst->ReleaseInst(SELF);
                SELF->DelListElem(evntInst); };
        };

        ...

END; // RltvTimeEvnt_InstType
```

Method *SignalTime* makes use of method *GetConcurrentInsts* to determine all instances of the subscribed event class, for which a relative time event instance has to be created at point in time *time*. *GetConcurrentInsts* scans the property *actvTimes* for keys, that are equal to or earlier than *time*. The values associated with the keys found are accumulated to a set and returned to the calling *SignalTime*-method, then the keys (and their associated values) are removed from *actvTimes*. Object type *RltvTimeEvnt_InstInstType*, a subtype of *EvntInstType* and of *UnarySubscrInstType*, does not specify additional functionality for the instances of relative time-event classes.

Object type *PrdcTimeEvnt_InstType* specifies common structure and behaviour of user-defined periodic time event classes. Like relativ time-event classes, periodic time-event classes administer instances and represent unary event subscribers:

```
OBJECTTYPE PrdcTimeEvnt_InstType SUBTYPEOF Metaclass_InstType,
           EvntInstAdminType, UnarySubscrClsType;
  INTERFACE
    METHODS
        SignalTime(time: AbsTime);

        // For each start event instance in <prdcFirings>, whose
        // associated point in time is equal to (or earlier than)
        // <time>, a new periodic time event instance is created.
        // The associated point in time is increased by <period>.

        SignalEvnt(evntInst: OID);

        // <guardFunction> is called for <evntInst> and sequentially
        // every key in <prdcFirings>. A key, for which <guardFunction>
        // evaluates to true, is removed from <prdcFirings>, i.e., the
        // respective periodic firing is terminated. A new periodic
        // firing - with <evntInst> as start event (key) - is initiated.
```

```
        Release(evntInst: OID, subscriber: OID);

        // <evntInst> is released by <subscriber>. If <evntInst> is
        // released by all subscribers, the event instance on which
        // <evntInst> is based, is released and <evntInst> is deleted.

        SetPeriod(prdcTimeAsStr: STRING);

        // <prdcTimeAsStr> is converted to RelTime and assigned
        // to <period>

        GetGuardFunc():STRING READONLY;

        // <guardFunction> is returned

        AnyObject():BOOL READONLY;

        // guard-function that evaluates to TRUE in any case

        SameObject():BOOL READONLY;

        // guard-function that evaluates to TRUE, if <evntToCheck> and
        // <firingToCheck> have been bound to the same <o>

        Init(evntCls: OID, prdcTimeAsStr: STRING, guardFunc: STRING);

        // <evntCls> is subsribed (by means of method <AddTimeEvntSubscr>)
        // and assigned to <evnt>; <prdcTimeAsStr> is converted to
        // RelTime and assigned to <period>;
        // <guardFunc> is assigned to <guardFunction>

  IMPLEMENTATION
    PROPERTIES
        prdcFirings: ||OID --> AbsTime||;
            // mapping from start event instances to points in
            // time representing the periodic time event's next
            // instantiations

        period: RelTime;
            // temporal interval between two instantiations

        guardFunction: STRING;
            // name of the function which is called to determine
            // whether an existing periodic firing has to be
            // replaced

        evntToCheck: OID;
            // new start event instance that has to be check by
            // the <guardFunction>

        firingToCheck: OID;
            // start event instance of a periodic firing that has
            // to be checked by the <guardFunction>

    METHODS
```

```
        ...

        SignalTime(time: AbsTime);
        {
            VAR newPrdcTimeInst:OID;
            VAR firings: {OID};
            VAR i: OID;

            firings:=SELF->GetConcurrentFirings(time);
            FORALL (i IN firings) {
                newPrdcTimeInst:=SELF->NewListElem();
                newPrdcTimeInst->SYS_set_timeStamp(time);
                newPrdcTimeInst->SYS_set_evntInst(i);
                newPrdcTimeInst->SetAllSubscribers(SELF->AllSubscribers());
                SELF->InformSubscribers(newPrdcTimeInst); };
        };

        ...

        Release(evntInst: OID, subscriber: OID);
        {
            evntInst->ReleasedBy(subscriber);
            IF (evntInst->(BOOL)IsReleased()) {
                evntInst->ReleaseInst(SELF);
                SELF->DelListElem(evntInst); };
        };

        ...

END; // PrdcTimeEvnt_InstType
```

Method *GetConcurrentFirings*—called by *SignalTime*—determines all instances of the subscribed start event, for which the periodic time event has to be raised at the point in time *time*. Therefore, *GetConcurrentFirings* searches the property *prdcFirings* for keys (start-event instances) with an associated value earlier than or equal to *time*. Each value found is increased by *period*, and thus, it represents the point in time of the next instantiation. The keys are accumulated and returned to the calling method *SignalTime*. Object type *RltvTimeEvnt_InstInstType*—a subtype of *EvntInstType* and of *UnarySubscrInstType*—does not define additional functionality for the instances of user-defined periodic time-event classes.

In the sequel, the process of time-event detection and instantiation and the process of releasing time-event instances are summarized:

1. *Detection and instantiation of time events:* As explained in subsection 5.2.1.1, the finest granularity in specifying points in time is one second. Consequently, to detect time events precisely, it would be necessary to perform method *SignalTime* on metaclass *TimeEvnt* every second. Hardware and database restrictions make such a precise solution unpracticable: First, perfoming method *SignalTime* (and all subsequent methods!) every second, would overload many hardware platforms. Second, the clock process performing method *SignalTime* makes use of transaction

semantics. Situations may arise, where a clock-process transaction is in conflict with other transactions running on the database. If the clock-process transaction is prevented from signalling for more than a second, critical points in time may be overlooked.

This problem is encountered in ActiveVODAK by adopting a less precise strategy. The clock process does not perform method *SignalTime* every second, but in intervals specified according to application needs and hardware performance (cf. section 6.4). When the point in time t is signalled to a user-defined time-event class, an instance is not created only, if the time-event class's point in time of instantiation $tInst$ exactly matches t, but also if $tInst$ is earlier than t. The timestamp bound to every time-event instance corresponds to t—the point in time signalled to the time event class.

2. *Release of time event instances:* The release of event instances has to be explained separately for the different types of time events: As already mentioned, a user-defined absolute time event class simultaneously represents its own single instance. Since a class can never be deleted, keeping track of subscribers which have/have not released the absolute time event instance is irrelevant.

   The releasing strategy for relative and periodic time event instances is largely equivalent to the strategy adopted for method event instances. But, before a relative/periodic time event instance can be made invisible (by performing method *DelListElem*), the underlying event instance has to be released (by performing method *ReleaseInst* defined by the object type *UnarySubscrInstType*).

Abstract Events. User-defined abstract event classes are instances of metaclass *AbstrctEvnt*. Figure 6.3 illustrates the object type hierarchy and the class hierarchy established to support abstract events.

Object type *AbstrctEvnt_InstType*—a subtype of *EvntClsType* and of *EvntInstAdminType*—defines the common structure and behaviour of user-defined abstract event classes:

```
OBJECTTYPE AbstrctEvnt_InstType SUBTYPEOF Metaclass_InstType,
           EvntClsType, EvntInstAdminType;
  INTERFACE
    METHODS
        NewEvntInst():OID;

        // An instance is made visible. The default parameters
        // (<timeStamp> and <transId>) are bound. The new instance's
        // set of subscribers is set to the conjunction of all
        // subscribers of the abstract event class.


        Release(evntInst: OID, subscriber: OID);

        // The instance <evntInst> is released by the subscriber
        // <subscriber>. <subscriber> is removed from the subscriber
```

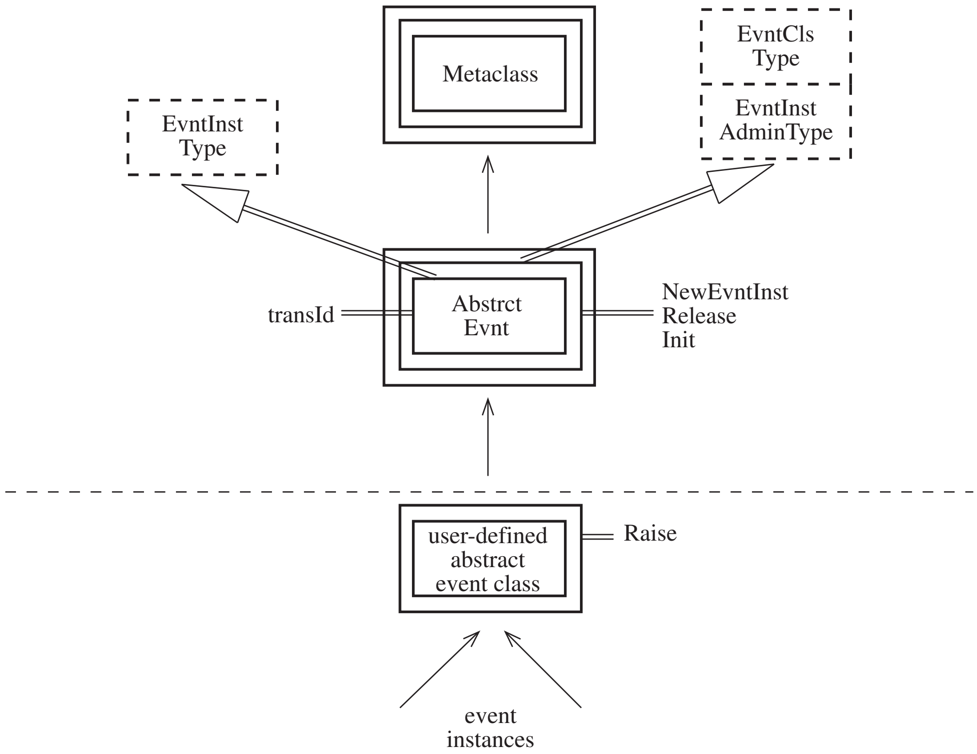


Figure6.3: Implementation of abstract events

```
        // list of <evntInst>. If <evntInst> has been released
        // by all subscribers, <evntInst> is made invisible.

        Init();

        // The instance list is initialized.

  IMPLEMENTATION

    ...

END; // AbstrctEvnt_InstType
```

The INSTINSTTYPE of metaclass *AbstrctEvnt* defines a single property for the instances of user-defined abstract event classes—*transId*. When a new abstract event instance is created (by means of method *NewEvntInst*), the identifier of the transaction embracing the execution of method *Raise* is assigned to *transId*. Object type *AbstrctEvnt_InstInstType* is a subtype of *EvntInstType*.

Method *Raise*, specified within the OWNTYPE of each user-defined abstract event class, plays the central role in instantiating abstract events (cf. subsection 5.2.1.1). After a new abstract event instance has been created (*NewEvntInst*), event-specific parameters are bound to the new event instance. *Raise* concludes by informing all subscribers (*InformSubscribers*). The releasing strategy for abstract-event instances is equivalent to that for method-event instances.

6.1.1.2 Monitoring Intervals

User-defined monitoring intervals are instances of metaclass *MonitInt*. Figure 6.4 illustrates the object-type hierarchy and the class hierarchy established to support monitoring intervals.

Object type *MonitInt_InstType* defines the common structure and behaviour of monitoring intervals. To maintain subscription relationsships, all user-defined event classes introduced so far make use of the functionality defined within object type *EvntClsType*. Only two kinds of subscribers are based on monitoring interval classes: negations and events with associated monitoring intervals. Consequently, the functionality defined by object type *EvntClsType* is not necessary and, thus, not available for monitoring interval classes.

```
OBJECTTYPE MonitInt_InstType SUBTYPEOF Metaclass_InstType,
           BinarySubscrClsType, EvntInstAdminType;
  INTERFACE
    METHODS
        SignalEvnt(evntInst: OID);

        // Instantiations of the begin/end event class are
        // signalled to the monitoring interval:
```

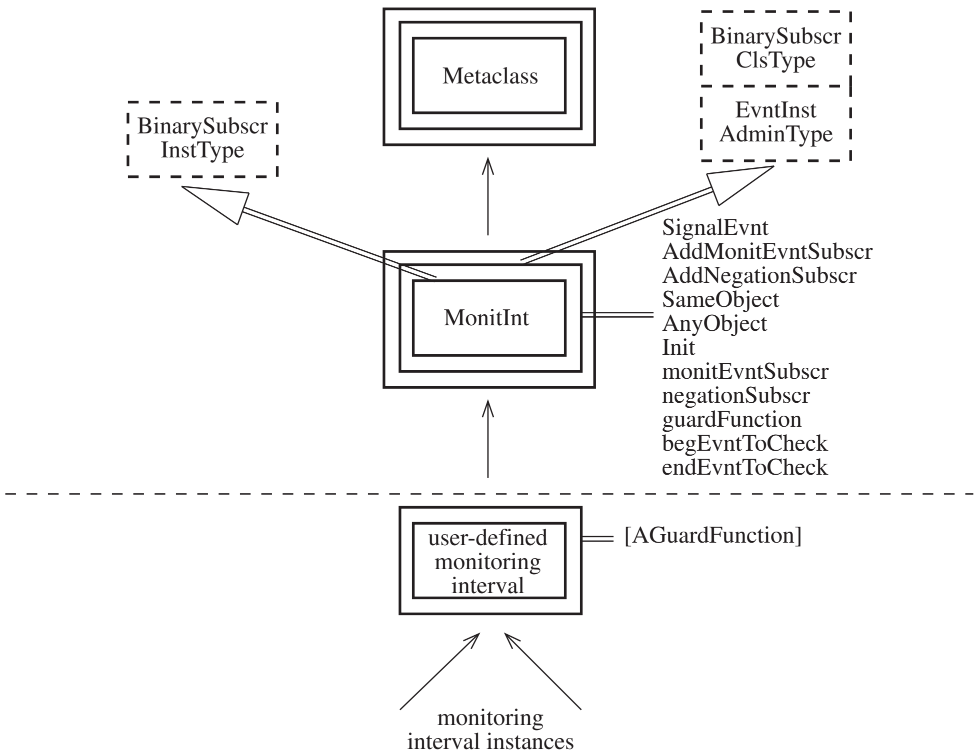


Figure6.4: Implementation of monitoring intervals

```
    AddMonitEvntSubscr(monitEvnt: OID);

    // The monitored event class <monitEvnt> is added to
    // <monitEvntSubscr>.

    AddNegationSubscr(negation: OID);

    // The negation event class <negation> is added to
    // <negationSubscr>.

    Init(bEvnt: OID, eEvnt: OID, guardFunc: STRING);

    // <bEvnt> is assigned to <beginEvnt>;
    // <eEvnt> is assigned to <endEvnt>;
    // <bEvnt> and <eEvnt> are subscribed (by means of method
    // <AddMonitIntSubscr>; <guardFunc> is assigned to <guardFunction>


IMPLEMENTATION
  PROPERTIES
    monitEvntSubscr: {OID};
        // monitored event classes subscribing the monitoring
        // interval

    negationSubscr: {OID};
        // negation event classes subscribing the monitoring
        // interval

    guardFunction: STRING
        // name of the method which is called to determine,
        // whether a begin event instance (<begEvntToCheck>)
        // and an end event instance (<endEvntToCheck>) are
        // compatible

    begEvntToCheck: OID;
        // begin event instance which is checked by
        // <guardFunction> (together with <endEvntToCheck>)

    endEvntToCheck: OID;
        // begin event instance which is checked by
        // <guardFunction> (together with <endEvntToCheck>)

  METHODS
    SameObject(): BOOL READONLY;

    // predefined guard function which evaluates to TRUE, if
    // <begEvntToCheck> and <endEvntToCheck> have been bound
    // to the same <o>

    AnyObject(): BOOL READONLY;

    // predefined guard function which evaluates to TRUE
    // in any case

   ...
```

```
END; // MonitInt_InstType
```

Instances of monitoring interval classes represent states of the begin event for which the monitoring interval is "on". Monitoring interval instances do not bind a timestamp. Object type *MonitInt_InstInstType*—a subtype of *BinarySubscrInstType*—does not specifiy additional properties or methods for monitoring interval instances. Creation and deletion of monitoring interval instances is realized by method *SignalEvnt*. An instantiation of a begin/end-event class is signalled to a monitoring interval class by performing method *SignalEvnt(evntInst)* on it. Depending on the class of *evntInst*, method *SignalEvnt* executes the following actions:

1. If *evntInst* is an instance of the monitoring interval's begin event class (*beginEvnt*), a new monitoring interval instance is created and, thus, switched to "on". The instantiation is signalled to all subscribers (negation classes and event classes with associated monitoring intervals) by performing method *SignalMonitIntOn* on them.
2. If *evntInst* is an instance of the monitoring interval's end event class (*endEvnt*), an existing interval instance is searched for completion. In the case that more than one monitoring interval instances are compatible with *evntInst*, the oldest open interval instance is selected. If an instance could be completed (i.e., *guardFunction* evaluates to TRUE), completion is signalled to all subscribers by perfoming method *SignalMonitIntOff* on them. The completed instance is made invisible.

6.1.1.3 Complex Events

The complex event operators supported in ActiveVODAK are *conjunction*, *sequence*, and *negation*. Since there are considerable differences in the implementation of conjunctions/sequences on the one hand and negations on the other, conjunctions/sequences and negations are described separately.

Conjunctions and Sequences. User-defined conjunctions are instances of the predefined metaclass *Conjunction*; user-defined sequences are based on metaclass *Sequence*. The metaclasses *Conjunction* and *Sequence*, in their turn, are instances of metaclass *ComplxEvnt*. Figure 6.5 illustrates the object-type hierarchy and the class hierarchy established to support conjunctions and sequences.

Object type *ComplxEvnt_InstInstType* is a subtype of object type *TidCheckingType*, an object type not introduced so far. *TidCheckingType* provides the functionality to determine, whether an event instance of arbitrary type possesses a transaction binding or not. Whether an instance of a conjunction/sequence binds *transId* can be determined only at runtime. If the completing basic event instance binds *transId*, the instance of the conjunction/sequence binds *transId*, too. Consequently, depending on the completing basic event instance, some instances of a conjunction/sequence may possess a transaction binding, others may not.

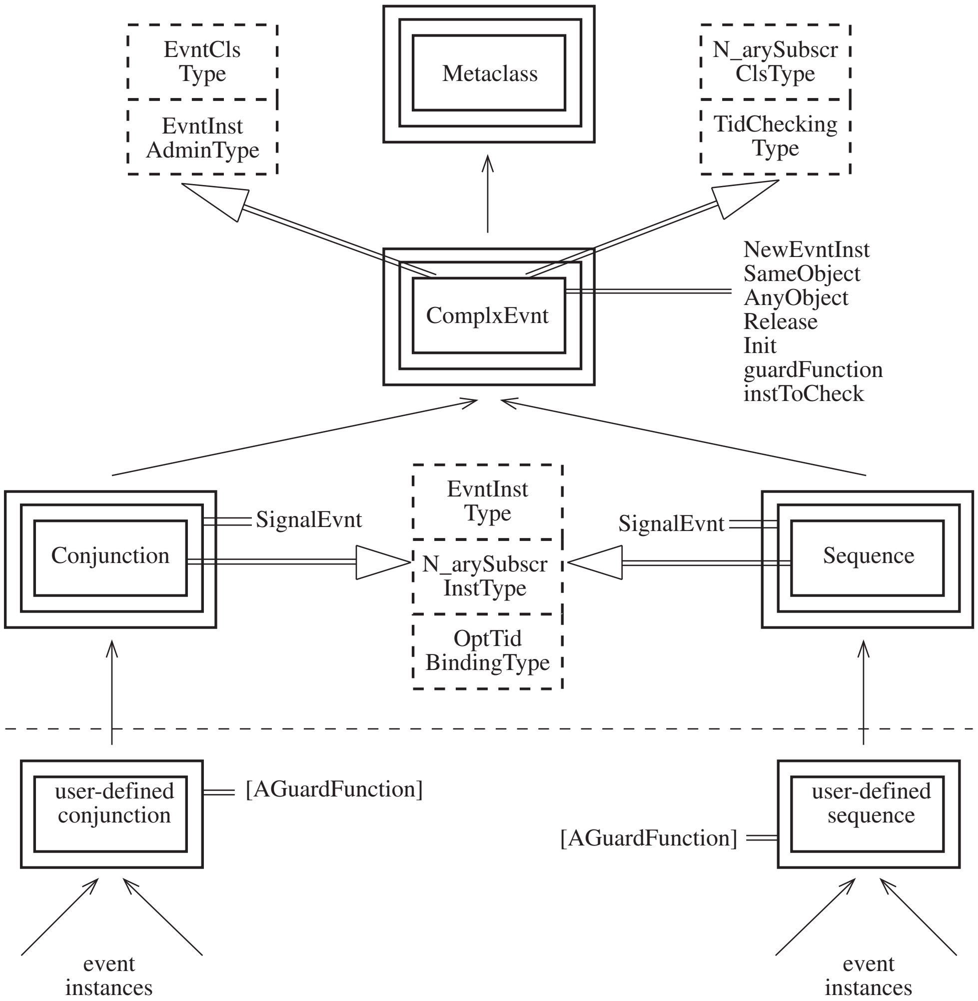


Figure6.5: Implementation of conjunctions and sequences

```
OBJECTTYPE TidCheckingType;
  INTERFACE
    METHODS
        CheckTidBinding(evntInst: OID):BOOL READONLY;

        // Check, whether <evntInst> possesses a transaction binding:
        // Method event instances and abstract event instances always
        // bind <transId>. Time event instances and instances of
        // negations never bind <transId>. Whether an instance
        // of a conjunction/sequence possesses a transaction binding
        // must be checked.

  IMPLEMENTATION
    METHODS
        CheckTidBinding(evntInst: OID):BOOL READONLY;
        {
            VAR mClsOfInst:OID;
            VAR mmClsOfInst:OID;

            mClsOfInst:=evntInst->(OID)class()->(OID)class();
            mmClsOfInst:=mClsOfInst->(OID)class();

            IF (mmClsOfInst==MethEvnt OR mClsOfInst==AbstrctEvnt) {
                RETURN TRUE; };
            IF (mmClsOfInst==TimeEvnt OR mClsOfInst==Negation) {
                RETURN FALSE; };

            RETURN evntInst->(BOOL)IsTidBound();
        };

END; // TidCheckingType
```

Method *IsTidBound*—defined within object type *OptTidBindingType*—is utilized to determine, whether an instance of a conjunction/sequence possesses transaction binding. The object types *Conjunction_InstInstType* and *Sequence_InstInstType*, which determine the common structure and behaviour of instances of user-defined conjunctions and sequences, are subtypes of *OptTidBindingType*:

```
OBJECTTYPE OptTidBindingType;
  INTERFACE
    PROPERTIES
        transIdBound: BOOL;
            // does the event instance bind <transId>?

        transId: INT;
            // identifier of the transaction, the event instance
            // is bound to

    METHODS
        IsTidBound(): BOOL READONLY;

        // <transIdBound> is returned
```

```
      IMPLEMENTATION
        ...

END; // OptTidBindingType
```

The INSTINSTTYPE of metaclass *ComplxEvnt* defines common methods and properties for user-defined conjunctions and sequences:

```
    OBJECTTYPE ComplxEvnt_InstInstType SUBTYPEOF Metaclass_InstInstType,
               EvntClsType, EvntInstAdminType, N_arySubscrClsType,
               TidCheckingType;
      INTERFACE
        METHODS
            NewEvntInst():OID;

            // A new (incomplete) complex event instance is created. The
            // basic event instances constituting the complex event instance
            // are initialized with NULL.

            Release(evntInst: OID, subscriber: OID);

            // The instance <evntInst> is released by the subscriber
            // <subscriber>. <subscriber> is removed from the subscriber
            // list of <evntInst>. If <evntInst> has been released
            // by all subscribers, all basic event instances of <evntInst>
            // are released and <evntInst> is made invisible.

            Init(evntList: LIST OF OID, guardFunc: STRING);

            // <evntList> is assigned to <evnts>; all basic event classes
            // contained in <evntList> are subscribed (by means of method
            // <AddCompEvntSubscr>; <guardFunc> is assigned to <guardFunction>;


      IMPLEMENTATION
        PROPERTIES
            guardFunction: STRING;
                // name of the method which is called to determine, whether
                // the basic event instances of <instToCheck> are compatible

            instToCheck: OID;
                // complex event instance which has to be checked by
                // <guard function>

        METHODS
          ...

            SameObject():BOOL READONLY;

            // guard-function which evaluates to TRUE, if all basic event
            // instances of <instToCheck> have been bound to the same <o>

            AnyObject():BOOL READONLY;
```

```
        // guard-function which evaluates to TRUE in any case

       ...

   END; // ComplxEvnt_InstInstType
```

Object type *Conjunction_InstType* defines a single method for user-defined conjunctions: *SignalEvnt*. Every time, a basic event class is instantiated, method *SignalEvnt* is called. The new basic event instance is passed as argument *evntInst*. *SignalEvnt* performs the following steps: First, the position at which *evntInst* must be inserted into incomplete instances is determined. Then, *SignalEvnt* searches for an incomplete instance which is empty at the required position and which fulfills the (predefined or user-defined) guard-function when filled up with *evntInst*. In the case suited incomplete instances could be found, the oldest instance is filled up with *evntInst*. If, thereby, the instance could be completed, the completed instance's *timeStamp* and—if *CheckTidBinding* evaluates to TRUE for *evntInst*—its *transId* are set to the values bound in *evntInst*. Instantiation of the conjunction is signalled to its subscribers (*InformSubscribers*). If no proper incomplete instance could be found, a new incomplete instance is created (by means of method *NewEvntInst*) and *evntInst* is inserted at the required position.

Object type *Sequence_InstType* defines method *SignalEvnt* for user-defined sequences. The implementation of *SignalEvnt* is similar to the implementation of the *SignalEvnt*-method defined for conjunctions. Contrary to conjunctions, *evntInst* can only be inserted into an incomplete instance of a sequence, if the incomplete instance is filled until the required insert position. The chronological order defined on the basic event classes must be reflected by each instance of the sequence! Therefore, if no proper incomplete instance could be found, a new incomplete instance can be created only, if *evntInst* is an instance of the first basic class subscribed by the sequence. Otherwise, *evntInst* cannot contribute to an instantiation of the sequence; *evntInst* is released.

Negations. User-defined negations are instances of metaclass *Negation*. Figure 6.6 illustrates the object-type hierarchy and the class hierarchy established to support negations.

Object type *Negation_InstType* defines the common structure and behaviour of user-defined negations. Negations are unary subscribers, they are based on a single event class. Additionally, each negation "subscribes" a monitoring interval.

```
   OBJECTTYPE Negation_InstType SUBTYPEOF Metaclass_InstType,
              EvntClsType, EvntInstAdminType, UnarySubscrClsType;
     INTERFACE
       METHODS
         SignalEvnt(evntInst: OID);

         // Instantiation of <evnt> is signalled. If any instance
         // of <monitInt> is ''on'' (i.e., <monitIntsOn> is not
         // empty), <evntInst> is appended to <evntsOccurred>.
```

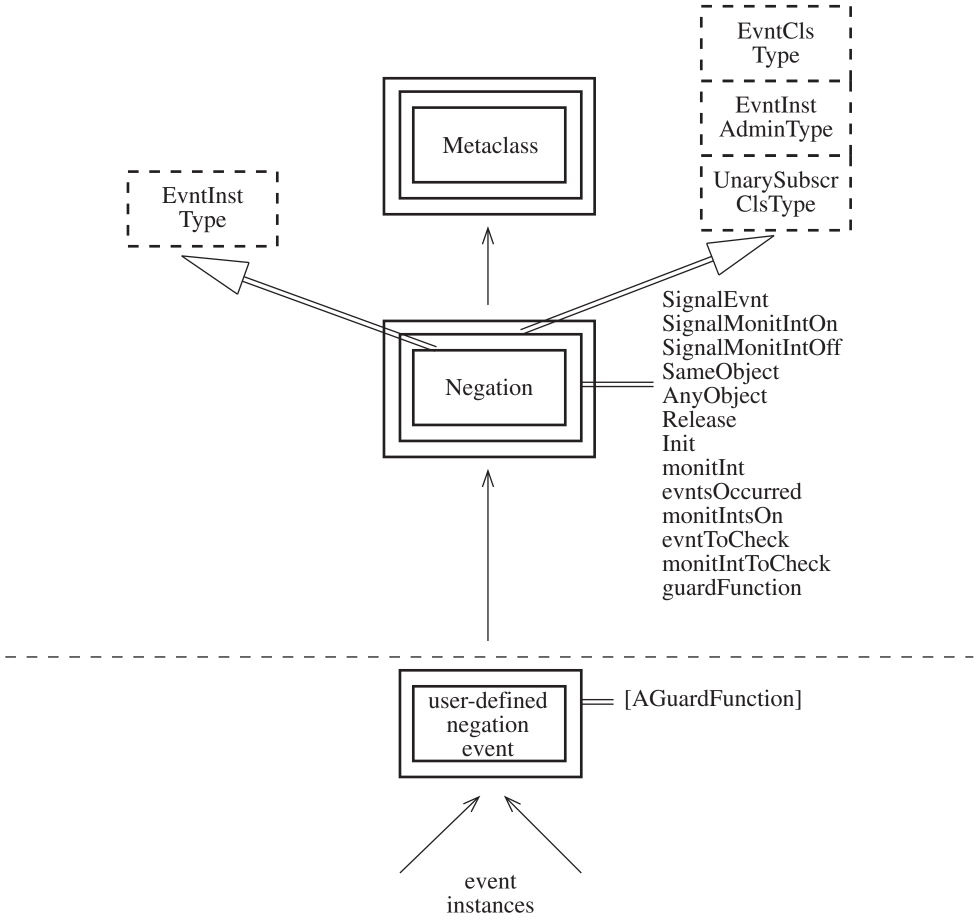


Figure6.6: Implementation of negations

```
      SignalMonitIntOn(monitIntInst: OID);

      // The instance <monitIntInst> of <monitInt> switches
      // to ''on''. <monitIntInst> is appended to <monitIntsOn>.

      SignalMonitIntOff(monitIntInst: OID);

      // The instance <monitIntInst> of <monitInt> switches
      // to ''off''.

      Release(evntInst: OID, subscriber: OID);

      // The instance <evntInst> is released by the subscriber
      // <subscriber>. <subscriber> is removed from the subscriber
      // list of <evntInst>. If <evntInst> has been released
      // by all subscribers, <evntInst> is made invisible.

      Init(notEvnt: OID, notMonitInt: OID, guardFunc: STRING);

      // <notEvnt> is assigned to <evnt>; <notEvnt> is subscribed by
      // means of method <AddCompEvntSubscr>;
      // <notMonitInt> is assigned to <monitInt>; <notMonitInt> is
      // subscribed by means of method <AddNegationSubscr>;
      // <guardFunc> is assigned to <guardFunction>;

IMPLEMENTATION
  PROPERTIES
      monitInt: OID;
          // monitoring interval the negation is based on

      evntsOccurred: LIST OF OID;
          // instantiations of <evnt>

      monitIntsOn: LIST OF OID;
          // instances of <monitInt> that are ''on''

      evntToCheck: OID;
          // instance of <evnt> which is checked by <guardFunction>
          // (together with <monitIntToCheck>)

      monitIntToCheck: OID;
          // instance of <monitInt> which is checked by <guardFunction>
          // (together with <evntToCheck>

      guardFunction: STRING;
          // name of the function which is called to determine
          // whether <evntToCheck> and <monitIntToCheck> are
          // compatible

  METHODS
      SameObject():BOOL READONLY;

      // guard-function which evaluates to TRUE, if <evntToCheck>
      // and <monitIntToCheck> have been bound to the same <o>
```

```
        AnyObject():BOOL READONLY;

        // guard-function which evaluates to TRUE in any case

        ...

END; // Negation_InstType
```

Every time, the negation's underlying monitoring interval switches "off" for a specific state, method *SignalMonitIntOff* is called. The monitoring interval instance representing the state for which the monitoring interval is switched "off" is passed as argument *monitIntInst*. Method *SignalMonitIntOff* checks, whether the negation has to be raised, i.e., whether no compatible basic event instantiation occurred during *monitIntInst*. Therefore, method *SignalMonitIntOff* considers all basic event instantiations (elements of *evntsOccurred*) that occurred during the interval delimited by the begin-event instance and the end-event instance of *monitIntInst*. If the guard-function does not evaluate to TRUE for any basic event instance considered, a new instance of the negation is created and signalled to the negation's subscribers. In the case that *monitIntInst* represents the oldest monitoring interval instance that has been "on" (i.e., *monitIntInst* is the first element in *monitIntsOn*), all basic event instances older than the begin-event instance of *monitIntInst* are released. *monitIntInst* is removed from *monitIntsOn*.

6.1.1.4 Events with Associated Monitoring Intervals

User-defined events with associated monitoring intervals are instances of metaclass *MonitEvnt*. Figure 6.7 illustrates the object type hierarchy and the class hierarchy established to support monitored events.

Object type *MonitEvnt_InstType* defines the common structure and behaviour of user-defined monitored events. Monitored events are unary subscribers, they are based on a single event class. Additionally, each monitored event "subscribes" a monitoring interval.

```
OBJECTTYPE MonitEvnt_InstType SUBTYPEOF Metaclass_InstType, EvntClsType,
           EvntInstAdminType, UnarySubscrClsType, TidCheckingType;
  INTERFACE
    METHODS
        SignalEvnt(evntInst: OID);

        // Instantiation of <evnt> is signalled to the monitored event.

        SignalMonitIntOn(intInst: OID);

        // The instance <intInst> of <monitInt> is switched ''on''.
        // <intInst> is inserted into <monitIntsOn>.

        SignalMonitIntOff(intInst: OID);

        // The instance <intInst> of <monitInt> is switched ''off''.
```

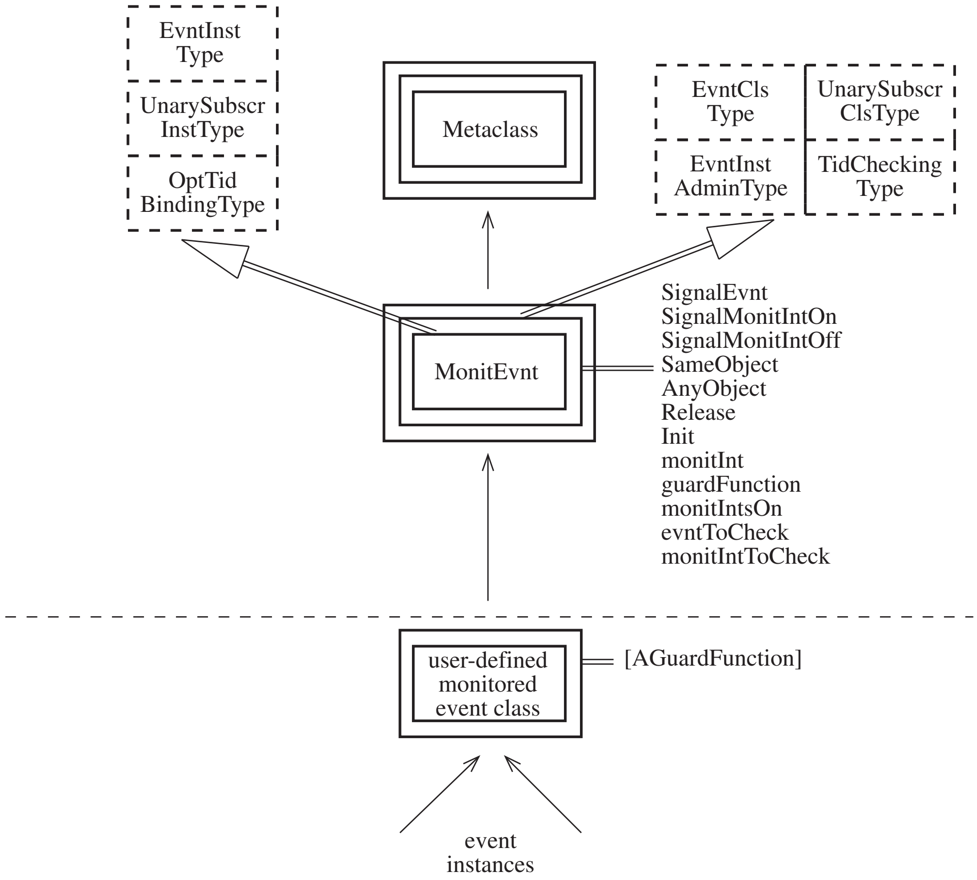


Figure6.7: Implementation of monitored events

```
        // <intInst> is removed from <monitIntsOn>.

        Release(evntInst: OID, subscriber: OID);

        // The instance <evntInst> is released by the subscriber
        // <subscriber>. <subscriber> is removed from the subscriber
        // list of <evntInst>. If <evntInst> has been released
        // by all subscribers, the event instance of the subscribed
        // event class is released, and <evntInst> is made invisible.

        Init(monitEvnt:OID, interv:OID, guardFunc: STRING);

        // <monitEvnt> is assigned to <evnt>; <evnt> is subscribed;
        // <interv> is assigned to <monitInt>; <monitInt> is subscribed;
        // <guardFunc> is assigned to <guardFunction>

  IMPLEMENTATION
    PROPERTIES
        monitInt: OID;
            // monitoring interval subscribed by the monitored event

        guardFunction: STRING;
            // name of the method which is called to determine, whether
            // <evntTocheck> and <monitIntToCheck> are compatible

        monitIntsOn: LIST OF OID;
            // list of monitoring interval instances which are ''on''

        evntToCheck: OID;
            // instance of <evnt> which has to be checked by
            // <guardFunction> (together with <monitIntToCheck>)

        monitIntToCheck: OID;
            // instance of <monitInt> which has to be checked by
            // <guardFunction> (together with <evntToCheck>)

    METHODS
        SameObject():BOOL READONLY;

        // guard-function which evaluates to TRUE, if <evntToCheck> and
        // the begin event of <monitIntToCheck> have been bound to
        // the same <o>

        AnyObject():BOOL READONLY;

        // guard-function which evaluates to TRUE in any case

        ...

END; // MonitEvnt_InstType
```

Every time, the monitored event's underlying basic event is instantiated, method *SignalEvnt* is called. The new basic event instance is passed as argument *evntInst*. Property *monitIntsOn* contains instances of the associated monitoring interval, i.e., *monitIntsOn*

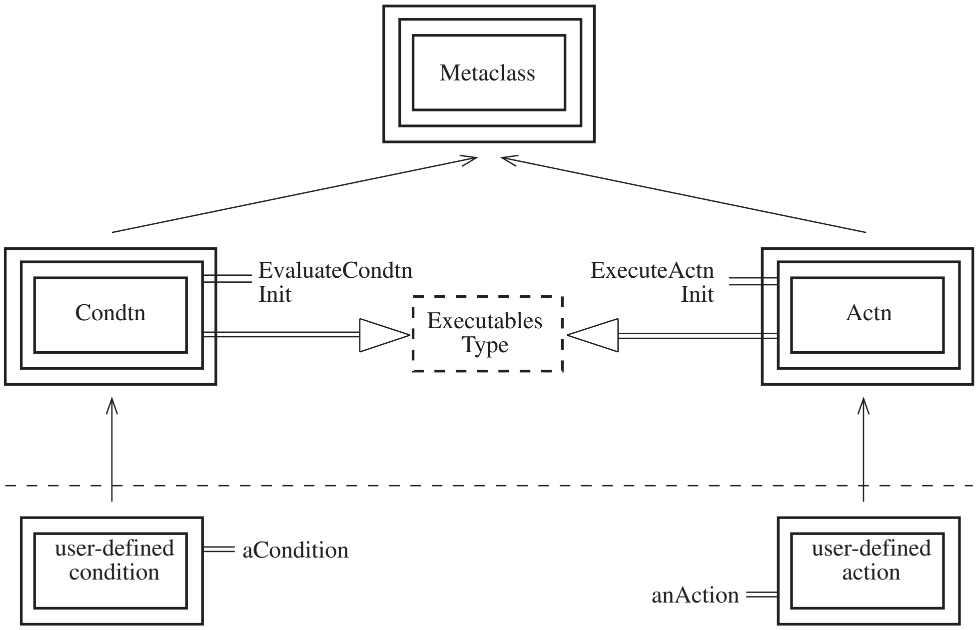


Figure6.8: Implementation of conditions and actions

represents all the states for which *monitInt* is "on". If the guard-function evaluates to TRUE for *evntInst* and an element of *monitIntsOn*, the monitored event is raised. The new instance's *timeStamp* and—if *CheckTidBinding* evaluates to TRUE for *evntInst*—its *transId* are set to the values bound in *evntInst*. Instantiation of the monitored event is signalled to its subscribers (*InformSubscribers*). Otherwise, the guard-function did not evaluate to TRUE, *evntInst* is released.

6.1.2 Conditions and Actions

User-defined conditions are instances of metaclass *Condtn*; user-defined actions are based on metaclass *Actn*. Figure 6.8 illustrates the object type hierarchy and the class hierarchy established to support conditions and actions.

Both *Condtn_InstType* and *Actn_InstType* are subtypes of the object type *ExecutablesType*. Object type *ExecutablesType* has not been introduced so far:

```
OBJECTTYPE ExecutablesType;
  INTERFACE
    PROPERTIES
        forEvntInst: OID;
            // event instance for which the condition/action
            // must be executed

        methToBeExec: STRING;
            // name of the method implementing the condition/
```

```
                // action; (A method with exactly this name must be
                // defined in the OWNTYPE of the condition/action.)

        METHODS
            Event():OID READONLY;

            // <forEvntInst> is returned

      IMPLEMENTATION
        ...

    END; // ExecutablesType
```

User-defined condition/action classes specify a method implementing the condition/action. This method, specified with the condition's/action's OWNTYPE (named *methToBeExec*) is parameterless. Method *Event* provides the possibility to access the event instance for which the condition is evaluated or for which the action is executed (*forEvntInst*) from within method implementation.

Object type *Condtn_InstType* specifies two methods for user-defined conditions:

```
    OBJECTTYPE Condtn_InstType SUBTYPEOF Metaclass_InstType, ExecutablesType;
      INTERFACE
        METHODS
            EvaluateCondtn(forInstnc: OID):BOOL READONLY;

            // The condition is evaluated for the event instance
            // <forInstnc>. <forInstnc> is assigned to <forEvntInst>, then
            // the method with name <methToBeExec> is executed. The
            // boolean value returned by <methToBeExec> is returned to
            // the calling rule.

            Init(nameOfImplMethod: STRING);

            // <nameOfImplMethod> is assigned to <methToBeExec>

      IMPLEMENTATION
        ...

    END; // Condtn_InstType
```

When a rule is fired, the rule's associated condition is evaluated, i.e., method *EvaluateCondtn* is performed on the associated condition class. The event instance that caused the rule firing is passed as argument *forInstnc*. If the condition evaluates to TRUE, the rule's associated action is executed. Action execution is induced by perfoming method *ExecuteActn* (with argument *forInstnc*) on the associated action class. The functionality of action classes is specified by the object type *Actn_InstType*:

```
    OBJECTTYPE Actn_InstType SUBTYPEOF Metaclass_InstType, ExecutablesType;
      INTERFACE
```

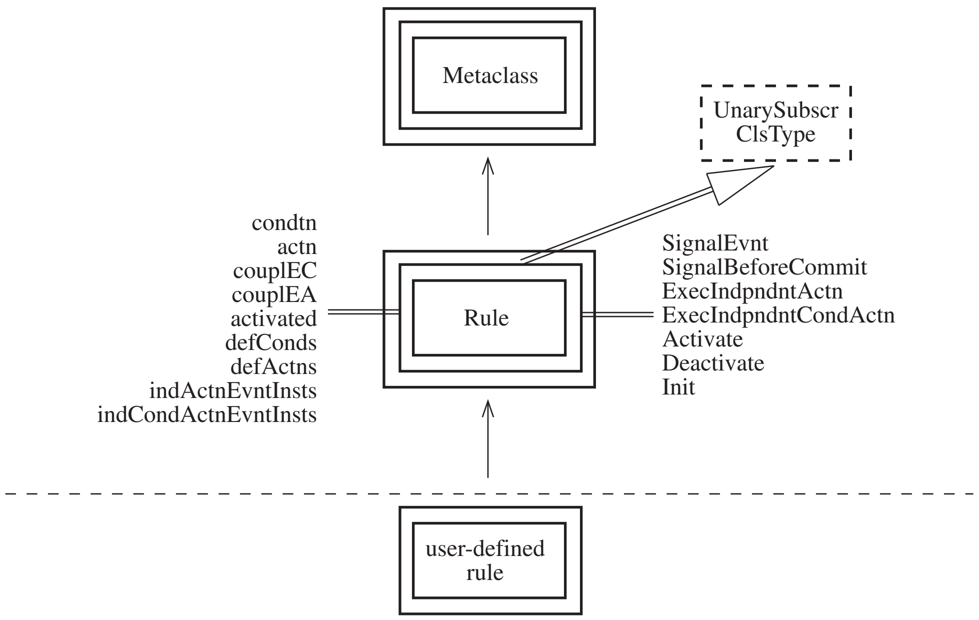


Figure6.9: Implementation of rules

```
    METHODS
        ExecuteActn(forInstnc: OID);

        // The action is executed for the event instance
        // <forInstnc>. <forInstnc> is assigned to <forEvntInst>, then
        // the method with name <methToBeExec> is executed.

        Init(nameOfImplMethod: STRING);

        // <nameOfImplMethod> is assigned to <methToBeExec>

  IMPLEMENTATION
    ...

END; // Actn_InstType
```

6.1.3 Rules

User-defined rules are instances of metaclass *Rule*. Figure *ruleImpl* illustrates the object type hierarchy and the class hierarchy established to support rules.

When a rule is fired, the rule's associated condition is evaluated. If the condition is satisfied, the rule's associated action is executed. The temporal and transactional relationships between event instantiation and condition evaluation/action execution entirely depend on the rule's coupling modes. Object type *Rule_InstType* implements condition evaluation and action execution for the different coupling modes introduced in subsection 5.3.4:

```
OBJECTTYPE Rule_InstType SUBTYPEOF Metaclass_InstType, UnarySubscrClsType;
  INTERFACE
    PROPERTIES
        condtn: OID;
            // condition class associated with the rule

        actn: OID;
            // action class associated with the rule

        couplEC: STRING;
            // coupling between event instantiation and condition
            // evaluation; three different modes are allowed:
            // 'imm' (immediate), 'def' (deferred) and
            // 'ind' (independent)

        couplEA: STRING;
            // coupling between event instantiation and action
            // execution; modes like <couplEC>

        activated: BOOL;
            // flag, which indicates whether the rule is activated

    METHODS
        SignalEvnt(evntInst: OID);

        // Instantiation of <evnt> is signalled to the rule. If the rule
        // is activated, the rule fires. Depending on the coupling modes,
        // different execution strategies are implemented.

        ExecIndpndntActn();

        // <actn> is executed independently for the first event instance
        // contained in <indActnEvntInsts>.

        ExecIndpndntCondActn();

        // <condtn> is evaluated independently for the first event
        // instance contained in <indCondActnEvntInsts>. If <condtn>
        // is satisfied, <actn> is executed for the same event
        // instance.

        SignalBeforeCommit(befComTransId: INT):BOOL;

        // Transaction with id <befComTransId> signals its ready-to-
        // commit status. Depending on the rule's coupling modes,
        // condition and/or action are executed for the event instances
        // contained in <defConds>/<defActns>. If a condition has
        // been evaluated or an action has been executed, TRUE is
        // returned.

        Activate();

        // The rule is activated.

        DeActivate();
```

```
        // The rule is deactivated. If there are event instances, for
        // which the condition and/or action have to be executed
        // deferredly, these conditions and/or actions are executed
        // nevertheless. Only new event instantiations are ignored.

        Init(event: OID, condition: OID, ec: STRING, action: OID, ea: STRING);

        // <event> is assigned to <evnt>; <evnt> is subscribed;
        // <condition> is assigned to <condtn>; <ec> is assigned
        // to <couplEC>; <action> is assigned to <actn>; <ea> is
        // assigned to <couplEA>; the rule is activated; if <couplEC>
        // =='def' or <couplEA>=='def', class <BeforeCommit> is
        // subscribed

  IMPLEMENTATION
    PROPERTIES
        defConds: ||INT --> LIST OF OID||;
            // mapping from transaction identifiers to lists of event
            // instances; this mapping is only relevant, if EC-coupling
            // is 'def'

        defActns: ||INT --> LIST OF OID||;
            // mapping from transaction identifiers to lists of event
            // instances; this mapping is only relevant, if EC-coupling
            // is 'imm' and EA-coupling is 'def'

        indActnEvntInsts: LIST OF OID;
            // list of event instances, for which the action has to
            // be executed independently; this list is only relevant,
            // if EA-coupling is 'ind'

        indCondActnEvntInsts: LIST OF OID;
            // list of event instances, for which the condition has
            // to be evaluated independently; if the condition
            // evaluates to TRUE, the action is executed
            // independently, too; this list is only relevant, if
            // EC-coupling is 'ind' (and, therefore, EA-coupling
            // is 'ind', too)

    EXTERN startIndependentTA(theObject:OID, theMethod:STRING, giveAck:BOOL): INT;

    METHODS
        ExecIndpndntActn();
        {
            VAR myEvntInst:OID;

            myEvntInst:=indActnEvntInsts->>get(0);
            indActnEvntInsts->>remove(0,1);
            actn->ExecuteActn(myEvntInst);
            evnt->Release(myEvntInst,SELF);
        };

        ExecIndpndntCondActn();
        {
            VAR myEvntInst:OID;
```

```
            myEvntInst:=indCondActnEvntInsts->>get(0);
            indCondActnEvntInsts->>remove(0,1);
            IF (condtn->(BOOL)EvaluateCondtn(myEvntInst)) {
                actn->ExecuteActn(myEvntInst); };
            evnt->Release(myEvntInst,SELF);
        };

        ...

    END; // Rule_InstType
```

In the case of deferred event/condition-coupling or deferred event/action coupling, a rule must be informed when a transaction is ready-to-commit. Therefore, rules with deferred couplings subscribe the transaction event *BeforeCommit*. *BeforeCommit* is represented by a class which is informed about transactions that are ready-to-commit by the transaction commitment notifier (cf. section 6.3). Class *BeforeCommit*—of which no instances are created—distributes the identifier of the committing transaction to all rules subscribing it.

```
OBJECTTYPE BeforeCommit_OwnType;
  INTERFACE
    PROPERTIES
        rules: {OID};
            // set of rules with deferred condition evaluation
            // or action execution;

    METHODS
        AddRuleToInform(rule: OID);

        // <rule> is added to <rules>

        TransBeforeCommit(committingTid: INT);

        // This method is called by the transaction commitment
        // notifier:
        // Transaction with id <committingTid> is ready-to-commit.
        // Method <SignalBeforeCommit> is perfomed on every rule
        // contained in <rules>. <committingTid> is passed as
        // argument. If any subsequent condition or action
        // had to be executed, execution of method
        // <SignalBeforeCommit> is repeated for all rules in
        // <rules>.

  IMPLEMENTATION
    ...

END; // BeforeCommit_OwnType
```

When a top-level transaction t is ready to commit, the transaction commitment notifier performs method *TransBeforeCommit* on the class *BeforeCommit*. The identifier of the committing top-level transaction t is passed as argument *committingTid*. Method

TransBeforeCommit informs its subscribers (i.e., rules with deferred couplings) by performing method *SignalBeforeCommit* on them. Subsequent condition evaluations/action executions may cause new rule firings. Consequently, execution of method *SignalBeforeCommit* on all subscribing rules must be repeated, until no conditions have been evaluated and no conditions have been executed.

To be able to perform condition evaluation/action execution in an independent top-level transaction, the function *startIndependentTA* has been implemented[2]. The function *startIndependentTA* allows to perform a parameterless method in an independent top-level transaction. It takes the following arguments: The identifier of the *object*, on which the method should be performed, the *name of the method* to be executed, and a flag indicating whether the calling transaction should wait for *acknowledgement* of successful execution. Since the methods for condition evaluation (*EvaluateCondtn*) and action execution (*ExecuteActn*) are not parameterless, two auxiliary methods had to be introduced. These auxiliary methods are *ExecIndpndntActn* and *ExecIndpndntCondActn*. The event instance, for which the condition evaluation and/or action execution has to be performed, is passed to the auxiliary methods via the rule properties *indActnEvntInsts* and *indCondActnEvntInsts*.

Here, method *SignalEvnt* is explained in more detail since it is central for the implementation of coupling modes. If a rule is activated, depending on the rule's coupling modes, different actions are performed[3]:

1. *EC-coupling: "imm", EA-coupling: "imm"*:

```
IF ((couplEC=='imm') AND (couplEA=='imm')) {
    IF (condtn->(BOOL)EvaluateCondtn(evntInst)) {
        actn->ExecuteActn(evntInst); };
    evnt->Release(evntInst,SELF); };
```

2. *EC-coupling: "imm", EA-coupling: "def"*: If the immediately executed condition is satisfied, the pair of triggering transaction and event instance is inserted into *defActns*:

```
IF ((couplEC=='imm') AND (couplEA=='def')) {
    IF (condtn->(BOOL)EvaluateCondtn(evntInst)) {
        triggTransId:=evntInst->(INT)SYS_transId();
        IF (defActns->>includes(triggTransId)) {
            newListOfInsts:=defActns->>getvalue(triggTransId); };
        newListOfInsts->>append(evntInst);
        defActns->>insert(triggTransId,newListOfInsts); };};
```

[2] *startIndependentTA* represents a function implemented in C++, which is made available in VML by the EXTERN declaration.

[3] The auxiliary variable *newListOfInsts* is local to method *SignalEvnt*, it is defined as a list of object identifiers, and it is intiatialized as an empty list.

3. *EC-coupling: "imm", EA-coupling: "ind"*: If the immediately executed condition is satisfied, the triggering event instance is inserted into *indActnEvntInsts*. Then, method *ExecIndpndntActn* is perfomed on the rule class in an independent top-level transaction:

```
IF ((couplEC=='imm') AND (couplEA=='ind')) {
    IF (condtn->(BOOL)EvaluateCondtn(evntInst)) {
        indActnEvntInsts->>append(evntInst);
        startIndependentTA(SELF,'ExecIndpndntActn',FALSE);};};
```

4. *EC-coupling: "def"*: The pair of triggering transaction and event instance is inserted into *defConds*.

```
IF (couplEC=='def') {
    triggTransId:=evntInst->(INT)SYS_transId();
    IF (defConds->>includes(triggTransId)) {
        newListOfInsts:=defConds->>getvalue(triggTransId); };
    newListOfInsts->>append(evntInst);
    defConds->>insert(triggTransId,newDefConds); };
```

5. *EC-coupling: "ind"*: The triggering event instance is inserted into *indCondActnEvntInsts*. Then, method *ExecIndpndntCondActn* is performed on the rule class in an independent top-level transaction:

```
IF (couplEC=='ind') {
    indCondActnEvntInsts->>append(evntInst);
    startIndependentTA(SELF,'ExecIndpndntCondActn',FALSE);};}
```

When a transaction is ready-to-commit, method *SignalBeforeCommit* is performed on each rule class with deferred couplings. The identifier of the committing transaction is passed as argument *befComTransId*. Depending on the coupling modes defined, method *SignalBeforeCommit* takes the pending event instances associated with the key *befComTransId* either from *defConds* or *defActns*. In the case that the event instances are taken from *defActns* (i.e., EC-coupling is "imm" and EA-coupling is "def"), the rule's associated action is executed for each event instance. If the event instances are taken from *defConds*, (i.e., EC-coupling is "def") the rule's associated condition is evaluated for each event instance. If the condition is satisfied for an event instance, depending on the EA-coupling ("def" or "ind"), action execution is performed either in the same or in an independent top-level transaction.

6.2 Method Execution Notifier

The method execution notifier—an extension of the *Message Handler* module—informs active databases about method executions. How the Message Handler performs method executions has already been explained in subsection 4.2—five steps are performed sequentially. Intuitively, there are two possible points where method executions can be signalled to active databases:

1. The execution of the method is signalled to the active database before the Message Handler has checked, whether the method to be executed belongs to the interface of the object's class or whether the NOMETHOD clause must be called.
2. The execution of the method is signalled to the active database after the Message Handler has determined to execute the method without calling the NOMETHOD clause.

Alternative 1 allows to reflect semantic relationships in the definition of method events. The implementation of the NOMETHOD-clause does not have to be considered. Alternative 2 means that the implementation of NOMETHOD-clauses has to be considered. Semantic relationships have to be resolved.

Example: Figure 4.2 shows the semantic relationships established between publications, books, and journals. Alternative 1 allows, for example, to define a method event on method *Sell* for all instances of class *Book*. Method *Sell* is not actually executed on books. The semantic relationship guarantees that method *Sell* is passed to the books' generalized counterparts, i.e., that *Sell* is performed on an instance of class *Publication*. Alternative 2 requires the definition of a method event on method *Sell* for all instances of class *Publication*. So, the method event is raised, whenever method *Sell* is performed on a book or journal. Then, the actual kind of publication (book or journal) has to be determined in the rule's condition/action.

Alternative 1 is realized in ActiveVODAK, since it allows to reflect semantic relationships in defining method events. When a method m is performed on object o, the method execution notifier performs the following steps:

1. It determines the database d, to which o belongs. The database to which an object belongs is encoded in the object's identifier (cf. Figure 6.10). Then, class *MethEvnt* of database d is determined.
2. It performs method *IsEvnt* on class *MethEvnt*. *IsEvnt* returns a structure consisting of two sets: A set named *before* containing all user-defined method event classes that have to be informed before method execution, and a set named *after* containing all user-defined method event classes that have to be informed after method execution.
3. Each method event class $methEvntCls$ contained in *before* is instantiated: Therefore, method *Before_Raise* is executed on $methEvntCls$; a new instance of $methEvntCls$ is returned. Then, method *Raise* is performed on $methEvntCls$. Together with the original arguments of m, the new event instance is passed as argument *newEvntInst* of *Raise*. Method *After_Raise* executed on class $methEvntCls$ concludes the instantiation.

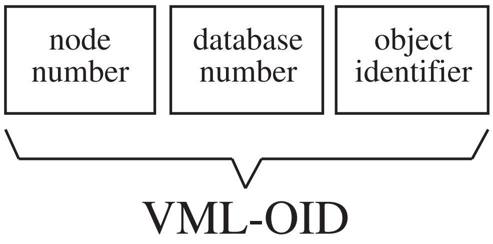


Figure6.10: The structure of VML-OIDs

4. Method m is actually executed on object o.

5. Each method event class *methEvntCls* contained in *after* is instantiated: Therefore, the same methods are engaged as in step 3.

6. The result of the execution of m is returned.

6.3 Transaction Commitment Notifier

The transaction commitment notifier represents an extension of the VODAK Transaction Manager (cf. Figure 5.10), which provides the functionality to begin, to end (commit), and to abort (sub-)transactions. To realize the deferred coupling mode, top-level transactions that attempt to commit have to be monitored in ActiveVODAK. Since ActiveVODAK does not support transaction events, begin and abort of transactions do not represent occurrences requiring special reactions.

When the Transaction Manager is called to commit a top-level transaction t, the following steps are performed: First, all objects that have been loaded into the object buffer within transaction t are made persistent. Then, the Recovery Manager (a submodule of the Transaction Manager) is informed about commitment of t. Finally, the Lock Manager (a submodule of the Transaction Manager, too) is engaged to release all locks obtained by t. Deferred condition evaluations/action executions are part of the triggering top-level transaction. Within condition evaluation/action execution new objects may be loaded into the object buffer and new locks may be requested. Consequently, a top-level transaction's ready-to-commit status has to be signalled to active databases before the three steps enumerated above can be executed.

Within a single top-level transaction objects of different databases may be accessed, i.e., rules defined in different databases may be triggered. Consequently, all databases accessed by the top-level transaction t must be informed about commitment of t. The implementation of a strategy, in which only actually accessed databases are informed, proved to be complicated: It would have been necessary to explicitly keep track of the databases accessed. The strategy adopted in ActiveVODAK informs all databases that potentially can be accessed by a transaction t, i.e., all databases that have been opened by the VODAK process to which t belongs.

Each active database contains the class *BeforeCommit*. When a top-level transaction t is ready-to-commit, method *TransBeforeCommit* is performed on the *BeforeCommit*-class of each database that has been opened by the VODAK process to which t belongs. To perform method *TransBeforeCommit*, the transaction commitment notifier utilizes the services of the Message Handler.

6.4 Clock Process

A clock process—a complete VODAK process—periodically sends time pulses to active databases. There is only one clock process per node, i.e., a single clock process serves all databases located at the clock process' node. Further, a clock process represents a daemon process: Once it is started, it is never aborted.

Two environment variables determine the behaviour of a clock process: The variable *DB_UPDATE_INT* specifies the interval (in seconds) between two points in time, at which the set of active databases is updated. Every time, a new database is created on node n, a description of the new database is inserted into the system database (data dictionary) of node n. The clock process periodically (every *DB_UPDATE_INT*-seconds) determines the set of existing active databases by consulting the system database. The clock process opens databases not yet opened and periodically sends time pulses to them.

The environment variable *SIGNAL_INT* specifies the interval (in seconds) between two points in time, at which time pulses are sent to the existing databases. Every *SIGNAL_INT*-seconds, the clock process performs method *SignalTime* on class *TimeEvnt* of each existing active database. To perform method *SignalTime*, a clock process utilizes the services of the Message Handler.

Chapter 7

Outlook

ActiveVODAK, as described in the previous chapters, represents an active database system providing *basic* active capabilities. The active kernel of ActiveVODAK is not tailored to requirements of specific fields of application. Future work on ActiveVODAK can proceed in two directions: It may either attempt to enhance the kernel capabilities of ActiveVODAK, or to tailor the active capabilities of ActiveVODAK to the special requirements of specific fields of application. In the sequel, basic ideas for future work are briefly presented:

1. *Enhancing the kernel system:* Starting points for enhancing the active kernel of ActiveVODAK may be:

   - *Value events:* The current version of ActiveVODAK does not support value events (cf. subsection 5.2.1.1). If value events are needed, they must be mapped to method events. But, for performance reasons a direct support of value events might be desireable.

     System internal methods represent potential anchors to detect READ and WRITE events. To assign a value *val* to a property *prop* (i.e., to WRITE the property *prop*) ActiveVODAK internally calls the predefined method *SYS_set_prop(val)*. To access the value of *prop* (i.e., to READ the property *prop*) the predefined method *SYS_prop()* is engaged. Those system internal methods are utilized irrespective of the property's type (primitve data type, structured data type, object type). Besides reading or writing, manipulation of structured data types is manifold. For example, the following primitive functions are defined for *sets*: union, intersect, diff, includes, insert, remove, and size. All these primitive functions have to be considered, when ActiveVODAK is enhanced with value events.

   - *Additional complex event operators:* The current version of ActiveVODAK provides operators for conjunctions, sequences, and negations. Other event operators explained in the diverse literature are ([SR94]):

     - *Disjunction:* A disjunction of two or more basic events represents a complex event that occurs, when an arbitrary basic event has occurred. Thus, the disjunction of n basic events corresponds to n rules with equal conditions and actions.

  – *Star:* A star, consisting of a basic event and a monitoring interval, represents a complex event that occurs after termination of the monitoring interval, if the basic event has occurred an arbitrary number of times, but at least once, during the monitoring interval.
  – *History:* A history, consisting of a basic event, a monitoring interval, and a number of occurrences n, represents a complex event that occurs after the nth occurrence of the basic event during the monitoring interval.

- *Subtransactions:* The current implementation of schema "Rules" (cf. section 6.1) does not make use of the open nested transactions provided by VODAK (cf. subsection 4.1.2). A yet incomplete implementation of the VODAK Transaction Manager made it impossible to fully exploit open nested transactions within this thesis. Supposed that a complete and correct implementation were provided in the future, the definition of commutativity tables and of inverse methods would increase transaction parallelity and, thus, system performance.
- *Event locking mechanism:* To avoid transaction anomalies, a transaction is prevented from raising a complex event that has partially been raised by another transaction not yet committed (cf. subsection 5.3.3). This strategy is more restrictive than necessary: It locks a complex event class instead of *critical* complex event instances. A complex event instance is *critical* for transaction t, if it is incomplete and if a transaction not yet committed and different from t has contributed to it. Only event instances critical for transaction t must be "protected" from completion by t. Locking event instances instead of event classes would increase transaction parallelity and, thus, system performance.

  The event locking mechanism adopted in ActiveVODAK influences the instantiation of primitive events, too. When a primitive event is instantiated within transaction t, a method m (depending on the event type) is performed on the primitive event class, i.e., the primitive event class is locked for transaction t. Unless method m is defined as a commuting subtransaction, all transactions different from t are prevented from raising the primitive event. As mentioned above, a full exploitation of subtransactions would increase transaction parallelity.

2. *Tailoring the system to specific fields of application:* As mentioned in the introducing chapter 1, multimedia synchronization and workflow management are promising fields of application for active capabilities in an integrated publication environment:

   - *Multimedia synchronization:* General problems of synchronizing the presentation of multimedia objects are discussed in [Hoe91]. Aberer/Klas ([AK94]) make concrete considerations for supporting temporal multimedia operations in VODAK. They introduce *schedules* to model the time-dependent behaviour associated with multimedia data. ECA rules, as supported in ActiveVODAK, seem to be a solid basis for the realization of the proposed schedules. First attempts to implement schedules by means of ECA rules have already been made during the development of ActiveVODAK. These attempts, which should not be described in more detail here, proved to be promising.

- *Workflow management:* The central role of active databases in workflow management is emphasized, when workflow management is named as an application that will certainly be supported by active database systems in the not-too-distant future ([Wid94]). Bussler/Jablonski engage ECA rules to implement agent coordination in workflows, i.e., in multi-step, multi-user activities ([BJ94]). Information retrieval, editing, and publication represent workflows in an integrated publication environment that potentially can be supported by ECA rules as they are realized in ActiveVODAK.

Appendix A

Bookseller

Throughout this thesis, a bookseller's repertoire of tasks served to illustrate active capabilities. This chapter presents the bookseller's basic data model and the rules defined to support the tasks introduced in chapter 1.

A.1 Basic Data Model

In the bookseller's basic data model some simplifying assumptions have been made:

- The only way to decrease the number of a book's copies in stock is to sell one or more copies. Faulty books that are returned to a supplier are not considered.
- An order placed with a supplier concerns only a single publication. If different publications should be ordered, one order has to be placed for each publication.
- A sale concerns only one publication, too. If a customer buys copies of more than one publication, several sales have to be carried out.

```
SCHEMA BookSeller

IMPORT Rules FROM Rules;
IMPORT Invent FROM Invent;

DEFINE THRESHOLD = 5;
DEFINE REORDERQUNTY = 5;

//*** Generalization **************************************************

CLASS Generalization METACLASS Metaclass
    INSTTYPE Gen_InstType
    INSTINSTTYPE Gen_InstInstType
END;
```

```
OBJECTTYPE Gen_InstType SUBTYPEOF Metaclass_InstType;

    INTERFACE
      METHODS
        AddSpecCls(specCls: OID);
          // adds spec to the set of classes (specClss) representing
          // specializations of the generalization class

        GetSpecClss(): {OID}  READONLY;
          // returns the set of classes (specClss) representing
          // specializations of the generalization class

    IMPLEMENTATION
      PROPERTIES
        specClss: {OID};
          // classes representing specializations of the
          // generalization class

      METHODS
        AddSpecCls(specCls: OID);
        {
            specClss->>insert(specCls);
        };

        GetSpecClss(): {OID}  READONLY;
        {
            RETURN specClss;
        };
END;


OBJECTTYPE Gen_InstInstType SUBTYPEOF Metaclass_InstInstType;

    INTERFACE
      METHODS
        SetSpec(specObj: OID);
          // sets the specialization of the generalized instance
          // (spec) to specObj

        AsSpec(): OID READONLY;
          // returns the specialization of the generalized
          // instance (spec)

    IMPLEMENTATION
      PROPERTIES
        spec: OID;
          // specialization of the generalized instance

      METHODS
        SetSpec(specObj: OID);
        {
            spec:=specObj;
        };
```

```
        AsSpec(): OID READONLY;
        {
            RETURN spec;
        };
END;


//*** Specialization ***********************************************

CLASS Specialization METACLASS Metaclass
    INSTTYPE Spec_InstType
    INSTINSTTYPE Spec_InstInstType
END;


OBJECTTYPE Spec_InstType SUBTYPEOF Metaclass_InstType;

    INTERFACE
      METHODS
        SetAsSpecClsOf(genClass: OID);
          // defines the specialized class to be a specialization
          // of genCls; this method is called for initialization
          // purpose only (cf. INIT clause of the specialized classes)

        CreateSpecInstOf(genInst: OID): OID;
          // creates a specialized instance for genInst

        GetGenCls(): OID READONLY;
          // returns the class representing the generalization
          // of the specialized class


    IMPLEMENTATION
      PROPERTIES
        genCls: OID;
          // generalization class

      METHODS
        SetAsSpecClsOf(genClass: OID);
        {
            genCls:=genClass;
            genClass->AddSpecCls(SELF);
        };


        CreateSpecInstOf(genInst: OID): OID;
        {
            VAR specObj: OID;

            specObj:=SELF->new();
            specObj->SetGen(genInst);
            genInst->SetSpec(specObj);
            RETURN specObj;
        };

        GetGenCls(): OID READONLY;
```

```
        {
            RETURN genCls;
        };
END;


OBJECTTYPE Spec_InstInstType SUBTYPEOF Metaclass_InstInstType;

    INTERFACE
      METHODS
        SetGen(genInst: OID);
          // sets the generalization of the specialized instance
          // (gen) to genInst

        AsGen(): OID;
          // returns the generalization of the specialized
          // instance (gen)

    IMPLEMENTATION
      PROPERTIES
        gen: OID;

    METHODS
        SetGen(genInst: OID);
        {
            gen:=genInst;
        };

        AsGen(): OID;
        {
            RETURN gen;
        };

        NOMETHOD

        { RETURN gen->currentMeth(arguments);}

END;


//*** Publication ****************************************************

CLASS Publication METACLASS Generalization
    OWNTYPE  Publication_OwnType
    INSTTYPE Publication_InstType
END;


OBJECTTYPE StockItem;
    INTERFACE
      PROPERTIES
        price: REAL;
        supplier: STRING;
        threshold: INT;
        itemsOnStock: INT;
```

```
        itemsOrdered: INT;

      METHODS
        SellQuntty(noOfItems: INT);
        OrderQuntty(noOfItems: INT);
        RecordItemArrival(noOfItems: INT);
        Available(no: INT):BOOL;

    IMPLEMENTATION
      METHODS
        SellQuntty(noOfItems: INT);
        {
            itemsOnStock:=itemsOnStock-noOfItems;
        };

        OrderQuntty(noOfItems: INT);
        {
            itemsOrdered:=itemsOrdered+noOfItems;
        };

        RecordItemArrival(noOfItems: INT);
        {
            itemsOrdered:=itemsOrdered-noOfItems;
            itemsOnStock:=itemsOnStock+noOfItems;
        };

        Available(no: INT):BOOL;
        {
            RETURN itemsOnStock >= no;
        };
END;


OBJECTTYPE Publication_OwnType;
    INTERFACE
      METHODS
        Create(tPrice: REAL, tSupplier:STRING, tISBN: STRING,
               tTitle: STRING, tPublishers: STRING): Publication;
          // a new publication is created

        PubWithISBN(tISBN: STRING):Publication;
          // publication with the specified ISBN is returned

    IMPLEMENTATION
      METHODS
        Create(tPrice: REAL, tSupplier:STRING, tISBN: STRING,
               tTitle: STRING, tPublishers: STRING): Publication;
        {
            VAR newPub:Publication;

            newPub:=SELF->new();
            newPub.price:=tPrice;
            newPub.supplier:=tSupplier;
            newPub.threshold:=5;
            newPub.ISBN:=tISBN;
            newPub.title:=tTitle;
```

```
            newPub.publishers:=tPublishers;
            RETURN newPub;
        };

        PubWithISBN(tISBN: STRING):Publication;
        {
            VAR i:Publication;

            FORALL (i IN SELF->allInstances())
            {
                IF (i.ISBN == tISBN) RETURN i;
            };
            RETURN NULL;
        };
END;

OBJECTTYPE Publication_InstType SUBTYPEOF StockItem;
    INTERFACE
      PROPERTIES
        ISBN: STRING;
        title: STRING;
        publishers: STRING;

    IMPLEMENTATION
END;


//*** Book ***********************************************************

CLASS Book METACLASS Specialization
    OWNTYPE  Book_OwnType
    INSTTYPE Book_InstType
    INIT Book->SetAsSpecClsOf(Publication)
END;


OBJECTTYPE Book_OwnType;
    INTERFACE
      METHODS
        NewBook(tPrice: REAL, tSupplier: STRING, tISBN: STRING,
                tTitle: STRING, tPublishers: STRING, tAuthor: STRING,
                tYear: INT, tPlace: STRING): Book;
          // a new book is created:
          // - a generalized instance of class Publication is created
          // - the new book is set to be the specialized instance
          //   of this new publication

    IMPLEMENTATION
      METHODS
        NewBook(tPrice: REAL, tSupplier: STRING, tISBN: STRING,
                tTitle: STRING, tPublishers: STRING, tAuthor: STRING,
                tYear: INT, tPlace: STRING): Book;
        {
            VAR newPub: Publication;
            VAR newBook: Book;
```

```
            newPub:=Publication->Create(tPrice,tSupplier,tISBN,
                    tTitle,tPublishers);
            newBook:=SELF->CreateSpecInstOf(newPub);
            newBook.author:=tAuthor;
            newBook.year:=tYear;
            newBook.place:=tPlace;
            RETURN newBook;
        };
END;


OBJECTTYPE Book_InstType;
    INTERFACE
      PROPERTIES
        author: STRING;
        year: INT;
        place: STRING;

    IMPLEMENTATION
END;


//*** Journal *********************************************************

CLASS Journal METACLASS Specialization
    OWNTYPE  Journal_OwnType
    INSTTYPE Journal_InstType
    INIT Journal->SetAsSpecClsOf(Publication)
END;


OBJECTTYPE Journal_OwnType;
    INTERFACE
      METHODS
        NewJournal(tPrice: REAL, tSupplier: STRING, tISBN: STRING,
                   tTitle: STRING, tPublishers: STRING, tEditor: STRING,
                   tIssue: INT): Journal;
          // a new journal is created:
          // - a generalized instance of class Publication is created
          // - the new journal is set to be the specialized instance
          //   of this new publication

    IMPLEMENTATION
      METHODS
        NewJournal(tPrice: REAL, tSupplier: STRING, tISBN: STRING,
                   tTitle: STRING, tPublishers: STRING, tEditor: STRING,
                   tIssue: INT): Journal;
        {
            VAR newPub: Publication;
            VAR newJournal: Journal;

            newPub:=Publication->Create(tPrice,tSupplier,tISBN,
                    tTitle,tPublishers);
            newJournal:=SELF->CreateSpecInstOf(newPub);
            newJournal.editor:=tEditor;
            newJournal.issue:=tIssue;
```

```
            RETURN newJournal;
        };
END;


OBJECTTYPE Journal_InstType;
    INTERFACE
      PROPERTIES
        editor: STRING;
        issue: INT;

    IMPLEMENTATION
END;


//*** Order **********************************************************

CLASS Order
    OWNTYPE  Order_OwnType
    INSTTYPE Order_InstType
END;


OBJECTTYPE Order_OwnType;
    INTERFACE
      METHODS
        GetNewOrderId():INT;
          // each order placed at a supplier has a unique number;
          // a number not used so far is returned by this method

        GetOrderWithId(orderId: INT):Order;
          // the order with number 'orderId' is returned

        NewOrder(orderId:INT, tSupplier:STRING, tPub:Publication,
                 tNoOfItems:INT):Order;
          // a new order is composed; the new order is not
          // automatically placed at the supplier

    IMPLEMENTATION
      PROPERTIES
        lastOrderId:INT;
          // id of the last order composed

      METHODS
        GetNewOrderId():INT;
        {
            lastOrderId:=lastOrderId+1;
            RETURN lastOrderId;
        };

        GetOrderWithId(orderId: INT):Order;
        {
            VAR i:Order;

            FORALL (i IN SELF->allInstances())
            {
```

```
                IF (i.id == orderId) {RETURN i;};
            };
            RETURN NULL;
        };

        NewOrder(orderId:INT, tSupplier:STRING, tPub:Publication,
                 tNoOfItems:INT):Order;
        {
            VAR newOrder:Order;

            newOrder:=Order->new();
            newOrder.id:=orderId;
            newOrder.supplier:=tSupplier;
            newOrder.pub:=tPub;
            newOrder.noOfItems:=tNoOfItems;
            newOrder.status:='registered';

            RETURN newOrder;
        };
END;


OBJECTTYPE Order_InstType;
    INTERFACE
      PROPERTIES
        id: INT;
        supplier: STRING;
        pub: Publication;
        noOfItems: INT;
        status: STRING;

      METHODS
        PlaceOrder();
          // the order is placed at the supplier

        RecordShipment();
          // order shipment is recorded

    IMPLEMENTATION

      EXTERN prints(s: STRING);
      EXTERN printi(i: INT);
      EXTERN endline();

      METHODS
        PlaceOrder();
        {
            pub->OrderQuntty(noOfItems);
            prints('Order number ');
            printi(id);
            prints(' has been placed!');
            endline();
            status:='placed';
        };

        RecordShipment();
```

```
        {
            pub->RecordItemArrival(noOfItems);
            prints('Order number ');
            printi(id);
            prints(' has been shipped!');
            endline();
            status:='shipped';
        };
END;


//*** Sale ***********************************************************

CLASS Sale
    OWNTYPE  Sale_OwnType
    INSTTYPE Sale_InstType
END;


OBJECTTYPE Sale_OwnType;
    INTERFACE
      METHODS
        GetNewSaleId():INT;
          // each sale is identified by a unique number; a number
          // not used so far is returned by this method

        GetSaleWithId(saleId: INT):Sale;
          // the sale with number 'saleId' is returned

        NewSale(saleId:INT, tCustomer:STRING, tPub:Publication,
                 tNoOfItems:INT):Sale;
          // a new sale is composed; it is not executed automatically

    IMPLEMENTATION
      PROPERTIES
        lastSaleId:INT;

      METHODS
        GetNewSaleId():INT;
        {
            lastSaleId:=lastSaleId+1;
            RETURN lastSaleId;
        };

        GetSaleWithId(saleId: INT):Sale;
        {
            VAR i:Sale;

            FORALL (i IN SELF->allInstances())
            {
                IF (i.id == saleId) {RETURN i;};
            };
            RETURN NULL;
        };

        NewSale(saleId:INT, tCustomer:STRING, tPub:Publication,
```

```
                 tNoOfItems:INT):Sale;
        {
            VAR newSale:Sale;

            newSale:=Sale->new();
            newSale.id:=saleId;
            newSale.customer:=tCustomer;
            newSale.pub:=tPub;
            newSale.noOfItems:=tNoOfItems;
            newSale.status:='open';
            newSale.pub->SellQuntty(tNoOfItems);

            RETURN newSale;
        };
END;


OBJECTTYPE Sale_InstType;
    INTERFACE
      PROPERTIES
        id: INT;
        customer: STRING;
        pub: Publication;
        noOfItems: INT;
    status: STRING;

      METHODS
        SendInvoice();
          // a sale is executed

        RecordPayment();
          // a sale is payed for

    IMPLEMENTATION

      EXTERN prints(s: STRING);
      EXTERN printi(i: INT);
      EXTERN endline();

      METHODS
        SendInvoice();
        {
            prints('Sale number ');
            printi(id);
            prints(' has been invoiced!');
            endline();
            status:='invoiced';
        };

        RecordPayment();
        {
            prints('Sale number ');
            printi(id);
            prints(' has been payed for!');
            endline();
            status:='payed';
```

```
        };
END;


END_SCHEMA;
```

A.2 Active Extensions

Section 5.2 has illustrated the definitions of events, actions, conditions, and rules by means of examples. All the examples contribute to cope with the bookseller's tasks of inventory control, keeping track of orders placed, reminding for payment, price reduction, and book announcement. This section groups all the definitions according to the tasks they support.

A.2.1 Inventory Control

```
//*** Method Event: SellEvent ***************************************

CLASS SellEvent METACLASS MethEvntOnInst
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Raise(newEvnt: OID, noOfItems: INT);

  IMPLEMENTATION METHODS
    Raise(newEvnt: OID, noOfItems: INT);
    {
        newEvnt->SYS_set_noOfItemsBnd(noOfItems);
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE PROPERTIES
     noOfItemsBnd: INT;

  IMPLEMENTATION
END;
INIT SellEvent->Init(Publication,'Sell',AFTER)
END;



//*** Condition: ThresholdCheck **************************************

CLASS ThresholdCheck METACLASS Condtn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    BelowThreshold():BOOL READONLY;

  IMPLEMENTATION METHODS
```

```
    BelowThreshold():BOOL READONLY;
    {
        VAR publication: OID;
        VAR book: OID;
        VAR threshold: INT;
        VAR itemsOnStock: INT;
        VAR itemsOrdered: INT;

        publication:=SELF->(OID)Event()->(OID)SYS_o();
        book:=publication->(OID)AsSpec();
        IF (book->(OID)class()==Book) {
            threshold:=publication->(INT)SYS_threshold();
            itemsOnStock:=publication->(INT)SYS_itemsOnStock();
            itemsOrdered:=publication->(INT)SYS_itemsOrdered();
            RETURN itemsOnStock<threshold AND
               itemsOrdered==0; };
        RETURN FALSE;
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT ThresholdCheck->Init('BelowThreshold')
END;



//*** Action: Reorder ************************************************

CLASS Reorder METACLASS Actn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    InitiateReorder();

  IMPLEMENTATION METHODS
    InitiateReorder();
    {
        VAR publication: OID;
        VAR newOrderNo: INT;
        VAR newOrder: OID;
        VAR supplier: STRING;

        publication:=SELF->(OID)Event()->(OID)SYS_o();
        supplier:=publication->(STRING)SYS_supplier();
        newOrderNo:=Order->GetNewOrderId();
        newOrder:=Order->NewOrder(newOrderNo,supplier,
                      publication,REORDERQUNTY);
        newOrder->Place();
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
```

```
END;
INIT Reorder->Init('InitiateReorder')
END;



//*** Rule: InventoryControl ****************************************

CLASS InventoryControl METACLASS Rule
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT InventoryControl->Init(SellEvent,ThresholdCheck,'imm',
                            Reorder,'imm')
END;
```

A.2.2 Keeping Track of Orders Placed

```
//*** Method Event: PlaceOrderEvent **********************************

CLASS PlaceOrderEvent METACLASS MethEvntOnInst
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Raise(newEvnt: OID);

  IMPLEMENTATION METHODS
    Raise(newEvnt: OID);
    {
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT PlaceOrderEvent->Init(Order,'PlaceOrder',AFTER)
END;



//*** Relative Time Event: TwoWeeksAfterOrder ************************

CLASS TwoWeeksAfterOrder METACLASS RltvTimeEvnt
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT TwoWeeksAfterOrder->Init(PlaceOrderEvent,'14 00:00:00')
END;



//*** Monitoring Interval: OrderExecInt ******************************
```

```
CLASS OrderExecInt METACLASS MonitInt
INSTTYPE OBJECTTYPE
  INTERFACE
    SameOrder():BOOL;

  IMPLEMENTATION
    SameOrder():BOOL;
    {
        RETURN endEvntToCheck->(OID)SYS_evntInst() ==
               begEvntToCheck;
    };
END;
INIT OrderExecInt->Init(PlaceOrderEvent,TwoWeeksAfterOrder,'SameOrder')
END;



//*** Method Event: ShipOrderEvent ***********************************

CLASS ShipOrderEvent METACLASS MethEvntOnInst
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Raise(newEvnt: OID);

  IMPLEMENTATION METHODS
    Raise(newEvnt: OID);
    {
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT ShipOrderEvent->Init(Order,'RecordShipment',AFTER)
END;



//*** Sequence: PlacedOrderShipped ***********************************

CLASS PlacedOrderShipped METACLASS Sequence
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT PlacedOrderShipped->
     Init([<PlaceOrderEvent,ShipOrderEvent>],'SameObject')
END;



//*** Negation: DeliveryDelay ****************************************

CLASS DeliveryDelay METACLASS Negation
```

```
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT DeliveryDelay->Init(ShipOrderEvent,OrderExecInt,'SameObject')
END;



//*** Condition: EmptyCondition ***************************************

CLASS EmptyCondition METACLASS Condtn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    AlwaysTrue():BOOL READONLY;

  IMPLEMENTATION METHODS
    AlwaysTrue():BOOL READONLY;
    {
        RETURN TRUE;
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT EmptyCondition->Init('AlwaysTrue')
END;



//*** Action: InformSupplier ******************************************

CLASS InformSupplier METACLASS Actn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Inform();

  IMPLEMENTATION METHODS
    Inform();
    {
        // At this position, writing of a notification must be
        // initiated. Notifications are not part of the data
        // model considered!
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT InformSupplier->Init('Inform')
END;
```

```
//*** Rule: SupplierNotification ************************************

CLASS SupplierNotification METACLASS Rule
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT SupplierNotification->Init(PlacedOrderShipped,EmptyCondition,'def',
                                InformSupplier,'def')
END;



//*** Action: UrgeSupplier ******************************************

CLASS UrgeSupplier METACLASS Actn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Urge();

  IMPLEMENTATION METHODS
    Urge();
    {
        // At this position, the supplier has to be urged
        // on delivery. Urging is not part of the data model
        // considered!
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT UrgeSupplier->Init('Urge')
END;



//*** Rule: UrgeOnDelivery ******************************************

CLASS UrgeOnDelivery METACLASS Rule
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT UrgeOnDelivery->Init(DeliveryDelay,EmptyCondition,'ind',
                          UrgeSupplier,'ind')
END;
```

A.2.3 Reminding Policy

```
//*** Abstract Event: DelayForPayment ********************************
```

```
CLASS DelayForPayment METACLASS AbstrctEvnt
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Raise(sale: INT, amount: REAL);

  IMPLEMENTATION METHODS
    Raise(sale: INT, amount: REAL);
    {
        VAR newEvnt: OID;

        newEvnt:=SELF->NewListElem();
        newEvnt->SYS_set_saleNo(sale);
        newEvnt->SYS_set_openAmount(amount);
        SELF->InformSubscribers(newEvnt);
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE PROPERTIES
    saleNo: INT;
    openAmount: REAL;

  IMPLEMENTATION
END;
INIT DelayForPayment->Init()
END;



//*** Method Event: SalePayed ****************************************

CLASS SalePayed METACLASS MethEvntOnInst
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Raise(newEvnt: OID);

  IMPLEMENTATION METHODS
    Raise(newEvnt: OID);
    {
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT SalePayed->Init(Sale,'RecordPayment',AFTER)
END;



//*** Periodic Time Event: ReminderEvent *****************************

CLASS ReminderEvent METACLASS PrdcTimeEvnt
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
```

```
    SameSale():BOOL;

  IMPLEMENTATION
    SameSale():BOOL;
    {
        RETURN evntToCheck->(INT)SYS_saleNo() ==
               firingToCheck->(INT)SYS_saleNo();
    };
END;
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT ReminderEvent->Init(DelayForPayment,'21 00:00:00','SameSale')
END;



//*** Monitoring Interval: OpenSaleInt ******************************

CLASS OpenSaleInt METACLASS MonitInt
INSTTYPE OBJECTTYPE
  INTERFACE
    SameSale():BOOL;

  IMPLEMENTATION
    SameSale():BOOL;
    {
        VAR saleNo: INT;
        VAR openSale: OID;

        saleNo:=begEvntToCheck->(INT)SYS_saleNo();
        openSale:=Sale->GetSaleWithId(saleNo);

        RETURN endEvntToCheck->(OID)SYS_o() == openSale;
    };
END;
INIT OpenSaleInt->Init(DelayForPayment,SalePayed,'SameSale')
END;



//*** Monitored Event: ReminderToSendEvent ***************************

CLASS ReminderToSendEvent METACLASS MonitEvnt
INSTTYPE OBJECTTYPE
  INTERFACE
    SameSale():BOOL;

  IMPLEMENTATION
    SameSale():BOOL;
    {
        VAR saleNo: INT;
        VAR startEvnt: OID;

        startEvnt:=evntToCheck->(OID)SYS_evntInst();
```

```
        saleNo:=startEvnt->(INT)SYS_saleNo();

        RETURN monitIntToCheck->(OID)SYS_beginEvnt()->
               (INT)SYS_saleNo() == saleNo;
    };
END;
INIT ReminderToSendEvent->Init(ReminderEvent,OpenSaleInt,'SameSale')
END;



//*** Condition: EmptyCondition *************************************

CLASS EmptyCondition METACLASS Condtn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    AlwaysTrue():BOOL READONLY;

  IMPLEMENTATION METHODS
    AlwaysTrue():BOOL READONLY;
    {
        RETURN TRUE;
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT EmptyCondition->Init('AlwaysTrue')
END;



//*** Action: SendReminder *******************************************

CLASS SendReminder METACLASS Actn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Send();

  IMPLEMENTATION METHODS
    Send();
    {
        // At this position, writing of a reminder must be
        // initiated. Reminders are not part of the data
        // model considered!
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT SendReminder->Init('Send')
END;
```

```
//*** Rule: RemindingPolicy ****************************************

CLASS RemindingPolicy METACLASS Rule
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT RemindingPolicy->Init(ReminderToSendEvent,EmptyCondition,'ind',
                           SendReminder,'ind')
END;
```

A.2.4 Price Reduction

```
//*** Absolute Time Event: On1Jan95 **********************************

CLASS On1Jan95 METACLASS AbsTimeEvnt
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT On1Jan95->Init('95-01-01')
END;



//*** Condition: EmptyCondition **************************************

CLASS EmptyCondition METACLASS Condtn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    AlwaysTrue():BOOL READONLY;

  IMPLEMENTATION METHODS
    AlwaysTrue():BOOL READONLY;
    {
        RETURN TRUE;
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT EmptyCondition->Init('AlwaysTrue')
END;



//*** Action: ReduceBookPrice ****************************************

CLASS ReduceBookPrice METACLASS Actn
```

```
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Reduce();

  IMPLEMENTATION METHODS
    Reduce();
    {
        VAR aBook:Book;
        VAR aPubl:Publication;
        VAR yearOfPubl:INT;
        VAR bookPrice:REAL;

        FORALL (aBook IN Book->allInstances()) {
        yearOfPubl:=aBook->(INT)SYS_year();
        IF (yearOfPubl+5 < 95) {
            aPubl:=aBook->(Publication)AsGen();
            bookPrice:=aPubl->(REAL)SYS_price();
            bookPrice:=bookPrice*0.8;
            aPubl->SYS_set_price(bookPrice); }; };
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT ReduceBookPrice->Init('Reduce')
END;



//*** Rule: PriceReduction *******************************************

CLASS PriceReduction METACLASS Rule
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT PriceReduction->Init(On1Jan95,EmptyCondition,'ind',
                          ReduceBookPrice,'ind')
END;
```

A.2.5 Book Announcement

```
//*** Method Event: NewBookEvent ************************************

CLASS NewBookEvent METACLASS MethEvntOnObject
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Raise(newEvnt: OID, price: REAL,
          supplier: STRING, title: STRING,
          publishers: STRING, ISBN: STRING,
          author: STRING, year: INT,
          place: STRING);
```

```
  IMPLEMENTATION METHODS
    Raise(newEvnt: OID, price: REAL,
          supplier: STRING, title: STRING,
          publishers: STRING, ISBN: STRING,
          author: STRING, year: INT,
          place: STRING);
    {
          newEvnt->SYS_set_priceBnd(price);
          newEvnt->SYS_set_supplierBnd(supplier)
          newEvnt->SYS_set_titleBnd(title);
          newEvnt->SYS_set_publishersBnd(publishers);
          newEvnt->SYS_set_ISBNBnd(ISBN);
          newEvnt->SYS_set_authorBnd(author);
          newEvnt->SYS_set_yearBnd(year);
          newEvnt->SYS_set_placeBnd(place);
     };
END;

INSTTYPE OBJECTTYPE
  INTERFACE PROPERTIES
    priceBnd: REAL;
    supplierBnd: STRING;
    titleBnd: STRING;
    publishersBnd: STRING;
    ISBNBnd: STRING;
    authorBnd: STRING;
    yearBnd: INT;
    placeBnd: STRING;

  IMPLEMENTATION
END;
INIT NewBookEvent->Init(Book,'NewBook',AFTER)
END;



//*** Condition: EmptyCondition ***************************************

CLASS EmptyCondition METACLASS Condtn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    AlwaysTrue():BOOL READONLY;

  IMPLEMENTATION METHODS
    AlwaysTrue():BOOL READONLY;
    {
        RETURN TRUE;
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT EmptyCondition->Init('AlwaysTrue')
```

```
END;



//*** Action: AnnounceNewBook ****************************************

CLASS AnnounceNewBook METACLASS Actn
OWNTYPE OBJECTTYPE
  INTERFACE METHODS
    Announce();

  IMPLEMENTATION METHODS
    Announce();
    {
        // At this position, the new book must be set on
        // a list of new books. This list of new books
        // is not part of the data model considered!
    };
END;

INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT AnnounceNewBook->Init('Announce')
END;



//*** Rule: BookAnnouncement *****************************************

CLASS BookAnnouncement METACLASS Rule
INSTTYPE OBJECTTYPE
  INTERFACE
  IMPLEMENTATION
END;
INIT BookAnnouncement->Init(NewBookEvent,EmptyCondition,'def',
                            AnnounceNewBook,'def')
END;
```

Appendix B

EBNF Notation

The formal notations used to specify the syntax of ECA rules corresponds to the *Extended Backus Naur Formalism (EBNF)* introduced by Nikolaus Wirth (cf. for example [Wir83, pages 10–11]).

The constructs that can be expressed in EBNF are:

- *Concatenation:* A construct $< A >$ consisting of $< B >$ followed by $< C >$ is denoted as:

  $$< A >=< B >< C >$$

- *Choice:* A construct $< A >$ consisting of $< B >$ or, alternatively, of $< C >$ is denoted as:

  $$< A >=< B > | < C >$$

- *Option:* A construct $< A >$ consisting of $< B >$ or nothing (empty) is denoted as:

  $$< A >= [< B >]$$

- *Repetition:* A construct $< A >$ consiting of any number of $< B >$ (including none) is denoted as:

  $$< A >= \{< B >\}$$

The syntactic entities available can be grouped into terminal symbols and non-terminal symbols. Non-terminal symbols are embraced by brackets. $< A >$, $< B >$, and $< C >$ in the above examples are non-terminal symbols. Rules must be defined for the construction of non-terminal symbols. Terminal symbols are embraced by quotation marks (e.g., “CLASS”). They represent the basic entities of a formal language.

Bibliography

[ABD+89] M. Atkinson; F. Bancilhon; D. DeWitt; K. Dittrich; D. Maier; S. Zdonik: *The Object-Oriented Database System Manifesto.* In: 1st Intl. Conference on Deductive and Object-Oriented Databases, Kyoto, 1989.

[AK94] K. Aberer; W. Klas: *Supporting Temporal Multimedia Operations in Object-Oriented Database Systems.* In: Proceedings of the IEEE Intl. Conference on Multimedia Computing and Systems, Boston, 1994.

[AF93] K. Aberer; G. Fischer: *VQL—VODAK Query Language, Version 1.0.* Working Paper, GMD-IPSI, 1993.

[AK92] K. Aberer; W. Klas: *The Impact of Multimedia Data on Database Management Systems.* Technical Report ICSI-TR-92-065, International Computer Science Institute, Berkeley, 1992.

[Ban92] F. Bancilhon (ed.): *Building an Object-Oriented Database System - The Story of O2.* Morgan Kaufmann Publishers, San Mateo, 1992.

[Bau90] B. Baumgarten: *Petri-Netze, Grundlagen und Anwendungen.* BI-Wiss.-Verlag, Mannheim, 1990.

[BJ94] C. Bussler; S. Jablonski: *Implementing Agent Coordination for Workflow Management Systems Using Active Database Systems.* In: Proceedings of The Fourth Intl. Workshop on Research Issues in Data Engineering, Texas, 1994.

[BM91] E. Bertino; L. Martino: *Object-Oriented Database Management Systems: Concepts and Issues.* In: IEEE Computer, April 1991.

[CAM93] S. Chakravarthy; E. Anwar; L. Maugis: *Design and Implementation of Active Capability for an Object-Oriented Database.* Technical Report, University of Florida, CIS, 1993.

[Cat91] R. G. G. Cattell: *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, Reading, 1991.

[CCS94] C. Collet; T. Coupaye; T. Svensen: *NAOS - Efficient and Modular Reactive Capabilities in an Object-Oriented Database System.*, Working Paper, private communication.

[CM93] S. Chakravarthy; D. Mishra: *Snoop: An Expressive Event Specification Language for Active Databases.* Technical Report, University of Florida, CIS, 1993.

[CN90] S. Chakravarthy; S. Nesson: *Making an Object-Oriented DBMS Active: Design, Implementation and Evaluation of a Prototype.* In: Proceedings of the Intl. Conference on Extending Database Technology, Venice, 1990.

[Day88] U. Dayal: *Active Database Management Systems.* In: Proceedings of the 3rd Intl. Conference on Data and Knowledge Bases, Jerusalem, 1988.

[DBB+88] U. Dayal; B. Blaustein; A. Buchmann et. al.: *The HiPAC Project: Combining Active Databases and Timing Constraints.* ACM SIGMOD RECORD, Vol. 17 No. 1, 1988.

[DKT88] H. Duchêne; M. Kaul; V. Turau: *VODAK Kernel Data Model.* In: K. R. Dittrich (Ed.), Advances in Object-Oriented Database Systems, Proc. of the 2nd Intl. Workshop on Object-Oriented Database Systems, Lecture Notes in Computer Science No. 334, Bad Münster, 1988.

[DPG91] O. Diaz; N. Paton; P. Gray: *Rule Management in Object-Oriented Databases: A Uniform Approach.* In: Proceedings of the 17th Intl. Conference on VLDB, Barcelona, 1991.

[Gat92] S. Gatziu: *Eine Ereignissprache für das aktive objektorientierte Datenbanksystem SAMOS.* Technical Report, Institute for Computer Science, University of Zürich, 1992.

[GD94] S. Gatziu; K. R. Dittrich: *Detecting Composite Events in Active Database Systems Using Petri Nets.* In: Proceedings of The Fourth Intl. Workshop on Research Issues in Data Engineering, Texas, 1994.

[GGD91] S. Gatziu; A. Geppert; K. R. Dittrich: *Integrating Active Concepts into an Object-Oriented Database System.* In: Proceedings of the 3rd Intl. Workshop on Database Programming Languages, Nafplion (Greece), 1991.

[GJ91] N. H. Gehani; H. V. Jagadish: *Ode as an Active Database: Constraints and Triggers.* In: Proceedings of the 17th VLDB Conference, Barcelona, 1991.

[GJS92a] N. H. Gehani; H. V. Jagadish; O. Shmueli: *Composite Event Specification in Active Databases: Model & Implementation.* In: Proceedings of the 18th VLDB Conference, Vancouver, Canada, 1992.

[GJS92b] N. H. Gehani; H. V. Jagadish; O. Shmueli: *Event Specification in an Object-Oriented Database.* Technical report, AT & T Bell Labs, 1992.

[GR83] A. Goldberg, D. Robson: *SMALLTALK-80: The Language and its Implementation.* Reading, Mass. [u.a.]; 1983.

[Heu92] A. Heuer: *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme.* Addison-Wesley, Bonn-München-Paris [u.a.], 1992.

[HLM88] M. Hsu; R. Ladin; D. R. McCarthy: *An Execution Model for Active Database Management Systems.* In: Proceedings of the 3rd Intl. Conference on Data and Knowledge Bases, Jerusalem, 1988.

[Hoe91] P. Hoepner: *Synchronizing the Presentation of Multimedia Objects — ODA Extensions.* In: Multimedia — Systems, Interaction, and Application; Ed.: L. Kjelldahl; Springer, Stockholm, 1991.

[KRR+93] G. Kappel, S. Rausch-Schott, W. Retschitzegger, S. Vieweg: *The TriGS Execution Model: Implementing Active Concepts on Top of a Commercial OODBMS.*, Technical Report, Research Institute for Applied Knowledge Processing, University of Linz, 1993.

[KRR+94] G. Kappel, S. Rausch-Schott, W. Retschitzegger, S. Vieweg: *TriGS: Making a Passive Object-Oriented Database System Active.* In: Journal of Object-Oriented Programming, Vol. 7, No. 4, July/August, 1994.

[Kim90] W. Kim: *Introduction to Object-Oriented Databases.* MIT-Press, 1990.

[Kla90] W. Klas: *A Metaclass System for Open Object-Oriented Data Models.* Doctoral Thesis, Technical University of Vienna, 1990.

[KNA+93] W. Klas, E. J. Neuhold, K. Aberer, R. Bahlke, P. Fankhauser, G. Fischer, M. Kaul, P. Muth, T. Rakow, V. Turau: *VML – VODAK Model Language.* Internal Report, GMD–IPSI, Darmstadt, Sept. 1993.

[KNS88] W. Klas; E. J. Neuhold; M. Schrefl: *On an Object-Oriented Data Model for a Knowledge Base.* In: R. Speth (Ed.), Research into Networks and Distributed Applications, EUTECO 88, North Holland, 1988.

[KNS89] W. Klas; E. J. Neuhold; M. Schrefl: *Tailoring Object-Oriented Data Models through Metaclasses.* In: Proc. of the Advanced Database Systems Symposium, Kyoto, 1989.

[Kot89] A. M. Kotz: *Triggermechanismen in Datenbanksystemen.* Informatik-Fachberichte, Springer, Berlin, 1989.

[MD89] D. R. McCarthy, U. Dayal: *The Architecture of an Active Database Management System.* In: Proceedings of the ACM SIGMOD Intl. Conference of Management of Data, Portland, 1989.

[Mey91] K. Meyer-Wegener: *Multimedia-Datenbanken.* Teubner, Stuttgart, 1991.

[MRK+94] P. Muth; T. C. Rakow; W. Klas; E. J. Neuhold: *A Transaction Model for an Open Publication Environment.* In: Database Transaction Models For Advanced Applications, Ed.: A. K. Elmagarmid, San Mateo, 1994.

[MRW+93] P. Muth; T. C. Rakow; G. Weikum; P. Brössler; C. Hasse: *Semantic Concurrency Control in Object-Oriented Database Systems.* In: Proceedings of the 9th Conf. on Data Engineering, Vienna, 1993.

[NT92] E. J. Neuhold; V. Turau: *Database Research at IPSI.* In: SIGMOD RECORD Vol. 21, No. 1, March 1992.

[Obj93] Object Design, Inc.: *ObjectStore*^{TM}. ObjectStore Release 3 for UNIX Systems Documentation, Burlington, 1993.

[Ora93] Oracle Corporation: *Oracle7 Symmetric Replication - Asynchronous Distributed Technology.* Part A13824, September 1993.

[SR94] M. Schrefl; B. Röck: *Active Object-Oriented Database Systems.* Manuscript, Department of Information Systems - Data & Knowledge Engineering, University of Linz, 1994.

[SRH90] M. Stonebraker; L. A. Rowe; M. Hirohama: *The Implementation of POSTGRES.* In: IEEE Transactions on Knowledge and Data Engineering, Special Issue on Database Prototype Systems, March 1990.

[Str87] B. Stroustrup: *Die Programmiersprache C++.* Reading, Mass.; 1987.

[US90] R. Unland; G. Schlageter: *Object-Oriented Database Systems: Concepts and Perspectives.* In: Database Systems of the 90s, Ed.: A. Blaser, Springer Verlag, Berlin, 1990.

[VG93] G. Vossen; M. Gross-Hardt: *Grundlagen der Transaktionsverarbeitung.* Addison-Wesley, Bonn, 1993.

[Wei88] G. Weikum: *Transaktionen in Datenbanksystemen.* Addison-Wesley, Bonn, 1988.

[Wid94] J. Widom: *Research Issues in Active Database Systems: Report from the Closing Panel at RIDE-ADS 94.* In: SIGMOD RECORD, Vol. 23, No. 3, September 1994.

[Wir83] N. Wirth: *Programming in MODULA-2.* Second, Corrected Edition, Springer, New York, 1983.