Active Object-Oriented Databases: From Conceptual Design to Logical Design

DISSERTATION
zur Erlangung des akademischen Grades
eines Doktors der Sozial- und Wirtschaftswissenschaften

Eingereicht an der Johannes Kepler Universität Linz
Institut für Wirtschaftsinformatik
Data & Knowledge Engineering

Betreuung:
o.Univ.-Prof. Dipl.-Ing. Dr. Michael Schrefl
o.Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel

Verfaßt von: **Mag. Werner Obermair**

Linz, im Februar 1998

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, daß ich die vorliegende Dissertation selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe. Diese Dissertation habe ich bisher weder im Inland noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt.

Linz, am 15. Juni 1998

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die zum Gelingen dieser Arbeit beigetragen haben. Ein ganz besonderer Dank gilt Michael Schrefl. Er hat mir nicht nur im Rahmen eines Projektes die Möglichkeit zu dieser Arbeit gegeben, sondern er hat auch durch zahlreiche Diskussionen mit vielen wertvollen Anregungen einen wesentlichen Beitrag zu dieser Arbeit geleistet. Besonders bedanken möchte ich mich auch bei meinem Projektkollegen Peter Lang. Sein Ideenreichtum, seine Ausdauer und seine Hilfsbereitschaft haben ganz wesentlich zur inhaltlichen und zur menschlichen Qualität der Projektarbeit beigetragen. Bei Prof. Gerti Kappel bedanke ich mich für die Übernahme der Zweitbegutachtung.

Weiters möchte ich mich bei allen Kolleginnen und Kollegen am Institut für Wirtschaftsinformatik der Universität Linz (Forschungsschwerpunkt Data & Knowledge Engineering), Ingeborg, Margit, Günter, Peter, Roman und Thomas, für das sehr angenehme Umfeld, in dem diese Arbeit entstehen konnte, bedanken.

Werner Kraus und Thomas Thalhammer danke ich für ihren Einsatz bei der Entwicklung des grafischen Editors. Die intensive Zusammenarbeit mit ihnen hat immer wieder neues Verständnis geschaffen.

Besonderer Dank gebührt meinen Eltern Helga und Günter. Sie haben mich über all die Jahre hinweg unterstützt und in meinen Entscheidungen bekräftigt.

Ein liebevolles Dankeschön an
Alexandra!

Kurzfassung

Der Entwurf von aktiven objektorientierten Datenbanken umfaßt die Modellierung der Objektstruktur, des passiven Objektverhaltens in Form von Operationen und des aktiven Objektverhaltens in Form von Geschäftsregeln. Geschäftsregeln beschreiben die Geschäftspolitik eines Unternehmens. Derartige Regeln werden als Ereignis-Bedingung-Aktions-Regeln von aktiven Datenbanksystemen unterstützt und erlauben, auf bestimmte vordefinierte Ereignisse mit der Ausführung einer Aktion zu reagieren, falls eine angegebene Bedingung beim Eintritt des Ereignisses erfüllt ist.

Ein Ansatz, der erfolgreich im Entwurf konventioneller Datenbanken verfolgt wird, ist, den Datenbankentwurf in zwei Phasen zu gliedern: In der ersten Phase, dem *konzeptuellen Entwurf*, wird ein ausdrucksstarkes graphisches Datenbankschema entwickelt. In der zweiten Phase, dem *logischen Entwurf*, wird das entwickelte Schema in das Datenmodell des Systems abgebildet, das für die Implementierung verwendet wird. In der vorliegenden Arbeit wird ein solcher Ansatz für den Entwurf aktiver objektorientierter Datenbanken verfolgt.

Aktive Objekt/Verhaltensdiagramme werden als Sprache für den konzeptuellen Entwurf aktiver objektorientierter Datenbanken eingeführt. Aktive Objekt/Verhaltensdiagramme erweitern Objekt/Verhaltensdiagramme, eine bestehende graphische Sprache für den konzeptuellen Entwurf von Objektstruktur und passivem Objektverhalten, um Konzepte zur Modellierung von Geschäftsregeln. Die Analyse von Fallbeispielen zeigt, daß die Modellierung von Geschäftsregeln auf konzeptueller Ebene andere Konzepte verlangt als sie zur Zeit von aktiven objektorientierten Datenbanksystemen und von aktuellen Ansätzen im Entwurf von aktiven objektorientierten Datenbanken zur Verfügung gestellt werden. Aktive Objekt/Verhaltensdiagramme umfassen eine graphische Regel- und Ereignissprache, die den identifizierten Anforderungen entspricht.

Im logischen Entwurf wird ein mit Aktiven Objekt/Verhaltensdiagrammen entworfenes Datenbankschema in ein logisches Datenbankschema eines aktiven objektorientierten Datenbanksystems abgebildet. In der vorliegenden Arbeit wird eine solche Abbildung für *TriGS* präsentiert. TriGS ist ein Prototyp eines aktiven objektorientierten Datenbanksystems, der auf der Basis des kommerziellen objektorientierten Datenbanksystems GemStone (GemStone Systems, Inc.) entwickelt wurde. Die vorgestellte Abbildung behandelt alle drei Dimensionen eines mit Aktiven Objekt/Verhaltensdiagrammen entworfenen Datenbankschemas: Objektstruktur, passives Objektverhalten und aktives Objektverhalten.

Abstract

The design of active object-oriented databases includes modeling the structure of objects, their passive behavior in the form of operations that can be performed on them, and their active behavior in the form of business rules. Business rules are statements about business policies and can be formulated according to the event-condition-action structure of rules provided by active database systems, which allow to react to predefined situations by performing an operation if a certain condition is satisfied when the event occurs.

An approach that has been followed successfully in conventional database design is to perform database design in two phases: In the first phase, the *conceptual design*, a high-level graphical representation of the database schema is developed. In the second phase, the *logical design*, the developed schema is mapped to the data model of the system used for the implementation. We adopt this approach for the design of active object-oriented databases.

We introduce *Active Object/Behavior Diagrams* for the conceptual design of active object-oriented databases. Active Object/Behavior Diagrams seamlessly extend Object/Behavior Diagrams, an existing high-level graphical language for modeling the structure and the passive behavior of objects, with concepts for modeling business rules. Modeling business rules at the conceptual level requires different concepts than currently provided by active object-oriented database systems and by recent approaches to active object-oriented database design. Active Object/Behavior Diagrams introduce a graphical rule and event language the meets the identified requirements.

During logical design, a schema developed with Active Object/Behavior Diagrams is mapped to a logical schema of an existing active object-oriented database system. We present such a mapping for TriGS, a prototype of an active object-oriented database system built on top of the commercial object-oriented database system GemStone (GemStone Systems, Inc.). The mapping decomposes the high-level constructs of Active Object/Behavior Diagrams into lower-level constructs of TriGS. The presented mapping covers all three dimensions of a schema specified with Active Object/Behavior Diagrams: object structure, passive object behavior, and active object behavior.

Contents

1 Introduction **3**

2 Active Object-Oriented Database Systems **9**

2.1 From Database Systems to Active Object-Oriented Database Systems . . . 10

2.2 Concepts for Modeling Active Behavior 16

2.3 Related Work on Active Object-Oriented Database Systems 29

3 Active Object-Oriented Database Design **33**

3.1 Information-System Design 34

3.2 From Conventional Database Design to Object-Oriented Database Design . 40

3.3 The Open Challenge: Business Rules 45

3.4 From Active Conventional Database Design to Active Object-Oriented Database Design 49

3.5 Tool Support 54

3.6 Related Work on Active Object-Oriented Database Design 55

4 Object/Behavior Diagrams **61**

4.1 Running Example: Communication System 63

4.2 Object-Type Diagrams 63

4.3 Object-Class Diagrams 71

4.4 Behavior Diagrams 73

4.5 Query Diagrams 80

5 Situation Diagrams **95**

5.1 Situation-Type Diagrams 97

5.2 Situation Classes 99

5.3 Situation Diagrams 103

5.4 Summary 115

6 Activation Diagrams **119**

6.1 Start-Rules and Do-Rules 120

6.2 Graphical Elements 123

6.3 Execution Semantics 127

6.4 Summary 131

7 Mapping Environment **133**

7.1 Target Model 133

7.2 Mapping Strategy 140

7.3 Related Work 146

8 Mapping Object Structure **149**

8.1 Types and Properties 150

8.2 Derived Properties 153

8.3 Query diagrams and OQL 157

9 Mapping Passive Object Behavior **171**

9.1 Activities and States 172

9.2 Integrity Maintenance 175

9.3 Task Model 199

10 Mapping Active Object Behavior **229**

10.1 Rationale 230

10.2 Situation Types and Situation Classes 235

10.3 Detecting Basic Database Situations 238

10.4 Detecting Logical Situations 240
10.5 Handling Rules 255

11 Conclusion **273**

11.1 Summary 273
11.2 Tool Support 278
11.3 Outlook 279

Chapter 1

Introduction

The domain of business applications usually defines statements about business policies. These statements, so-called "business rules," require to perform operations upon predefined situations, for example, to update selling prices at 12:00 noon. Conventional business applications have the user to invoke these operations upon certain situations. Since active database systems provide the ability to react to predefined situations without any user interaction, business rules can be formulated according to the event-condition-action structure of rules in active database systems. However, modeling business rules at the conceptual level from an external user's perspective requires a different set of concepts than currently provided by active object-oriented database systems [BZBW95, Cha97, GD94, LGA96] or by recent approaches to active object-oriented database design [BS94, CBFP95, Müc94, SC96].

After examining the basic features provided by event and rule languages of active database systems, we identify the requirements on the event language and on the semantics of rule execution for modeling business rules at the conceptual level and present a high-level graphical language meeting these requirements.

Active Object-Oriented Database Systems. Object-oriented database systems and active database systems aim at representing real-world semantics, which is otherwise hidden in application programs, in the database schema. Contrary to conventional database systems, object-oriented database systems model the (passive) behavior of objects next to their structure. Active database systems provide the ability to monitor the database state and to react to predefined situations without explicit user action or application requests. Situations to be observed and actions to be initiated are defined in the database schema (cf. [BS94]).

Active object-oriented database systems combine the capabilities of object-oriented database systems and active database systems. They represent objects of the "universe of discourse" by their structure and two forms of behavior: passive behavior and active behavior. The *passive behavior* describes that objects react to messages by invoking methods.

The *active behavior* describes that objects act autonomously by performing certain actions, if some predefined situation occurs.

Passive behavior is specified according to the object-oriented paradigm by a set of methods associated with object classes. Active behavior is usually specified by a set of event-condition-action rules (ECA rules) associated with object classes or individual objects. Objects act autonomously upon the occurrence of an event by performing the specified action if the condition is satisfied.

Active Object-Oriented Database Design. Modeling the universe of discourse, which comprises the set of problem-relevant objects of the business area, in a database schema is called database design. In addition to object-oriented database design, where the objects are modeled by means of object structure and passive behavior, active object-oriented database design considers the business rules defined in the business area. Business rules are represented by means of the active behavior of objects.

Two possible approaches for the design of active object-oriented databases are:

1. Use the schema definition language of a favorite active object-oriented database management system (cf. [WC96a]).
2. Develop a high-level graphical representation of the database schema during conceptual design. Have this schema mapped to the object model of the system used for the implementation during logical design.

The first approach faces two major inherent design difficulties because schema definition languages of active object-oriented database management systems provide only lower-level concepts particularly suitable for the logical design and not for the conceptual design of databases: (a) It is generally known from AI systems that large sets of rules are hard to design, hard to understand, and hard to maintain (cf. [KS93]). (b) In many cases rules and methods are alternative ways to express the same effect (cf. [KS96a]). E.g., in the case of method events, the action to the event may be specified immediately in the body of the method or in the action part of a rule triggered by the method event. If clear design guidelines are missing, large systems will be hard to maintain as different designers may model similar cases differently.

These difficulties are tried to be overcome by providing rule analysis tools to predict their behavior in realistic situations, further by providing simulation, testing, and debugging tools, and by using abstraction techniques like rule stratification (cf. [BCP96]), which enables the partitioning of complex rule sets into modules of rules with comprehensible and predictable behavior, or rule patterns (cf. [KRSRS96]), which allow to factor out the common behavior of rules. Nevertheless, the rule and event languages provided by active object-oriented database systems are usually too low-level to model business rules at the conceptual level. Rather, these rule and event languages are usually used in logical design,

for example, for checking integrity constraints and for maintaining views. By means of the term "business rules at the conceptual level" we refer to business policies defined in the domain of business application and not, for example, to integrity constraints, which are usually specified declaratively at the conceptual level. Further, we do not assume business rules to deal with real-time aspects.

Active Object/Behavior Diagrams. We follow the second approach, which has been followed successfully in conventional database design, and introduce *Active Object/Behavior Diagrams*, a high-level graphical design notation for the conceptual design of active object-oriented databases. Our approach extends *Object/Behavior Diagrams* [KS91a], a graphical object-oriented design notation for modeling the object structure and the passive behavior, with *Situation/Activation Diagrams* [LOS96, LOS97] for modeling the active behavior of objects. Situation/Activation Diagrams include activation diagrams and situation diagrams. Activation diagrams allow to model ECA rules, which are triggered upon the occurrence of situations, which are specified by means of situation diagrams. Bichler and Schrefl have laid the foundations for our approach in [BS94].

Active Object/Behavior Diagrams are presented in their basic form within this thesis. Specialization issues are not considered. Specialization of object structure and passive behavior in the realm of Object/Behavior Diagrams is already discussed in [Sch90, KS94, SS95, KS96b, SS97]. Inspired by the work on the consistent specialization of passive object behavior presented in [SS95, SS97], Lang [Lan97] discusses the consistent specialization of active object behavior. He lists rules that have to be obeyed when subtypes redefine, extend, or refine features inherited from their supertypes.

Logical Design. In a second step, we map a conceptual database schema developed with Active Object/Behavior Diagrams to a logical database schema of an existing active object-oriented database system. Doing this, we decompose the high-level constructs of Active Object/Behavior Diagrams into lower-level constructs of a logical model. A principal goal in the development of Active Object/Behavior Diagrams has been to enable a smooth transformation from a conceptual schema to a logical schema. A conceptual schema designed with Active Object/Behavior Diagrams is detailed enough, so that its mapping yields a practicable logical model and not only a skeleton. The presented mapping covers all three dimensions of a schema specified with Active Object/Behavior Diagrams: object structure, passive object behavior, and active object behavior. The active capabilities of the target system are not used only for representing business rules modeled with Active Object/Behavior Diagrams. Rules are also generated for countering violations of static and dynamic integrity constraints, for deriving data, and for enhancing the functionality of the target system. Thus, the resulting logical database schema contains a large set of rules. But these rules are generated automatically, their consistency is guaranteed by the schema translator, and the designer is not responsible for maintaining them.

Outline. The remainder of this work is organized in three groups of chapters. Chapters 2 and 3 give an overview of active database technology, in particular of active object-oriented database systems and active object-oriented database design. Chapters 4, 5, and 6 introduce Active Object/Behavior Diagrams. Chapters 7, 8, 9, and 10 study how a conceptual schema specified with Active Object/Behavior Diagrams can be mapped to a logical schema supported by an active object-oriented database system. The following list gives a more detailed description of the successive chapters' contents.

- Chapter 2 introduces into the field of active object-oriented database systems, gives a historical overview of the evolution from individual file systems to active object-oriented database systems, and presents the rule and event languages provided by existing active database systems, which are not sufficient for the conceptual design of active object-oriented databases but enforce the need for high-level modeling concepts.

- Chapter 3 discusses the database design process in general and presents the evolution from conventional database design to active object-oriented database design especially with respect to the role that business rules play within the various approaches followed in database design. The requirements on a rule and event language for modeling business rules at the conceptual level by means of the active behavior of objects are identified.

- Chapter 4 presents Object/Behavior Diagrams [KS91a] as a diagramming technique (for the conceptual design of object-oriented databases) that serves as the basis for the conceptual design of active behavior in the following chapters.

- Chapter 5 introduces situation diagrams [LOS96] as a high-level graphical notation to model events and intervals in a declarative way similar to OQL [CB97]. Situation diagrams are integrated homogenously into Object/Behavior Diagrams.

- Chapter 6 introduces activation diagrams [LOS97] as a high-level graphical notation to model ECA rules. Like situation diagrams, activation diagrams are integrated homogenously into Object/Behavior Diagrams.

- Chapter 7 presents an environment for mapping a conceptual model specified with Active Object/Behavior Diagrams to a logical model supported by an active object-oriented database system. It selects TriGS (cf. [KRSRV94, Hie96, Ret96, KRSR98]), an active extension to the commercial object-oriented database system GemStone (GemStone Systems, Inc.) as target system.

- Chapter 8 shows how the concepts of Active Object/Behavior Diagrams for modeling the structure of objects can be mapped to GemStone/TriGS.

- Chapter 9 shows how the concepts of Active Object/Behavior Diagrams for modeling the passive behavior of objects can be mapped to GemStone/TriGS and introduces an execution model that extends the conventional way a user works with GemStone/TriGS.
- Chapter 10 shows how the concepts of Active Object/Behavior Diagrams for modeling the active behavior of objects can be mapped to GemStone/TriGS.
- Chapter 11 summarizes the work presented in this thesis, shortly describes the prototype of an editor for Active/Object Behavior Diagrams, discusses open points, and gives a short outlook to further work.

Work Environment. This thesis reports research results of a project on active object-oriented database design, funded by the Austrian Science Foundation (FWF) under contract P10320-TEC. Whereas this thesis gives emphasis to Active Object/Behavior Diagrams and their mapping to a logical schema of an active object-oriented database system, an accompanying thesis, written by Peter Lang [Lan97], focuses on conceptual design. Lang presents a complete overview of specializing object structure, passive behavior, and active behavior in Active Object/Behavior Diagrams and establishes guidelines for the conceptual design. The core concepts of Active Object/Behavior Diagrams have been developed jointly by the authors of both theses and appear in each thesis in order to make them self-contained.

Chapter 2

Active Object-Oriented Database Systems

Contents

2.1 From Database Systems to Active Object-Oriented Database Systems 10

2.1.1 Database Systems 10

2.1.2 Object-Oriented Database Systems 11

2.1.3 Active Database Systems 11

2.1.4 Active Object-Oriented Database Systems 15

2.2 Concepts for Modeling Active Behavior 16

2.2.1 Rule 16

2.2.2 Event 19

2.2.3 Condition 24

2.2.4 Action 25

2.2.5 Execution Model 25

2.3 Related Work on Active Object-Oriented Database Systems 29

The goal of this chapter is to give an overview of the field of active object-oriented database systems. The chapter is based on a manuscript surveying active object-oriented database systems [ORS95], which has been compiled in preparation of the research project "Active Object-Oriented Database Design", within which this thesis has been carried out. During the project, the survey has been repeatedly updated by the author of this thesis. The material from the preliminary survey has been thouroughly revised and recent developments in the field of active object-oriented database systems have been incorporated.

Active object-oriented database systems combine the capabilities of object-oriented database systems and active database systems. They represent entities of the application domain by their structure, their passive behavior, and their active behavior. Section 2.1 opens this chapter with a historical overview. The way from individual files to active object-oriented database systems is sketched out. Section 2.2 continues with a survey of the concepts that are provided by active object-oriented database systems for modeling active behavior. Section 2.3 concludes this chapter with an overview of related work on active object-oriented database systems.

2.1 From Database Systems to Active Object-Oriented Database Systems

Information systems should be easy to maintain, easy to extend, and easy to comprehend. Considerable progress in providing the proper technology to achieve these goals has been made since the first implementations of information systems based on individual files. In this section, the way to active object-oriented database systems is shown. This way is characterized by representing more and more semantics of real-world entities in the database schema, independent of application programs.

2.1.1 Database Systems

Early information systems consisted of a bunch of application programs with individual files. These information systems were hard to maintain as common data was stored redundantly. Later, file systems have been used which allowed sharing of data between different application programs. But programs and their data still depended on each other in the sense that changes in the physical organization of files as well as extensions to their record structure caused modifications of all programs accessing these files.

Database management systems following the ANSI/SPARC architecture provide the appropriate technology for decoupling data and application programs. The conceptual schema of a database system describes the logical structure of the entire set of data in a particular application environment. The physical schema of the database system, which defines the actual storage structures, is invisible to application programs. Application programs access the data by database management system calls according to the logical structure of the data in subsets of the conceptual schema, the view schemata. Thus, the physical schema can be modified without affecting the conceptual schema (*physical data independence*) and the conceptual schema can be extended without affecting the view schemata (*logical data independence*).

Various semantic data models (cf. [HK87]) have been developed for capturing more meaning of data in the conceptual schema of a database system. Semantic data models allow

to represent explicitly structural semantics of data, which formerly was hidden in application programs. Most prominent examples for structural semantics of data are referential integrity, aggregation, and generalization (cf. [SS77]). Semantic data models capture the structural semantics of data, but their behavioral semantics still resides in application programs.

2.1.2 Object-Oriented Database Systems

Conventional database systems support sharing of data between various applications. Object-oriented database systems (cf. [ABD+89, SRL+90, BM91a]) extend the idea of sharing to operations on data. The schema of an object-oriented database describes not only the structure of data shared by several application programs but also the basic operations for manipulating these data.

The schema of an object-oriented database consists of a set of object classes. An object class contains the description of the data structure used to model real-world entities of that class together with the description of the associated operations, also called methods. The data of one real-world entity is represented by an instance of an object class. The passive behavior of an object is reflected by the methods of its class. A particular method is invoked on an object by sending the object a message identifying the method selector.

Object-oriented data models incorporate the concept of generalization already present in semantic data models. But they go beyond the latter by considering methods next to attributes of classes. A subclass inherits the attributes and methods defined at or inherited by its superclass. Attributes and methods of a superclass may be overridden, i.e., redefined at a subclass. If a message is sent to an object, a method to handle the message is first searched for at the object's class. If no such method is found, the search is continued up the inheritance hierarchy.

2.1.3 Active Database Systems

Conventional database systems are passive. They execute only operations that are explicitly specified in user requests or in application programs. But many applications demand reactions upon specific changes to the database or the database environment without further user action. Such applications require continued automatic monitoring of the database state and database environment. Therefore, they cannot be well served by passive database systems.

A representative problem that cannot be handled satisfactorily by passive database systems is inventory control. It is a common reorder policy to reorder an item if the quantity on hand falls below a specified threshold. In absence of direct support for monitoring the database state, two typical ad-hoc solutions are (cf. [Cha89]):

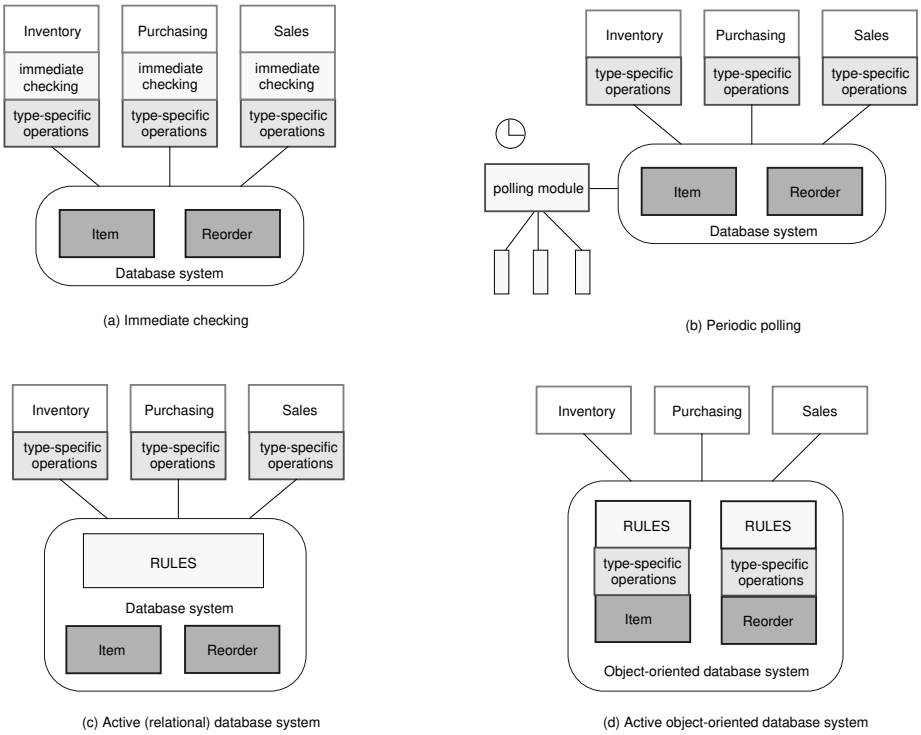


Figure 2.1: From immediate checking to active object-oriented database systems

- *Immediate checking*: Application programs compare the number of items in stock against the reorder threshold after each withdrawal and issue a reorder request if necessary (cf. Figure 2.1 (a)).

- *Periodic polling*: A special program, which runs periodically, issues reorder requests for all items whose quantity in stock is below the threshold (cf. Figure 2.1 (b)).

Immediate checking as well as periodic polling are unsatisfactory solutions. Using immediate checking, the reorder policy is buried into the application code, which makes both the reorder policy and the application code hard to maintain, hard to extend, and hard to comprehend. The success of periodic polling depends heavily on the appropriate polling period. Polling too seldom may miss transitions of items in stock beyond their thresholds. Polling too often burdens the system with unnecessary queries. Furthermore, to determine the right polling period is difficult if the frequency of withdrawals changes over time and differs for various items.

Active database systems monitor the database state and execute predefined actions if a given condition holds after a specified event has occurred. Events to be detected, conditions to be checked, and actions to be initiated are defined explicitly in the database schema using some form of *event-condition-action rules (ECA rules)* [DBB+88, Day88, DBM88]. The event of an ECA rule may be the start or the completion of a database operation, a

temporal event, or an arbitrary event defined by the user; the condition is a predicate or a query over event parameters and the database state; the action can be any user-defined executable method or predefined generic database operation.

ECA rules in active database systems are defined independently from application programs and are part of the database schema (cf. Figure 2.1(c)). Thus, the responsibility for event detection has migrated from application programs to the database system. In the above example of inventory control, an ECA rule would state the following: if an item is withdrawn (the event), and less pieces of an item than a given threshold are in stock (the condition), a certain number of items must be reordered (action). The active database system recognizes each withdrawal of an item and examines whether the current number of items is below the threshold. If this is the case, the database system initiates a reorder for that item automatically, and may inform the stock officer about the reorder. There is no need to initiate a polling-program or to encode checking-procedures in application programs.

Immediate checking, periodic polling and ECA rules of active database systems are alternative solutions to react to specific situations. Figure 2.2 evaluates each of these solutions against the following criteria: (1) specification technique for active capabilities, (2) responsibility for action initiation, (3) complexity, modularity, and locality of program code, (4) effectiveness and flexibility of event and condition monitoring, and (5) expected system load.

1. *Specification technique:* In an active database system, events, conditions, and actions are specified explicitly in the database schema using rules, whereas using immediate checking and periodic polling, they are represented implicitly within program code.

2. *Responsibility for action initiation:* After a specific event has been detected, an active database system automatically checks proper conditions and initiates proper actions. Using immediate checking (or periodic polling) each program (or the polling program) is responsible for taking care of checking proper conditions and executing proper actions on specific events. As a consequence, the knowledge which events require which conditions to be evaluated and actions to be executed is hidden in program code.

3. *Complexity, modularity, and locality of program code:* One advantage of active database systems is the reduction of the overall complexity of program code by extracting out a cohere part of complexity into the independent rule mechanism of the database system. Aside from achieving better maintainable and extendible systems, the modularity of the program code increases significantly. Therefore, programs are easier to comprehend, too. Knowledge represented in program code is hard to maintain. E.g., if the reorder policy is changed, programs must be investigated, rewritten, and recompiled. Using an active database system the user simply adds, modifies, or deletes a rule in the database schema to reflect the changed reorder policy.

	Passive database systems		Active database systems
	Immediate checking	Periodic polling	
Specification technique	Implicit in replicated program code; dependencies between different rules have to be considered explicitly.	Implicit in non replicated program code; dependencies between different rules have to be considered explicitly.	Explicit in the database schema; dependencies between different rules are considered by the database system.
Responsibility for action initiation	The knowledge which actions must be executed in response to occurred events is hidden in the program code. Each program is responsible for initiating the proper actions.	The knowledge which actions have to be initiated is placed at a central point. Actions must be initiated explicitly by the polling program.	The database system initiates the proper actions without further user or application program request.
Complexity of program code	The entire complexity of the application environment is present in program code.	A cohere part of complexity is extracted out into a separated program, but this program is not independent from programs of the application environment.	A cohere part of complexity is extracted out into an independent rule mechanism of the database system.
Modularity of program code	As all events and conditions beeing monitored and all corresponding actions may appear repeatedly in program code, modularity is violated.	Modularity is preserved.	Modularity is preserved.
Locality of program code	Modifications and extensions are an expensive task: program code has to be changed in several places.	Necessary changes are hard to locate in a monolitic polling program; if several polling programs are used, appropriate ones may be hard to detect.	Only the rule specifications must be changed in the database schema.
Effectiveness and flexibility of event and condition monitoring	Having programs continuously check if some event occurred can be ineffective or rather cumbersome if the checks are not optimized.	If the polling period can be determined appropriately, periodic polling represents an effective approach for event and condition monitoring.	The database management system monitors all known events in a timely and efficient manner. It can use various optimization techniques to avoid unnecessary checks.
System load	The same queries for condition checking may be issued from different programs redundantly.	The system load depends on the right polling period.	Event detection and condition evaluation are executed automatically by the database system.

Figure 2.2: Comparison of immediate checking, polling, and active database systems

4. *Effectiveness and flexibility:* Active database systems convince also by providing a more effective and flexible approach for monitoring the database state. Once defined, every rule is considered for event and condition monitoring by the database system all over the time, unless the user makes a contrary decision. Using immediate checking, each program implements its own events and conditions being monitored and is responsible for choosing the appropriate points in time when to check them. Using periodic polling, determining the proper polling period is difficult, as already discussed earlier.

5. *Expected system load:* In active database systems, event detection, condition evaluation and action execution is handled by the database system and can be optimized by the system. If immediate checking is used, identical queries for condition checking, issued from different programs, burden the systemload unnecessarily. If periodic polling is used, the database system can be overloaded by unnecessary queries because of a too short polling period.

Research on active capabilities in databases is not new at all. Active capabilities can be found already 1973 in the CODASYL project [Com73], which proposed the ON-condition for database procedures for the first time. System R used triggers for ensuring integrity constraints (assertions) (cf. [EC75, Esw76]). Later, this mechanism was generalized to integrity rules and "triggered" procedures (cf. [Dat83]). Triggers were used to maintain materialized views, database snapshots and derived attributes (cf. [BC79, KP81, Rou82, Mor83, BLT86, HK86, LHM86, Han87]. The term "active database" was first used in 1983 by Morgenstern [Mor83] to describe a system that propagates changes to the database automatically to materialized views and derived attributes. In [Sto82] and [Sto85] situation-action rules are used for supporting inference cycles based on forward and backward chaining.

Now, triggers are provided also by a series of commercial database systems, such as Oracle, Informix, OpenIngres, Sybase, Rdb, Allbase/SQL, or InterBase. An overview of trigger facilities in commercial systems is given in [CW96b]. Triggers are also part of the proposed SQL3 standard. Prominent academic prototypes of relational active database systems are ARIEL [Han87, Han89, Han96], Postgres [SR86, Sto87, SHP89, SJGP90, PS96], Starburst [WF90, WCL91, Wid96], or A-RDL [SK96].

2.1.4 Active Object-Oriented Database Systems

Early research in active database systems has been devoted to extend conventional database systems, such as relational ones, with active capabilities. Basic operations on relations, such as insert, delete, and update, are recognized by the database management system and trigger predefined actions. These systems miss the representation of the passive behavior of entities and lack encapsulation of data and operations. Data structures and associated

operations are defined separately in the database schema and in application programs, respectively.

Active object-oriented databases combine the research directions of active database systems and object-oriented database systems and promise a higher degree of expressive power than today's database systems offer. They complement the description of the structure and the passive behavior of an object by a description of its active behavior. Whereas the passive behavior describes how an object reacts upon specific requests, its active behavior defines how it acts autonomously upon the occurrences of predefined events. The semantics how real-world entities act is no longer hidden in application programs but expressed explicitly in the database schema during object definition (cf. Figure 2.1(d)). The principle of encapsulation of data and operations is extended to rules, which are commonly used to define an object's active behavior. As user-defined higher-level operations on objects are part of the database schema, the active mechanism becomes more general available than with relational systems. Not only basic database operations may be monitored, but also higher-level operations.

2.2 Concepts for Modeling Active Behavior

Active object-oriented database systems provide concepts for modeling the structure, the passive behavior, and the active behavior of real-world entities. In this section, the focus is on concepts for modeling active behavior. Therefore, different features of modeling active behavior are examined. The considered features concern a rule as a whole, the components of a rule, or the execution of rules. For each feature, concepts are described that are provided by active object-oriented database systems to support that feature. No architectural and implementational issues are discussed in this section. (For a comparison of architectures and monitoring techniques for active databases, see [Cha95].)

2.2.1 Rule

Features concerning a rule as a whole are the representation of the rule, the extensional scope of the rule, the rule's degree, the reusability of the rule's components, the way the rule can be manipulated, and the availability of provisions for assigning the rule to specific groups.

Rule Representation. Active object-oriented database systems extend object-oriented database systems with *active rules*. Rules, as they can be found in deductive database systems, are referred to as *passive rules*. Contrary to active rules, passive rules are triggered only on explicit program or user request. The most prominent approach to represent active

rules is to engage ECA rules. Besides ECA rules, alternative approaches can be found in the diverse literature:

- *CA rules:* A CA rule does not have an event part. The meaning is that the event triggering the rule is implicitly derived from the rule's condition part. The rule is triggered whenever data is modified so that changed data might satisfy the rule's condition. Widom and Ceri ([WC96b]) compare CA and ECA rules: If the event part is omitted, it is not possible to perform different actions depending on which event occurred. On the other hand, CA rules are considered more "declarative" and easier to program since the task of determining when to evaluate the condition is shifted from the user to the system.

- *ECEA rules:* In an ECEA rule, condition evaluation and action execution may be triggered by two different events. The event that triggers a condition evaluation is referred to as *condition event*, the event that triggers an action execution is referred to as *action event*. In TriGS ([KRSR98]), ECEA rules have been introduced to obtain more expressiveness and uniformity in specifying when rule conditions have to be evaluated and rule actions have to be executed (cf. Section 2.2.5).

The *Active Database Management System Manifesto* (cf. [Con96]) demands that an active database system supports the explicit definition of events. According to the manifesto, there is no general support for reactive behavior if explicitly definable events are not provided. The notion of "event" is one major distinction of active database systems from other rule-based database systems, such as knowledge-based systems.

Extensional Scope. The extensional scope of a rule constrains the set of objects for which the rule is defined. The most prominent scope, found in most active object-oriented database systems, is the extension of a class (*class-specific rule*). A class-specific rule is defined for all direct and indirect instances of a class. Generally, a class-specific rule may be overridden at a subclass. Variants that may be provided in addition to class-specific rules are:

- *Instance-specific rules:* Some recent object-oriented database systems allow to define specific attributes and methods for an object without the need to define a new class for that object [Deu91, KNS89]. Similarly, some active object-oriented database systems support the definition of rules for single objects.

- *Method-specific rules:* Some systems allow to define rules for all objects for which a method with a specific name is defined. Such rules are defined across class boundaries.

Rule Degree. The number of objects referenced by a rule's event expression is referred to as the rule's *degree*. A rule that references a single object in its event expression is referred to as an *intra-object rule*, a rule that references an arbitrary number of objects in its event expression is referred to as *inter-object rule*.

- *Intra-object rule:* A rule that references a single object can be associated clearly with a single class and, therefore, often is defined as part of the class definition. In this case, an intra-object rule is said to be "local".
- *Inter-object rule:* If the objects referenced by the event expression of an inter-object rule belong to different classes, the inter-object rule cannot be clearly associated with a single class. In this case, the inter-object rule is often defined outside class definitions, the rule is said to be "global" then.

Inter-object rules have been incorporated into diverse active object-oriented database systems with a view to enable symmetric designs: Usually, object-oriented languages provide no special constructs for representing inter-object structure and inter-object behavior. Recent papers argue that this may lead to assymmetric designs, which are hard to maintain and hard to extend. Symmetric relationship constructs (cf. [DG90, LSR87, Rum87]), as they are found in the Entity-Relationship Model, have been proposed to avoid mimicing them by directed references stored in attributes. Cooperation contracts (cf. [SK91]) have been proposed as the counterpart to symmetric relationships for modeling behavior to which several objects contribute. Similarly, inter-object rules have been proposed to model active behavior concerning several objects.

Modularity. Modularity of a rule language is a measure that indicates to which extent rule components can be reused. Rule components that a rule designer might be interested to reuse are event specifications, conditions, and actions. Prerequisites for reusing rule components are that rule components have unique names by means of which they can be identified and that rule components have well-defined interfaces. Typical degrees of modularity are (a) that reuse of components is not supported at all, (b) that event specifications can be reused in several rules and other event specifications, or (c) that event expressions, conditions, and actions can be reused.

Rule Manipulation. The way rules are incorporated into an active object-oriented database system determines the degree of uniformity in manipulating conventional objects and rules. If rules are treated as "first-class citizens", i.e., like conventional objects, they can be created, modified, and deleted dynamically.

Other generic operations specific to rules are rule firing and rule activation/deactivation.

- *Rule firing:* A rule can be fired explicitly without the specified event having occurred. If a rule is fired, its condition is evaluated and, if it holds, the action is executed.

- *Rule activation/deactivation:* A rule can either be *activated* or *deactivated*. If a rule is activated, the specified event is observed by the database system, and, if the event occurs, the rule is fired. If a rule is deactivated, any occurrence of the specified event is ignored, and no rule firing takes place. Therefore, deactivated rules have no effect on the current active behavior of objects. Activating a class-specific rule for one object only allows to simulate an instance-specific rule [GJ91]. Contrary to an instance-specific rule, the rule is potentially applicable (if activated) to all instances of the class.

Grouping of Rules into Rule Categories. The field of application of active rules is wide. In [CW96a], Ceri and Widom classify the kinds of database applications that make use of active rules into internal, extended, and external:

- *Internal:* Internal applications use active rules to implement conventional system-supported facilities, where active rules are used as a substitute for a special-purpose mechanism built into the database system (e.g., management of integrity constraints, computation of views and derived data, or authorization support).
- *Extended:* Extended applications use active rules as an implementation mechanism to support novel or non-standard database tasks (e.g., workflow management, version management, advanced transaction management, or replication management).
- *External:* External applications use active rules to obtain automatic reactive behavior in the context of a specific domain (e.g., manufacturing automation, inventory control, trading, finance, or air traffic control).

Depending on the application to which a rule belongs, different groups of users are interested in the rule. To keep the set of active rules defined in an active database manageable, an active object-oriented database system may allow to group rules into different sets. For example, these sets might be used for collective assignment of priorities, for collective authorization, or for collective activation/deactivation.

2.2.2 Event

Instead of talking commonly of events, it has to be distinguished between event specifications and event occurrences. An *event specification* defines which types of happenings (or history of happenings) cause the raising of an event occurrence. An event specification refers to specific points on the time line. References to activities which take some time, such as the execution of a method, must be complemented by event modifiers, to indicate time points and not intervals. An *event occurrence* is the occurrence of some happening (or history of happenings) at one specific point on the time line as defined by an event

specification. In the sequel, the term *event* will be used for both, an event specification and an event occurrence if its meaning is clear from the context.

Features concerning the event part of a rule are the kinds of primitive events that are supported, the parameters that are bound by primitive events, the event operators that are provided for building composite events, the event and parameter contexts that are supported to resolve ambiguities in building composite events, and the possibility to specify guards.

Basic Events. Basic events represent elementary happenings that are independent of other events. Generally, they are classified into database events, time events, and abstract events.

- *Database events:* Database events are based on the execution of database operations and can be detected by the database system. As the execution of a database command usually takes some amount of time, *event modifiers*, such as *before* and *after*, are used together with the database operation to specify an event that refers to specific points on the time line. In an object-oriented setting, database events can be further classified into value events, transaction events, and method events. Events generated by generic database operations, such as reading or writing an object's attribute using predefined access primitives, are called *value events*. Events generated by the execution of a method are called *method events*. Events generated by a transaction statement, such as *begin* transaction, *abort* transaction, and *commit* transaction are called *transaction events*.

- *Time events:* Time events represent the occurrence of *specific points* on the time line and are classified into *absolute time events*, *relative time events*, and *periodic time events*. The specification of an absolute time event refers to one absolute point in time. The specification of a relative time event refers to one or several points in time that are relative to the occurrence time(s) of a reference event. The specification of a periodic time event refers to several points in time that are located on the time line in periodic intervals to the occurrence time(s) of a starting event.

- *Abstract events:* Abstract events are defined and raised by users, by programs, or by external components.

Parameters of Events. In many cases, more information than the pure happening of an event is needed to specify the corresponding condition or action. For instance, the condition or action of a rule may need to know which database state or which user caused the event occurrence, or the condition or action may need to reference some parameters supplied by the method whose invocation caused the event occurrence. To relieve conditions and actions of rules from calculating information available already from event detection, events

are parameterized. When an event occurs, the formal parameters are bound, and the actual parameters are passed to condition evaluation and action execution.

The set of parameters bound by an event depend on the event's kind. The only parameter that is bound by all kinds of events is the point in time at which an event occurred. Typical parameters that are bound by other kinds of events are:

- *Method event:* The occurrence of a method event binds (1) the transaction triggering the event, (2) the session within which the triggering transaction runs, (3) the user who initiated the triggering session, (4) the object on which the method is performed, and (5) the parameters of the method. If the method event is raised after method execution (i.e., if a modifier such as *after* is applied), the return value of the method is bound as additional parameter.

- *Value event:* The occurrence of a value event binds (1) the transaction triggering the event, (2) the session within which the triggering transaction runs, (3) the user who initiated the triggering session, (4) the object on which the operation is performed, and (5) for read operations of an attribute, the value of the attribute, and for write operations of an attrbiute, the old and the new value of the attribute.

- *Transaction event:* The occurrence of a transaction event binds (1) the transaction triggering the event, (2) the session within which the triggering transaction runs, and (3) the user who initiated the triggering session.

- *Time event:* The occurrence of a time event does not bind parameters in addition to the point in time at which the event occurs.

- *Abstract event:* The occurrence of an abstract event binds (1) the transaction triggering the event, (2) the session within which the triggering transaction runs, (3) the user who initiated the triggering session, and (4) the parameters defined by the user with the event specification.

Composite Events. Composite events correspond to particular histories of events and are constructed from basic and other composite events using various *event operators*.

Composite events are usually not detected by keeping the complete log of basic event occurrences and by inspecting this log; rather, special data structures are used to facilitate detection of composite events and to avoid storage of events not needed any more. Typically, Petri nets [GD94], finite state machines [LGA96], or event graphs [CKAK94, KRSR98] are used to detect composite events.

In the diverse literature, operators of the same name convey different semantics. Therefore, typical event operators are not only enumerated below, but also their semantics are ex-

plained. Event operators, as they can be found in the diverse prototypes, can be classified into *boolean operators*, *history operators*, and *interval-based operators*[1]:

- *Boolean operators:* Disjunction, conjunction, and negation are used with the following semantics in event expressions:
  - $\vee$: A *disjunction* E of two events F and G, denoted $E = (F \vee G)$, occurs when either F or G occurs. Thus, the disjunction of two events corresponds to two rules with equal conditions and actions.
  - $\wedge$: A *conjunction* E of two events F and G, denoted $E = (F \wedge G)$, occurs when F and G occur at the same point on the time line.
  - $\neg$: A *negation* E of an event F, denoted $E = \neg F$, occurs every time an event other than F occurs.
- *History operators:* History operators select particular patterns of event occurrences in the event history.
  - A *relative sequence* E of two events F and G, denoted $E = (F,_{rel}\, G)$, occurs when first F occurred and thereafter G occurs. If F and G represent composite events, all partial events F_i of F must occur before the first partial event G_1 of G.
  - A *prior sequence* E of two events F and G, denoted $E = (F,_{prior}\, G)$, occurs when the last partial event F_n of F occurred before the last partial event G_n of G. Note: The relative sequence and the prior sequence of two basic events are the same.
  - The *historical conjunction* E of two events F and G, denoted $E = (F \triangle G)$, occurs when one of both events occurs and the other occurs, too, or has already occurred in the past; i.e., $F \triangle G = (F \wedge G) \vee (F,_{prior}\, G) \vee (G,_{prior}\, F)$.
  - A *choosing n sequence* E of an event F, denoted $E = (choose\ n\ F)$, occurs at every n^{th} occurrence of F.
- *Interval-based operators:* Interval-based operators select one or several event occurrences from one or several occurrences of a time interval. The operators differ whether the composite event is raised during or at the end of the time interval. Interval-based operators are *notOccurredIn* (interval-based negation), *occursDuring* (monitoring interval), * (interval-based sequence), and $[n_1,n_2]$ (interval-based counting sequence).
  - The event $E = F\ notOccurredIn\ [G..H]$ occurs when H occurs and F did not occur during the interval defined by G and H.

[1] In the definition of the event operators below, event specifications identified by capital letters may refer to basic as well as to composite events.

- The event $E = F$ *occursDuring* $[G..H]$ or shortly $E = F[G..H]$ occurs every time F occurs during the interval defined by G and H. The interval $[G..H]$ is often referred to as *monitoring interval*, as occurrences of F are "monitored" only during that interval and not always.
- The event $E = (*F)[G..H]$ occurs when H occurs. During the interval $[G..H]$ event occurrences of F are not raised but registered by E. After E occurred, the condition and/or action of a rule may refer to any individual occurrence of F.
- The event $E = ([n_1, n_2]F)[G..H]$, occurs when H occurs and F occurred for at least n_1 times but at most n_2 times during the interval defined by G and H.

Event and Parameter Contexts. Event and parameter contexts are needed to resolve ambiguities that result from several occurrences of the events used in the specification of a composite event in the event history. The (basic or composite) events that contribute to the occurrence of a composite event are referred to as *contributing events*. That contributing event which terminates the building of a composite event is referred to as the *terminating event*. An event and parameter context specifies how contributing events and their parameters are combined to build a composite event. No context needs to be specified for composite events for which each partial event occurs at most once.

Event and parameter contexts have been introduced in [CKAK94], where five different contexts are distinguished, the unrestricted context, the recent context, the chronicle context, the continuous context, and the cumulative context:

- *Unrestricted context:* At occurrence of a terminating event, a composite event is raised for every possible combination of occurrences of contributing events.
- *Recent context:* At occurrence of a terminating event, a composite event is raised with the most recent occurrences of the contributing events.
- *Chronicle context:* At occurrence of a terminating event, a composite event is raised with the oldest occurrences of the contributing events that have not yet contributed to an older occurrence of the composite event.
- *Continuous context:* At occurrence of a terminating event, a composite event is raised for every possible combination of occurrences of contributing events that have not yet contributed to an older occurrence of the composite event.
- *Cumulative context:* At occurrence of a terminating event, a composite event is raised that collects all the occurrences of contributing events that have not yet contributed to an older occurrence of the composite event.

In the cronicle context, in the continuous context, and in the cumulative context, a contributing event that has contributed to an occurrence of a composite event once is excluded from contributing to further occurrences of the same composite event. In the recent context, an event is excluded from contributing to further occurrences of a composite event as soon as there is a more recent occurrence of the contributing event. Events that are excluded from contributing to further occurrences of a composite event are said to be *consumed*.

The set of parameters bound by a composite event corresponds to the collection of the parameters of all constituent events.

Guarded Events. To restrict the occurrence of composite events, *guards* can be associated with event specifications. A guarded event occurs if it is detected and the associated guard is satisfied. A guard can be any boolean predicate over the parameters of the event to which it is associated. Events with guards are often referred to as *situations* and can be used in event expressions the same way as events without guards.

If both a guard and a context are present in the specification of a composite event, it has to be considered whether the guard influences the way constituent events are combined to an occurrence of the composite event or whether the guard is applied to a composite event only after its detection has been completed. For example, assume a relative sequence E of two basic events F and G, where event F has a parameter p, where event G has a parameter q, where the recent context is applied for composing occurrences of E, and where a guard requires parameter p and parameter q to be identical. First, consider the case that the guard influences the way occurrences of F and G are combined to occurrences of E: When G occurs, the occurrence is combined with the most recent occurrence of F that satisfies the guard. In this case, there may be a more recent occurrence of F that does not satisfy the guard. Second, consider the case that the guard is applied only after detection: When G occurs, the occurrence is combined with the most recent occurrence of F and event E is detected. Then, the guard is applied to the detected event. If the guard holds, event E occurs.

2.2.3 Condition

After an event has been signaled to the database system, the rules to be fired are determined by the database system. A rule firing causes the rule's condition to be evaluated and, if the condition is satisified, the rule's action to be executed. Conditions are parameterized. They receive the parameters bound during event occurrence and may define additional parameters. All parameters are then passed to action execution.

Features concerning the condition part of a rule are the general representation of a condition and the scope of a condition.

Result vs. Boolean Condition. A condition represents either a *predicate* or a *query*. A condition representing a predicate returns a boolean value (*boolean condition*), a condition representing a query returns a set of objects (*result condition*). Simple conditions do not make additional parameter bindings during condition evaluation. Result conditions may bind the set of objects that satisfy the condition query.

Scope of a Condition. The scope of a condition delimits the set of objects that can be referenced from within a condition. The scope may embrace the parameters bound during event occurrence, a single database object (generally the object for which the event triggering the condition evaluation occurred), the extension of a single class, or the entire database.

2.2.4 Action

For each rule fired by an event occurrence its condition is evaluated. If a simple condition evaluates to *true* or the returned set of a result condition is not empty, the action of the rule is executed. To reap the benefits of the parameter bindings made during event occurrence and condition evaluation, actions can access the bound parameters.

In the diverse systems, an action may be a user-defined method, a predefined database statement (e.g., a transactional primitive), the raising of an abstract event, or an arbitrary combination of these. Some systems allow the specification of an "else" action, which is executed if the rule's condition is not satisfied (cf. [DBM88]). Other systems allow to specify that the rule action has to be executed instead of the database operation that triggered the rule execution (cf. [KRSR98]).

2.2.5 Execution Model

The execution model of an active database system must satisfy different kinds of requests of various applications.

For applications that have to meet strong time requirements, it is very important to evaluate the condition and to execute the action immediately after the event triggering the rule has occurred. In this case, the execution of the triggering transaction is suspended until the execution of the rule terminates.

Other applications do not expect immediate reactions upon the occurrence of an event, but rather some deferred or detached reaction. E.g., integrity constraints may be checked only at the end of a transaction and not immediately after a data item involved in the constraint is changed, because consistency is maintained only at the transaction level and may be violated during transaction execution.

Response time and parallelism of the system can be increased if condition evaluation and/or action execution are processed in their own transactions, probably totally independent from the triggering transaction. E.g., for access control, it may be necessary to log accesses to special data even if the triggering transaction is aborted. This is possible if the rule execution takes place in an own independent transaction.

A rule execution can cause subsequent rule firings (*nested rule execution*). Thus, triggered transactions can become triggering transactions themselves, and the procedure for rule firing continues recursively. A subsequent rule firing may even cause the same rule to be fired again (*recursive rule execution*).

Features that are incorporated into the execution models of active object-oriented database systems to meet these requirements are coupling modes, enhanced transaction models, mechanisms for resolving conflicts among simultaneously triggered rules, mechanisms for imposing time restrictions on rule execution, and mechanisms for imposing restrictions on the interaction of rules.

Coupling Modes. Most active database systems introduce coupling modes to gain more flexibility in rule execution. Coupling modes describe the temporal and transactional relationships between event occurrence and condition evaluation and between event occurrence and action execution. Typically, for each rule two coupling modes, one between event and condition and one between event and action, are indicated.[2]

Coupling modes as they are supported in the diverse systems allow that condition evaluation and action execution may be done immediately after the event occurrence, may be delayed until the end of the triggering transaction, may be performed in an independent transaction, or may be performed in a separate transaction that is in some form causally dependent from a triggering transaction. Naming of coupling modes with equal semantics differs from system to system. Names and semantics as they are introduced here are borrowed from [BZBW95].

- *Immediate mode:* The condition/action is evaluated/executed immediately after the event occurrence within the triggering transaction.
- *Deferred mode:* The condition/action is evaluated/executed at the end of the triggering transaction as last operation before the commit. If the transaction is aborted, the condition evaluation/action execution is not initiated.
- *Detached mode:* The condition/action is evaluated/executed in an independent transaction.

[2] Alternatively, the coupling modes could be defined between event and condition and between condition and action, but the two alternatives are semantically equal.

- *Parallel causally dependent mode:* The condition/action is evaluated/executed in a separate transaction, which may begin in parallel but may not commit unless the triggering transaction commits.

- *Sequential causally dependent mode:* The condition/action is evaluated/executed in a separate transaction which may begin only after the triggering transaction has committed.

- *Exclusive causally dependent mode:* The condition/action is evaluated/executed in a separate transaction which may be executed only if the triggering transaction aborts.

Each of the coupling modes above can be defined between event occurrence and condition evaluation and between event occurrence and action execution. Since action execution has to take place after condition evaluation, only a subset of all possible pairs of coupling modes is meaningful. For example, it is obvious that an action cannot be executed immediately after event occurrence if the evaluation of the condition has been deferred until the end of the triggering transaction.

Similarly, not every combination of triggering event and coupling mode is allowed. Some combinations are impossible due to the nature of the triggering event. For example, time events are not related to triggering transactions and, therefore, a condition evaluation or action execution can be coupled to a time event exclusively in the *detached* mode. Some systems exclude other combinations due to performance issues. In the REACH system [BZBW95], for example, the *immediate* mode is excluded for composite events, even for those composite events where all contributing events originate in the same transaction.

Enhanced Transaction Model. In traditional database systems, transactions are initiated from users and are processed interleaved or in parallel guaranteeing serialization. Active database systems need the ability to initiate new transactions during transaction execution in order to support normal rule firing as well as nested and recursive rule firing. The new transactions can be a part of the triggering transaction or they represent own independent or dependent transactions depending on the coupling mode used in rule definition.

To meet these requirements, active database systems often provide an enhanced transaction model supporting *nested transactions* [Mos81]. In general, a transaction which is executed within another transaction is called a nested (sub)transaction. A nested transaction model allows to build *nested transaction forests*. In a nested transaction forest, each top transaction represents an independent tree which is subject to serialization.

Priorities. As several rules may share the same event, a single event occurrence may fire an arbitrary number of rules simultaneously. A mechanism for determining the execution

order of all fired rules is needed to avoid unpredictable execution sequences of rules (*conflict resolution*). Two approaches to deal with this situation are:

- No conflict situation arises since all simultaneously fired rules are executed in parallel transactions. This approach is restricted by the fact that these transactions must represent commutable actions, i.e., their execution does not result in different database states depending on their serialization order.

- The conflict resolution is obtained by means of priorities. The user defines priorities between pairs of rules. Two major approaches for defining priorities are:

  1. *Total ordering*: To each rule a unique system priority is associated at rule definition time. Later, users can override this priority in a way such that the ordering of the whole set of rules remains total. Therefore, the database system is able to determine the correct rule execution order at any point in time.
  2. *Partial ordering*: At rule definition time no priority is associated to rules, i.e., all rules have the same priority. Later, the user can specify priorities between pairs of rules to determine their execution order. Rules for which no priority is defined are executed by the database system in an indeterministic order.

  For the specification of priorities *absolute numbers* or a *<-relation* may be used (cf. [ACL91, GGD91, Wid92]).

Time Restrictions and Alternative Actions. To support time-sensitive environments, time constraints may be specified at rule definition time. The execution of a rule must be finished within this time restriction after an event occurred. An alternative action may be specified, which is executed in the case that the time constraint cannot be kept or the action execution fails for some reason.

Restrictions in the Interaction of Rules. Some systems allow to restrict the interaction of rules in order to avoid undesirable side effects of rules and to guarantee that the execution of rules remains understandable. In a stringent setting, rules may be restricted to not "cascade", which means that the execution of one rule must not trigger other rules. In a more flexible setting, rules may be restricted to not "loop", meaning that a rule must not directly or indirectly trigger itself during rule processing. To preventively avoid that rule execution results in an infinite loop, rules may be restricted to not loop with the same bindings. This means that a rule must not be triggered twice with exactly the same bindings during rule processing. Some restrictions in the interaction of rules can be enforced by syntactic examination of rules (i.e., at compile time) others can be enforced only while rules are being processed (i.e., at execution time).

2.3 Related Work on Active Object-Oriented Database Systems

In the area of active object-oriented database systems, research has been focused on designing and implementing prototype systems. Representative systems of active object-oriented databases are:

- *HiPAC:* HiPAC (High Performance ACtive database system) has been developed in cooperation of CCA (Computer Corporation of America) and the University of Wisconsin. Later, collaborators of Xerox and DEC were involved in the project as well. HiPAC provides a specification and several experimental implementations for an expressive rule language with a flexible execution semantics. The data model of HiPAC is based on an interpretation of the PROBE data model [MD86, Man87]. Details on HiPAC can be found in [DBB+88, Day88, DBM88, HLM88, Cha89, MD89, DD91, DBC96].

- *Sentinel:* Sentinel has been developed at the University of Florida within a follow-on project to HiPAC. A focus of the Sentinel project has been the development of an expressive event language (called *Snoop*) that incorporates sophisticated event operators and event contexts. The Sentinel prototype has been implemented on top of the Open OODB Toolkit from Texas Instruments [WBT92]. Details on Sentinel can be found in [CHS92, AMC93, CAM93, CKAK93, CKAK94, CM94, CKTB95, Cha97].

- *REACH:* REACH has been developed at the University of Darmstadt within another follow-on project to HiPAC. Work in the REACH project has been focused on efficiency issues in event detection, event composition, and rule firing. Contrary to many other prototypes, REACH proposes an architecture where event detection and rule support are tightly integrated with the core database functionality. Like Sentinel, the REACH prototype has been implemented on top of the Open OODB Toolkit from Texas Instruments [WBT92]. Details on REACH can be found in [BBKZ92, BBKZ93, BZBW95].

- *Ode:* Ode (Object Database and Environment) is a project of AT & T Bell Laboratories. Ode extends the O++ object-oriented database programming language with facilities for expressing rules in the form of *constraints* and *triggers*. Constraints use a language and execution semantics well-suited for enforcing integrity constraints on objects, whereas triggers use a language and execution semantics well-suited for monitoring complex events and invoking procedures when the events occur. Details on Ode can be found in [AG89, GJ91, GJS92b, GJS92c, GMS92, GJMS93, GJ96, LGA96].

- *SAMOS:* SAMOS (Swiss Active Mechanism-Based Object-Oriented Database System) is a research prototype developed at the University of Zürich. Central features of SAMOS are a rich event language and an efficient Petri-net based mechanism

for event detection. The prototype implementation is based upon the commercial object-oriented database system ObjectStore. The active components represent attached modules to the ObjectStore database system. Details on SAMOS can be found in [GGD91, GD92, DG93, GD93a, GD93b, GD94].

- *Chimera:* Chimera is the conceptual language of the IDEA Methodology [CF97]. Chimera incorporates passive and active rules into a language for conceptual object-oriented modeling. The target database management system used in the implementation of Chimera is ALGRES [CCRL+90]. Details on Chimera can be found in [CBFP95, CGMR95, CFPT96, MPC96].

- *TriGS:* TriGS (Trigger System for GemStone) is an active extension to the commercial object-oriented database system GemStone. It is developed at University of Linz. GemStone/TriGS plays the role of a target system when local design of active object-oriented database systems is discussed in this thesis. Therefore, GemStone/TriGS will be presented in Section 7.1.2. Details on TriGS can be found in [KRSRV94, Hie96, Ret96, KRSR98].

- *ACOOD2:* ACOOD2 (ACtive Object-Oriented Dbms 2) is a prototype developed at the University of Skövde. ACOOD2 adds a facility for active rule processing to the commercial object-oriented database ONTOS. Details on ACOOD2 can be found in [BL92, Ber94, BL95].

- *ADAM/EXACT:* ADAM is an object-oriented database system that has been developed at the University of Aberdeen. ADAM integrates active rules as objects in the data model seamlessly. EXACT, a follow-on project to ADAM, focuses on flexibility of rule specification and execution. Details on ADAM/EXACT can be found in [DPG91, DP91, PDB93, DJ94, DJPaQ94].

- *NAOS:* NAOS is an active extension to the commercial object-oriented database system O_2. In NAOS, rule management is integrated into the core of the database system, allowing for efficient detection of events. Details on NAOS can be found in [CCS94, CC96].

A series of European organizations performing research in the field of active database systems cooperate within *ACTNET*, the *Research Network on Active Databases*. Among the results of *ACTNET* there is an agreement on the characteristics of active database systems. This agreement is presented in *The Active Database Management System Manifesto* ([Con96]), where the features of active database systems are grouped into essential and optional ones. The manifesto is general in nature, since it represents a framework for relational, extended relational, and object-oriented active systems. Special aspects concerning active object-oriented database systems, such as the integration of rules into an object-oriented data model, are not considered by the manifesto.

All the prototype systems enumerated above share common features, however, a formal framework acting as a starting point for development is missing. No formal model has yet been developed that encompasses the wide range of functionality supported in different active object-oriented systems. If formal models are engaged in system description, they only cover parts of active behavior. An overview of approaches adopted for the formal specification of active behavior in diverse systems is given in [PCFW95]. In [PDW+93], Paton et al. provide a more informal framework for system comparison. They identify dimensions which relate to structure, execution model, and management of active rules. Dimensions of active behavior are also identified by Comai et al. ([CFPT95]). Both, Paton et al. and Comai et al. seek to provide a framework for active databases in general. They do not concentrate on systems with an object-oriented data model.

Chapter 3

Active Object-Oriented Database Design

Contents

3.1 Information-System Design 34

3.1.1 Elements of the Business Area 34

3.1.2 Database Design versus Application-Program Design 37

3.1.3 Database Design Process 39

3.2 From Conventional Database Design to Object-Oriented Database Design 40

3.2.1 Conventional Database Design 42

3.2.2 Object-Oriented Database Design 43

3.3 The Open Challenge: Business Rules 45

3.4 From Active Conventional Database Design to Active Object-Oriented Database Design 49

3.4.1 Active Conventional Database Design 50

3.4.2 Active Object-Oriented Database Design 52

3.5 Tool Support 54

3.6 Related Work on Active Object-Oriented Database Design . 55

Database systems in general aim at representing real-world semantics, which is otherwise hidden in application programs, in the database schema. The process of modeling the "universe of discourse," which comprises the set of problem-relevant entities of the real world, in a database schema is called database design. This chapter discusses the database design process in general and presents the evolution from conventional database design to

active object-oriented database design, especially with respect to the role that business rules play within the various approaches followed in database design.

The remainder of this chapter is organized as follows. Section 3.1 presents the basic components of an information system, discusses the difference between database design and application-program design, and finally introduces the database design process. Section 3.2 gives a brief overview of the evolution from conventional database design to object-oriented database design. Section 3.3 argues that business rules are better covered by database design than by application-program design and establishes the requirments on concepts to model business rules in conceptual database design. Section 3.4 consideres business rules and presents the evolution from active conventional database design to active object-oriented database design. Section 3.5 presents tools that may support the designer in using database modeling-concepts. Section 3.6 concludes this chapter with a short survey of related work on active object-oriented database design.

3.1 Information-System Design

The term "information system" is commonly misunderstood by practitioners because it can be recognized at two different levels [VS89]:

1. The "information system in the broader sense" is the totality of all formal and informal data representation and processing activities within an organization, including the associated communication, both internally and with the outside world.
2. The automated "information system in the narrower sense" is a computer-based sub-system, intended to provide recording and supporting services for organizational operation and management.

Herein, the term "information system" is used in the narrower sense.

Analyzing the business area results in more or less exact requirements. These requirements constitute the base for further design of information systems. Before discussing the design process itself, the main elements within the universe of discourse—in the following called "business area," which seems to be a more practitioner-friendly term [OHM+91]—are presented in Section 3.1.1 and the major differences between application-program design and database design are discussed in Section 3.1.2. Finally, the design process and its division into three successive design phases represents the focus of Section 3.1.3.

3.1.1 Elements of the Business Area

The business area defines the following elements as illustrated in Figure 3.1: inter-related structured objects together with static integrity constraints, behavior (operations on ob-

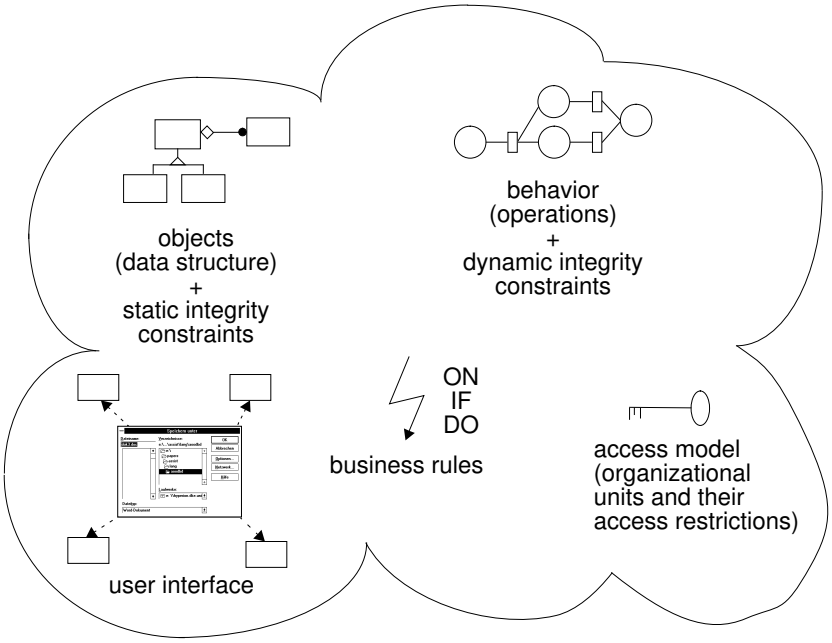


Figure 3.1: Elements of the business area

jects) together with dynamic integrity constraints, business rules, user interfaces with application logic, and an access model. These elements represent the main components of an information system.

Inter-related Structured Objects and Static Integrity Constraints. Objects (sometimes called "entities," "data objects," or simply "data") play the central role in the business area. They carry all static information about the business area in attributes and in relationships with which they refer to each other. Commonly, object types define the structure, (i.e., attributes and relationships) for a set of objects (i.e., for its instances). In the real world, relationships can have different semantics. E.g., some relationships may define an object to be existence-dependent on another object.

Static integrity constraints hold additional static information by constraining, for example, attributes to hold a small range of values of a certain data type only, or by constraining relationships to refer only to objects of a certain object type.

Object Behavior and Dynamic Integrity Constraints. The operations that can be executed on an object—perhaps depending on the actual state of an object—define its behavior. Operations may be simple reads or updates of attributes as well as complex computations. Public operations on objects, i.e., operations executable by some application program, should leave the database in a consistent state, supposed that they have started from a consistent database state.

Sometimes, executing an operation may be only possible if certain conditions are satisfied. Conditions restricting the execution of operations on objects are called dynamic integrity constraints (or "permissions to act" in [KS96a]).

Business Rules. Business rules are statements about business policies within the business area. They define when to do a certain task (operations on objects), whereas the task itself defines how a certain part of business is done.

In the diverse literature (e.g., [HKMS94, Her95]) different types of business rules are considered: consistency rules, rules on business procedures, and rules on business policies. Herein, only rules on business policies are called "business rules" as the following description of the three rule types shows.

- *Consistency rules* (integrity rules[1]) define static integrity constraints and dynamic integrity constraints that either prevent operations violating the constraints from taking place ("passive integrity constraints" in [HKMS94]) or react to constraint violations by performing alternative actions ("active integrity constraints" in [HKMS94]). Since integrity constraints only concern object structure and behavior, they are assumed to be part of them herein (see above) and thus are not referred to by the term "business rules."

- *Rules on business procedures* define in which order business activities may be performed. The order of business activities is either described by the order in which public operations on objects are executed within application programs or by explicitly specified constraints on the order in which operations may be invoked. In the former case, rules on business procedures are part of application logic, which may determine the order in which public operations are executed in a transaction. In the latter case, rules on business procedures are part of object behavior (e.g., expressed by some form of Statechart [Har87], Petri net [Pet86]), or automaton. But they are again not referred to by the term "business rules."

- *Rules on business policies* define which business procedures have to be performed at specific calendar times or which business procedures have to be initiated upon the occurrence of specific database events.

The term "business rule" used later on in this work refers to rules on business policies only. As it is shown in [KS96a] and shortly outlined in the above description, the other types of business rules are better represented by other high-level modeling concepts during conceptual database design. Static integrity constraints are defined declaratively with object

[1]Note that the term "consistency rule" may either denote integrity constraints or may denote rules given by some design method to enforce the correct application of modeling concepts in order to yield a consistent database schema. Consequently, rules concerning integrity constraints are better called "integrity rules."

structure, dynamic integrity constraints are specified declaratively with object behavior. Reactions to the violation of an integrity constraint should be either defined implicitly by means of relationship semantics or specified explicitly and declaratively by the designer. For instance, preventing an operation from setting an item's price lower than zero represents a "passive" static integrity constraint, which is specified declaratively for the involved attribute. Issuing a cascading delete of component objects if the superordinate object is destroyed represents an "active" static integrity constraint and can be defined implicity by the semantics of the component-relationship while modeling the object structure. Preventing an operation from reordering an item if enough items are in stock represents a dynamic integrity constraint, which may be specified declaratively as precondition of this operation.

Unlike integrity constraints and business procedures, business policies tend to change several times during lifetime of an information system. Therefore, business rules play a central role when maintaining information systems. They should be well-structured and well-organized such that information systems become easy to maintain.

User Interface and Application Logic. End users communicate with the information system through a user interface. Usually, users communicate by means of a conventional graphical user interface. But, in general, they may use any feature forwarding messages from them to the application program and vice versa. The application logic processes the user inputs and executes the intended business procedures by executing public operations on objects within transactions.

Access Model. In most cases, users are members of an organization. An organization consists of several organizational units, for instance, users and user groups. The access model defines restricted access to operations and objects for organizational units. An example for restricted access to an object attribute is that all users of a certain user group may read its value, but may not change it. Organizational units and their restricted access to objects and operations are not further considered in this thesis. Nevertheless, they represent an inherent part of the business area (see, for example, [Bic98]).

3.1.2 Database Design versus Application-Program Design

An information system supports storage, manipulation, and retrieval of information of the business area. Database systems allow to store information and provide basic access to it. Application programs allow the user to interact with the database system by issuing transactions with operations against the database, i.e., they provide convenient access to the information.

Developing an information system starts with defining the requirements. Usually, the

result of the requirements-definition phase addresses the information system as a whole. But database systems and application programs can cope with different requirements only. Therefore, the design phase following the requirements definition phase is divided into two (possibly parallel) design tasks (cf. [BCN92]): database design and application-program design.

Database design is mainly concerned with the design of problem-inherent semantics like object structures, integrity constraints, and (depending on the type of database system intended to use) object behavior. As objects (or data in general) play the central role in database design, it follows an object-driven (or a data-driven) approach.

Application-program design covers the design of user interfaces and the application logic lying behind the interface. Application logic defines the application's reactions to user inputs by means of transactions (business procedures) performing operations on objects within the associated database. Originally, modeling sequences of operations followed a function-driven approach with functional decomposition of user tasks. Since object-oriented concepts have been incorporated into application-programs, the design of application programs follows also an object-driven approach now. The behavior of the application object is driven by user inputs and thus determines the application logic.

Database Systems and application programs together have to cover all elements of the business area like object structure, object behavior, integrity constraints, business rules, application logic, and user interfaces. Database technology primarily determines which elements are subject to database systems and which elements are subject to application programs. Due to increasing capabilities of database systems, database design can cover more and more issues previously considered by application-program design. For instance, object-oriented databases incorporate the behavior of objects. The design of object-oriented databases has made modeling the behavior of objects a substantial part of conceptual database design and thus removed it from application-program design. The major advantage of this enhancement of database design is that the elements of the business area that are handled by the database system can be shared among all application programs accessing the database. These elements need not be designed separately for each application program.

> The overall objective of information-system design is to put as much semantics of the business area as possible into the database schema and to make application programs responsible for providing convenient access to the information managed by the database system.

Although database design is of major interest in this work, application-program design is further considered in the following sections to point out which elements originally treated within application-program design are covered by database design now.

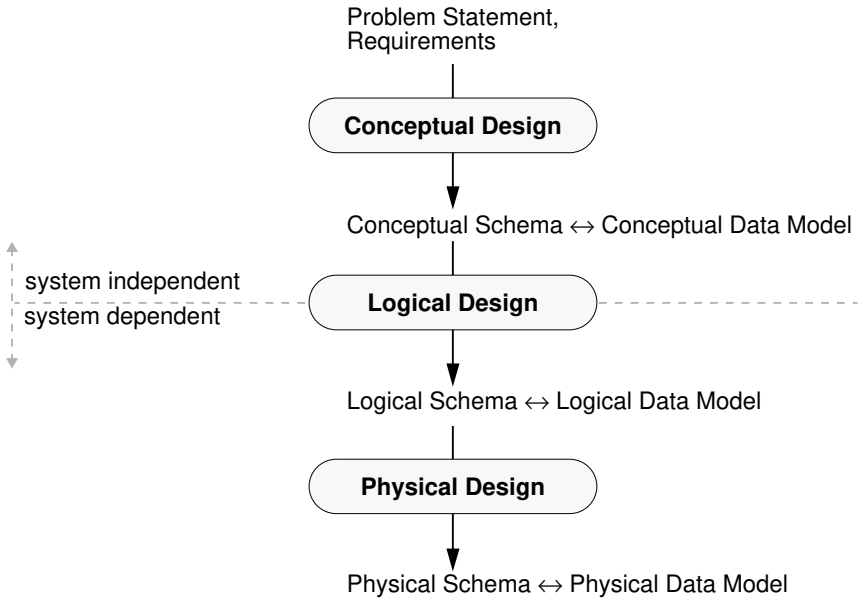


Figure 3.2: Database design process

3.1.3 Database Design Process

For larger information systems, the problem-statement and the requirements cannot be directly translated into a specific database language. Objects, their relationships, their behavior, and business rules as defined in the business usually are too high-level to be described directly by most database languages. Therefore, a design process is introduced that bridges the gap between the requirements definition and the implementation in some database language.

Figure 3.2 shows the sequential steps within the database design process: conceptual design, logical design, and physical design. The result of each step in the design process is a schema, which is represented in some model. Usually, a model is represented either textually by some textual language or graphically by a set of diagrams. Diagrams make modeled schema easier to comprehend as they hide much complexity in expressive graphical constructs and allow the designer to recognize and to control the dependencies between objects more easily.

1. *Conceptual design* is the first design step resulting in a conceptual schema, the most high-level specification of the real-world semantics within the database design process. A conceptual model represents the conceptual database schema independently from a particular database system or from a particular database technology that will be used for implementing the database schema. The conceptual database schema constitutes the base for communication between end users and database designers to clarify the requirements for the information system. Therefore, a conceptual model is represented by some expressive language, usually a set of diagrams.

2. Within *logical design*, the conceptual database schema is transformed into a logical database schema with respect to the target database technology (e.g., relational database systems) and perhaps to a particular target database system. Then, the logical database schema, which is specified by the logical model, can be implemented directly in the target database system.

3. *Physical design* defines physical aspects of the implemented database system like storage structures and access methods that allow fast storage, fast retrieval, and fast manipulation of data.

The modeling concepts of a specific model have to allow the designer to model all relevant elements of the higher-level model without loosing any relevant semantics. If the designer obeys the consistency rules (i.e., rules given by the design method to enforce the correct application of the modeling concepts), he or she yields a consistent database schema. But different designers may yield different consistent database schemata, which may differ in their quality. Therefore, design guidelines should assist the designer to achieve a "high-quality" database schema. A conceptual database schema is of high quality if it is comprehensible and maintainable. Guidelines for logical or physical database design aim at different qualities of the resulting database schema. For example, normalizing relations helps to get rid of redundancy in relational database schemata resulting from logical database design, whereas high performance and low storage costs determine the quality of database schemata resulting from physical database design. The desire to determine the quality of design results is not new. Boehm [BBL76] has already set up a number of criteria like comprehensibility and maintainability, which help to determine the quality of software in general.

Models are represented in textual or graphical languages. Tools like text editors for textual language specifications and diagram editors for graphical specifications support the designer in generating a database schema. Consistency-checking facilities help the designer to produce a consistent database schema (cf. Section 3.5).

The terms "design" or "database design" used in the sequel refer only to conceptual database design.

3.2 From Conventional Database Design to Object-Oriented Database Design

The evolution of database systems from conventional database systems to object-oriented database systems has aimed at incorporating more and more real-world semantics into the database schema and at removing elements of the business area with this semantics from application programs. For example, with relational database systems transferring money between two bank accounts has to be implemented in the application program by issuing

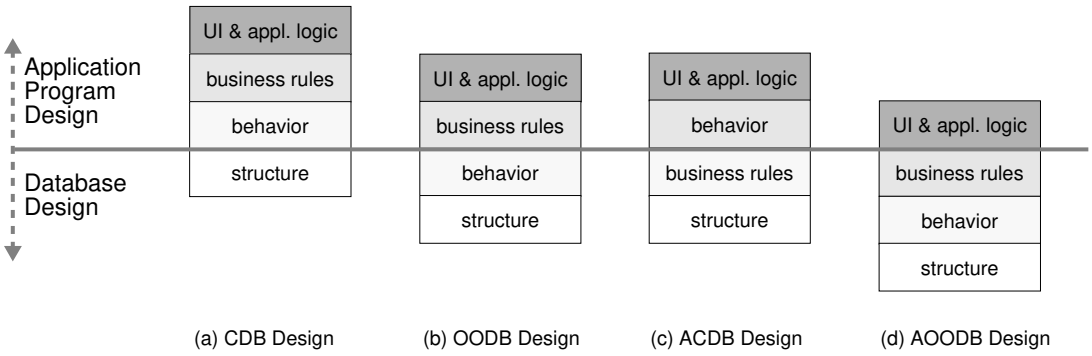


Figure 3.3: Where the elements of the business area are reflected in information-system design

two operations (one operation withdraws the money from the first account and the other operation deposits the money on the second account) in a transaction. With object-oriented database systems, transferring money can be implemented as a single public operation of some object (e.g., the bank) stored in the database, and this public operation invokes the withdrawal and the deposit of the money on the involved bank accounts.

Database design has been inspired by the evolution of database systems. But in general, conceptual design of information systems should not be influenced by the database system used for implementing the conceptual database schema. However, increasing functionality of database systems, for example, by incorporating integrity constraint maintenance supporting different semantics of relationships or by incorporating behavior, has called for more expressive conceptual database-design concepts, because many features of information systems previously modeled in application-program design are now considered in database design. The positive effect for application-program designers: They can concentrate on better, more human-centered user interfaces with appropriate application logic, and do not have to implement database functionality like integrity maintenance any more. Furthermore, the features (once implemented in the database) are common to all application programs that access this database.

The connection between the type of database system used for implementing the conceptual design and the approach followed in conceptual database design is twofold: First, new features of database systems have required additional modeling concepts in conceptual database design as mentioned above. Second, higher-level modeling concepts of conceptual database design are best implemented by supporting design concepts directly in the database system. E.g., results of conventional database design are best implemented in relational database systems, and results of object-oriented database design are best implemented in object-oriented database systems.

Subsequent Sections 3.2.1 and 3.2.2 present the evolution from conventional database design to object-oriented database design with respect to the elements of the business area that are considered by the actual approach, as illustrated in Figure 3.3. Each subsection

is structured as follows: First, the elements of the business area that are considered by the actual approach and the basic modeling concepts used for specifying these elements are introduced. Then, examples of design guidelines on how to use the introduced concepts are presented, examples of concrete design methods are shown, and related work is discussed. Issues concerning logical design conclude each subsection.

Remember that in the following the terms "design" and "database design" refer only to conceptual design and not to logical design or to physical design. If the term "design" is used differently, it will be said explicitly.

3.2.1 Conventional Database Design

Conventional database design assumes that application programs implement the behavior (operations) and the business rules besides the user interface and the application logic. Thus, application-program design techniques have to consider these aspects and have to provide concepts to model them. Conventional database design provides concepts only for modeling object structure in terms of attributes and relationships and for modeling static integrity constraints (cf. Figure 3.3 (a)).

Since the real world defines relationships with different semantics, a conceptual data model for conventional database design has to provide modeling concepts representing relationships with different semantics. Data models that support relationships with different semantics are called *semantic data models* [HK87].

Design Guidelines. Design guideline help to generate a conventional database schema that is easy to extend later on. For example, the designer has to decide whether to model a certain property of an object as an attribute of that object or to model it as an object of its own and to define a relationship between them. To illustrate this, consider the color of a car. If the color is represented by an object of its own to which a car refers, it is easy to add attributes like the identification number or the producer to the color later on. If the color of a car is only represented as attribute of a car, more effort is required to extend the database schema if colors should get attributes.

Another design guideline is presented by Batini et al. in [BCN92]. Normalization, originally a design guideline for logical database design to exclude storing redundant data because of functional dependencies between data, is also a matter of concern in conceptual database design. If functional dependencies between attributes of an object are detected, the object structure is restructured (e.g., by introducing new objects) in order to achieve "normalized" objects.

Related Work. The today's most famous semantic data model is the *Entity-Relationship Model* (ER model) [Che76, Che77]. The ER model describes entities (objects), their struc-

	cDB Design	ooDB Design	acDB Design	aooDB Design
Conceptual Model	ER Diagrams	OMT Diagrams	$(ER)^2$	Chimera
Logical Model	SQL	ODL, OQL	Ingres SQL	Ode

Table 3.1: Examples of conceptual and logical models in database design

ture (attributes), and relationships between entities. The ER model is visually represented in ER diagrams. The original ER model [Che76, Che77] has been tailored explicitly to conventional database design and therefore does not provide any information about the behavior of entities. Several extensions (e.g., [SNF80, SSW80, KS89]) to the original ER model incorporate new relationship semantics and/or behavior of entities. Basic modeling concepts of the ER model are, for example: entities, attributes, key attributes, relationships.

Remarks on Logical Design. The column labeled "cDB Design" in Table 3.1 presents the conceptual model and the logical model typically used in conventional database design: ER diagrams (the visual representation of an ER model) are translated into data definition statements and data manipulation statements specified in some SQL dialect of a relational database system, for example, in the SQL dialect of the relational database management system Oracle (Oracle, Inc.).

But relational database systems providing an ordinary SQL interface cannot represent directly each kind of relationship used in a semantic data model. Usually, relational database systems (especially former versions of today's commercial relational database systems) do not support relationships with the same semantics as defined for semantic data models. Thus, the relationship semantics not supported by the database system that finally implements the conceptual design has to be implemented within the application program. For example, consider a database system that does not provide a concept to specify a relationship with existence-dependent component objects. Thus, the application program has to implement cascading-delete semantics, which deletes all component objects if the superordinate object is deleted. However, modeling such semantics is still part of conceptual database design even if it is implemented by the application program during logical design.

Most relational database systems of the newer generation (also Oracle) already allow to specify relationships with different semantics using trigger mechanisms (e.g., for implementing cascading-delete semantics). Their data definition and manipulation languages resemble a newer generation of SQL, SQL-92 [MS93].

3.2.2 Object-Oriented Database Design

Object-oriented database design includes modeling structure and behavior of objects as

shown in Figure 3.3 (b). Static and dynamic integrity constraints are considered to be part of structure and behavior, respectively. The application program may use only consistency-preserving operations on objects, i.e., operations that observe integrity constraints. Business rules (rules on business policies) are still covered by application-program design with the consequence that business rules are hidden in several places within application programs. Business rules may change during lifetime of an information system but are hard to maintain if they are hidden within application-program code. Nevertheless, including business procedures, i.e., operations on objects, into database design is a step further towards considering all relevant semantics of the real world in database design.

The structure of objects is modeled in according to the same principles as defined for conventional database design. In object-oriented systems, the behavior of an object is usually modeled by a set of methods understood by the object. In conceptual design, the behavior of an object is modeled additionally in terms of a life cycle denoting the possible states an object may reside in. Executing operations (methods) on an object may change its state. Commonly, object life cycles are represented by formalisms based on either Petri nets [Pet86], Statecharts [Har87], or automata. Dynamic integrity constraints are usually specified as preconditions of operations.

A key concept that has already been introduced for conventional database design in the diverse literature is generalization or specialization of objects, respectively. The more special object inherits the structure of the more general object. Object-oriented design enlarges the notion of specialization by inheritance of object behavior. The inheritance of object life-cycles addresses new problems, which still require more research to be done on inheritance of object life-cycles [SS95, SS97]. Many of today's object-oriented design techniques do not care about the inheritance of object life cycles at all, but consider only the inheritance of methods. Generalization and specialization is again defined at object-type level and thus applied to all instances of the involved object types. Specialization and generalization of object types is ususally represented in type hierarchies.

Design Guidelines. Incorporating modeling concepts that are not available in conventional database design like object life-cycles, requires additional design guidelines addressing their usage. Riel [Rie96] gives a good overview of guidelines for designing object structure and behavior. He defines, for example, a guideline that requires to factor behavior common to several object types as high as possible in the type hierarchy.

Lieberherr and Holland [LH89] define the "Law of Demeter" restricting the set of possible receivers of messages sent within an object's method, so that the interaction between objects can be identified statically, i.e., by inspecting the relationships between objects and the method interfaces.

Related Work. Numerous object-oriented analysis and design methods have been developed in recent years. [KS96b] includes a short description of a large number of object-

oriented analysis and design methods. It is beyond the scope of this section to describe all known object-oriented methods. Only two of them are mentioned, the first one because it is one of the most popular methods, and because this approach is referenced several times throughout this work, the second one because it represents a precursor of the object-oriented approach underlying that one used in this thesis.

The *Object Modeling Technique* (OMT) [RBP+91] uses an object model based on the ER model for specifying the structure of objects and the relationships between them. The so-called dynamic model represents an object's dynamics, i.e., the object's state transitions as reaction to the execution of operations on it. The functional model allows to describe operations on objects as processes transforming input data into output data by means of dataflow diagrams. Each model is represented by appropriate diagrams. OMT is not tailored to the design of object-oriented databases. Rather, it is a technique for designing object-oriented systems in general.

The *Behavior Integrated Entity-Relationship* (BIER) Approach [KS89] extends the Entity-Relationship model, which is used for modeling the structure of objects and their relationships, with Petri nets representing the behavior of objects. This approach is the predecessor of *Object/Behavior Diagrams* [KS91a] (cf. Chapter 4).

Remarks on Logical Design. The column labeled "ooDB Design" in Table 3.1 shows that a conceptual model specified in OMT may be translated into the object definition language ODL and into the object manipulation and query language OQL, which are defined by the ODMG-93 standard [CB97]. The logical database schema can be implemented directly by database systems providing an ODMG-93 interface such as the object-oriented database system O_2 (O_2 Technology, Inc.). A conceptual model specified in OMT may be translated also into a conventional SQL dialect. As a consequence, implementing object behavior and dynamic integrity constraints again becomes subject to application-program design.

3.3 The Open Challenge: Business Rules

Neither conventional database design nor object-oriented database design consider business rules specified in the business area. Business rules remain subject to application-program design. Business rules have to be implemented in application programs (once in each application program) and are therefore hidden somewhere in the application-program code. As a consequence, the information system becomes hard to maintain. The deficiencies of hiding business rules in the application-program code have been recognized and novel approaches in database design provide new concepts that allow to omit business rules in application-program design and make their design an inherent part of database design.

At the conceptual level, business rules are usually formulated from an *external observation*

perspective. They express how an external observer, e.g., a swift clerk, could monitor calendar time and the database state to recognize predefined situations and to react to them accordingly in order to meet a company's business policy. Observing the database from an external perspective means to observe database changes only at the level of transactions. Thus, transactions have a great impact on the execution semantics of rules although transactions are usually not modeled explicitly in conceptual database design. The role that transactions play for modeling business rules will be addressed below when the requirements on the concepts for conceptual rule design are introduced in detail.

In the following, we discuss the requirements on rule concepts for modeling business rules at the conceptual level. These requirements have been established for the development of Situation/Activation Diagrams, a diagramming technique for active object-oriented database design, which has been presented in [LOS96, LOS97] and which is further elaborated in Chapters 5 and 6.

1. *Viewing events from the external user's perspective*: Database changes should be observed only at the level of transactions that are not further decomposable. In the realm of ACID-transactions, which are assumed, all actions performed within a transaction occur instantaneously for the external observer. Thus, all these actions should carry the *same* occurrence time-stamp, the occurrence time of the transaction (which is from a user's perspective the time of the commit of the transaction, or—as discussed later in Chapter 6—if commits are delayed, the time when a transaction requests to commit). Otherwise, a database containing the effects of all actions having occurred up to some point in time would be inconsistent (e.g., by containing the withdrawal action but not the deposit action of a banking transaction). Active object-oriented database systems, however, usually time-stamp an action event with the processing time within the transaction issuing the action. One can easily find examples showing that both approaches may yield different results when time-stamps of events are compared to recognize sequences of events. In the absence of a bitemporal model covering "effective time" as well as "internal processing time" the first approach seems to be more appropriate for modeling business rules. By the way, it is evident that this approach can be implemented in an active object-oriented database system by introducing complex events that are defined upon the action event and the commit event of a transaction. Yet, such a semantics should be provided at first hand for modeling business rules at the conceptual level.

2. *Future calendar events and intervals*: Business rules often refer to calender time, including future calendar events (e.g., 2001/01/01 00:00) as well as calendar intervals (e.g., May 1998). Event languages presented so far [CKAK94, GD94, GJS92a, MPC96] may not reference future calendar events and calendar intervals.

3. *Events and intervals as first-class objects*: When the incorporation of events and intervals into a database system is considered, it has to be distinguished between the *internal representation* and the *external representation* of events and intervals.

The internal representation denotes how events and intervals are represented during event detection, rule triggering, and rule execution. In most systems, events and intervals are treated as first-class objects internally. The external representation denotes how events and intervals are represented in the data model of a system, how events and intervals can be defined and used by a modeler. Treating events and intervals as first-class objects externally means that events and intervals are described by types and are collected in sets, usually called classes. In most systems, events and intervals are not treated as first-class objects externally. For example, the structure of an event, i.e. the form in which the context of an event occurrence is bound, usually is derived automatically from the specification of the event and cannot be specified explicitly (cf. [CKAK94, GD94, GJS92a, MPC96]). Collections of events and intervals are not available externally at all. Treating events and intervals as first-class objects externally brings a series of benefits. First, events and intervals as first-class objects represent a homogenous extension to the passive data model. Second, the structure of events and intervals and, thus, the information bound at event and interval occurrence can be defined explicitly. This allows to selectively bind meaningful information instead of simply binding all the information that is potentially senseful. Third, collecting events and intervals in classes allows to range over events and intervals the same way as over plain objects. Ranging over events and intervals may be useful in composing more complex events and intervals or in performing statistical analysis.

4. *Abstracting from granularity of time*: The smallest temporal unit that is not further decomposable from a user's perspective can often be of a coarser granularity than the temporal unit defined by two successive clock ticks of an internal system clock. E.g., time-stamps showing only minutes but not seconds are sufficient for many business applications. The time model lying behind the conceptual model should allow to abstract from the granularity of the time axis defined by an internal system clock.

5. *Declarative event language versus operational event language*: Similarly as QBE and SQL are better accepted by end users than Relational Algebra, end users are assumed to prefer a *declarative* event language over an operational one. The event languages of most prototypes of active object-oriented database systems follow an operational approach. The semantics of an operational language is determined by the semantics of the fixed set of operators provided. Generally, the semantics of an operator in turn is specified declaratively as a formula on the event history (cf. [CKAK94, GJS92a]). But the task of the rule modeler when specifying a composite event is to apply event operators step-by-step on event sets generated by other operators (down to basic events). Event languages based on an operational approach are very well suited to describe and to implement active capabilities of database systems, but might be less accepted by many end users for stating business rules at the conceptual level. The approach presented in [SW95] is declarative; it uses a special temporal logic to define situations. However, end users often have difficulties with pure logic, although they

accept languages that are based on logic but hide it, such as QBE and SQL.

6. *Reliable and controllable rule execution*: Modelers and system users expect business rules to be executed *reliably*, i.e., transactions performing rule executions are restarted automatically if they are aborted due to serialization failures, and *controllably*, i.e., specific types of exceptional situations occurring during rule execution are reported by events, which can again be handled by business rules. E.g., a delay event should be raised if a rule could not be executed successfully up to some user-specified temporal event. It is unclear from the research papers [CKAK94, GD94, GJS92a, MPC96] whether active object-oriented database systems execute rules reliably or whether aborts due to concurrency control conflicts have to be taken care of explicitly by additional rules. At least, this should be clear to the user at the conceptual modeling level.

7. *Asynchronous and immediate rule execution in top-level transactions*: Two rule categories are needed to formulate business policies: (1) rules that are considered and executed asynchronously to user transactions (e.g., reordering an item if the stock level has dropped below some threshold) and (2) rules that are considered and executed immediately upon temporal events (e.g., adjusting selling prices at specific hours). Active object-oriented database systems provide a different set of rule categories including the former but not the latter. Depending on the coupling mode [Day88], rules are considered either immediately, i.e., just after an event occurs within a transaction, or deferred, i.e., before the commit of a transaction, or detached, i.e., asynchronously and outside of the transaction raising the event. For the immediate and the deferred mode, rule actions can be either executed within the triggering transaction or in an independent top-level transaction. In the detached mode, rule actions can only be executed in a top-level transaction. Only the detached coupling mode is relevant for business rules that monitor the database state from an external perspective. In addition, business policies may require that certain business rules are considered and executed immediately upon temporal events. This feature is currently not supported by active object-oriented database systems and does not necessarily require the capabilities of real-time database systems. A less stringent criterion than real-time execution will be sufficient for many business applications: *temporal faithfulness*. Informally, temporal faithfulness means that transactions requesting to be committed after the temporal event triggering some business rule are serialized after the transaction performing the rule action and vice versa. E.g., if a business rule requires that selling prices of products are to be adjusted at 12:00 noon, sales transactions performed before 12:00 noon (i.e., the customer has decided to buy and thus requested to commit the sales transaction before 12:00 noon) should be serialized before the transaction adjusting the selling prices and sales transactions performed after 12:00 noon should be serialized after the transaction adjusting the selling prices. This ensures that sales, although possibly delayed, are always carried out at "up-to-date" prices.

We believe that the concepts required for modeling business rules at the conceptual level are not yet appropriately provided by current prototypes of active object-oriented database systems (cf. Chapter 2) and by recent approaches to active object-oriented database design (cf. Section 3.6). On the one hand, the rule concepts provided by active database systems can be used for implementing database issues such as view maintenance or intregrity maintenance. However, these issues are usually specified declaratively (e.g., integrity constraints) or defined implicitly within relationship semantics (e.g., cascading-delete semantics of component relationships) but not by means of rules in conceptual design. On the other hand, the rule concepts provided by active database systems either are too low-level to implement business rules directly or do not support the execution semantics as expected for business rules at the conceptual level.

Simon and Kotz-Dittrich [SKD95] have pointed out the need for conceptual active database design methods. We agree that existing rule languages of active database systems lack expressiveness and clear semantics. In most cases, these trigger languages are part of or built on top of existing database management systems. They may be appropriate for logical database design but usually are too low-level for conceptual database design.

3.4 From Active Conventional Database Design to Active Object-Oriented Database Design

Neither conventional database design nor object-oriented database design copes with business rules. Application-program designers model business rules, and programmers include them somewhere in the application programs. The newer generation of database systems and database design methods pay already attention to rules. Nevertheless, we believe that their rule concepts are not always appropriate to model business rules at the conceptual level. For appropriate modeling, rule concepts should meet the requirements presented above in Section 3.3.

Two approaches followed in active database design can be distinguished: Section 3.4.1 presents active conventional database design, which enriches conventional database design by adding rule-design concepts. Section 3.4.2 presents active object-oriented database design, which enriches object-oriented database design by adding rule-design concepts. Each subsection is again structured as follows: First, the elements of the business area that are considered by the actual approach and the basic modeling concepts are introduced. Then, examples of design guidelines on how to use the introduced concepts are shown. Related work on active conventional database design is described in Section 3.4.1. Related work on active object-oriented database design is discussed separately in Section 3.6 as this approach represents the focus of this thesis. Issues concerning logical design conclude each subsection.

Remember again that in the following the terms “design” and “database design” refer to

conceptual design only and not to logical design or to physical design. If the term "design" is used differently, it will be said explicitly.

3.4.1 Active Conventional Database Design

In addition to conventional database design, *active conventional database design* provides concepts to model rules, usually an event language and a rule language. Figure 3.3 (c) illustrates that besides structure also business rules are now considered in active conventional database design. Modeling the behavior of objects, the application logic, and the user interface, however, remains part of application-program design.

Rules are usually specified in an ECA-like structure and their execution semantics is specified by coupling modes. Besides not fully meeting the requirements on rule concepts for modeling business rules at the conceptual level (cf. Section 3.3), active conventional database design has to cope with the lack of concepts for modeling behavior. Thus, behavior modeling is subject to application-program design. Business procedures, which are modeled by the behavior of objects and which are sometimes initiated by business policies, are modeled in application-program design. Application-program designers and database designers have to cooperate in implementing business procedures and business rules. The more business rules defined within the business area the higher becomes the coordination effort.

Further, business procedures might be modeled twice: within database design if a certain procedure is initiated by an event and within application-program design if the same procedure is invoked explicitly by the user. Even if this is not the case, the design lacks homogeneity because some business procedures are covered by database design, others are covered by application-program design. Usually, active conventional database design methods overcome these deficiences by providing a concept to specify procedures, which can be invoked by rules as well as by the user.

Design Guidelines. Incorporating the rule paradigm into database design requires appropriate design guidelines addressing rule modeling. In general, the larger the set of rules and the more inter-dependent the rules, the more difficult to supervise and to maintain the resulting model. For example, the decision whether or not to split a rule with a complex event, a complex condition, or a complex action into several inter-dependent rules with simpler events, simpler conditions, and simpler actions should be supported by design guidelines. Splitting "big" rules into "small" rules or merging small rules into big rules influences the amount of rules specified in the conceptual model. More information on design guidelines for active database design is presented in the next subsection about active object-oriented database design and concrete guidelines can be found in [Lan97].

Related Work. Tanaka et al. in [TNCK91, Tan92] initiated the work on active conventional database design. Events and rules are integrated into the ER model. The resulting *Entity-Relationship-Event-Rule* ($(ER)^2$) model provides an extended architecture of tools for assisting the database designer in the task of specifying, analyzing, and translating rules into executable data definition statements in specific relational database systems. An $(ER)^2$ model can be specifed declaratively in a textual specification language or in $(ER)^2$ diagrams, which can be translated automatically into SQL statements. Rules in the $(ER)^2$ model follow the ECA paradigm.

The textual specification allows to combine several primitive operations on the database in procedures, which can be invoked by rules. Thereby, this approach overcomes a major deficiency of conventional database design: Usually, no concepts are provided for specifying business procedures. Then, in order to invoke a business procedure by a rule, it must be broken down into primitive operations, each of them invoked by a single rule (see also the remarks on logical design below).

This approach meets partially the requirements specified in Section 3.3. For external observers, database events are time-stamped with transaction time. Rules, however, are processed within the transaction that raises the rule-triggering event. Although temporal and external events are distiguished from database events, it is not clear whether rules can be defined upon temporal events in the future. Futhermore, it is not clear if actions are executed reliably and controllably because no comment is made on what happens if a transaction that executes a rule action fails. Different rule execution modes similar to asynchronous or immediate rule execution in top-level transactions are not available. Nevertheless, this approach has constituted the base for further work in the area of conceptual active database design.

Remarks on Logical Design. The column labeled "acDB Design" in Table 3.1 identifies $(ER)^2$ diagrams as a representative conceptual model for active conventional database design and Ingres SQL (Computer Associates International, Inc.) as a representative logical model for active conventional database design. Ingres SQL provides a concept to specify rules with simple events and business procedures directly within the database. A mapping between these models has been conducted in [Tan92].

If the target database system does not provide a concept for the specification of procedures executing primitive database operations or other procedures, logical database design is faced with the following difficulty. Rules that implement business policies are reflected in the database system and therefore cannot trigger business procedures that are not reflected in the database but are part of some application program. A business procedure represents a collection of several execution units (primitives like retrieval- and update-operations) processed by an application program in a predefined order within a single transaction. In order to trigger business procedures in the database system, they have to be broken down into their execution units, which can be processed directly by the active database system.

Then, for each execution unit, a rule is defined with the execution unit as action. The execution sequence of the actions or rules, respectively, is reflected in the rules events. Imagine a business procedure with three successive execution units U_1, U_2, and U_3. Three rules will be required to process these sequence in the desired order. The first rule fires upon occurrence of the event decribed in the business policy and executes U_1. The second rule reacts to the execution of U_1 and executes U_2. The third rule reacts to the execution of U_2 and executes U_3. Notice that the events of the two latter rules may fire only if the business policy event *and* the event raised by the execution of the previous execution unit occur. Otherwise they would fire every time after their preceeding execution unit is processed even if the business process is invoked explicitly by the user. One can easily recognize that the implementation of triggering business procedures with more execution steps and/or with more complex control structures ends up in a large number of complex and highly inter-dependent rules. Then, the active behavior of the resulting information system becomes rather incomprehensible. In $(ER)^2$, this problem is solved by adding a language concept for procedure modeling and by making procedures as actions of rules. These procedures can be translated directly into database procedures specified in Ingres SQL.

3.4.2 Active Object-Oriented Database Design

Adding concepts for modeling business rules to object-oriented database design finishes the evolution from conventional database design to active object-oriented database design. *Active object-oriented database design* comprises concepts for the specification of object structure, object behavior (also called the "passive behavior" of objects), and business rules (also called the "active behavior" of objects). As a consequence, application-program design has to consider "only" the design of the application logic and of the user interface as illustrated in Figure 3.3 (d). Business procedures as well as business rules are given in an object-centered view, because they represent the passive behavior and active behavior of objects, respectively.

The requirements on a rule concept suitable for modeling business rules have already been discussed in Section 3.3. These requirements can be applied for rule concepts in active object-oriented database design, too.

Business rules are again best specified in an ECA-like structure. The deficiencies of active conventional database design, where object behavior is not modeled in database design but in application-program design, are overcome by modeling behavior in database design: First, business rules and business procedures are now subject to database design. Thus, the coordination effort between database designers and application-program designers is reduced. Further, business rules and business procedures are not hidden in application-program code any more. They are all part of the database, which makes the resulting information system easier to maintain. Second, business procedures need not be modeled twice, once for the application program when business procedures are invoked explicitly by

the user and once for the database when business procedures are invoked automatically by some rule. A business procedure stored in the database may be invoked either by a user or by a rule.

In contrast to active conventional database design, where rules are always specified globally for the database, active object-oriented database design allows us to organize rules by attaching them to object types. Then, rules associated with an object type are handled only for objects of this type.

Core concepts of object-orientation like inheritance and overriding have to be adapted or extended if new concepts like rules are embedded into object-oriented models. If rules are attached to object types they are inherited by more special object types. Inherited rules must not be freely overridden. Overriding inherited rules requires additional restrictions. For details on the inheritance of rules, see [Lan97].

Design Guidelines. The following list presents a few examples of design guidelines with respect to rule modeling in active object-oriented database design. Some of them have already been presented in [KS96a].

- Static and dynamic integrity constraints are best modeled declaratively when specifying the structure or the behavior of objects and not by means of rules. Of course, integrity maintenance is often implemented by means of rules, but this is subject to logical database design and not to conceptual database design.
- Business procedures represent a collection of several execution steps, processed by the database system in a predefined order. Business procedures and thus the execution sequence of their steps should be modeled as the passive behavior of objects the database. As already mentioned, using rules to specify the sequence of execution steps leads to a large set of highly inter-dependent rules, whose execution behavior is hard to comprehend.
- Business rules represent obligations to act as they say when a certain task has to be done. Business rules should be modeled explicitly by a rule concept.
- As rules initiate operations on objects, they should be specified locally to the object type on whose instances they execute an operation. On the contrary, rules initiating global operations within the system should be specified globally. By specifying rules locally to object types, the large set of rules, which is addressed as the main problem of active non-object-oriented database design, is divided into smaller sets of rules local to specific object types.
- Instead of specifying and executing a rule for each member in a set of objects, it is more efficient to specify a single rule that executes an operation which performs the required operation on all members of the set.

Related Work. Related work on active object-oriented database design is discussed separately in Section 3.6 as this approach represents the focus of this thesis.

Remarks on Logical Design. The column labeled "aooDB Design" in Table 3.1 shows Chimera [CBFP95, CFPT96] as a conceptual for active object-oriented database design. A conceptual database schema specified in Chimera may be implemented in the active object-oriented database system Ode [LGA96] as taken into consideration in [CBFP95]. Chimera will be discussed in more detail in Section 3.6.

Implementing business procedures of a conceptual schema in active conventional database design has been identified as a severe problem without a concept to implement business procedures in the target database system. Using an active object-oriented database system like Ode for implementing business rules eliminates this problem. Business procedures may be directly specified by means of operations on objects and need not be broken down into several database primitives. The number of business rules is kept low. The large number of inter-dependent rules (caused by rules defining the execution sequence of single steps) is replaced by a lower number of less inter-dependent rules, one rule for each business rule.

3.5 Tool Support

Tools should support the designer in using the modeling concepts in a comfortable way and in representing the conceptual model appropriately. In addition, tools should help the designer to check the consistency of the developed database schema, and they should assist the designer in applying modeling concepts according to specific design guidelines. The following kinds of tools are usually provided:

- Diagram-editor tools let the designer specify structure, behavior, or rules in some diagramming technique. Text-editor tools let the designer specify structure, behavior, and rules in some textual language. Both should assist in checking whether the specified database schema is consistent and whether the design guidelines are obeyed.
- Schema generators translate the conceptual database schema into a logical database schema more or less automatically.
- Static rule-analysis tools help the designer to predict the behavior of rules. The set of rules is evaluated at compile-time to check whether the rules consider the following properties [AWH92]: termination, confluence, and observable determinism. *Termination* requires that rule processing terminates for a given set of rules in a final database state. *Confluence* requires that rule processing terminates in a unique final state independent from the order in which rules are executed. *Observable determinism* requires that all visible actions performed by rules are the same. For more details on these properties, see [AWH92].

- Dynamic rule-analysis tools (cf. [Ret96]) allow to supervise the active database system at run-time by tracing already triggered rules, or by manually debugging rule execution.

The importance of tools in active database design is underlined in [ABJZ96] and in [SKD95]. After conducting case studies on active information systems, Appelrath et al. [ABJZ96] point out the lack of methods and tools for active database modeling and the lack of transformation tools (schema generators), which translate higher-level specifications into lower-level specifications. Simon and Kotz-Dittrich [SKD95] identify, besides the insufficient methodological support in analysis and design of active databases, the insufficient security, reliability, and predictability, and the low performance of triggers in active database systems as the major problems. Rule analysis, simulation, testing, debugging, and optimization techniques should help to solve these problems.

3.6 Related Work on Active Object-Oriented Database Design

So far, a lot of work has been done in the field of active databases. The rule and event languages introduced in the diverse literature (cf. Chapter 2) are tailored to the logical design of databases and thus usually are too low-level for the specification of business rules at the conceptual level. This section presents several approaches to the conceptual design of active object-oriented database design.

As far as possible, the presented approaches are examined whether they meet the requirements on rule concepts defined in Section 3.3. But note that these requirements have been set up for the development of Situation/Activation Diagrams, which will be presented in Chapters 5 and 6. Other approaches to the conceptual design of active object-oriented database design are not be able to meet exactly these requirements because they usually do not view events from an external user's perspective, which represents the premise for most of the other requirements. These approaches provide different concepts to formulate business rules and therefore need not be less powerful.

Active Object-Oriented Database Modeling Technique. Silva and Carlson [SC96] introduce the *Active Object-Oriented Database Modeling Technique* (A/OODBMT), which is based on OMT [RBP+91] and integrates four kinds of graphical models, namely a nested object model, a behavior model, a nested rule model, and a nested event model. The nested object model extends OMT with nesting capabilities and allows to incorporate rule specifications for checking integrity constraints. The behavior model exists of a dynamic model, a functional model, and transaction diagrams. The dynamic model and the functional model (both borrowed from OMT) describe the dynamics of objects and the data-flow

between operations, respectively. Transaction diagrams extend interaction diagrams borrowed from [JCJO92] with transaction commands to begin and end transactions. The nested rule model allows to specify rules attached to object types. The nested event model serves for the specification of primitive and composite events.

This approach meets the requirements defined in Section 3.3 as follows:

- Events are not viewed from the external user's perspective. They are detected within a transaction. Whether a database event is time-stamped with effective time or with processing time, is unclear.
- Calendar events are modeled by means of absolute, periodic, and relative temporal events [Day88]. Whether events preceding a future temporal event by a certain time-span can be used, is unclear.
- Events and intervals are not treated as first-class objects. Intervals are represented only by a starting and an ending point in time within a composite event, the monitoring interval event.
- Whether this approach can abstract from the time granularity of a future underlying system, is unclear because events are always detected within transactions and time-stamping of events is not treated.
- Composite events are specified graphically with an operational event language using event constructors. They are not specified declaratively as end users usually prefer.
- The reliability and controllability of the rule execution is given implicitly by coupling modes [BBKZ93] defined between the event-triggering transaction and the action. But it is not possible to define a rule's action transaction to restart automatically after being aborted or to raise an event if the rule execution has failed. Exception rules and contingency rules allow the designer to specify whether a rule raises exception events upon a certain result of the action or to restrict the execution time of a rule to a certain time limit.
- Coupling modes define a rule's execution semantics. The "sequential causually dependent" mode is similar to our asynchronous execution because the rule's action is executed after the event-triggering transaction has committed (but not aborted). A coupling mode ensuring the temporal faithful execution of rule transactions and user transactions is not supported.

The above discussion shows that this approach does not meet all the requirements. Partially, this is because rules are used also for modeling static and dynamic integrity constraints and not only for modeling business rules, for which events should be viewed from the external user's perspective. Rules implementing integrity constraints have to be executed within event-triggering transactions and thus require different concepts.

The nested rule model allows to attach each rule to some object type. However, events are specified globally and are used for triggering rules that are specified locally to object types. Since rules are attached to object types, they are inherited and can be overidden at more special object types.

Chimera. Ceri et al. [CBFP95, CFPT96] present *Chimera* as the (textual) conceptual language of the Esprit IDEA Methodology. Chimera includes deductive rules and active rules ("triggers") into object-oriented database design. Chimera provides, besides language constructs for the specification of object structure and passive behavior, language constructs for the specification of rules: Datalog-like deductive rules are used for specifying integrity constraints, derived attributes, and views in a declarative way. Active rules following the ECA paradigm serve as specification purpose for business rules. Deductive rules are translated semi-automatically into active rules. In addition, rule metrics can be defined to measure the progress in achieving a business procedure's objective.

The rule and event language of Chimera (see also [CGMR95, MPC96]) meets the requirements defined in Section 3.3 as follows:

- Events are not detected outside a transaction and thus are not observed from the external user's perspective. Since integrity constraints are modeled by deductive rules, which are then (during logical design) translated into active rules, events have to be recognized within transactions. Nevertheless, events of business rules are detected also inside transactions, and rules are, depending on the coupling mode, processed either immediately after the event occurrence or deferred before the commit of the event-triggering transaction.
- Calendar events or intervals, a substantial part of the specification of business rules, are not supported at all.
- Intervals are not treated as first-class objects. If an event is required to occur during an interval, its occurrence has to be compared with the interval-ending events.
- Since events are detected inside transactions, time-stamping of them is not made explicit. Whether Chimera allows to abstract from the time granularity of a future underlying system, is therefore unclear.
- Composite events [MPC96] are specified operationally. They are specified by means of four predicates, so-called boolean operators: conjunction, disjunction, negation, and precedence. Consumption modes [CKAK94], which can be specified for the rule execution, change the event history by removing event occurrences from the history.
- Rule transactions are not restarted automatically after being aborted due to serialization failures and therefore are not executed reliably in our sense. Further, as rules always are executed inside the event-triggering transaction, this transaction cannot

commit if executing the rule's action fails. Then, neither the occurrence of the event nor the execution of the rule's action (even tried) will become public. Therefore, rules cannot be executed controllably in our sense.

- Chimera does not provide concepts comparable to the asynchronous or to the immediate rule execution because neither rules can be executed in their own top-level transaction nor temporal events are available.

Surprisingly, Chimera does not meet any of the requirements identified in Section 3.3. Chimera represents a language that can be translated easily into existing active database languages and thus is characterized by their features like an operational event language that supports only intra-transactional events. The concepts provided by Chimera are not sufficient to model some kinds of business rules like rules that are triggered upon calendar events.

Chimera provides targeted and untargeted triggers [MPC96]. Unlike untargeted triggers, targeted triggers are attached to a certain object type and are triggered only for its instances. Further, Chimera allows to specify the execution sequence of rules by giving rules higher priority relative to other rules. But specifying priorities between rules usually is not necessary at the conceptual level. Instead of requiring rules to be executed in a certain order, this order should be reflected in the passive behavior of objects, which implement the rules' actions. At the conceptual level, rules with different execution semantics, like asynchronous and immediate execution, define implicitly priorities for their execution.

Priorities for rule executions are typically used in logical database design. For example, rules that implement integrity constraint maintenance have to be scheduled before rules that implement business rules. Before a business procedure that is triggered by a business rule can be performed, the transaction signaling the business rule's event has to be checked whether it violates integrity constraints. The rules implementing integrity constraints must have higher priority than rules implementing business rules.

Since Chimera considers also the automatic generation of active rules, prototyping of active and deductive rules (which are translated into active rules) requires static and dynamic rule analysis. *Arachne* [CBFP95] is a static rule analyzer for Chimera supporting event-based analysis (simple syntax-based analysis of the triggering graph) and condition-based analysis (refines event-based analysis by removing arcs of the triggering graph according to condition evaluation).

To cope with large collections of active rules, Chimera introduces the concept of *modularization* [BCP96]. Stratification of rules is the key principle: rules are partitioned into disjoint strata. The designer can abstract rule behavior by first reasoning about each stratum, and then by globally reasoning on the behavior across strata. Behavioral stratification is used for verifying that rule processing terminates for each strata.

ERSC Monitors. Mück [Müc94, Müc95] presents a database design formalism based on the concept of *ERSC monitors*. ERSC monitors include extended ER diagrams for modeling the structural aspects of the conceptual schema, and modified Statecharts [Har87] for modeling the behavior of entities. This approach is in some way object-oriented, even the term "object-oriented" is never used and typical issues like inheritance are never addressed. A graphical representation of ERSC monitors supports the designer in specifying the conceptual schema. An equivalent algebraic representation is used for analyzing the schema and used as source for translating it into ECA rules. Two types of ERSC monitors are distinguished: policy monitors and initiative monitors. Policy monitors guarantee that integrity constraints hold, and initiative monitors start internal or external activities; thus, the system may communicate also with other system components.

The concept of ERSC monitors is not evaluated with respect to the requirements defined in Section 3.3. First, ERSC monitors are a more complex concept than rules. As opposed to rules, the semantics of an ERSC monitor is defined by a user-definable execution life cycle, which is controlled by events. Second, additional features like restricting the lifetime of a monitor by creating and destroying it upon certain events, make this concept more flexible. Third, execution semantics based on database transactions are not defined for ERSC monitors.

The BROCOM Approach. Herbst presents in [Her97] the business rule-oriented conceptual modeling approach (*BROCOM*) based on previous work presented in [Her95, HM96]. The structure of rules in BROCOM (ECA^2 structure) differs from the conventional ECA structure in the additional action part, which is executed if the condition fails. The BROCOM approach goes a step further towards workflow modeling as it requires to associate the event, the condition, and the action of a rule with different "processors." Processors are human beings or machines, they belong to organizational units, and are responsible either for raising the event, for evaluating the condition, or for performing the action. This approach provides different types of simple events like data manipulation events, (future) temporal events, user-defined events and composite events, which are composed of other events using several operators. As details of the execution behavior of rules specified at the conceptual level are not discussed, especially with respect to transactions, it is not possible to relate the BROCOM approach to the requirements defined in Section 3.3.

Rule Patterns. Kappel et al. [KRSRS96] and Retschitzegger [Ret96] introduce *rule patterns* as generic rule-based solution for specifying business policies in general. Rule patterns are descriptions of ECA rules that capture business policies in a generic application-independent manner. They represent an abstraction of several similar business rules by parametrizing the parts of an ECA rule and by providing predefined application-independent components. Applying application-dependent components to a generic rule pattern results in appliation-dependent, concrete rules. Furthermore, rule patterns are

organized in sets of patterns, each set of patterns dealing with a specific kind of business policy like integrity patterns, activity ordering patterns etc. The pattern sets are collected in a pattern library.

Since rule patterns address the organizational aspect regarding business rules in active (object-oriented) database design, but not the design of concrete business rules for active databases, the requirements defined in Section 3.3 are not applicable. Examples illustrate the applicability of rule patterns in logical database design, using the event and rule language of TriGS [KRSRV94, Hie96, Ret96, KRSR98]. Nevertheless, the princple of rule patterns is applicable to conceptual active (object-oriented) database design, too.

Rule patterns represent a very powerful mechanism to cope with business policies in diverse applications that would otherwise lead to a large number of rules. Although grouping of rule patterns into pattern sets dealing with specific business policies reduces the set of rules into smaller partitions, rule specifications cannot be attached to object types even if they specify the active behavior of a single object only.

Active Object/Behavior Diagrams. Activation scripts, which have been presented by Bichler and Schrefl in [BS94], constituted the base for further work on active object-oriented database design, presented in this thesis. Activation scripts are built on top of Object/Behavior Diagrams [KS91a]. In Object/Behavior Diagrams, the structure of object types is specified by means of object-type diagrams, the passive behavior by means of behavior diagrams (life cycle diagrams, activity-specification diagrams, and activity-realization diagrams). Activation scripts allow to specify the active behavior of objects. They consist of an event, a guard, and an activity corresponding to the ECA paradigm. The event-specification capabilities are very limited (but declarative and graphical), and different rule execution semantics are not discussed.

This approach constituted more or less the starting point for the research project in this thesis and does not yet meet most of the requirements identified in Section 3.3.

Chapter 4

Object/Behavior Diagrams

Contents

4.1 Running Example: Communication System 63
4.2 Object-Type Diagrams 63
4.2.1 Properties 64
4.2.2 Relationships 66
4.2.3 Type Hierarchies 71
4.3 Object-Class Diagrams 71
4.4 Behavior Diagrams 73
4.4.1 Life-Cycle Diagrams 74
4.4.2 Activity-Specification Diagrams 76
4.4.3 Activity-Realization Diagrams 79
4.4.4 Compound Activities 79
4.5 Query Diagrams 80
4.5.1 Elementary Structure 81
4.5.2 Expressions 82
4.5.3 Domain Box 83
4.5.4 Condition Box 85
4.5.5 Result Box 89
4.5.6 Advanced Concepts 91

This chapter presents Object/Behavior Diagrams, a diagramming technique for the conceptual design of object-oriented databases. Object/Behavior Diagrams have been introduced in [KS91a] and much work has been done in extending and refining them since then

(cf. [Kap91, KS91b, KS92, KS94, Sch90, SS95, SS97]). A thorough presentation of the original definition of Object/Behavior Diagrams as well as a summary of the extensions and refinements made to Object/Behavior Diagrams is given in [KS96b]. The focus of this chapter is to describe Object/Behavior Diagrams so that they can serve as a basis for the conceptual design of active features in the following chapters. Some aspects of Object/Behavior Diagrams discussed in the diverse literature are left out, others are discussed in depth. Also extensions to the original definition of Object/Behavior Diagrams are made. Object/Behavior Diagrams are extended by query diagrams for specifying derivation expressions for derived properties, by class diagrams for expressing local referential integrity constraints, and by an enhanced way for specifying pre- and postconditions of activities.

Object/Behavior Diagrams provide a uniform graphical representation of the structure and the passive behavior of objects, strictly following the object-oriented paradigm. Contrary to many object-oriented data models, Object/Behavior Diagrams distinguish clearly between object types and object classes (cf. [KS92]). An object type defines the structure and the behavior of its potential instances. An object class serves as container, comprising a set of objects of a certain object type. Object classes are properties of objects, and therefore they are called local object classes. The members of a local object class are dependent on and exclusively belong to the object of which the local object class is a property. To treat global collections of objects uniformly as local object classes, the database is considered an object itself that is instance of a *global object type*. Its local object classes correspond to the classes of a conventional object-oriented database, and members of these local object classes correspond to independent objects stored in the database.

The remainder of this chapter is organized as follows: Section 4.1 starts with a short excursus: It introduces a running example by means of which the concepts of Object/Behavior Diagrams are illustrated. Section 4.2 presents object-type diagrams describing the structure of an object type. An object-type diagram depicts the properties of an object type with their names and domains and classifies them, according to their semantics, into one of several predefined property types. Additionally, object types may be organized in subtype-supertype relationships, where the subtype inherits both the structure as well as the behavior of its supertype. Section 4.3 presents object-class diagrams, an extension to the original definition of Object/Behavior Diagrams. An object-class diagram depicts local referential integrity constraints that apply to the instances of the object type to which the object-class diagram is associated. Section 4.4 presents behavior diagrams. A behavior diagram specifies the behavior of an object type's instances by a set of activities and a set of states. Section 4.5 concludes this chapter by introducing a further extension to the original definition of Object/Behavior Diagrams, namely query diagrams. A query diagram allows the graphical specification of a derivation expression for a derived property. Query diagrams lay the foundations for situation diagrams, a high-level graphical language for describing scenarions upon which activities have to be invoked automatically (cf. Section 5).

4.1 Running Example: Communication System

We use a communication system to illustrate the concepts of Object/Behavior Diagrams. Whereas details of the communication system are explained when they are used for exemplifying a concept in the subsequent session, the basic motivation of the communication system is sketched out briefly. The presented communication system allows users to post notices to notice boards and to send messages to other users in the form of mails and circulars:

- *Notice:* A notice is a message that is sent to a notice-board administrator to be posted to one or more notice boards. A notice-board administrator—a role that is played by selected users—maintains a set of notice boards, each dedicated to a specific topic. A notice board belongs exclusively to one notice-board administrator, who is the only one allowed to post notices to and take notices from the notice board.

- *Mail:* A mail is a message with several users as receivers. After a mail has been composed, the initiator of the mail has to deliver it explicitly. When a mail is delivered, a delivery is created for every receiver. With every delivery it is recorded whether the receiver has read the message.

- *Circular:* A circular is a message with several users as receivers. A circular is not delivered to all receivers simultaneously but is delivered to one receiver after the other. An order is specified among the receivers. After a circular has been composed, the initiator of the circular has to start the circulation explicitly. Each receiver is invited to comment on the message, but he/she might give no commentary.

The structure as well as the passive behavior of the communication system will be shown in the course of this chapter. The following Chapters 5 and 6, which introduce concepts for modeling the active behavior of a database, will come back to this example and will extend it by events, intervals, and rules.

4.2 Object-Type Diagrams

Object-type diagrams rely on semantic data models (cf. [HK87]) and show the properties defined for instances of an object type. Properties describe the conceptual structure of an object type, not its physical (internal) structure. Consequently, properties are considered to be abstract concepts and should not be confused with instance variables as found in diverse object-oriented models. Rather, every property is to be interpreted as a public retrieval operation returning an instance or a set of instances of its domain, depending on whether the property is single-valued or multi-valued.

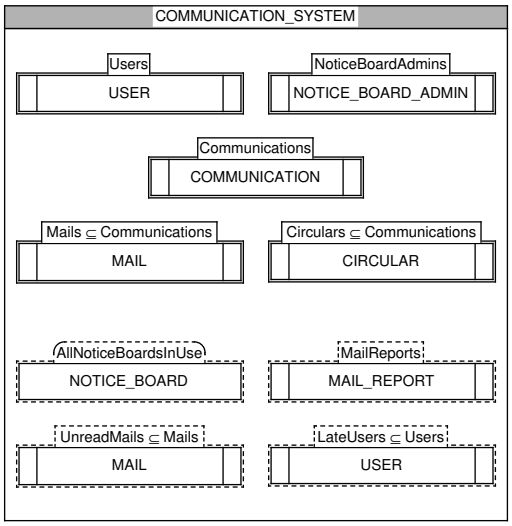


Figure 4.1: The global object type COMMUNICATION_SYSTEM

Graphically, an object-type diagram represents an object type by a box giving the name of the object type in a shaded bar at its top.

Example: Figure 4.1 shows the object-type diagram of the global object type COMMUNICATION_SYSTEM, which represents the whole communication system. Figure 4.2 shows the object-type diagrams of the object types modeling users, notice-board administrators, notice boards, notices, and communications in the form of mails and circulars.

4.2.1 Properties

Properties of an object type are distinguished in *attribute properties* and *relationship properties*. Every property has a name and a domain and is single-valued or multi-valued. Attribute properties model descriptive features and have a predefined data type (INTEGER, REAL, STRING, BOOLEAN, DATE, TIME, or TIMESTAMP) as domain. Relationship properties represent references to other objects and have an object type as domain. Properties are depicted by symbols, that are nested within the box representing an object type. An attribute is represented by a rounded rectangle, a relationship is represented by a rectangle. Generally, the name of a property is placed above the graphical symbol representing it, the domain of the property is placed within the symbol. Single-valued properties are depicted by single-line borders, multi-valued properties by double-line borders.

Unlike most notations for representing a schema in object-oriented modeling, relationships are not represented by arrows or solid lines, but by nested graphical boxes in Object/Behavior Diagrams. This approach is favoured by two main reasons: First, nesting of

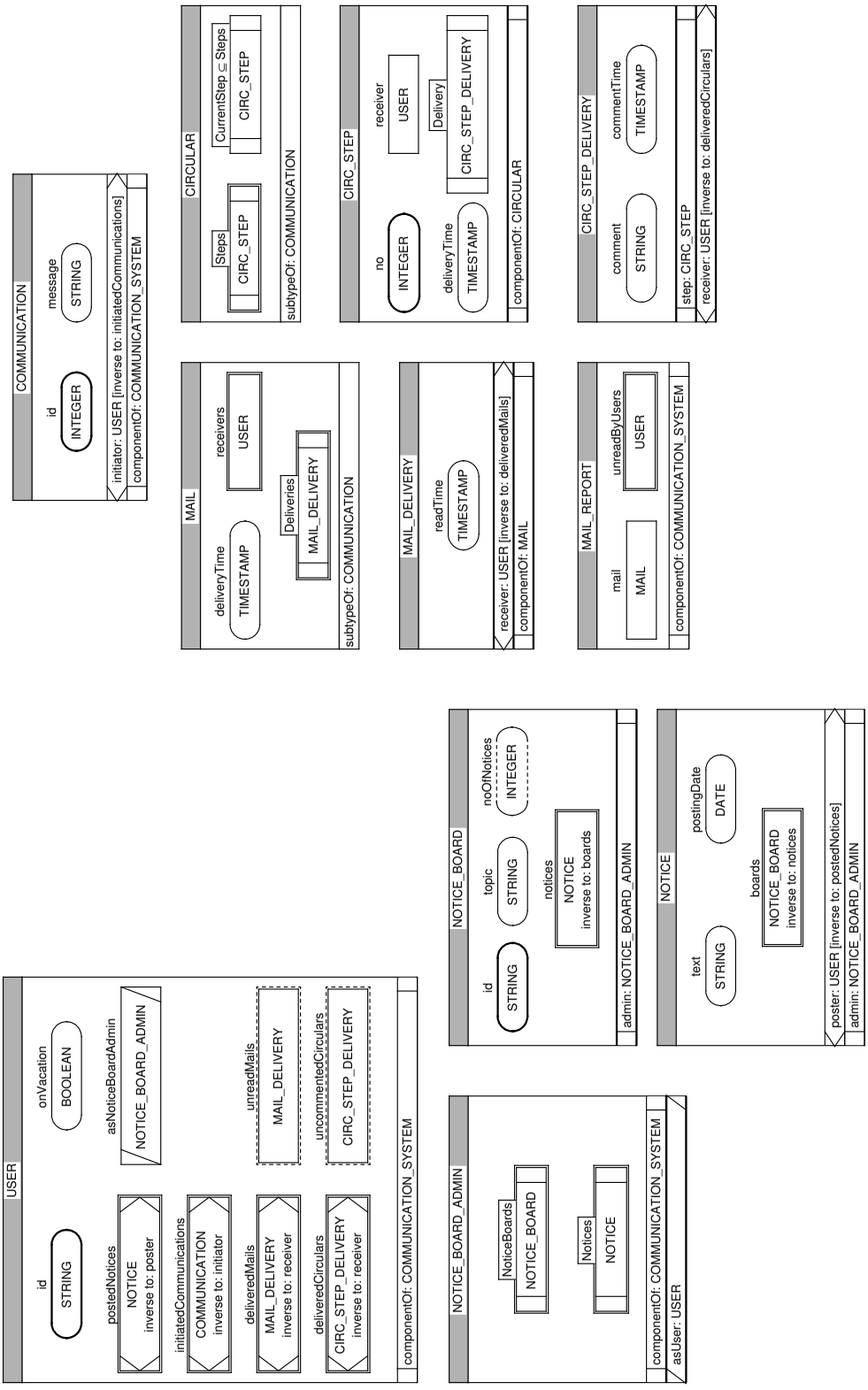


Figure 4.2: Object types representing users and communications

properties reflects the object-oriented concept of encapsulation. Second, nesting allows to use object-type diagrams to depict an object-centered view on instances of an object type as it is required in several diagrams (see below).

Single-valued properties may be defined to be *key properties*. Key properties uniquely identify an instance of an object type. They have to be initialized when an object is created and cannot be modified later on. In object-type diagrams, key properties are depicted by boldface borders.

The value of a property may be derived. In this case, the property is referred to as a *derived property*. A derived property has an associated derivation expression, specified by means of a query diagram (cf. Section 4.5). Within an object-type diagram, derived properties are graphically depicted by a dashed border.

Example: Consider object type NOTICE_BOARD in Figure 4.2. Object type NOTICE_BOARD defines attributes id, topic, and noOfNotices and relationships notices and admin. All attributes as well as relationship admin are single-valued, relationship notices is multi-valued. Attribute id and relationship admin represent key properties. Attribute noOfNotices is derived. For the moment ignore the special location and the special graphical representation of relationship admin.

4.2.2 Relationships

Relationships model references between instances of object types. In Object/Behavior Diagrams, relationships are directed. This is a major difference to conceptual design methods where relationships are bi-directional (e.g., associations in OMT [RBP+91]). In Object/Behavior Diagrams, bi-directional relationships are modeled by means of object types. The instances of an object type representing a bi-directional relationship are called *relationship objects*. As will be shown later in this section, a relationship object references the objects participating in the relationship via has-constituent relationships.

If update operations should maintain the correspondence between two directed relationships, the two relationships have to be declared to be *inverses*. In the case that relationship r_1 of object type O_1 and relationship r_2 of object type O_2 are declared to be inverses, O_2 must be the domain of r_1 and O_1 the domain of r_2. If r_1 and r_2 are inverse relationships and relationship r_1 of object o_1 (of type O_1) references o_2 (of type O_2), then the relationship r_2 of o_2 must reference o_1, and vice versa.

References between objects may have different semantics, so several types of relationships are provided in Object/Behavior Diagrams. Generally, the relationship types provided can be classified into two categories: object-defining relationships and object-describing relationships. *Object-defining relationships* model on which referenced objects a referencing object is existence dependent. Thus, object-defining relationships have to be initialized when creating an instance (*obligatory relationship*) and cannot be changed throughout the

lifetime of an instance (*invariant relationship*). *Object-describing* relationships are optional (i.e., they can be nil) and they can be changed during the lifetime of an instance. Object-describing relationships and object-defining relationships are distinguished graphically in that the former are depicted by rectangles in the body of an object-type diagram and the latter are depicted by bars placed at the bottom of an object-type diagram. Whereas object-defining relationships are always single-valued, object-describing relationships may be single-valued or multi-valued.

In the following, the semantics of the various types of relationships supported in Object/Behavior Diagrams is described.

General Relationship. A *general relationship* models a directed relationship between two independent objects. General relationships are object-describing. A general relationship is depicted by a rectangle. The name of the relationship is given above the rectangle, the name of the domain type within the rectangle.

Two general relationships may be declared to be inverses. In an object-type diagram, an inverse declaration between two general relationships is specified by annotating at least one of the relationships. A relationship is annotated by giving the keyword inverse to and the name of the general relationship to which the relationship is inverse within the rectangle representing the relationship.

Example: Consider object types NOTICE_BOARD and NOTICE in Figure 4.2. Every notice board refers to the notices posted to it by means of general relationship notices. Every notice refers to the notice boards to which it is posted by means of general relationship boards. These two general relationships are declared to be inverses. To increase readability, both relationships are annotated to be inverses.

Component-Of Relationship. A *component-of relationship* models a directed relationship of a component object to a superordinate composite object. A component object is an object that exclusively belongs and, therefore, is existence dependent on a single superordinate object. Consequently, there may be defined only one component-of relationship for a single object type. Component-of relationships are object-defining. In an object-type diagram, a bottom bar representing a component-of relationship is depicted with side borders. Since there may be defined only one component-of relationship for an object type, naming the relationship is optional. If the relationship remains unnamed, the superordinate object can be referenced by means of the predefined name componentOf.

Relationships inverse to component-of relationships are has-component relationships. A *has-component relationship* models a directed relationship of a composite object to one or many component objects. Has-component relationships are object-describing. There is an inverse has-component relationship for every component-of relationship, and vice versa. Since there may be defined only one component-of relationship for an object type,

and since there may be only one inverse of a relationship, the inverse of a component-of relationship is available implicitly and need not be indicated: The inverse of a component-of relationship defined at object type O_1 with domain type O_2 corresponds to that has-component relationship of the superordinate object type O_2 which has object type O_1 as domain type. Thus, seen from a superordinate object type, one object type may be used as the domain type of only one has-component relationship. Within an object-type diagram, a has-component relationship is depicted by a rectangular box with a tag placed above the box holding the name of the relationship.

Has-component relationships are also referred to as *local object classes* (cf. [KS92]). The term "local object class" gives emphasis to the fact that the objects referenced by a has-component relationship depend on and exclusively belong to the object of which the relationship is a property. The objects contained in a local object class are also called *members* of the local object class, and hence the object type serving as domain type of a local object class is also referred to as the class's *member type*. A local object class is maintained for each instance of the object type for which it is defined.

Example: Users, notice-board administrators, and communications are components of the communication system. Therefore, object types USER, NOTICE_BOARD_ADMIN, and COMMUNICATION define a component-of relationship with the global object type COMMUNICATION_SYSTEM as domain (cf. Figure 4.2). The global object type defines has-component relationships (local classes) for maintaining users, notice-board administrators, and communications (cf. Figure 4.1). For the moment ignore all other properties of the global object type. Notice boards and notices are components of a notice-board administrator. Object type NOTICE_BOARD and object type NOTICE name their component-of relationships explicitly as admin.

Local object classes of an object type may be organized in a class hierarchy, where the extension of a subclass (i.e., the set of its members) must be a subset of the extension of its superclass. Of course, the restriction that an object type can serve as member type of only one local object class does not apply to subclasses. A subclass may restrict its member type to be a subtype of the member type of its superclass (cf. Section 4.2.3). A subclass is depicted graphically as a local object class that shows the mathematical symbol for a subset relationship ($\subseteq$) and the name of its superclass in the name tag.

Example: Object type COMMUNICATION_SYSTEM defines a local class named Communications, which collects communications in the form of mails *and* circulars (cf. Figure 4.1). Additionally, the global object type defines two subclasses of local class Communications: subclass Mails, which contains the members of Communications that are mails, and subclass Circulars, which contains the members of Communications that are circulars. Both subclasses restrict their member type to the corresponding subtype of COMMUNICATION. (As will be explained later in this section, object type MAIL and object type CIRCULAR are subtypes of object type COMMUNICATION.)

In addition to has-component relationships, there are so-called has-indirect-component relationships. A *has-indirect-component relationship* models a directed relationship of a composite object to one or several indirect component objects. An object o is a *direct component* of some other object o' if o is related to o' via a component-of relationship. An object o is an *indirect component* of o', if there is another object o'', where o is related to o'' via a component-of relationship, and where o'' is a direct or indirect component of o'. Has-indirect-component relationships are object-describing. There is no explicit inverse relationship of a has-indirect-component relationship. Actually, the "inverse" of a has-indirect-component relationship corresponds to a cascade of component-of relationships starting at the object type of the indirect component objects and ending at the object type for which the has-indirect-component relationship is defined. A has-indirect-component relationship is depicted like a has-component relationship. But to distinguish a has-indirect-component relationship from a has-component relationship, the tag of a has-indirect-component relationship is accentuated by rounded top corners.

Example: Property AllNoticeBoardsInUse of the global object type COMMUNICATION_SYSTEM represents a has-indirect-component relationship (cf. Figure 4.1). Notice boards are components of notice-board administrators and, thus, indirect components of the global communication system. As will be shown later in this section, the derivation expression associated with property AllNoticeBoardsInUse selects all notice boards to which at least one notice is posted.

Role-Of Relationship. A *role-of relationship* models a directed relationship of an object to another object that represents the same real-world entity, but in a more general context. Role-of relationships utilize the concept of object specialization, in which a real-world entity is represented by an instance of a root object type and an arbitrary number of instances of role types. Every object type may have at most one role-of relationship. Role-of relationships are object-defining. In an object-type diagram, a bottom bar representing a role-of relationship is depicted with an inner structure in the shape of a parallelogram. Since there may be defined only one role-of relationship for an object type, naming this relationship is optional. If the relationship remains unnamed, a role object's representation in a more general context can be referenced by means of the predefined name roleOf.

Relationships inverse to role-of relationships are has-role relationships. A *has-role relationship* models a directed relationship of an object to one or more role objects each representing the same real-world entity, but in a specific context. Has-role relationships are object-describing. There is an inverse has-role relationship for every role-of relationship, and vice versa. Since there may be defined only one role-of relationship for an object type, and since there may be only one inverse of a relationship, the inverse of a role-of relationship is available implicitly and does not have to be indicated. The inverse of a role-of relationship defined at object type O_1 with domain type O_2 corresponds to that has-role relationship of object type O_2 which has object type O_1 as domain type. Thus,

one object type may be used as the domain type of only one has-role relationship of any object type. A has-role relationship is depicted like a general relationship, but with an inner structure in the shape of a parallelogram.

Example: Users may play the role of notice-board administrators. Consider object type USER and object type NOTICE_BOARD_ADMIN in Figure 4.2: Object type USER defines a has-role relationship named asNoticeBoardAdmin with object type NOTICE_BOARD_ADMIN as domain. The inverse to this relationship is role-of relationship asUser of object type NOTICE_BOARD_ADMIN.

Has-Constituent Relationship. A *has-constituent relationship* models a directed relationship of an association object to one of its constituent objects. An association object represents a relationship between two or more objects. The objects participating in the association are referred to as constituents, as they taken together "constitute" the association. Has-constituent relationships are object-defining. A has-constituent relationship is depicted by a bar with a hexagonal inner structure showing the name of the relationship and the name of the domain type.

Relationships inverse to has-constituent relationships are constituent-of relationships. A *constituent-of relationship* models a directed relationship of an object to one or more association objects of a specific type. Constituent-of relationships are object-describing. They are depicted like a general relationship but with a hexagonal inner structure.

There is an inverse constituent-of relationship for every has-constituent relationship, and vice versa. Inverse declarations are obligatory in the case of pairs of has-constituent and constituent-of relationships. Inverses are always indicated by annotating both sides of a pair.

Besides a relationship from an association object to one of its constituents, also another kind of relationship is modeled by a has-constituent relationship. A has-constituent relationship is used for modeling a relationship that is obligatory and invariant but that is not (a) a component-of relationship, (b) a role-of relationship, or (c) a relationship of an association object to one of its constituents. This semantics applies especially to object references that are part of a key. Like an object participating in an association, an object that is referenced via such an obligatory and invariant relationship is referred to as a constituent, and an inverse constituent-of relationship is defined for it.

Example: Object types that represent associations have not been used in modeling the communications system and, therefore, no has-constituent relationships have been introduced for modeling the relationship of an association object to one of its constituents. However, has-constituent relationships and constituent-of relationships have been used extensively for modeling relationships that are obligatory and invariant. Consider for example object types USER and NOTICE in Figure 4.2. When a

notice is created, the user posting the notice has to be known. The poster of a notice cannot be changed during the lifetime of a notice. Every notice refers to its poster by means of the has-constituent relationship poster, and every user refers to the notices that he/she has posted by means of the constituent-of relationship postedNotices.

4.2.3 Type Hierarchies

Object types may be organized in type hierarchies. Each object is a *direct instance* of exactly one object type and an *indirect instance* of all predecessors of this object type in the type hierarchy. A subtype inherits the property definitions, the local referential integrity constraints (cf. Section 4.3), and the behavior (cf. Section 4.4) of its supertype. A property definition may be specialized at a subtype in that the domain is redefined to some subtype of the domain at the supertype. According to the classification of inheritance concepts presented in [KS96b], the inheritance concept of Object/Behavior Diagrams provides a "covariant subtypeOf" relationship between a subtype and its supertype. Type specialization is denoted by giving the name of the supertype in a bar labeled subtypeOf, located at the bottom of the object-type diagram representing the subtype. Inherited property definitions may, but need not, be depicted with object-type diagrams of subtypes. Usually, an inherited property is depicted only if its domain is restricted.

Example: Object type MAIL and object type CIRCULAR are defined to be subtypes of object type COMMUNICATION (cf. Figure 4.2). MAIL and CIRCULAR inherit from COMMUNICATION that they are components of the global object type, that they are initiated by a user, that they are uniquely identified by an id, and that they convey a message. Both, MAIL and CIRCULAR do not specialize the domain of the inherited properties but only add new properties.

4.3 Object-Class Diagrams

By indicating domain types for relationships, Object/Behavior Diagrams incorporate referential integrity constraints. Another important kind of integrity constraints, *local referential integrity constraints*, cannot be expressed in Object/Behavior Diagrams as they have been introduced so far. Local referential integrity extends the concept of referential integrity for composite objects with dependent and exclusive components.

Informally, a local referential integrity constraint requires two objects taking part in a directed relationship to be direct or indirect components of the same composite object.

Example: A notice-board administrator maintains its notice boards and its notices as dependent and exclusive components. Each notice references the notice boards to which

it is posted. A local referential integrity constraint may require the referencing notice and the referenced notice boards to belong to the same notice-board administrator.

The specification of a local referential integrity constraint requires a construct by means of which objects that are direct or indirect components of a composite object can be identified. Object/Behavior Diagrams provide locality paths for that purpose. A *locality path* is of the form $O.c_1[O_1].c_2[O_2].\cdots.c_n[O_n]$: The path starts at an object type O and navigates along an arbitrary number of local object classes c_i. Each local object class c_i may be restricted to members that are direct or indirect instances of type O_i. Local object class c_1 has to be a local object class defined for O. Every further local object class $c_i (i \geq 2)$ has to be a local object class defined for the member type of c_{i-1} or, if c_{i-1} is restricted, for object type O_{i-1}. The local object class at which a locality path ends, c_n, is called the path's *leaf class*.

A locality path P starting at an object type O applies to every instance o of O. $P(o)$ represents the set of leaf-class objects that are obtained when P is evaluated for o. A locality path is evaluated for an object o by traversing the local object classes $c_1.c_2.\cdots.c_n$ starting at o. The semantics of traversing a class c_i depends on whether c_i is a leaf class or a non-leaf class. In the case that class c_i is a non-leaf class, traversing c_i returns the union of the results of traversing c_{i+1} at every member of c_i or, if c_i is restricted, at those members of c_i that are direct or indirect instances of O_i. In the case that class c_i is the leaf class, traversing class c_i returns the members of c_i or, if c_i is restricted, those members of c_i that are direct or indirect instances of O_i.

The definition of a local referential integrity constraint that applies to the instances of an object type O consists of a locality path P_{source} starting at O, a relationship r, and another locality path P_{target} also starting at O. Locality path P_{source} identifies the set of direct or indirect components of instances of O for which the domain of relationship r is constrained. Relationship r has to be defined for the type of the objects identified by P_{source}. It may be a general relationship, a role-of relationship, a has-role relationship, a has-constituent relationship, or a constituent-of relationship. Locality path P_{target} identifies the set of direct or indirect components of instances of O serving as domain for relationship r of the objects identified by P_{source}. The type of objects in P_{target} has to correspond to the domain type of relationship r or to a subtype of it. A local referential integrity constraint defined with an object type O is applied to every instance o of O. For every object o, the domain of relationship r of objects in $P_{source}(o)$ is restricted to $P_{target}(o)$.

Extending the original definition of Object/Behavior Diagrams, *object-class diagrams* are introduced to model local referential integrity constraints. The term "object-class diagram" emphasizes that at its top level, an object-class diagram shows the local object classes of the object type with which the object-class diagram is associated.

A locality path is depicted by nesting local object classes: In a nested representation, the member type of a local object class C is expanded within the box representing local object class C. In the expanded representation, all local object classes defined for the member

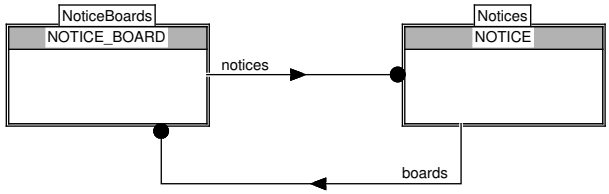


Figure 4.3: Object-class diagram of object type NOTICE_BOARD_ADMIN

type are shown. A nested representation of local object classes may be of any depth. To restrict a path element to objects of a specific type, the type hierarchy spanned by the path element's member type can be expanded within a local class. In this case, nesting of local object classes can be restricted to selected subtypes. If several locality paths navigate along the same local object classes, the common parts are depicted only once.

A local referential integrity constraint is depicted by a line connecting the goal classes of the constraint's source path and target path. The connecting line symbolizes the constrained relationship. The name of the relationship and a direction indicator are associated with the line. To show the multiplicity of a constrained relationship as specified in the type diagram, multiplicity indicators are associated with the connecting line. The notation for multiplicity indicators, a filled bullet for "zero to many", an unfilled bullet for "zero or one", and no bullet for "exactly one" are borrowed from OMT [RBP+91].

Example: Figure 4.3 shows the object-class diagram associated with object type NOTICE_BOARD_ADMIN. The object-class diagram shows the two local classes defined by object type NOTICE_BOARD_ADMIN, namely NoticeBoards with member type NOTICE_BOARD and Notices with member type NOTICE. A notice board contained in NoticeBoards is restricted to hold only such notices that are components of the notice-board administrator to which the notice board belongs. In the other direction, a notice contained in Notices is restricted to be posted only to such notice boards that are components of the notice-board administrator to which the notice belongs.

4.4 Behavior Diagrams

Behavior diagrams rely on Petri nets and show the passive behavior of instances of an object type by states and activities, corresponding to places and transitions of a Petri net. Each state represents a particular period, each activity an update operation in the life cycle of an object type's instances. Unlike Petri nets, where a transition is fired automatically if a prestate contains a token, an activity in a behavior diagram must be explicitly invoked for an object that is in every prestate of the activity.

As already mentioned in Section 4.2.3, a subtype inherits the behavior specified for its supertype. A subtype may specialize the inherited behavior in two ways: by refinement

and by extension. Refinement means expanding inherited activities and states into subdiagrams. Extension means adding activities and states. Refinement of behavior diagrams has been treated in [Sch90], different approaches for a consistent extension of object behavior have been presented in [SS95, SS97]. Specialization of object behavior by means of refinement and extension are not discussed in this fundamental chapter on Object/Behavior Diagrams. For details on the specialization of object behavior, see [Lan97].

For readability, the behavior diagram of an object type is split up into three parts: a life-cycle diagram, several activity-specification diagrams, and several activity-realization diagrams. Activity-specification diagrams and activity-realization diagrams are specified for activities that are defined in the life-cycle diagram of an object type and also for compound activities. A compound activity represents a means to aggregate the basic behavior of objects into higher-level behavioral units.

4.4.1 Life-Cycle Diagrams

The *life-cycle diagram* identifies update operations and their possible execution sequences. It depicts for an object type its states, its activities, and their connecting arcs. States represent necessary pre- and postconditions of activities.

In a life-cycle diagram, a state is depicted by a rectangle with a trapezoid at its top and bottom. The top trapezoid contains the name of the object type for which the state is specified, the bottom trapezoid contains the name of the state. An activity is depicted by a named rectangle, which is connected by a set of arcs to its pre- and poststates. The names of states and activities have to be unique within an object type.

For every object type, at least one creation activity has to be defined. A *creation activity* is an activity with no prestates that generates a new instance of an object type. The life cycle of an object can be terminated by means of activities for which no poststates are specified. Such (optional) activities are referred to as *removal activities*. Activities that are neither creation nor removal activities are *inner activities*. An inner activity defines prestates as well as poststates.

Often, updates to several objects have to be made within a single activity. Typical operations that involve several objects are the maintenance of inverse relationships and the removal of objects. Regarding inverse relationships and object removal, Object/Behavior Diagrams follow the principle of *object autonomy* (cf. [KS96b]). Object autnonomy means that inverse relationships are not maintained automatically and that objects are not removed automatically according to existence dependences. Rather, the inverse relationships of an object can be updated and an object can be removed only be means of activities provided by the object's type. This respects the principles of encapsulation and information hiding: An activity directly accesses only the object on which it is performed and updates all other participating objects only through activities provided by their types. Thus, a subset of the activities defined by an object type are activities that do not belong to the public

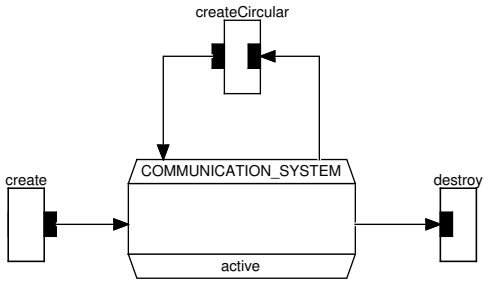


Figure 4.4: Life-cycle diagram of object type COMMUNICATION_SYSTEM (selection)

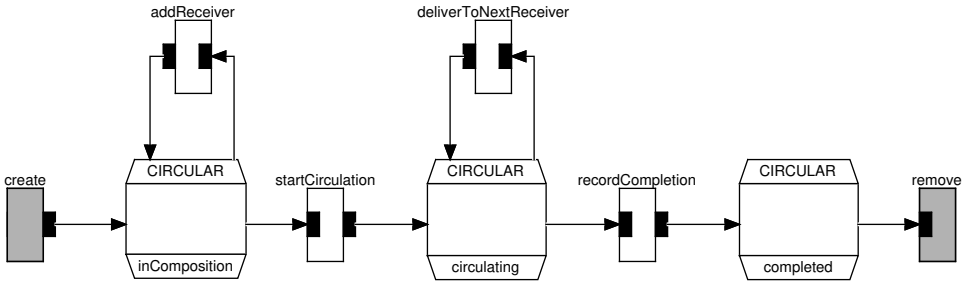


Figure 4.5: Life-cycle diagram of object type CIRCULAR

interface but that only contribute to other activities. Object/Behavior Diagrams provide the concept of activities with restricted visibility to model such activities. An *activity with restricted visibility* (also referred to as a *restrictedly public activity*) is an activity that can be invoked only by a single other activity. In the life cycle of an object type, a restrictedly public activity is shaded. The specification of a restrictedly public activity is characterized by an additional input port (cf. Section 4.4.2).

Example: Figure 4.4 shows a selection of the life-cycle diagram of object type COMMUNICATION_SYSTEM. After a communication system is created (creation activity create), the communication system is in state active. In this state, all the activities for creating and removing users, different kinds of communications, and notice-board administrators can be performed. Only inner activity createCircular is shown in the life-cycle diagram since this activity is further elaborated later in this section. Removal activity destroy terminates the existence of a communication system. All activities of object type COMMUNICATION_SYSTEM are part of the public interface. Figure 4.5 shows the life-cycle diagram of object type CIRCULAR. After performing activity create, a circular is in state inComposition. In this state, receivers can be added by repeatedly performing activity addReceiver. With activity startCirculation, the circular is delivered to its first receiver. Activity deliverToNextReceiver delivers a circular to the next receiver. After a circular has been delivered to all receivers, completion of the circulation is recorded by means of activity recordCompletion. Activity remove terminates the life cycle of a circular that is not further used. Creation activity create and removal activity remove do not belong to the public interface of a circular, they

are restrictedly public. As will be shown later in this section, activity create can be invoked only within activity createCircular of object type COMMUNICATION_SYSTEM. Activity remove can be invoked only within activity removeCircular of object type COMMUNICATION_SYSTEM (not shown).

4.4.2 Activity-Specification Diagrams

An *activity-specification diagram* specifies the formal input and output parameters of an activity. An activity-specification diagram is depicted by a named rectangle representing each formal parameter by a named port. Input ports are drawn as squares aligned to the left inner border of the rectangle, output ports are aligned to the right outer border. A port which is both, an input port and an output port, is drawn twice, each time carrying the same name. Single-valued ports are depicted by squares with single-line borders, multi-valued ports by rectangles with double-line borders. Every port has a domain. A *data port* has a data type as domain, an *object port* has an object type restricted to some states as domain.

An input object port may be connected to a prestate by a transfer arc or by a read arc. A *transfer arc* indicates that the object assigned to the input port has to reside in the connected state before activity execution but must not reside therein after activity execution unless the port is also an output port and the state is contained in the port's poststates. Like arcs connecting an output port to its domain or a data input port to its data type, transfer arcs are depicted by solid lines. A *read arc* indicates that the object assigned to the input port has to reside in the connected state before activity execution and still has to reside therein during and after activity execution. Read arcs are depicted by dashed lines.

Every activity has a distinguished single-valued port, the *primary port*. The primary port—depicted in black—identifies the object type to which the activity belongs. The primary port is an output port in the case of a creation activity, an input and an output port in the case of an inner activity, and an input port in the case of a removal activity. The poststates of a primary port and the prestates of a primary port that are connected via transfer arcs correspond to these states specified in the life-cycle diagram. Additional prestates of a primary port may be connected to the primary port only via read arcs.

A formal return value of an operation is represented by a *return port*. A return port is depicted by a semicircle. An arbitrary output port may serve as an activity's return port.

Example: Figure 4.6 shows the activity-specification diagram of activity createMail of object type COMMUNICATION_SYSTEM. Since activity createMail is an inner activity, its primary port system is both, an input port and an output port. Input and output port initiator requires the initiator of the circular, to reside in state created before and after activity execution. The message of the new circular is assigned to data port

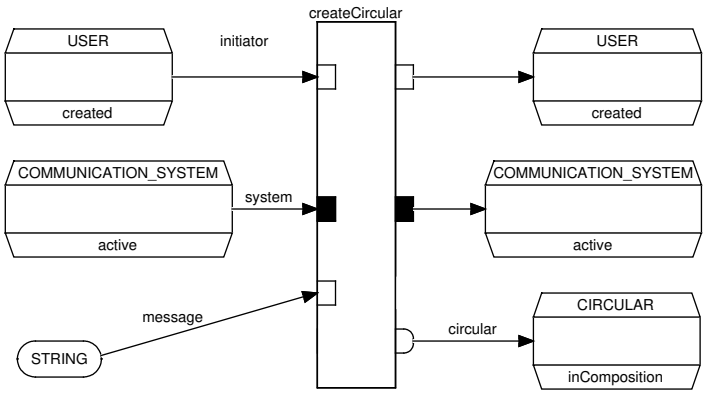


Figure 4.6: Specification of activity createCircular

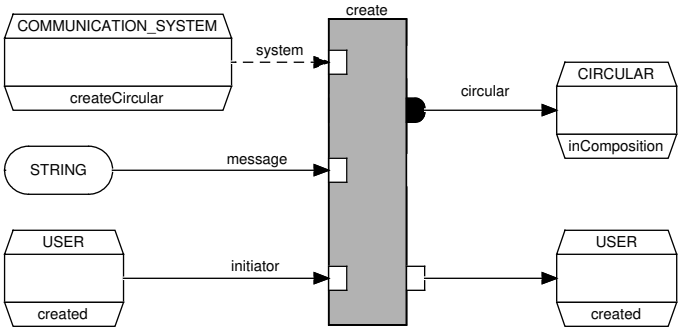


Figure 4.7: Specification of activity create

message. The formal return value of activity createCircular is represented by return port circular. A created and returned circular has to reside in state inComposition.

In Section 4.4.1, restrictedly public activities have been introduced. A restrictedly public activity a that is visible only to a' is modeled by means of an additional input port. The domain of this port is the object type O at which activity a' is defined. The prestate of the port is the activity state of a'. Restricting the port to the activity state ensures that activity a can be invoked only from objects of type O on which activity a' is being executed. Activity states are defined implicitly and are not indicated in the life cycle. An activity state is named after the activity it represents.

Example: Figure 4.7 shows the activity-specification diagram of activity create of object type CIRCULAR. Activity create is restrictedly public. It can be performed only as part of activity createCircular of object type COMMUNICATION_SYSTEM. As has been shown in Figure 4.6, a communication system on which activity createCircular is performed has to reside in state active before and after activity execution. During activity execution, however, the communication system resides in the activity state createCircular. To ensure that activity create can be performed only as part of createCircular,

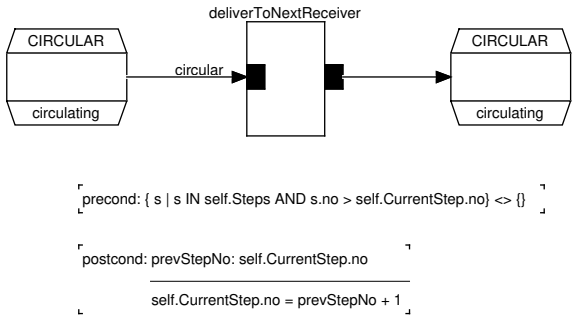


Figure 4.8: Specification of activity deliverToNextReceiver

input port system is introduced and the prestate of port system is set to the activity state createCircular of object type COMMUNICATION_SYSTEM.

A *precondition* and a *postcondition* may be specified with every activity. A precondition represents a condition on the objects and values assigned to the input ports of an activity. An activity cannot be executed unless its precondition is satisfied. A postcondition expresses a transitional constraint. The postcondition can refer to both, the database as it was before activity execution and the database as it is after activity execution. If the postcondition of an activity is violated, an inconsistency between the implementation of an activity (specified procedurally) and the transitional constraint (specified declaratively) becomes apparant. Pre- and postconditions are boolean expressions that are stated textually. Literals, functions, navigation paths, and placeholders for query diagrams as they will be introduced in Section 4.5 can be used to compose a pre- or postcondition. In the postcondition of an activity, it may be necessary to refer to an object as it has been before activity execution. Since it is impossible to maintain a copy of the whole database, the designer has to specify the objects of which the state before activity execution is accessed. This specification is given in the preamble of a postcondition by defining placeholders and by associating expressions with them. Before an activity is executed, the expressions in the preamble are evaluated and the results are bound to the respective placeholders. The placeholders can be used in an expression representing a postcondition to refer to old states.

Example: Figure 4.8 shows the activity-specification diagram of activity deliverToNextReceiver of object type CIRCULAR. Activity deliverToNextReceiver can be performed on a circular only if the circular has not already been delivered to all receivers, in other words, if there is a circulation step with a higher number than the current step (i.e., the step referenced in property CurrentStep). Activity deliverToNextReceiver is executed correctly if the current step after activity execution is the successor of the current step before activity execution. Therefore, the number of the current step before activity execution is bound to placeholder prevStepNo in the preamble of the postcondition. The postcondition itself checks whether the number of the current step after activity execution corresponds to prevStepNo + 1.

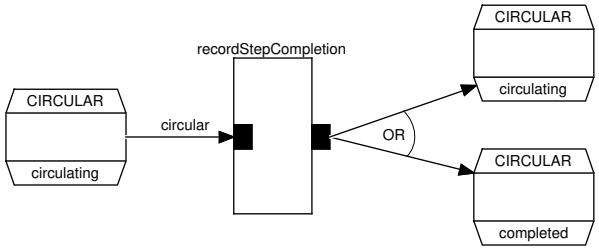


Figure 4.9: Specification of compound activity recordStepCompletion

4.4.3 Activity-Realization Diagrams

Activity-realization diagrams represent activity implementations. To specify the control flow of activities, activity-realization diagrams follow structured diagrams. Activity-realization diagrams are not discussed in this fundamental chapter on Object/Behavior Diagrams since they are not considered throughout this thesis. The interested reader is referred to [KS96b].

4.4.4 Compound Activities

Compound activities represent a means for behavior composition. The idea of *behavior composition* is that the same basic activities—specified in the life-cycle of an object type—can be used in several higher-level behavioral units. A compound activity is built from basic activities and other compound activities of one or several object types by specifying a particular order of execution on them. Like a basic activity, a compound activity belongs to exactly one object type. Compound activities do not augment the basic behavior of an object type and are not shown in the life-cycle diagram.

Like a basic activity, a compound activity is defined by an activity-specification diagram and is implemented by an activity-realization diagram. Generally, the specification of a compound activity does not differ from the specification of a basic activity. The only difference is that objects entering input ports and leaving output ports of a compound activity may belong to alternative states. By definition, several prestates of an input port and several poststates of an output port are interpreted as conjunctive condition. Thus, a way is needed to specify alternative pre- or poststates: In the specification of a compound activity, disjunctive states can be specified by connecting the alternative states by a solid line and by annotating the connection with the term "or". Like for basic activities, activity-realization diagrams are not considered for compound activities.

Example: Figure 4.9 shows the activity-specification diagram of compound activity recordStepCompletion of object type CIRCULAR. When activity recordStepCompletion is performed on a circular upon completion of a circulation step, the actual action depends on whether the circular has already been delivered to all receivers. If the circular has

already been delivered to all receivers, the circulation is terminated by performing activity recordCompletion. If the circular has not been delivered to all receivers, the circular is delivered to the next receiver by performing activity deliverToNextReceiver. Depending on the activity invoked, the circular resides in state circulating or in state completed after execution of the compound activity (cf. Figure 4.5.

4.5 Query Diagrams

Query diagrams represent a graphical language to specify derivation expressions for derived properties. A query diagram is evaluated for every instance of the object type for which the property derived by the query diagram is defined. Depending on its result, a query diagram may be *selecting* or *generating*. In the first case, a query diagram selects a single value, a single existing object, a set of values, or a set of existing objects from the database. In the second case, new instances of an object type are generated. These two approaches correspond to the object-preserving and the object-generating semantics of object-oriented database views.

As already mentioned above, a dashed property border indicates that the property is derived. Some restrictions apply to the specification of derived properties: Attributes and general relationships may be derived only by a selecting query diagram. Key properties and object-defining relationships cannot be derived since they must not be changed throughout the lifetime of an instance. Relationships that are inverses to object-defining relationships also cannot be derived since updating them would require updating object-defining relationships. The only exception is a has-component relationship: Components of an object may be derived by associating a generating query diagram with a local object class (i.e., with a has-component relationship). In such a case, the associated generating query diagram populates the local object class. The component-of relationship of the generated object is set to the object for which the generating query diagram is evaluated. Subclasses of local object classes as well as has-indirect-component relationships may be derived by selecting query diagrams.

Several query diagrams may be associated with a single derived property with the semantics that the value of the derived property corresponds to the union of the query results.

The following discussion on query diagrams is organized as follows: In Section 4.5.1, the elementary structure of query diagrams is explained. The different kinds of expressions that can be used in query diagrams are enumerated in Section 4.5.2. The parts of a query diagram, which are the domain box, the condition box, and the result box are described in detail in Sections 4.5.3 to 4.5.5. The discussion on query diagrams is concluded by presenting advanced concepts—query diagrams with alternative conditions and nested query diagrams—in Section 4.5.6.

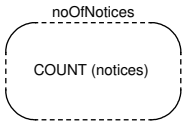


Figure 4.10: Query diagram for property noOfNotices

4.5.1 Elementary Structure

The conceptual structure of a query diagram resembles an OQL query [CB97]. The parts of a query diagram, which are the result box, the domain box, and the condition box, correspond to the SELECT-clause, the FROM-clause, and the WHERE-clause of an OQL query, respectively. The evaluation of a query diagram is similar to OQL, too.

A primitive derivation expression may perform computations without ranging over sets of objects. A query diagram describing such a primitive derivation expression consists of a result box only. A *result box* specifies the result of a derivation and is depicted like a property in an object-type diagram. The property that should by derived is shown as a box with a dashed border and with the property's name placed above. The property's type inside the box is replaced by an expression. A primitive query diagram is evaluated in that the expression specified in the result box is computed.

Example: Figure 4.10 shows the query diagram for property noOfNotices of object type NOTICE_BOARD. The primitive query diagram, which consists only of a result box, computes the number of notices posted to a notice board.

More complex derivation expressions may iterate over sets of objects and may impose conditions on these objects. Iterating over sets of objects is specified by defining variables in the *domain box* of a query diagram. The set of objects or values over which a variable iterates is specified by an expression. A domain box is depicted by a rectangle that is located below the result box on the left side.

Conditions over objects bound to variables are specified in the *condition box* showing these variables expanded. We call these variables *expanded variables*. Expanded variables are depicted by a rectangular body with a trapezoid at the top and at the bottom (cf. states in life-cycle diagrams and activity-specification diagrams). The top trapezoid contains the name of the variable together with its type. The rectangular body shows the properties defined with the variables's type. No domain types are shown within the property boxes. Conditions on values of attributes and on participation of objects in relationships are associated with the attributes and relationships shown in the rectangular body. Usually, only those attributes and relationships are depicted on which conditions are specified. The bottom trapezoid allows to specify conditions on the object state.

Informally, a query diagram is evaluated as follows (a more detailed description will be given later): The query diagram is evaluated by an iteration over sets of objects as specified

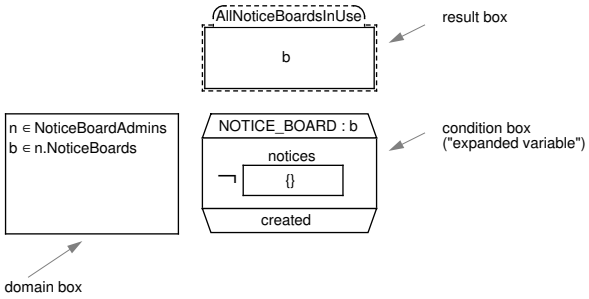


Figure 4.11: Query diagram for property AllNoticeBoardsInUse

in the diagram's domain box. For each binding to the variables, each condition in the condition box is checked, and bindings that do not meet all conditions are discarded. The expression specified in the result box of a query diagram is computed either for each remaining binding or—in the case of an aggregate function—for the remaining bindings together. In an expression associated with the result box of a query diagram, the set of bindings satisfying all conditions can be referred to by the special symbol *.

Example: Figure 4.11 shows the query diagram for property AllNoticeBoardsInUse of the global object type COMMUNICATION_SYSTEM. In the domain box of the query diagram, two variables are defined: Variable n iterates over local class NoticeBoardAdmins, variable b iterates over local class NoticeBoards of the notice-board administrators. The condition box of the query diagram consists of expanded variable b. The conditions imposed on a notice board represented by b require that the notice board is in state created and that notices are posted to it. All notice boards that satisfy these conditions are selected by the result box.

The specification of query diagrams is not allowed to be recursive. If derived property p of object type O is used directly or indirectly in the derivation of derived property p' of object type O', then, property p' is not allowed to be used directly or indirectly in the derivation of property p. Property p is used directly in the derivation of property p' if p is referenced in a query diagram for p'. Property p is used indirectly in the derivation of property p' if p is used directly in the derivation of another derived property p'' of object type O'', which is used directly or indirectly in the derivation of property p'.

4.5.2 Expressions

A query diagram is built from typed objects and typed data values by using them in expressions. Expressions may appear in the domain box, in the condition box, as well as in the result box of a query diagram. Expressions that can be used in query diagrams are literals, functions, navigation paths, and other query diagrams:

Literals. A *literal* is an expression whose value is the literal itself. Scalar literals are available for every data type, i.e., for INTEGER (e.g., 5), REAL (e.g., 10.25), STRING (e.g., 'University of Linz'), BOOLEAN (e.g., true), DATE (e.g., '1996/12/03'), TIME (e.g., '11:45:00'), and DATETIME (e.g., '1996/12/03 11:45:00'). Set literals can be constructed by giving scalar literals of the same type (separated by commas) within braces (e.g., {'Vienna', 'Salzburg', 'Linz'}).

Functions. A *function* is an expression whose value is the result of applying the function to its operands. Functions that can be used in query diagrams are boolean functions ($\neg$, $\vee$, $\wedge$), unary arithmetic functions ($-$, ABS), binary arithmetic functions ($+$, $-$, /, *, MOD, DIV), set functions ($\cap$, $\cup$, $\setminus$), aggregate functions (COUNT, SUM, MIN, MAX, AVG), comparison predicates ($<$, $\leq$, $=$, $==$, $\geq$, $>$, $\in$, $\subset$, $\subseteq$, $\supset$, $\supseteq$), state functions (STATES, IS_IN_STATE), date/time-manipulation functions (DATE_OF_TIMESTAMP, TIME_OF_TIMESTAMP, YEAR_OF_DATE, MONTH_OF_DATE, DAY_OF_DATE, WEEKDAY_OF_DATE, HOUR_OF_TIME, MINUTE_OF_TIME, SECOND_OF_TIME), and date/time-construction functions (DAY, TIME, TIMESTAMP, CURRDATE, CURRTIME), with their intuitive semantics.

Navigation Paths. A *navigation path* starts at an object, traverses an arbitrary number of relationships, and ends at a target property. Navigation is indicated by using a dot-notation. The object for which a query diagram is evaluated can be referred to as self. If a navigation path starts at self, the keyword self can be omitted. So, a primitive navigation path can be specified by solely giving the name of a property defined for the type of self. A navigation path is an expression whose value is a single value/object if all the traversed relationships and the target property are single-valued. A navigation path is an expression whose value is a set of values/objects if at least one of the traversed relationships or the target property is multi-valued. If a navigation path traverses a multi-valued relationship, navigation iterates over the multi-valued relationship and computes the union of the target-property values selected by each iteration step.

Query Diagrams. A query diagram itself represents an expression, too. The value of a query diagram is its result. Nesting of query diagrams will be discussed as an advanced concept in Section 4.5.6.

4.5.3 Domain Box

Within the domain box of a query diagram, variables are defined. A variable can be an iterator variable or a reference variable: An *iterator variable* is a variable iterating over a set of values/objects (iteration set) specified by an expression. Iteration is indicated by means of the element symbol ($\in$). A *reference variable* may be defined as shorthand to

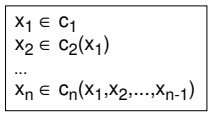


Figure 4.12: Skeleton of a domain box

```
FORALL x1 IN c1
    FORALL x2 IN c2(x1)
      ⋮
        FORALL xn IN cn(x1, x2, ..., xn−1)
            tuple(x1, x2, ..., xn−1)
        END
      ⋮
    END
END
```

Figure 4.13: Tuple creation

avoid repeating long expressions several times within a query diagram. Reference variables, which are defined by means of the assignment symbol (:=), may be single-valued or multi-valued.

A variable can be used at a given location in a query diagram if it is *visible* there. A variable is visible within the condition box or result box of a query diagram if it is defined in the diagram's domain box. A variable is visible within the domain box of a query diagram below its definition. If an iterator variable is used in the domain box below its definition to specify the iteration set of another iterator variable, iteration is nested.

Figure 4.12 shows the skeleton of a query diagram's domain box. In the domain box, variables $x_1, x_2, ..., x_n$ are specified to iterate over collections obtained by expressions $c_1, c_2, ..., c_n$. In an expression c_i, variables $x_1, x_2, ..., x_{i-1}$, which are defined above c_i, can be used. When a query diagram is evaluated, the first evaluation step is the creation of a tuple for each possible binding to the iterator variables. Figure 4.13 shows how the tuples are created by a nested iteration over $c_1, c_2, ..., c_n$. Constructor tuple is assumed to provide the required functionality for composing a tuple. In the presented skeleton of a domain box, only iterator variables are defined. Reference variables can be incorporated into the query evaluation easily. A reference variable does not open a nested iteration loop. Rather, the expression associated with a reference variable is evaluated, and the result is assigned to the reference variable within that loop where all variables used in the expression are bound.

Example: Consider the query diagram for property AllNoticeBoardsInUse presented in Figure 4.11. In the domain box of the query diagram, two iterator variables are defined. When the query diagram is evaluated, variable n iterates over local class NoticeBoar-

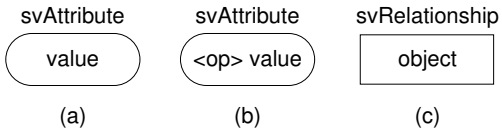


Figure 4.14: Scalar-comparison predicates

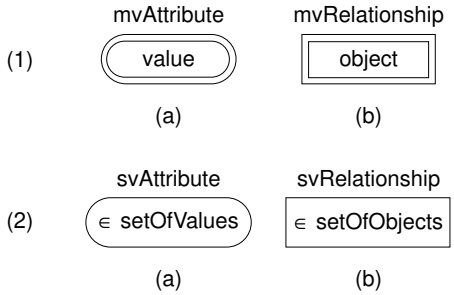


Figure 4.15: Scalar/set-comparison predicates

dAdmins and, in a nested loop, variable b iterates over local class NoticeBoards of the notice-board administrator currently bound to variable n.

4.5.4 Condition Box

Since conditions on attribute values and on the participation of objects in relationships are typically used in the condition box of query diagrams, such conditions should be easy to recognize and easy to specify. The graphical representation of comparison predicates, the graphical navigation along relationships, and a shorthand notation for defining variables provide for an intuitive specification of conditions in the condition box of a query diagram:

Comparison Predicates. In query diagrams, conditions on attribute values and on the participation of objects in relationships can be specified graphically as described below. Comparison predicates can be classified into three groups: (1) comparison predicates between scalars, (2) comparison predicates between a scalar and a set, and (3) comparison predicates between sets. Generally, comparison predicates are associated with object properties. A comparison predicate can be negated by placing symbol ¬ to the left of the property.

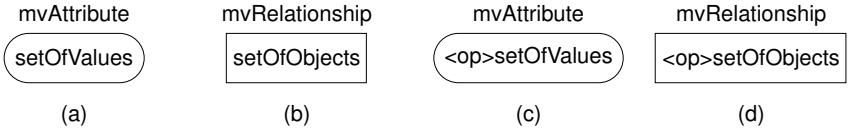


Figure 4.16: Set-comparison predicates

1. *Scalar-comparison predicates:* A scalar-comparison predicate is a comparison predicate between a single-valued property (i.e., a single-valued attribute or a single-valued relationship) and a single value/object. Figure 4.14 shows how scalar-comparison predicates are depicted within the expanded representation of an object. Figure 4.14 (a) requires svAttribute to be equal to value. Besides equality, diverse other comparison operators can be applied by prefixing expression value with a comparison symbol (cf. Figure 4.14 (b)). The set of operators that are applicable depends on the domain of svAttribute. Figure 4.14 (c) requires svRelationship to be identical to object. Two objects are identical if they have the same object identifier.

2. *Scalar/Set-comparison predicates:* Scalar/set-comparison predicates can be classified into two groups:

   (a) Predicates that require a multi-valued property (i.e., multi-valued attribute or multi-valued relationship) to *contain* a single value/object: Part (1) of Figure 4.15 presents predicate *contains* for (a) a multi-valued attribute and for (b) a multi-valued relationship.

   (b) Predicates that require a single-valued property to be *contained in* a set of values/objects: Part (2) of Figure 4.15 presents predicate *contained-in* for (a) a single-valued attribute and for (b) a single-valued relationship. To distinguish predicate *contained-in* from scalar-comparison predicates, the is-element-in symbol ($\in$) has to be indicated in the former case.

3. *Set-comparison predicates:* A set-comparison predicate is a comparison predicate between a multi-valued property and a set of values/objects. Figure 4.16 shows how set-comparison predicates are depicted within the expanded representation of an object. To distinguish set-comparison predicates from predicate *contains*, multi-valued properties are depicted without double-line borders in set-comparison predicates. Parts (a) and (b) of Figure 4.16 show set-equality predicates. Two sets of values are equal if they contain the same values. Two sets of objects are equal if they contain identical objects. Note: Sets do not have an identity in Object/Behavior Diagrams and, therefore, identity of sets cannot be compared. Parts (c) and (d) of Figure 4.16 show how set-comparison operators other than equality can be applied. The operators provided are $\subset$, $\subseteq$, $\supset$, and $\supseteq$.

If a condition refers to property values of different objects, the question arises with which object the condition should be associated. To provide for a symmetric specification, conditions on several objects should be specified in external condition boxes. An *external condition box* is depicted by a rounded rectangle that is connected with all the expanded variables to which the condition applies.

Navigation along Relationships. One way to define a variable is to specify it textually in the domain box of a query diagram. Another way to define a variable that references

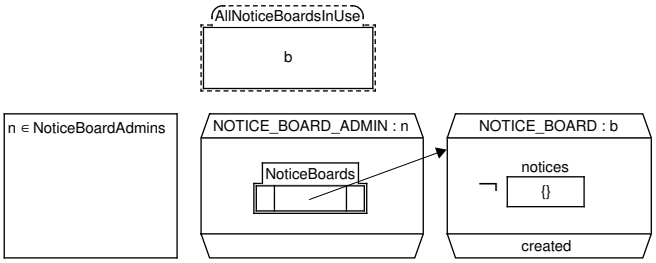


Figure 4.17: Query diagram for property AllNoticeBoardsInUse

or iterates over a relationship of an object bound to an expanded variable is to navigate graphically along a relationship of the expanded variable.

Navigating along relationship r of an expanded variable v in the condition box of a query diagram implicitly defines a variable (*implicit variable*). The implicit variable is a reference variable if r is single-valued, the implicit variable is an iterator variable if r is multi-valued. To visualize the navigation, the implicitly defined variable is expanded within v. If relationship r is single-valued, the nested expansion is shown with a single-line border, if relationship r is multi-valued, the nested expansion is shown with a double-line border. Implicitly defined variables can be named. Like for a variable defined in the domain box, the name of an implicit variable is shown behind the type name of the expanded variable separated by a colon. Semantically, there is no difference between defining a variable textually in the domain box and defining a variable implicitly by navigating along a relationship in the condition box. When a query diagram is evaluated, the two kinds of variable definitions are treated equally. All the variables implicitly defined by navigating along relationships of an expanded variable v are considered to be defined in an arbitrary order after v in the domain box.

Navigation along relationships in the condition box can be repeated to any depth. If a navigation path gets very long, a nested representation of expanded implicit variables may become clumsy and hard to read. As an alternative, an implicit variable may be expanded also outside the variable for which it is defined. In this case, the relationship property and the expanded implicit variable are connected by an arrow.

Example: Figure 4.17 shows the query diagram for property AllNoticeBoardsInUse of the global object type COMMUNICATION_SYSTEM. This diagram could be defined for property AllNoticeBoardsInUse instead of the diagram presented in Figure 4.11. Instead of defining variable b in the domain box of the query diagram, variable b is defined implicitly by expanding relationship NoticeBoards in the condition box.

Shorthand Variables. Besides defining variables in the domain box of a query, and besides defining variables implicitly by navigating along relationships of expanded variables, there is an additional shorthand notation for defining variables. By means of this shorthand

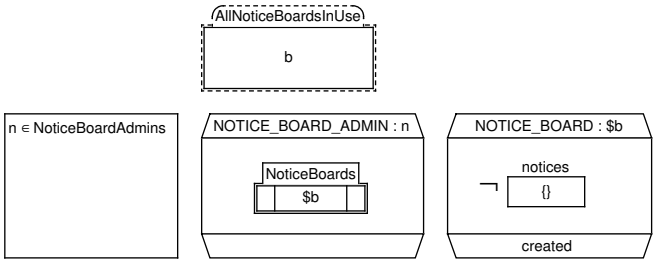


Figure 4.18: Query diagram for property AllNoticeBoardsInUse

notation, so-called *shorthand variables* are defined. A shorthand variable is defined by indicating its name (starting with $) in a property box of an expanded variable. Like other variables, shorthand variables may be expanded.

Like a variable defined in the domain box of a query, a shorthand variable is either an iterator variable or a reference variable. The semantics of a shorthand variable can be derived from the property for which the variable is defined:

- A shorthand variable depicted in a single-valued property is a reference variable that represents the single-valued property.
- A shorthand variable depicted in a multi-valued property with a double-line border is an iterator variable that iterates over the multi-valued property.
- A shorthand variable depicted in a multi-valued property with a single-line border is a reference variable that represents the multi-valued property.

Semantically, there is no difference between defining a variable in the domain box and defining a shorthand variable in the condition box. When a query diagram is evaluated, the two kinds of variable definitions are treated equally. All the shorthand variables that are defined for properties of an expanded variable v are considered to be defined in an arbitrary order after v in the domain box.

Example: Figure 4.18 shows the query diagram for property AllNoticeBoardsInUse of the global object type COMMUNICATION_SYSTEM. This diagram could be defined for property AllNoticeBoardsInUse instead of the diagrams presented in Figure 4.11 and Figure 4.17. In this variation, shorthand variable b is defined for property NoticeBoards of expanded variable n. Since multi-valued property NoticeBoards is depicted with a double-line border, variable b is an iterator variable. Shorthand variable b is further expanded.

4.5.5 Result Box

All the query diagrams introduced so far have been selecting query diagrams. The result box of a selecting query diagram is depicted like a property in an object-type diagram. The property that should by derived is shown as a box with a dashed border and with the property's name placed above. The property's type inside the box is replaced by an expression. When a selecting query diagram is evaluated, the following steps are passed through:

- *(1) Tuple creation:* As described in Section 4.5.3, a tuple is created for each possible variable binding. Besides bindings to the variables defined in the domain box, also bindings to implicit variables and to shorthand variables are part of the created tuples.
- *(2) Tuple filtering:* The tuples created in step (1) are filtered by retaining only those tuples that satisfy all the conditions imposed in the condition box of the query diagram.
- *(3) Result computation:* To each of the filtered tuples, the expression indicated in the query diagram's result box is applied.

Generating query diagrams differ from selecting query diagrams in the appearance of the result box and in their evaluation. The result box of a generating query diagram shows the object-type diagram of the instances to be generated. In this object-type diagram also inherited properties are shown. The domains of properties are replaced by expressions, which determine the property values of the new instances. Since the component-of relationship of a generated instance is implicitly set to the object for which the query is evaluated (i.e., to self), a component-of relationship is not shown within the object-type diagram. When a generating query diagram is evaluated, the first two steps—tuple creation and tuple filtering—correspond to the first two steps of evaluating a selecting query diagram. In the third step, however, a new instance is generated for each tuple filtered in step (2). A property p of an instance generated for a tuple t is set to the result of evaluating the expression associated with p on t.

Since query diagrams do not assume a specific implementation, instances derived by generating query diagrams need not be persistent across two successive database states. They could as well be dynamically re-generated from basic objects. Two derived objects are considered identical if their property values are identical.

Like in OQL, tuples can be grouped according to specific variables in order to generate a complex result structure. Grouping corresponds to the nest-operator for NF^2-relations [OY87, RKB87]. Grouping can be done only according to variables, but not according to other expressions. When a generating query diagram with a grouped result is evaluated, the first two steps—tuple creation and tuple filtering—correspond to the first

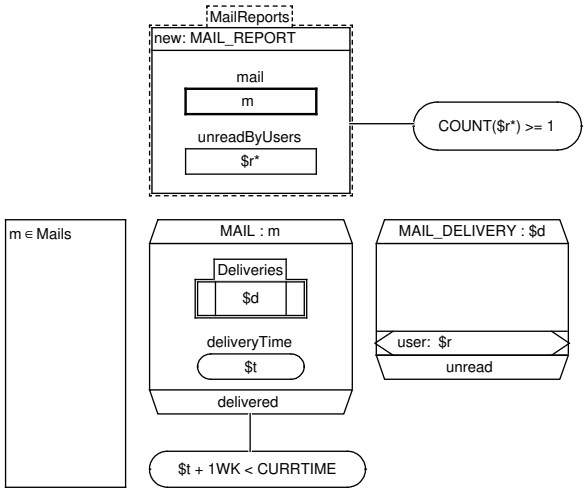


Figure 4.19: Query diagram for property MailReports

two steps of evaluating a selecting query diagram. Tuple filtering is succeeded by three further steps:

- *(3) Grouping:* The tuples filtered in step (2) are grouped and nested according to those variables which are assigned to the single-valued properties marked with a boldface border in the result box. A variable that is assigned to a property marked with a boldface border is referred to as *group variable*. All other variables are referred to as *nested variables*.

- *(4) Group filtering:* The groups computed in step (3) are filtered by retaining only those groups that satisfy the *having clause*. A having clause is specified in an external condition box (depicted by rounded rectangles) that is connected with a query diagram's result box. A grouped variable v can be used in a having clause by referring to it as v^*. The asterisk indicates that variable v actually stands for a set of values or objects after grouping.

- *(5) Instance creation:* An instance is created for each group filtered in step (4). A property p of an instance generated for a group g is set to the result of evaluating the expression associated with p on g. Like in a having clause, a grouped variable v can be used in an expression by referring to it as v^*.

Example: Figure 4.19 shows the query diagram for property MailReports of the global object type COMMUNICATION_SYSTEM. The query diagram generates instances of object type MAIL_REPORT. For a mail that has been delivered more than a week ago, a mail report collects the set of receivers that have not yet read the mail. Therefore, the query iterates over all mails and their deliveries. Iteration over local

class Deliveries of a mail is done by means of shorthand variable $d. The condition of the query diagram selects all tuples of mails and associated deliveries where the mail has been delivered more than a week ago and where reading of the mail has not been recorded with the delivery. The selected tuples are grouped according to mails. Shorthand variable $r is introduced to facilitate referencing of the receiver of a delivery. The having clause associated with the result box ensures that a mail report is generated only for mails that have not been read by at least one receiver.

4.5.6 Advanced Concepts

Advanced concepts of query diagrams allow to model alternative conditions by splitting the condition box of a query diagram into several branches, and to model complex derivation expressions in a modular way by nesting query diagrams.

Alternatives. One way to express alternatives is to associate several query diagrams with a single derived property. Another way to express alternatives is to describe several alternative conditions in one query diagram. Then, the query diagram is split into an arbitrary number of branches. When the query diagram is evaluated, the branches are combined by a logical *or*. The difference between associating several query diagrams with a single derived property and splitting a query diagram into several branches is that in the former case each query diagram is evaluated separately and that in the latter all branches are evaluated collectively. Thus, the two ways may yield a different number of result tuples and different values for nested variables.

Besides splitting the condition box of a query diagram into several branches, a branch-specific domain box can be specified for each branch. A branch-specific domain box defines the domain of variables which are used exclusively in the associated branch. To yield a uniform result structure, branch-specific variables cannot be used in the result box. This is another major difference to associating several query diagrams with one derived property.

Example: Figure 4.20 shows the query diagram for property LateUsers of the global object type COMMUNICATION_SYSTEM. A user is late if he/she has not read a mail that has been delivered more than three days ago or if he/she has not commented on a circular that has been delivered more than three days ago. Each of these two alternatives is captured by a branch in the query diagram. Variable m, which iterates over the unread mails of a user, is used exclusively in the upper branch and, thus, is defined in upper branch's domain box. Variable c, which iterates over the uncommented circulars of a user, is used exclusively in the lower branch and, thus, is defined in the lower branch's domain box. Only variable u, which is defined in the global domain box, is used in the result box of the query.

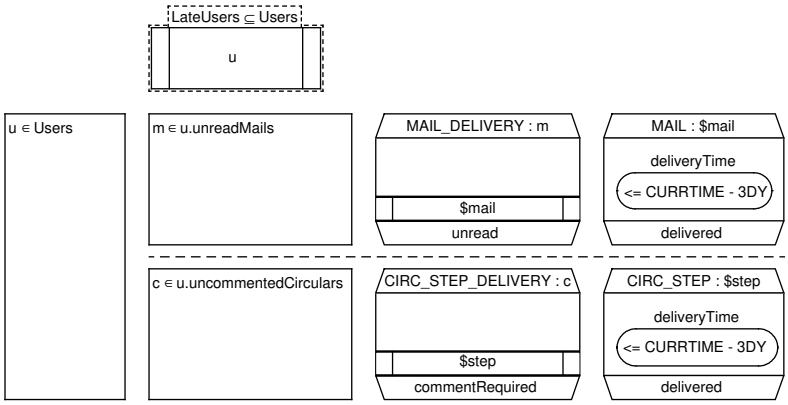


Figure 4.20: Query diagram for property LateUsers

Nested Query Diagrams. Often, complex expressions are easier to understand if they are described by modular subexpressions rather than by a single monolithic expression. To support modular subexpressions similar to subqueries in OQL, nesting of query diagrams is supported. Since a query diagram itself represents an expression, a nested query diagram can be used anywhere in a query diagram, i.e., in the domain box, in the condition box, and in the result box of a query diagram.

A nested query diagram is defined the same way as the main query diagram, except that also variables (including implicitly defined variables and shorthand variables) already bound by its enclosing environment can be used. For all parts of a nested query diagram, all variables visible at the location where it is embedded in the surrounding query diagram are visible, too.

For better readability, nested query diagrams are not depicted within the enclosing query diagram. They are labeled and described separately under this label. Nested query diagrams always are selecting. If the result type of a nested query diagram is a data type, the diagram's result box is depicted in the shape of a rounded rectangle. If the result type of a nested query diagram is an object type, the diagram's result box is depicted in the shape of a rectangle. The label under which a nested query diagram is used in its enclosing diagram is placed above the box. Like for properties, a multi-valued result is depicted by double-line borders.

Example: Figure 4.21 shows the query diagram for property UnreadMails of object type COMMUNICATION_SYSTEM. The query diagram selects all mails that have been delivered but that have not been read by a single receiver. The nested query diagram [deliveriesRead] is used within an external condition box. The nested query diagram selects for every mail the deliveries that have been read. Variable m can be used in the nested query diagram since m is visible in the condition box of its surrounding query diagram.

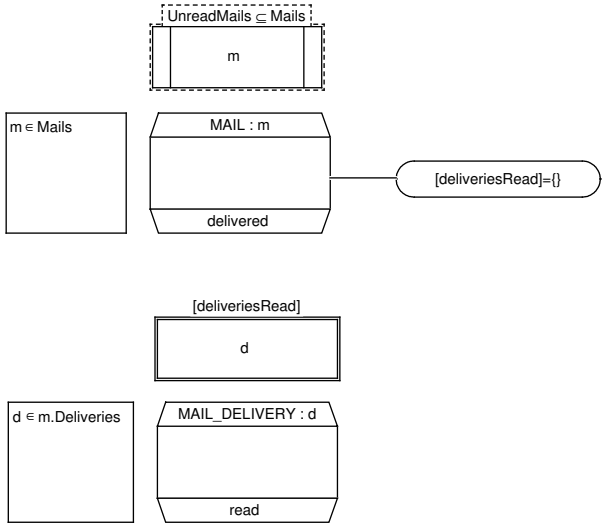


Figure 4.21: Query diagram for property UnreadMails

Chapter 5

Situation Diagrams

Contents

5.1 Situation-Type Diagrams 97
5.1.1 Properties 97
5.1.2 Situation-Type Hierarchies 98
5.2 Situation Classes 99
5.3 Situation Diagrams 103
5.3.1 Query Basics 104
5.3.2 Additional Expressions 107
5.3.3 Condition Box 108
5.3.4 Advanced Features 112
5.4 Summary 115

This chapter presents *situation diagrams* introduced in [LOS96]. Situation diagrams fill the need of a high-level declarative language for the specification of situations in conceptual database design. The notion of "situation" is used as a general term for events, which are characterized by an occurrence time, and intervals, which are characterized by an occurrence period. An event is said to occur if its occurrence time is the current system time, an interval is said to occur if its occurrence period includes the current system time, it is said to start if its start time is the current system time and it is said to end if its end time is the current system time. Rules specified in an ECA-like structure by means of *activation diagrams*, which will be presented in Chapter 6, are triggered by events specified by means of situation diagrams.

Situation diagrams extend Object/Behavior Diagrams in two directions: by adding new features to diagrams already existing in Object/Behavior Diagrams and by adding new diagrams at all. Situations are treated as "first-class objects" that are instances of some

situation type. A situation type is depicted by a *situation-type diagram*, which differs slightly from an object-type diagram. Situations are organized in situation classes. A *situation class* belongs to every instance of the object type for which it is defined. An object may react to situation occurrences belonging to one of its situation classes. Each situation class has an associated situation type specifying the type of the situation class's potential members. Situations have a static nature; they bind only the occurrence context and, thus, behavior diagrams cannot be specified for situation types.

Two types of situations are distinguished: basic situations and logical situations. *Basic situations* comprise database situations (e.g., events representing the completion of an activity), calendar situations (e.g., predefined intervals representing days), and explicitly raised situations (e.g., an alert raised manually). The classes in which basic situation classes are collected are called *basic situation classes*. *Logical situations* are derived from basic situations and more primitive logical situations by applying logical and temporal conditions. The classes in which logical situations are collected are called *logical situation classes*. Logical situation classes are populated by *situation diagrams*.[1] A situation diagram visualizes a query with temporal and logical conditions over situations and over conventional database objects. Situation diagrams extend query diagrams introduced in Section 4.5 by graphical concepts to apply temporal predicates to situations. Using names in strict analogy to Object/Behavior Diagrams, the term "derived situation" should be used instead of "logical situation." But the term "logical event" (and thus logical situation) is already established in the active database field [BL95]. Nevertheless, logical situation classes are depicted as derived classes.

In the sequel, the terms "situation," "event," and "interval," denote instances, i.e., particular occurrences of a situation type, event type, or interval type.

The running example of the communication system introduced in Chapter 4 is continued by extending it with situation types, situation classes, and situation diagrams. Some of the situation classes introduced here will be used in Chapter 6 for specifying business rules that trigger upon the occurrence of their situations.

The remainder of this chapter is organized as follows: Section 5.1 presents situation-type diagrams and shows how they differ from object-type diagrams. Section 5.2 presents situation classes, which represent additional properties of object types. Section 5.3 presents situation diagrams as query diagrams especially tailored to the temporal comparison of situations. Situation diagrams provide new concepts that are more convenient for the temporal comparison of situations than the concepts provided by "conventional" query diagrams. Section 5.4 concludes this chapter with a brief summary of the main contributions of situation diagrams.

[1] The term "situation diagram" is used in a broader and in a narrower sense. Situation diagrams in the broader sense comprise all concepts for specifying situations, i.e., (basic and logical) situations, situation classes, situation types, and derivation expressions for logical situations. Situation diagrams in the narrower sense represent the declaratively specified query, which determines logical situations.

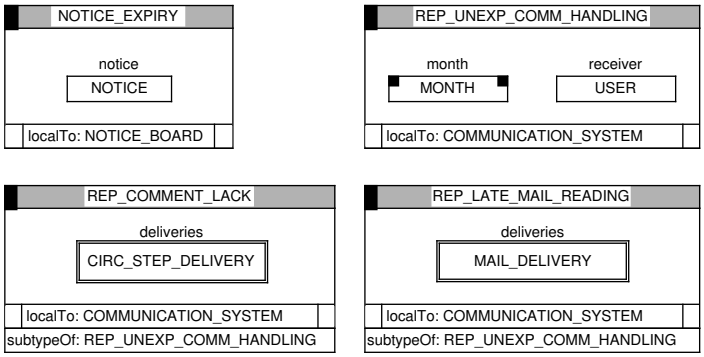


Figure 5.1: Situation-type diagrams

5.1 Situation-Type Diagrams

Like an object type, a situation type defines the structure of its instances by a set of properties. Situation types may also be organized in type hierarchies where a subtype inherits the properties of its supertype. Further, each situation type is defined locally to exactly one object type.

Event types implicitly define attribute occtime, which holds the occurrence time of an instance. Interval types implicitly define two attributes start and end, which hold the bounds of the occurrence period of an instance.

Situation-type diagrams differ from object-type diagrams as follows: An event-type diagram contains a small filled square at the upper left corner beside the type name, and an interval-type diagram contains two small filled squares, one at the upper left and one at upper right corner.

Example: Figure 5.1 depicts the situation-type diagrams of event types NOTICE_EXPIRY, REP_UNEXP_COMM_HANDLING (repeated unexpected communication handling), REP_COMMENT_LACK (repeated comment lack), and REP_LATE_MAIL_READING (repeated late mail reading). Their semantics will be explained later.

5.1.1 Properties

Like object types, situation types may define *general relationships* and *attributes* (either single-valued or multi-valued).

Example: Event type NOTICE_EXPIRY (Figure 5.1) defines general relationship notice with domain NOTICE.

Besides attributes and general relationships, a situation type may define two kinds of situation-specific relationships: contributing-situation relationships and local-to relationships.

Contributing-Situation Relationship. A *contributing-situation relationship* allows a situation to reference other situations that contribute to its occurrence. Thus, the domain of a contributing-situation relationship has to be a situation type. A contributing-situation relationship is either single-valued or multi-valued.

The visual representation of a contributing-situation relationship differs from the representation of a general relationship only in a small filled square depicted at the upper left corner if the domain is an event type, and in two small filled squares, one depicted at the upper left and the other depicted at the upper right corner, if the domain is an interval type.

Example: Event type REP_UNEXP_COMM_HANDLING (Figure 5.1) defines the contributing-situation relationship month with domain MONTH.

Local-To Relationship. Every situation type has to be defined locally to exactly one object type. This locality is captured by an obligatory *local-to relationship*. A situation that is local to an object is said to occur "directly" at this object. The domain of a local-to relationship has to be an object type. If a situation type S defines a local-to relationship to some object type O, O has to define a "situation class with local members" (cf. Section 5.2) with S as member type. This class represents the inverse relationship to the local-to relationship. Regarding this semantics the local-to relationship between a situation type and an object type corresponds to the component-of relationship between object types.

The visual representation of a local-to relationship corresponds to the visual representation of a component-of relationship. A local-to relationship is depicted by a bar (accentuated by side borders) at the bottom of a situation-type diagram. Since there may be defined only one local-to relationship for a situation type, naming this relationship is optional. If the relationship remains unnamed, the superordinate object can be referenced by means of the keyword localTo.

Example: Event type NOTICE_EXPIRY (Figure 5.1) defines a local-to relationship (labeled localTo) to object type NOTICE_BOARD.

5.1.2 Situation-Type Hierarchies

Like object types, situation types may be organized in hierarchies by means of a subtype-supertype relationship where the subtype inherits all properties of the supertype. The

domain of an inherited property may be redefined at the subtype by a direct or indirect subtype of the inherited domain (cf. Section 4.2.3).

The subtype-supertype relationship between situation types is depicted like the subtype-supertype relationship between object types. The name of the supertype is given in a bar labeled subtypeOf, located at the bottom of the situation-type diagram of the subtype. Usually, inherited properties are depicted at the subtype only if their domain is redefined.

Example: Event types REP_COMMENT_LACK and REP_LATE_MAIL_READING are subtypes of event type REP_UNEXP_COMM_HANDLING. Both subtypes inherit properties month and receiver from their supertype and add property deliveries (cf. Figure 5.1).

5.2 Situation Classes

Like objects, situations are organized in classes, which we refer to as situation classes. Situation classes are defined locally to object types. A situation class that is local to the "global" object type concerns a database as a whole and is therefore called a global situation class. Situation classes are defined at object types like relationships by giving a name for the situation class, by specifying a member type, and by determining whether it is single-valued or multi-valued. The situations collected in a situation class must be direct or indirect instances of the member type. Each instance of the object type possesses an extension for each situation class local to it. An object possessing a local situation class may react on the occurrence of situations that are contained in the class by triggering a rule (cf. Chapter 6).

Like properties, situation classes that are local to an object type O are inherited by the direct and indirect subtypes of O. In addition, the member type of an inherited situation class may be redefined at a subtype to a direct or indirect subtype of the inherited member type. How situation diagrams of logical situation classes, which are also inherited by the subtypes, may be redefined at subtypes, is discussed in [Lan97].

Situation classes are categorized into situation classes with local members, situation classes with indirectly local members, and situation classes with non-local members.

Situation Class With Local Members. Situations collected within a *situation class with local members* occur directly at the object that holds the situation class. The object referenced by the local-to relationship of a situation is the object holding the situation class to which the situation belongs.

A situation class with local members is depicted by a rectangle showing its name as a tag at the top and the name of the member type inside. One small filled square at the upper left corner identifies an event type as member type, two small filled squares, one

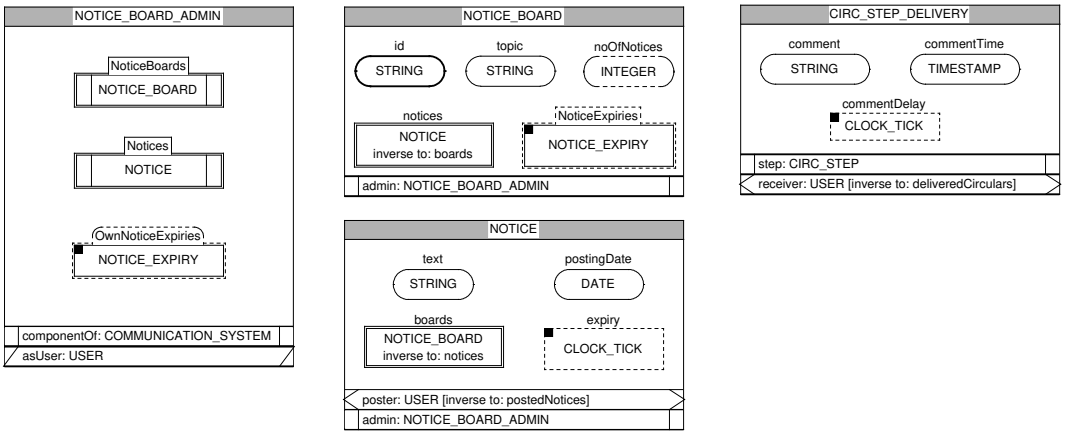


Figure 5.2: Object-type diagrams including situation classes

at the upper left and one at the upper right corner, identify an interval type as member type. A single-valued situation class (a situation class referencing only one situation, also called a singleton class) is depicted by a single-line border, a multi-valued situation class (a situation class referencing a set of situations) is depicted by a double-line border. A logical situation class is accentuated graphically by a dashed border.

Example: Figure 5.2 depicts object-type diagrams of object types already introduced in Chapter 4, but now including situation classes. Logical situation class NoticeExpiries of object type NOTICE_BOARD collects events of event type NOTICE_EXPIRY. Such an event occurs directly at a notice-board and describes that a notice posted to the board has expired (cf. Section 5.3). NoticeExpiries is a situation class with local members. Each member of situation class NoticeExpiries of some object o is local to o, i.e., the member's local-to relationship has value o. This event class will be used in Chapter 6 to formulate a business rule that requires to remove expired notices after a certain period of time has passed.

A situation class S_2 with local members may be defined to be a subclass of another situation class S_1 with local members. S_2 may be introduced at the same object type O at which S_1 is introduced or may also be introduced at some subtype of O, O', which inherits S_1 from O. S_2 may contain only situations that are members of S_1. Both situations classes may be logical situation classes.

A subclass constraint between situation classes is represented graphically like a subclass constraint between object classes. The tag at the top of the rectangle describing the subclass contains besides the name of the subclass also the name of the superclass with the mathematical symbol for a subset relationship ($\subseteq$) placed between them.

Example: Consider the global object type COMMUNICATION_SYSTEM extended with situation classes in Figure 5.3. Situation classes (with local members) RepeatedLateMail-

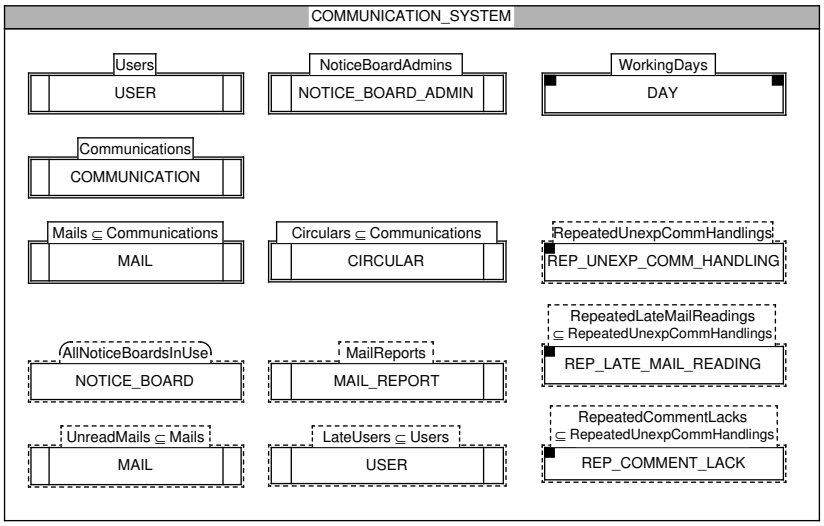


Figure 5.3: The global object type including situation classes

Readings and RepeatedCommentLacks represent subclasses of situation class RepeatedUnexpCommHandlings. Situation class RepeatedLateMailReadings holds situations of type REP_COMMENT_LACK (cf. Figure 5.1) that occur at the end of each month for each user who has not read some mails delivered to him/her. Situation class RepeatedCommentLacks holds situations of type REP_LATE_MAIL_READING (cf. Figure 5.1) that occur at the end of each month for each user who is late with commenting circulars or has refused to comment on circulars delivered to him/her. Their detailed semantics will be explained in Section 5.3.

We assume a predefined set of basic situation classes with local members for each object type: a basic event class for each activity (an *activity-event class*) and a basic interval class for each state (a *state-interval class*) defined in an object type's life-cycle diagram. Names of activity-event classes are constructed from the activity name (starting with a capital letter) followed by postfix _Events ($\langle ActivityName \rangle$_Events). Names of state-interval classes are constructed from the state name (starting with a capital letter) followed by postfix _Intervals ($\langle StateName \rangle$_Intervals). Also the situation types describing the members of these basic situation classes are predefined. There is an *activity-event type* for every activity, and there is a *state-interval type* for every state. An activity-event type defines attribute occtime holding the occurrence time of an event instance. Additionally, an activity-event type defines a property for every non-primary port of the activity for which it is defined. The name and the domain of a property correspond to the name and the domain of the port the property represents. A state-interval type defines attributes start and end holding the bounds of the occurrence period of an instance. The name of an activity-event type is constructed from the name of the object type for which the activity is defined, from the keyword _ACTIVITY_, and from the name of the activity, i.e., the name is of the form $\langle ObjectTypeName \rangle$_ACTIVITY_$\langle ActivityName \rangle$. The name of an

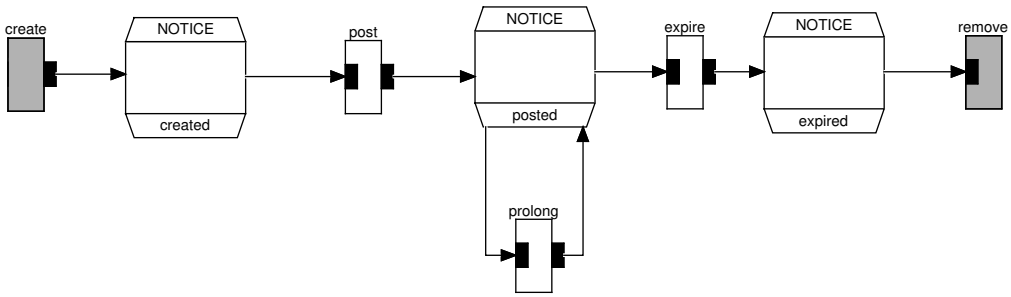


Figure 5.4: Life-cycle diagram of object type NOTICE

state-interval type is constructed from the name of the object type for which the state is defined, from the keyword _STATE_, and from the name of the state, i.e., the name is of the form ⟨*ObjectTypeName*⟩_STATE_⟨*StateName*⟩. Following the conventions for names of object types, the names of activity-event types and state-interval types are capitalized.

These basic situation classes are assumed to be populated as follows: (1) If the execution of an activity on an object o is completed, a basic event is created and inserted into that local event class of o which is implicitly named after the activity. The associated event type defines an attribute for each data port and a general relationship for each object port. The domains of the event type's properties correspond to the domains given for the ports (a data type for data ports and an object type for object ports). The event's properties corresponding to the input ports are set to the objects/values with which the activity is invoked. The event's properties corresponding to output ports are set to the object/values assigned to the ports when activity execution is completed. (2) If an object enters a state, a basic interval is created, the interval's end time is set to the special value "open" (which is always dynamically resolved to the current system time), and the interval instance is inserted into that local interval class which is named after the state. If an object leaves a state, the end time of the corresponding basic interval is set.

Example: Consider the life cycle of object type NOTICE depicted in Figure 5.4. Each instance of object type NOTICE holds event class Prolong_Events for activity prolong and interval class Posted_Intervals for state posted.

Situation Class With Indirectly Local Members. An object may reference situations that occur locally to a direct or indirect component of it. A *situation class with indirectly local members* allows an object to reference these indirectly local situations. A situation class with indirectly local members may again be basic or logical.

The graphical representation corresponds to the representation of has-indirect-component relationships. The tag at the top of the rectangle has rounded top corners. Disregarding the tag's rounded top corners, a situation class with indirectly local members is represented like a situation class with local members.

Example: Each instance of object type NOTICE_BOARD_ADMIN holds an extension for logical situation class OwnNoticeExpiries, which collects indirectly local events (cf. Figure 5.2). These events occur locally to the notice boards that are components of the administrator (expressed by means of the has-component relationship NoticeBoards) when a notice posted to such a board expires. Only those notices are considered which are posted by the administrator himself/herself.

Situation Class With Non-Local Members. The situation classes introduced so far reference directly or indirectly local situation occurrences. A *situation class with non-local members* allows an object to reference situations that occur (directly or indirectly) at another object. An object holding a situation class with non-local members is said to subscribe another object's situations and, thus, may react also to the occurrence of the subscribed situations by triggering a rule (cf. Chapter 6).

Situation classes with non-local members are represented similarly to general relationships. The name of the situation class is given above the rectangle (without a tag). Disregarding the missing tag, a situation class with non-local members is represented like a situation class with local members.

Example: Situation class expiry of object type NOTICE holds for each notice the (non-local) event denoting the clock tick at which the notice expires. Events denoting clock ticks are (predefined) basic events and are implicitly local to the global object. Their event type CLOCK_TICK is predefined, too. Event class expiry will be used in Chapter 6 to formulate a business rule that requires to have a notice expired after a certain amount of time has passed since the notice has been posted or prolonged the last time.

5.3 Situation Diagrams

The extension of a logical situation class (either with local members, with indirectly local members, or with non-local members) is determined by a graphical query or by the union of several graphical queries over members of object classes and situation classes. Such a graphical query is called *situation diagram*. Situation diagrams are derived from conventional query diagrams introduced in Section 4.5. They are structured equally; both consist of a result box, a domain box, and a condition box. The major difference to conventional query diagrams is that the condition box of situation diagrams is tailored to the temporal comparison of situations. Instead of displaying a set of "expanded" objects, a single box including situations as small boxes depicted from left to right in several horizontal lines is used for comparing the occurrence times or the ocurrence periods of situations.

Since situation diagrams are based on query diagrams, their basic structure is not rediscussed. Section 5.3.1 outlines only the basics of situation diagrams and points out the

similarities and differences of situation diagrams and conventional query diagrams. Section 5.3.2 shortly introduces expressions that are usually used for deriving logical situations. Section 5.3.3 presents how the occurrence times of events and the occurrence periods of intervals can be compared in the condition box. This subsection starts with the time model on which the temporal comparison of situation occurrences is based and continues with the graphical concepts provided by the condition box to compare situation occurrences. Finally, Section 5.3.4 concludes this section with a repetition of advanced features also available in situation diagrams: nesting, grouping, and alternatives.

5.3.1 Query Basics

As already mentioned, situation diagrams are structured like conventional query diagrams and thus resemble an OQL query [CB97], too. A situation diagram consists of a result box, a domain box, a condition box, and an auxiliary-condition box.

Variables bound to events are called *event variables*, variables bound to intervals are called *interval variables*. Both event variables and interval variables may also be called *situation variables*. Conditions over variables in situation diagrams are specified in the condition box. In the condition box, event variables are depicted by squares (*event boxes*) containing a small filled square at the upper left corner; interval variables are depicted by rectangles (*interval boxes*) containing two small filled squares, one at the upper left and one at the upper right corner. The name of a variable is usually given within the square or rectangle represented by the variable. If the name of a situation variable is longer than the width of its box (square or rectangle), it may also be placed outside the box. Situation variables are temporally compared by arranging them within the condition box. Section 5.3.3 presents the condition box of situation diagrams in detail.

In conventional query diagrams, a condition over several objects is specified by connecting an external condition box to the involved objects and by giving the condition within this box. Similarly, logical conditions over situations are given in a rounded rectangle, called *logical-condition box*, connected to all situation boxes whose variables are used in the condition. Large conditions may be given a label within a logical-condition box and then be specified under this label in an *auxiliary-condition box* of a situation diagram. The auxiliary-condition box of a situation diagram is depicted to the right of the situation diagram's condition box.

Corresponding to selecting and generating conventional query diagrams, situation diagrams are either selecting or generating. A *selecting situation diagram* selects situation instances from other situation classes, a *generating situation diagram* generates new situation instances of a given situation type upon the occurrence of other situations. Note that both selecting and generating situation diagrams consitute views on the database and, therefore, the extension of logical situation classes changes if the extensions of basic situation classes or objects in the database change when time proceeds (see also below).

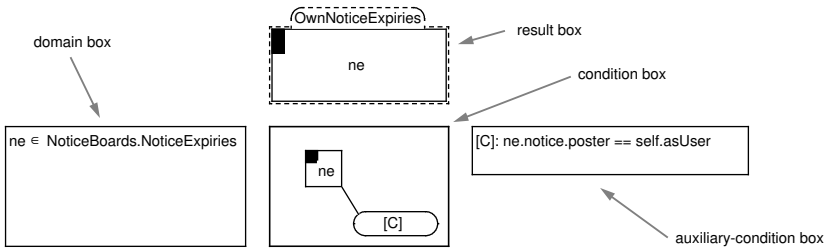


Figure 5.5: Situation diagram for situation class OwnNoticeExpiries

The result box of a situation diagram specifies the result of the situation derivation and is depicted like the situation class whose extension is derived, but without the name of its member type inside. In the selecting case, an expression (cf. Section 5.3.2) is specified within the result box instead of the member type's name.

Example: Figure 5.5 depicts a situation diagram for logical situation class OwnNoticeExpiries of object type NOTICE_BOARD_ADMIN. This selecting situation diagram defines event variable ne ranging over the expiries of the notices posted to the notice boards of an administrator. Within the condition box (the lower box in the middle), a logical condition over event variable ne is given in a logical-condition box connected to the event box of ne. The condition is not specified within the logical condition box, but is given the label [C] and is specified under this label in the auxiliary-condition box. This condition requires that the user who posted an expired notice is identical to the user who administrates the board to which the notice is posted, i.e., only the expiries of notices posted by the administrator are considered. The result box determines that all these expiries are selected and thus populate the situation class.

In the generating case, the situation-type diagram of the situation instances to be generated is depicted within the result box instead of the member type's name. The property domains of the situation type are replaced by expressions. The values of the implicit property occtime (for events) and the implicit properties start and end (for intervals) of the generated situations are determined by a shaded vertical bar in the condition box, the *occurrence bar*. Property occtime of events is set to the occurrence time of the event covered by the occurrence bar. Properties start and end of intervals are set to the start time and the end time of the interval covered by the occurrence bar (see also Section 5.3.3).

Example: Figure 5.6 depicts a generating situation diagram for logical situation class NoticeExpiries of object type NOTICE_BOARD. An instance of event type NOTICE_EXPIRY is generated for the expiry e of each notice n posted to the board. The occurrence time of the generated event is set to the occurrence time of e. Property notice is set to the expired notice n.

Generating situation diagrams may be attached only to a logical situation class with local members because each situation instance (and thus also the automatically generated ones)

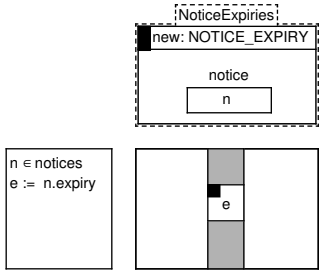


Figure 5.6: Situation diagram for situation class NoticeExpiries

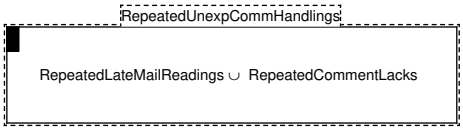


Figure 5.7: Situation diagram of situation class RepeatedUnexpCommHandlings

have to be local to some object. Selecting situation diagrams may be attached to a logical situation class with local members only if they select situations that are already local to the object that holds this situation class. The members of situation classes with indirectly local or non-local members may be derived only by selecting situation diagrams.

Example: Figure 5.7 depicts the selecting situation diagram for situation class RepeatedUnexpCommHandlings of the global object type COMMUNICATION_SYSTEM. For the moment assume that it holds events that occur at the end of the month and document that a user has not read several mails in time or has not commented circulars several times. The degenerated situation diagram, which consists only of a result box, selects all instances of situation classes RepeatedlateMailReadings and RepeatedCommentLacks by merging their extensions with operator ∪.

Like the specification of conventional query diagrams, the specification of situation diagrams is not allowed to be recursive.

A logical situation class constitutes a view on the current database state, and, therefore, the extension of a logical situation class changes if the extensions of basic situation classes or objects in the database change when time proceeds. Whether the extensions of logical situation classes are materialized or derived on demand is irrelevant at the conceptual level. An efficient implementation will, of course, make use of various strategies to update materialized views incrementally (cf. [CW91]). Since situation diagrams do not assume a specific implementation, situations created by generating situation diagrams need not be persistent across two successive database states. They could as well be re-generated later from basic situations (which are kept over the entire history). Thus, two logical situations are considered identical if their occurrence time (for events) or their occurrence period (for intervals) and their property values are identical.

5.3.2 Additional Expressions

The expressions (literals, functions, navigation paths, variables, and nested query diagrams) available in conventional query diagrams are also available in situation diagrams. Deriving logical situations requires a few further expressions, which are presented in the following.

Functions. A calendar system is assumed that provides a set of predefined calendar events and calendar intervals of the past and of the future compared to the current system time. *Situation-retrieving functions* select either single situations or a set of situations from the predefined set of basic calendar classes.

Example: Function CLOCKTICKS returns a set of clock-tick events (of event type CLOCK_TICK) according to a given time-stamp (e.g., '1997/12/03 11:45') or to a string pattern of a time-stamp (e.g., 'YYYY/12/31 12:00' stands for all clock-tick events denoting 12:00 noon of the last day of every year).

Function DAYS returns all day-intervals (of interval type DAY) according to a given date (e.g., '1997/12/31') or to a string pattern of a date (e.g., 'YYYY/MM/01').

Function MONTHS returns all month-intervals (of interval type MONTH) according to a given month (e.g., '1997/12') or to a string pattern of a month (e.g., 'YYYY/12'). The situation types of the returned calendar situations are predefined.

Additional *date/time-construction functions* like OCCTIME, OCCDATE, and OCCTIMESTAMP etc. return the time, the date, or the time-stamp of an event's occurrence respectively. Similarly functions STARTTIME, STARTDATE, and STARTTIMESTAMP return the time, the date, or the time-stamp of an interval's start and functions ENDTIME, ENDDATE, and ENDTIMESTAMP return the time, the date, or the time-stamp of an interval's end. The time, the date, and the time-stamp are values of data types TIME, DATE, or TIMESTAMP, respectively.

If a situation diagram refers to members of object classes, these members reflect the current database state. Generally, an object's property values change as time proceeds. In many situation diagrams, it may be desirable to refer to past property values. Therefore, *past-property-value-accessing function* AS_OF is provided, which, with object o and time-stamp t as operands, returns a version of object o with property values as they have been at time-stamp t.

Navigation Paths. As already mentioned above, each object type defines for each of its activities an activity-event class, and for each of its states a state-interval class. Navigation is usually indicated by using the dot-notation. To make navigation along activity-event classes and state-interval classes more intelligible, activity-event classes and

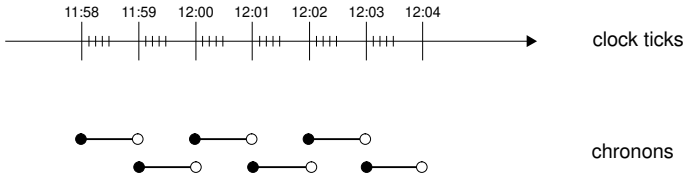


Figure 5.8: Clock ticks and chronons

state-interval classes may be referenced using the at-notation instead of the dot-notation. Then, an activity-event class may be referenced by $\langle activityName \rangle @object$ instead of $object.\langle ActivityName \rangle$_Events and, correspondingly, a state-interval class may be referenced by $\langle stateName \rangle @object$ instead of $object.\langle StateName \rangle$_Intervals. The rest of the path, again may follow the dot-notation. Properties of activity events may be accessed by adding the property name in dot-notation.

Example: Assume navigation path self.create@notices.poster specified in some query at object type NOTICE_BOARD (cf. Figure 5.2). The navigation path obtains the values of property poster of all events in class Create_Events of all notices posted to the notice board for which the path is evaluated. Navigation path self.notices.Create_Events.poster may be used instead.

5.3.3 Condition Box

The condition box of a situation diagram allows to compare the occurrence times of events and the occurrence periods of intervals by arranging situation boxes within the condition box. Before going into details of temporal comparison, we present the time model, which lays the foundations for the temporal comparison.

Time Model

Time is considered as a finite sequence of chronons, each lasting from one tick of the system clock (the chronon's start time) to immediately before the n-th subsequent tick (the chronon's end time). n may be greater than one because the smallest temporal unit that is not further decomposable from a user's perspective can often be of a coarser granularity than the temporal unit defined by two successive clock ticks of an internal clock of some target system. E.g., time-stamps showing only minutes but not seconds are sufficient for many business applications. Chronon borders have to match with the borders of calendar intervals in the real world. This means that the durations of the calendar intervals with a fixed duration (minutes, hours, etc.) must be an integer multiple of a chronon duration. E.g., the duration of calendar minutes is an integer multiple of a chronon lasting 15 seconds.

Consider Figure 5.8. The upper row depicts an arbitrary segment of the time line with

clock ticks shown on it. The clock ticks that represent the beginning of a minute are emphasized by time-stamps. The lower row depicts the chronons contained in the segment of the time line as short lines in the granularity of minutes. We assume that time-stamps showing only minutes but not seconds are sufficient for many business applications. The filled bullet at the begin of a chronon line means that the clock tick starting a chronon is also part of the chronon. The unfilled bullet at the end of a chronon line means that the clock tick ending a chronon is not part of the chronon.

In the condition box, time flows from left to right. Event boxes represent exactly one chronon; interval boxes represent a finite sequence of chronons. The depicted chronon of an event is the chronon during which the event occurred. The left side of the event box represents the occurrence time-stamp. An interval lasts during the depicted chronons. The left side of an interval box represents the start time-stamp, the right side represents the end time-stamp. The horizontal extension of an interval box depicted in the condition box does not denote the absolute number of chronons (compared to the length of event boxes) during which the interval lasts. As intervals usually last during several chronons, they are depicted generally by rectangles and not by squares. If necessary (e.g., for comparison with events), an interval box can also be depicted by a square.

To treat events and intervals uniformly as situations, the start time of a situation is defined to be the start time of its left-most chronon and the end time of a situation is defined to be the end time of its right-most chronon. The representation of an event as a chronon facilitates the graphical "occurs-*strictly*-before" comparison between events as shown below.

Temporal Comparison of Situations

Temporal and logical conditions on situations are specified in the condition box. The condition box itself is depicted by a simple rectangle. Temporal comparison of situations is expressed graphically within this rectangle by arranging the situation boxes according to a few rules and by applying following concepts: temporal precedence, temporal distance, temporal synchonization, and temporal adjacency.

Temporal Precedence. Typically, conditions on the temporal order of situations are used when defining logical situations. Thus, temporal precedence of situations should be easy to recognize and easy to specify. *Temporal precedence* is expressed by arranging situation boxes from left to right in the same horizontal line, which is referred to as *time line*. The end of a situation whose situation box is depicted to the left of another situation box in the same time line has to be less than or equal to the start of the situation whose situation box is depicted at the right. Therefore, if the two situations are events, the left event has to occur strictly before the right event, i.e., the occurrence time of the first event must be less (but not equal) to the occurrence time of the second event. If the two situations are intervals, the left interval has to end strictly before the right interval starts,

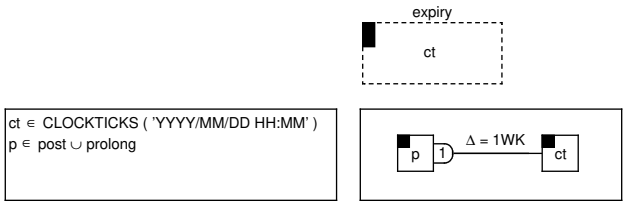


Figure 5.9: Situation diagram for situation class expiry

i.e, the occurrence time of the first interval's end chronon must be less (but not equal) to the occurrence time of the second interval's start chronon. If the left situation is an event and the right situation is an interval, the event has to occur strictly before the interval starts. If the left situation is an interval and the right situation is an event, the interval has to end strictly before the event occurs. Similarly, the concepts for temporal comparison introduced later on can be applied to both, events and intervals.

Example: Figure 5.9 depicts the situation diagram for situation class expiry of object type NOTICE. Consider only the two event boxes depicted in the condition box. Event variable p is required to occur strictly before event variable ct because p is depicted to the left of ct and both are depicted in the same time line. The detailed semantics of this situation diagram will be explained later in this section when further concepts for the temporal comparison of situations are introduced.

Whereas temporal precedence is expressed by depicting situation boxes in the same time line, *temporal independence* is expressed by depicting situation boxes in different time lines.

Temporal Distance. Time lines are usually not shown, except if two successive situations are compared with respect to a temporal condition, i.e., the *temporal distance* of two situations is constrained by some condition. Then, the temporal condition is written above the time line. The temporal condition is specified in the form of $\Delta\langle compPred\rangle\langle noOfChrs\rangle$ where Δ denotes the distance between the involved situations in number of chronons, $\langle compPred\rangle$ is one of the comparison predicates $<$, $\leq$, $=$, $\geq$, or $>$, and $\langle noOfChrs\rangle$ denotes an absolute number of chronons. The number of chronons is usually specified using a number n and a function describing a certain time-measure (WK for weeks, DY for days, HR for hours, and MN for minutes), which returns the result of n multiplied with the number of chronons a week, a day, an hour, or a minute represents. E.g., 2.5 HR results in 150 chronons if we assume minutes to be the granularity of time.

Example: Consider again Figure 5.9. Event variable ct is required to occur exactly one week after event variable p.

Temporal Synchronization. *Temporal synchronization* can be used to specify a partial temporal order between situations in different time lines. Temporal synchronization of time lines is expressed by a *synchronization line*, which is drawn vertically along a clock tick, i.e., along the border of two successive chronons.

Time lines need not be explicitly drawn. If a time line is not shown, all situations in the same horizontal position build an imaginary time line and any synchronization line crossing this imaginary time line synchronizes the time line at its crossing. All situation starts and situation ends depicted to the left of a synchronization line have to be less than or equal to the clock tick represented by the synchronization line. All situation starts and situation ends depicted to the right of a synchronization line have to be greater than or equal to the clock tick represented by the synchronization line. All situation starts and situation ends directly attached to a synchronization line have to coincide with to the clock tick represented by the synchronization line. If a time line is drawn, a special connector symbol, a small filled triangle having its base attached to the synchronization line and its top connected to the time line, is used to connect the time line only to those synchronization lines with which it is synchronized. Note that the occurrence bar in generating situation diagrams consists of two vertical synchronization lines, which divide all time lines of a situation diagram into past and future. Examples for temporal synchronization will be given in Section 5.3.4 when additional concepts of situation diagrams are introduced.

Temporal Adjacency. Formulating conditions like "the first response after the last offer" or "any response after the last offer" requires universal quantification, or negation and existential quantification, which is provided by specifying a logical condition that negates a nested situation diagram. Since conditions on the "*temporal adjacency*" of situations are used rather frequently in practice, they are supported directly by a graphical representation. Let $\langle c \rangle$ be a condition over event variable a and let $sat(a, \langle c \rangle)$ denote the set of objects in the domain of a (as specified in the domain box) that satisfy condition $\langle c \rangle$. Then, the predicate "event a is the n-th last event of the events belonging to $sat(a, \langle c \rangle)$ that occurs before event b" is expressed graphically by connecting these events by a line and by writing the number n and the condition $\langle c \rangle$ (which is "true" if omitted) next to event a. The predicate "event b is the n-th event of the events belonging to $sat(b, \langle c \rangle)$ that occurs after event a" is defined analogously with n and $\langle c \rangle$ associated with b. These predicates may also be used in an extended form: Instead of specifying that an event must be exactly the n-th event before or after another event, a set of numbers, e.g., "3–5" or "1,3", may be specified with the semantics that an event must be the k-th event for some k in this set.

Example: Consider again Figure 5.9. The event described by event variable `p` has to be the last event before the clock-tick event `ct`, which occurs exactly one week after the event described by `p`. Thus, the complete semantics of this situation diagram is as follows: It selects for each notice that clock-tick event which occurs exactly one week after the notice has been posted to one or more boards or after the notice has been prolonged or posted the last time.

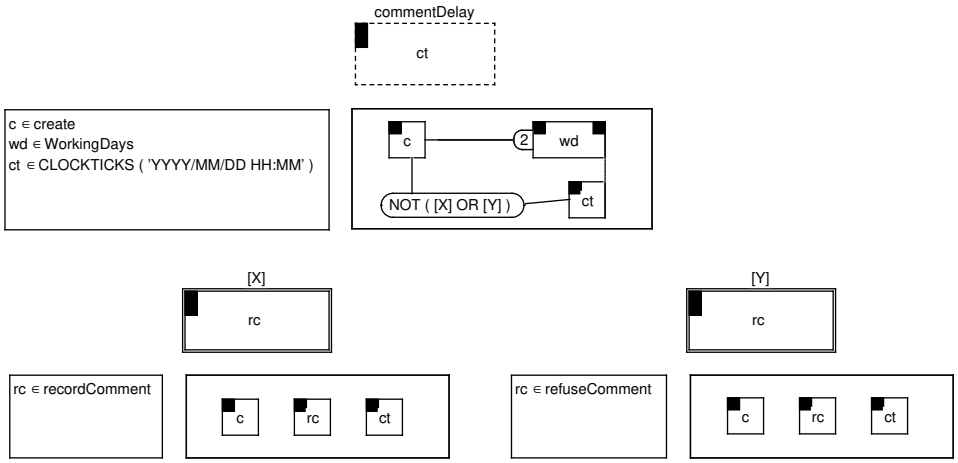


Figure 5.10: Situation diagram for situation class commentDelay

5.3.4 Advanced Features

Situation diagrams may be nested like conventional query diagrams (cf. Section 4.5.6). *Nested situation diagrams* represent expressions and may be used within other expressions. Nested situation diagrams differ from conventional nested query diagrams only in the result box, which depicts either a small filled square at the upper left corner if events are selected or two small filled squares, one at the upper left and one at the upper right corner, if intervals are selected. By giving an example of nested situation diagrams, we demonstrate also basic features like temporal synchronization, for which no example has been given so far.

Example: Figure 5.10 depicts the situation diagram for situation class commentDelay of object type CIRC_STEP_DELIVERY. The condition box shows two time lines, which are synchronized by a synchronization line. It synchronizes the end of interval box wd with the end of event box ct. Both are aligned to the synchronization line. Thus, ct represents the last clock-tick event of a working day wd.

Further, this situation diagram defines two nested situation diagrams labeled [X] and [Y]. The complete semantics of this situation diagram is as follows: Commenting a circular-step delivery is late at the end of the second working day following the creation of the delivery if neither a comment has been given nor a comment has been refused explicitly by the receiver since then. Situation class commentDelay will be used in Chapter 6 for implementing a business rule that requires to record the delay by executing activity recordDelay if such a situation occurs.

Like the result of generating conventional query diagrams, the result of generating situation diagrams may also be *grouped* and a *having clause* may be applied to the grouped result (cf. Section 4.5.5). Tuples are grouped according to those variables which are assigned to

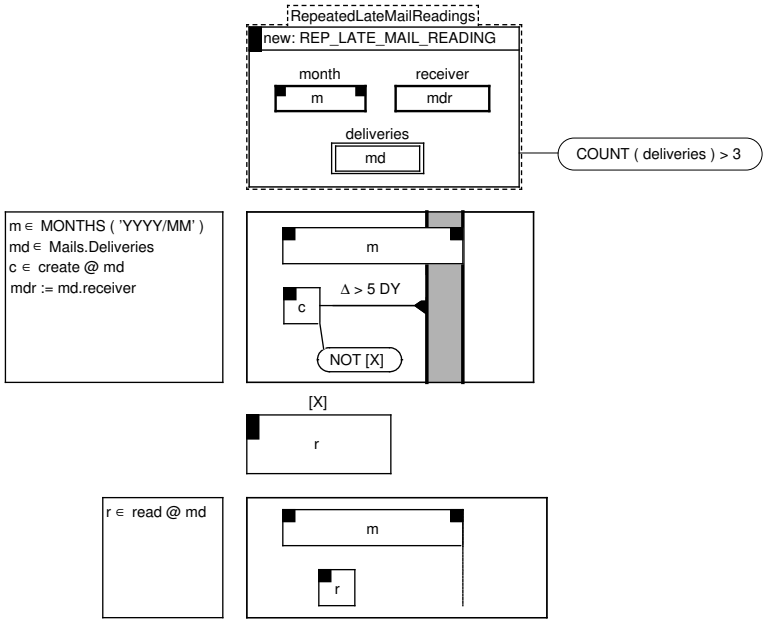


Figure 5.11: Situation diagram for situation class RepeatedLateMailReadings

single-valued properties marked with a boldface border in the result box. Different to the grouping of results in generating query diagrams, the occurrence time or the occurrence period is contained implicitly in the set of variables according to which the result is grouped. Correspondingly, the occurrence bar is depicted with a boldface border.

Remember the evaluation of generating query diagrams. (1) A tuple is generated for each binding to the variables specified in the domain box. (2) Tuples not satisfying the conditions specified in the condition box are discarded. (3) The remaining tuples are grouped. (4) Groups not satisfying the having clauses are discarded. (5) An instance of the result type is created for each remaining group. A property p of an instance generated for a group g is set to the result of evaluating the expression associated with p on g.

Grouping affects the number of new situations generated and provides similar features as the "cumulative context" in event languages based on an operational approach [CKAK94].

Example: Figure 5.11 depicts the situation diagram for situation class RepeatedLateMail-Readings of the global object type. The result is grouped by the occurrence times (depicted by the boldface border of the occurrence bar), by the months (m) and by the receivers (md.receiver). An event is generated for the end of each month and for each user that has received more than three mails (COUNT(deliveries) $>$ 3) that have been delivered more than five days before end of month ($\Delta >$ 5 DY) and that have not been read since then.

Remember that if a time line is drawn, it is connected to a synchronization line by means of a small filled triangle. The time line drawn between the event box c and

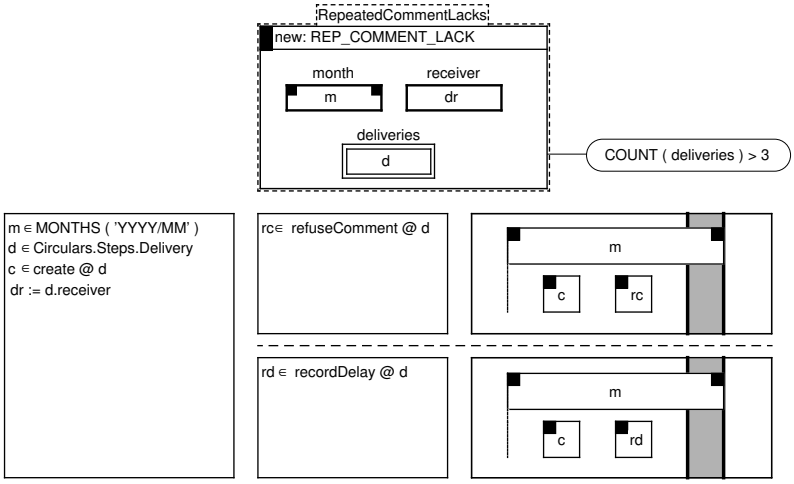


Figure 5.12: Situation diagram for situation class RepeatedCommentLacks

the occurrence bar is connected to the left synchronization line of the occurrence bar by such a triangle.

Note that the occurrence time or the occurrence period of the generated situations need not be determined only from a situation variable completely covered by the occurrence bar. The occurrence time or occurrence period may also be computed of situations aligned to the left or to the right side of the occurrence bar.

Example: Consider again Figure 5.11. Interval box m is aligned to the right side of the occurrence bar. The clock-tick event covered by the occurrence bar is assumed implicitly to be the last clock-tick event of the month.

Like conventional query diagrams, situation diagrams may define several alternative conditions by splitting the situation diagram into an arbitrary number of branches, so-called *alternatives* (cf. Section 4.5.6). Each branch may define its own branch-specific domain box (depicted to the right of the branch-specific condition box and to the left of the situation diagram's domain box) and its own branch-specific auxiliary-condition box (depicted to the right of the condition box).

Example: Figure 5.12 shows the situation diagram for situation class RepeatedCommentLacks of the global object type. Similar to the situation diagram depicted in Figure 5.11, an event is generated for the end of each month and for each user that has received more than three circulars (COUNT(deliveries) > 3) in this month for which a comment has been refused by the receiver or for which commenting already is late.

5.4 Summary

Situation diagrams (in the broader sense) represent a high-level graphical language for the specification of situations in conceptual database design. Situation diagrams allow to specify the event part of business rules. In the following, we summarize the main contributions of situation diagrams and indicate how the requirements on conceptual languages for the specification of business rules as defined in Section 3.3 are met. Of course, only those requirements are addressed that concern the event part of business rules. Chapter 6 represents activation diagrams, which allow to model business rules as a whole, using situation classes for the event part.

1. Whereas most event languages follow an operational approach [BL95, CKAK94, GD94, GJS92a, MPC96], situation diagrams are declarative. Similarly as QBE and SQL are better accepted by end users than Relational Algebra, end users are assumed to prefer a declarative language over operational ones for the definition of situations at the conceptual level.

2. Situation diagrams are homogeneously integrated into a design notation for object-oriented database schemes, Object/Behavior diagrams. Situation diagrams treat, next to events, intervals as "first-class objects." Like other objects, situations are described by types and are collected into classes. The distinction between types and classes is also a major difference to A/OODBMT, a recently proposed design notation for active object-oriented databases [SC96]. Situation classes are always specified locally to object types and thus help to increase a system's degree of locality by organizing situations around objects.

3. Situation diagrams, which derive the extension of logical situation classes, may be selecting or generating; hence, they support the object-preserving as well as the object-generating semantics of object-oriented database views.

4. Situation diagrams visualize the temporal order of situations such that temporal conditions, which are typically used in the definition of situations, are easy to recognize and easy to specify. The conceptual structure of situation diagrams follows the conceptual structure of (O)SQL queries, with which many users are familiar.

5. Sets of basic database situations (e.g., any state in the behavior diagram of an object type) and of basic calendar situations (e.g., year 1997) can be assumed to be predefined.

6. Our time model allows that the smallest temporal unit that is not further decomposable from a user's perspective can be of a coarser granularity than the temporal unit defined by two successive clock ticks of the internal system clock (e.g., time-stamps showing only minutes but not seconds are sufficient for many business applications).

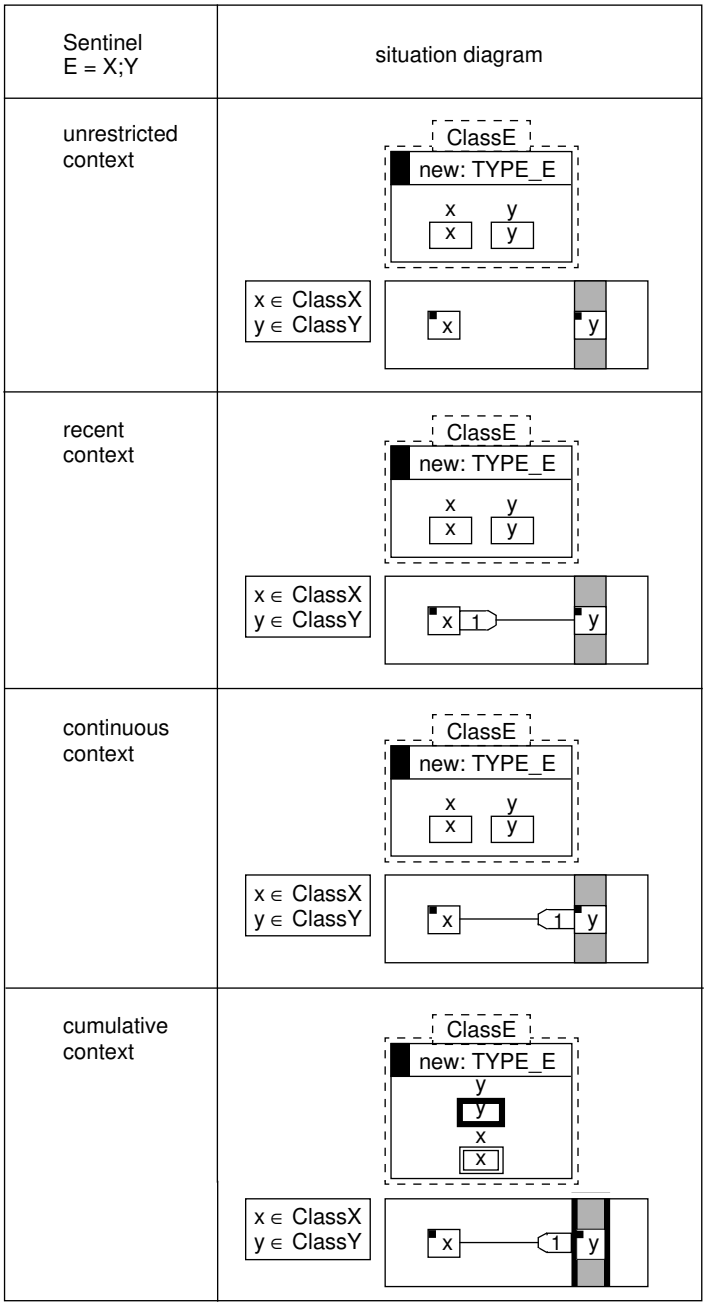


Figure 5.13: The declarative semantics of Sentinel's sequence operator

7. Situation diagrams homogeneously incorporate a calendar. Like any other object, calendar situations are described by a type and are collected in classes. Basic calendar situations are predefined (and can be retrieved using "situation-retrieving functions") and may be referenced also beyond current system time.

8. The notion of "context" has been introduced for operational event languages to give a unique semantics to operators referencing events that may have occurred several times in the past. Being declarative, situation diagrams do not need to distinguish different contexts explicitly. Even more, situation diagrams can be used to illustrate the semantics of event operators for different contexts, which is exemplified for the sequence operator of Sentinel [CKAK94] in Figure 5.13. Note that the "chronicle context" cannot be expressed by situation diagrams *alone* as this would require counting, which is not possible in predicate logic. It can, however, be expressed if situation diagrams, object diagrams, and behavior diagrams are considered together: Every time an initiating event occurs that has to be paired with a single terminating event in the future, an object is created that binds the context of the event occurrence (generally including the event's time-stamp). When a terminating event occurs, the appropriate initiator is determined by considering the created objects and their bindings (especially the time-stamps). Consumption of an initiator is reflected by a change in the life cycle of the initiator, which is effected by performing an activity on it.

9. Further, type and class hierarchies of situations support different levels of abstraction, nested situation diagrams support a modular description of situation expressions.

As far as situation diagrams as presented in this section, they leave some points open:

1. So far, situation diagrams cannot be parametrized and defined recursively. A later extension of situation diagrams should support some form of parametrization and recursion. Parametrization of situation diagrams would help to define generic situation patterns that can be reused several times to define similar situation diagrams (cf. "rule patterns" in [KRSRS96]).

2. Situation diagrams provide a high-level declarative respresentation of situations. Similiarly as SQL queries are usually mapped to Relational Algebra for processing them, situation diagrams can be mapped to an operational event language that is supported by an active database system. Unfortunately, there is no declarative event language directly supported by an active database system. Thus, implementing situation diagrams efficiently on top of some operational event language is a major challenge (cf. Chapter 10).

Chapter 6

Activation Diagrams

Contents

6.1 Start-Rules and Do-Rules 120
6.2 Graphical Elements 123
6.3 Execution Semantics 127
6.4 Summary 131

This chapter presents *activation diagrams* introduced in [LOS97]. Activation diagrams provide a graphical language for specifying business rules (statements about business policies) according to the event-condition-action structure of rules in active database systems. Situation diagrams, which have been presented in Chapter 5, provide the concepts for specifying events upon which rules specified by means of activation diagrams are triggered. Situations (events or intervals) are instances of a situation type, are collected in situation classes, and may be derived by means of a query diagram called "situation diagram." However, modeling business rules at the conceptual level from an external observer's perspective requires different concepts than currently provided by active database systems or by recent approaches to active object-oriented database design (cf. Section 3.3). Activation diagrams and situation diagrams are together referred as Situation/Activation Diagrams and meet the requirements for modeling business rules at the conceptual model. A detailed discussion of these requirements can be found in Section 3.3.

The term "business rule" refers only to rules on business policies and not to consistency rules or to rules on business procedures as introduced in Section 3.1.1. Formulating business rules from an external observer's perspective requires to observe database changes only at the level of transactions that are not further decomposable. In the realm of ACID-transactions, which we assume, all actions performed within a transaction occur instantaneously for the external user and therefore carry the same time-stamp. In contrast to this, active object-oriented database systems usually time-stamp an action-related event with the processing time within the transaction issuing the action.

The event-condition-action structure of rules is reflected in activation diagrams as follows: An activation diagram consists of an event class depicted by a situation-class box, a textual condition over event parameters and the database state, and an activity shown by its activity-specification diagram. An activation arrow drawn from the event to the activity identifies the rule category, which specifies when rules are handled and how rules are executed.

In general, when a rule is handled, a *rule consideration* is triggered for each event in the rule's event class whose occurrence time is the current chronon. When a rule is considered for some event, the condition of the rule is checked. If the condition is satisfied and the precondition of the rule's activity is met for some specified parameter binding, a *rule execution* is triggered. When a rule is executed—which may be some time after it has been triggered—its condition as well as the precondition of its action are checked again. If both still hold, the activity is performed.

Like situations, business rules may be global or local to object types. The locality of a rule determines which event classes and which activities may be used in a rule. A rule local to some type T may be triggered only upon events of an event class local to T and may perform only an activity defined for T. A local rule of some type T is handled for every instance of T. When a local rule is handled for some instance o, a rule consideration is triggered for every event in the rule's event class at o whose occurrence time is the current chronon.

The remainder of this chapter is structured as follows. Section 6.1 introduces the two rule categories, start-rules and do-rules, provided by activation diagrams. Section 6.2 presents the graphical elements of activation diagrams. Section 6.3 explains the execution semantics of both categories of rules. Section 6.4 summarizes the main achievements of activation diagrams and explains how activation diagrams meet the requirements identified for conceptual active object-oriented database design in Section 3.3.

6.1 Start-Rules and Do-Rules

Activation diagrams support two rule categories: start-rules and do-rules. *Start-rules* are handled and considered immediately before every chronon change. Rule executions for start-rules are performed asynchronously to other transactions some time after they have been triggered. Note that at the conceptual level, start-rules are assumed to be handled and considered immediately before every chronon change. A concrete implementation may use different strategies to handle and to consider start-rules (cf. Chapter 10).

Do-rules are handled, considered, and executed immediately after every chronon change and before any other transaction. As do-rules are handled and considered immediately after chronon changes, they may be based only on temporal event classes, whose extension is known already at the start of a chronon. Event classes whose members are calendar events

are *temporal event classes*. Further, logical event classes whose members with occurrence time i can be determined ultimately from database events with occurrence time $< i$ and from calendar events are temporal event classes.

Several kinds of exceptions may arise during rule consideration and execution that prevent a rule to be further considered or executed. Exceptions that may arise during rule consideration are:

1. *Object not enabled during consideration*: The object on which the rule's activity should be performed is not enabled if it is not in every prestate shown for the activity in the life-cycle diagram.

2. *Precondition violated during consideration*: The precondition of an activity is violated for a given set of parameter bindings if object self (the object on which the rule's activity should be performed) is not enabled or if any precondition given explicitly with the activity-specification diagram is not met.

Of course, raising both of these exceptions is not meaningful, as the second exception subsumes the first. Which exception is actually raised, depends on whether a rule is defined experimental or definite.

A rule is defined *experimental* whenever it is anticipated and desired that the rule is ignored without reporting an exception if object self is not enabled. If a rule is experimental and object self is not enabled at the occurrence time of its triggering event, the rule is simply not executed, and no exception is raised. But the exception "precondition violated during consideration" is raised if object self is enabled and the precondition of the rule's activity is violated. Using experimental rules relieves the end user from the task of checking within the condition of a rule whether object self is enabled.

A rule is defined *definite* whenever it is unacceptable that object self is not enabled for some triggering event if the condition of the rule is satisfied. If a rule is definite and object self is not enabled at the occurrence time of the triggering event, the exception "object not enabled during consideration" is raised. If object self is enabled but the precondition of the rule's activity is violated, the exception "precondition violated during consideration" is raised.

Exceptions that may arise during rule execution are:

1. *Object not enabled during execution:* Object self is not enabled for performing the rule's activity. Although object self has been enabled when the rule has been considered for execution, it may no longer be enabled if the rule is actually executed later.

2. *Precondition violated during execution:* The precondition of an activity to be performed is violated. Although the precondition has already been checked when the

rule has been considered for execution, the precondition may no longer be valid if the rule is actually executed later.

3. *Constraint violated:* A precondition or postcondition of some activity invoked (directly or indirectly) by the rule's activity is violated or an integrity constraint is violated while or after performing the activity. In this case, the execution of the activity is aborted. We assume that constraint violations are already taken care of by the passive object-oriented model during conceptual design (cf. [KS96a]) and that, once the precondition of an activity is met, the activity can be performed without causing a constraint violation. Thus, at the external observation perspective, from which business rules are modeled, constraint violations are truly exceptional and indicate a design or implementation error.

4. *Activity terminated:* The activity performed during rule execution is terminated internally by issuing some termination statement.

5. *Rule canceled:* The rule execution is canceled explicitly by a user.

6. *Maximum delay exceeded:* An initiated start-rule has not been executed up to some specified event indicating a maximum delay for executing the rule.

All exceptions, but the last one, can be raised upon information provided already for activities or upon a predefined external event (if the rule is canceled by a user). For the last exception ("maximum delay exceeded"), the event that defines the maximum delay must be specified explicitly with start-rules. Initiating a rule execution does not guarantee that the rule will ever be performed and, thus, that the effects of the rule will ever be reflected in the database. Resource conflicts may delay rule execution or even prevent it. Generally, performing the activity of a rule after some point in time may no longer make sense. At this point in time, referred to as *maximum delay*, a start-rule execution that is still not completed is aborted.

Exceptions are signaled as events, which are collected in predefined event classes. The event classes r_precondViolatedDuringConsid, r_precondViolatedDuringExec, r_constraintViolated, r_activityTerminated, and r_ruleCanceled, are predefined for every rule r. The event classes r_objectNotEnabledDuringConsid and r_objectNotEnabledDuringExec are predefined additionally for definite rules and the event class r_delayExceeded is predefined for start-rules. The exception-event classes of a rule r are always local to the same object type to which r is local. If r is global (i.e., local to the global object type), then the corresponding exception-event classes are also global.

The reaction to be taken when an exception arises depends on the kind of rule. In the case of a start-rule, rule consideration or rule execution is terminated by reporting the exception. Exception events may be used in situation diagrams just as any other event.

If an exception arises during the consideration or execution of a do-rule, the exception is reported as an error, which must be handled before any other transaction is performed.

Performing other transactions before handling the error would violate the semantics of do-rules, which requires that do-rules are executed immediately upon the triggering event before any other transaction. Once an error has been reported, the user must handle the error by issuing a repair transaction that is performed instead of the rule's activity.

6.2 Graphical Elements

The central element of an activation diagram is the *activation arrow*. It symbolizes whether a rule is a do-rule or a start-rule and whether a rule is experimental or definite. How these different kinds of rules are distinguished by means of the activation arrow will be explained later. The graphical elements grouped around the activation arrow (the event class, the condition, and the activity) are introduced before we go into details of the activation arrow. But first of all, we present a few business rules defined by our communication system:

(1) A notice has to expire immediately one week after the notice has been posted or after the notice has been prolonged the last time. A notice may be posted only once. Notices should expire immediately because the information (e.g., an offer containing a special price) presented by the notice may no longer be valid one week after its posting or its last prolongation.

(2) When a notice expires, it has to be removed from the notice boards to which it is posted. If a notice has actually not been removed until the end of the week following the week in which it expired, the administrators of the notice boards to which the notice was posted have to be informed about the delay by sending them a mail.

(3) The receiver of a circular has to give or to refuse to give a comment on the circular. Giving a comment or refusing to give a comment on a delivered circular is said to be late at the end of the second working day after the circular has been delivered. The comment delay has to be recorded at the delivered circular, provided that the receiver is not on vacation. If the delay is not recorded until two days after commenting has become late, the initiator of the circular has to be informed about the delay by sending him/her a mail.

These business rules are implemented by means of activation diagrams as follows.

Example: Rule Expiration of object type NOTICE depicted Figure 6.1 implements business rule (1). Rule NoticeRemoval of object type NOTICE_BOARD depicted in Figure 6.2 implements business rule (2). Rule DelayRecording of object type CIRC_STEP_DELIVERY depicted in Figure 6.3 implements business rule (3). These rules will be explained in more detail throughout this section. For the moment ignore the different representation of the activation arrows and the situation diagrams depicted to the left of the activation arrows.

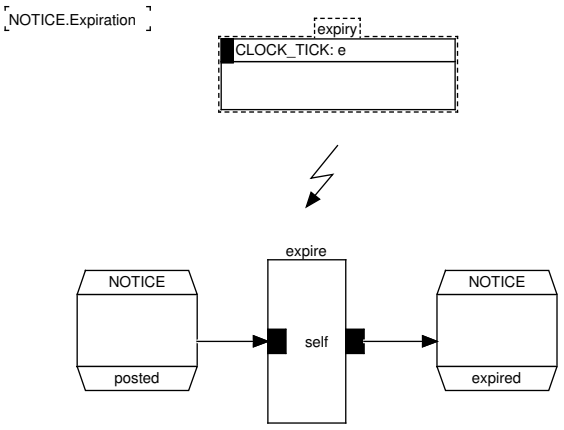


Figure 6.1: Rule Expiration of object type NOTICE

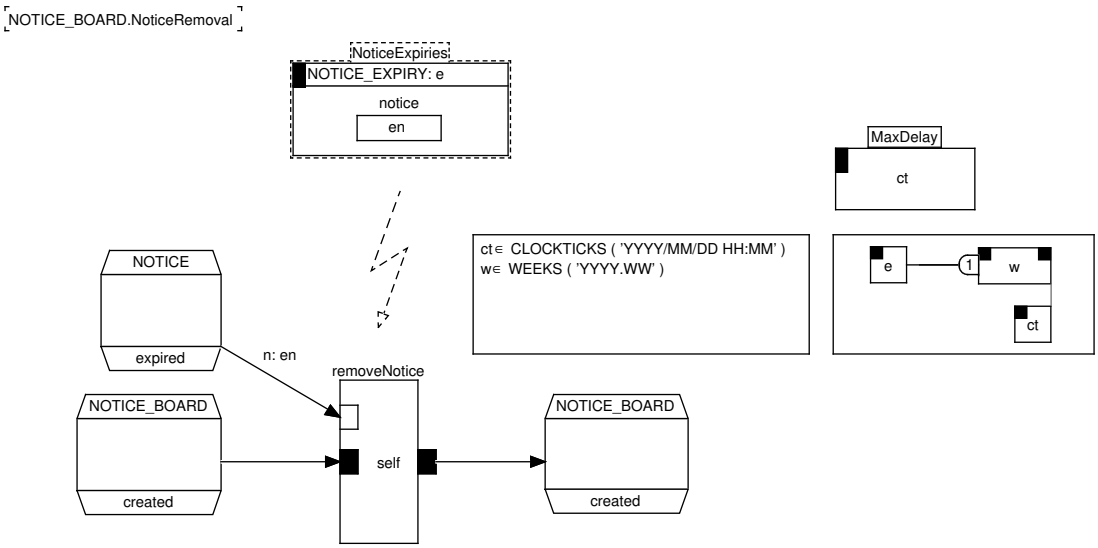


Figure 6.2: Rule NoticeRemoval of object type NOTICE_BOARD

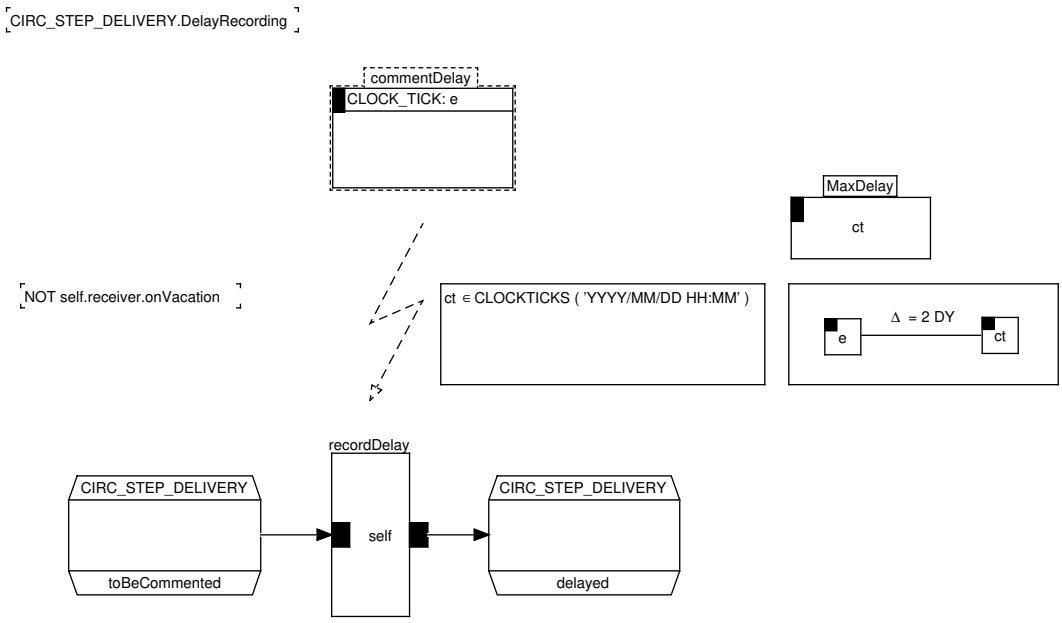


Figure 6.3: Rule DelayRecording of object type CIRC_STEP_DELIVERY

In the activation diagram of a rule, the *event class* of the rule is depicted by a situation-class box that is placed above the tail of the activation arrow and shows the class's member type. Each event in the event class whose occurrence time is the current system chronon is referred to as *triggering event*. The triggering event is named such that it can be referenced in other rule elements. The name of the triggering event is placed behind the type name. Similarly, names may be specified for property values of the triggering event. A name for a property value is given in the box representing the property.

Example: Rule NoticeRemoval (Figure 6.2) is defined upon event class NoticeExpiries, whose member type is NOTICE_EXPIRY. The triggering event is named e, and the expired notice referenced by the triggering event's property notice is named en.

The *condition* of a rule is a boolean expression and is placed to the left of the activation arrow. The condition may refer to properties of the triggering event, to object self and thus to its properties, and to global classes (i.e., classes that are local to the global object).

Example: The condition of rule DelayRecording (Figure 6.3) specifies that a delay is recorded only if the receiver of the circular-step delivery is currently not on vacation.

The *activity* of a rule is depicted by an activity-specification diagram placed below the head of the activation arrow. The activity-specification diagram is complemented by parameter bindings indicating with which actual parameters the activity should be performed during

rule execution. Actual parameters are placed after the names of formal input parameters separated by a colon. Any expression referring to property values of the triggering event (represented by the names specified with the member type of the event class), to literals, to properties of object self, or to properties of the global object may serve as actual parameter. Object self is implicitly assigned to the primary input port of the activity. Notice that the action of a rule is not restricted to an activity of the life-cycle diagram, but it may also be a compound activity (cf. Section 4.4.4).

Example: Rule NoticeRemoval (Figure 6.2) performs activity removeNotice on the notice board for which the rule is handled. The notice that has to be removed, represented by name en, is assigned to the input parameter n of activity removeNotice.

As already mentioned above, the activation arrow of an activation diagram symbolizes whether the specified rule is a do-rule or a start-rule and whether the rule is experimental or definite. Start-rules are depicted with an activation arrow with an unfilled head. Do-rules are depicted with an activation arrow with a filled head. Further, an experimental rule is depicted with a dashed activation arrow and a definite rule with a solid activation arrow.

Example: Rule DelayRecording (Figure 6.3) represents an experimental start-rule. It is handled immediately before every chronon change and is considered for every circular-step delivery whose local event class commentDelay refers to an event whose occurrence time is the current chronon. If the receiver is not on vacation, a rule execution that performs activity recordDelay is initiated, provided that the delivery is enabled, i.e., that it is in state toBeCommented. Otherwise, activity recordDelay is not executed and no exception is raised because the rule is experimental. The situation diagram depicted to the left of the activation arrow will be explained later.

Rule Expiration (Figure 6.1) represents a definite do-rule. It is handled immediately after every chronon change for each notice and is considered for a notice if the notice's local event class exiry refers to an event whose occurrence time is the current chronon. If the notice is in state posted, the rule is executed immediately and the notice expires. Otherwise, if the notice is not in state posted, an exception event is raised and inserted into class Expiration_precondViolatedDuringConsid, which is local to the notice.

The maximum delay for start-rules is specified graphically to the right of the activation arrow by means of a situation diagram named MaxDelay that selects a single calendar event. The triggering event of the rule may be used in the situation diagram. The situation diagram may refer only to situations that are already available when the triggering event occurs.

Example: Rule DelayRecording (Figure 6.3) specifies a maximum delay. If the rule execution has not been completed within two days after the triggering event e has

occurred, the rule execution is aborted, and a delay exception is raised by inserting the excpetion event into event class DelayRecording_delayExceeded, which is local to the circular-step delivery. An additional rule that is triggered upon the occurrence of such a delay event can inform the initiator about the delay that could not be recorded.

6.3 Execution Semantics

As activation diagrams provide a graphical language for specifying business rules during conceptual design, we intentionally describe their execution semantics declaratively and not by an operational model.

At first, we specify the semantics of do-rules and start-rules by using an ideal execution model in which transactions are performed instantaneously, i.e., in zero time, but not simultaneously. Then we relax these assumptions and define which schedule of rule executions in a realistic execution model correctly reflects the semantics of a schedule of rule executions in the ideal model. For simplicity, we will consider only global rules. An extension of the presented semantics to local rules is straightforward. For each local rule defined at some type T, a private copy of that rule must be introduced for every instance of T.

We define the semantics of do-rules and start-rules declaratively by considering the sequence of database states as they are observed at chronon changes and by identifying which kind of transactions must be present in which order in a log of committed transactions.

The following kinds of transactions may appear in the transaction log: *user transactions* (U), which are initiated by the user, *do-rule transactions* (D), which comprise the condition/activity pair of a do-rule (i.e., the condition of the rule is checked and the activity of the rule is executed if the rule condition is met), *start-rule transactions* (S), which comprise the condition/activity pair of a start-rule, *halt transactions* (H), which report the errors for a do-rule, *repair transactions* (R), which are initiated by the user upon a halt transaction, and *exception-reporting transactions* (E), which generate exception events for start-rules. Further, we consider a subset E^d of E, the set of transactions reporting a delay-exceeded exception.

The execution of a start-rule r triggered by some event e is reflected in the log either by a start-rule transaction $S_{(r,e)}$ or by an exception-reporting transaction $E_{(r,e)}$. If the start-rule transaction $S_{(r,e)}$ is aborted due to some exception, it will not appear in the log of committed transactions. But then, a transaction $E_{(r,e)}$ reporting the exception will be present in the log. Similarly, the execution of a do-rule r triggered by some event e is reflected in the log either by a do-rule transaction $D_{(r,e)}$ or by a halt transaction $H_{(r,e)}$ together with a repair transaction $R_{(r,e)}$.

We denote the database state as it is observed at the change from chronon i to chronon $i+1$

as db_i. Chronons, which are the smallest temporal unit that is not further decomposable from a user's perspective, may be of a coarser granularity than the temporal unit defined by two successive clock-ticks of the internal system clock. E.g., time-stamps showing only minutes but not seconds are sufficient for many business applications. All transactions that are performed during a chronon i receive the time-stamp of the start time of chronon i, i.e., i. Similarly, a letter posted just a few seconds before 12:00 is postmarked with a time-stamp of 11:59 if the chronon-size of the stamp is one minute. The set of database events that appear in the database of chronon i can be determined only at the end of chronon i. In our example, all letters with postmark 11:59 are known at the earliest at 12:00 sharp.

The rules defined for a database and a set of user transactions are executed correctly up to some database state db_i if the log of committed transactions satisfies the following five criteria (*ideal model*):[1]

1. *Transaction priorities:* Within transactions carrying the same time-stamp, every transaction with a higher priority is executed before any transaction with a lower priority. Any transaction from the set $D \cup H \cup R$ has a higher priority than any transaction from the set $U \cup S \cup (E \setminus E^d)$, and any transaction from the set $U \cup S \cup (E \setminus E^d)$ has a higher priority than any E^d-transaction.

2. *Do-rule consideration:* Assume (1) some database state db_{j-1} for $j \leq i$ contains an event e with occurrence time j, (2) event e belongs to the event class of some do-rule r, and (3) the condition of rule r is satisfied. Then, the log contains exactly one $H_{(r,e)}$-transaction and exactly one $R_{(r,e)}$-transaction in the following cases: (i) the rule is definite and the activity of the rule is not enabled, (ii) the activity is enabled, but the precondition of the activity is violated. Both transactions must carry time-stamp j.

3. *Do-rule execution:* Assume (1) some database state db_{j-1} for $j \leq i$ contains an event e with occurrence time j, (2) event e belongs to the event class of some do-rule r, (3) the condition of the rule is satisfied. Then the log contains exactly one $H_{(r,e)}$-transaction and exactly one $R_{(r,e)}$-transaction if (i) the rule is definite and the object on which the activity has to be performed is not enabled or (ii) the object is enabled, but the precondition of the activity is violated. The log contains either exactly one $H_{(r,e)}$-transaction together with exactly one $R_{(r,e)}$-transaction or exactly one $D_{(r,e)}$-transaction if the object is enabled and the precondition of the activity is satisfied. The transaction(s) must carry time-stamp j.

4. *Start-rule consideration:* Assume (1) some database state db_j for $j < i$ contains an event e with occurrence time j, (2) event e belongs to the event class of some start-rule r, (3) the condition of the rule r is satisfied, and (4) database state db_j contains a maximum delay event d for r where the occurrence time of d is k. Then, if $i < k$,

[1] For simplicity, we assume that an exception-reporting transaction always reports the kind of exception encountered.

the log contains at most one $E_{(r,e)}$-transaction, and if $i \geq k$, the log contains exactly one $E_{(r,e)}$-transaction in the following cases: (i) the rule is definite and the activity of the rule is not enabled, (ii) the activity is enabled, but the precondition of the activity is violated. The time-stamp of the $E_{(r,e)}$-transaction must be $> j$ and $\leq k$.

5. *Start-rule execution:* (1) some database state db_j for $j < i$ contains an event e with occurrence time j, (2) event e belongs to the event class of some start-rule r, (3) the condition of the rule is satisfied, and (4) database state db_j contains a maximum delay event d for r where the occurrence time of d is k. Then, if $i < k$, the log contains at most one $E_{(r,e)}$-transaction, and if $i \geq k$, the log contains exactly one $E_{(r,e)}$-transaction if (i) the rule is definite and the object on which the activity has to be performed is not enabled or (ii) the object is enabled, but the precondition of the activity is violated. If $i < k$, the log contains at most one $S_{(r,e)}$-transaction or at most one $E_{(r,e)}$-transaction, and if $i \geq k$, the log contains either exactly one $S_{(r,e)}$-transaction or exactly one $E_{(r,e)}$-transaction if the object is enabled and the precondition of the activity is satisfied. The time-stamp of the transaction contained in the log must be $> j$ and $\leq k$. If the log contains an $E^d_{(r,e)}$-transaction, its time-stamp must be k.

We assumed for our ideal execution model that transactions are executed instantaneously. We now drop this assumption but we still assign unique time-stamps to them as follows: D-transactions, H-transactions, and R-transactions receive as time-stamp the occurrence time of the triggering event of the rule execution to which these transactions belong. E^d-transactions are time-stamped with the occurrence time of the maximum delay event they report. All remaining kinds of transactions receive as time-stamp the current reading of the system clock truncated to the granularity of the chronon size (e.g., if the chronon size is one minute, seconds and milliseconds are truncated) when the commit-request is issued by the transaction. We also drop the assumption that transactions are not executed concurrently. To provide a reliable rule execution, all transactions (except user transactions) are automatically restarted in the case of a concurrency control conflict. Now, we can give the execution semantics for a realistic execution model.

A set of do-rules and start-rules defined for a database and a set of user transactions has been executed correctly up to database state db_i if the following criteria are met after all transactions with a time-stamp $\leq i$ have been aborted or committed (*realistic model*):

1. *Temporal faithfulness:* Transactions with different time-stamps are serialized according to their time-stamps, i.e., the effect of executing any two transactions T and T' such that the time-stamp of T is less than the time-stamp of T' must be the same as if transaction T' had been executed after T in serial order. Thus, user transactions and transactions performing rule executions are serialized in an order that rule modelers and system designers intuitively expect.

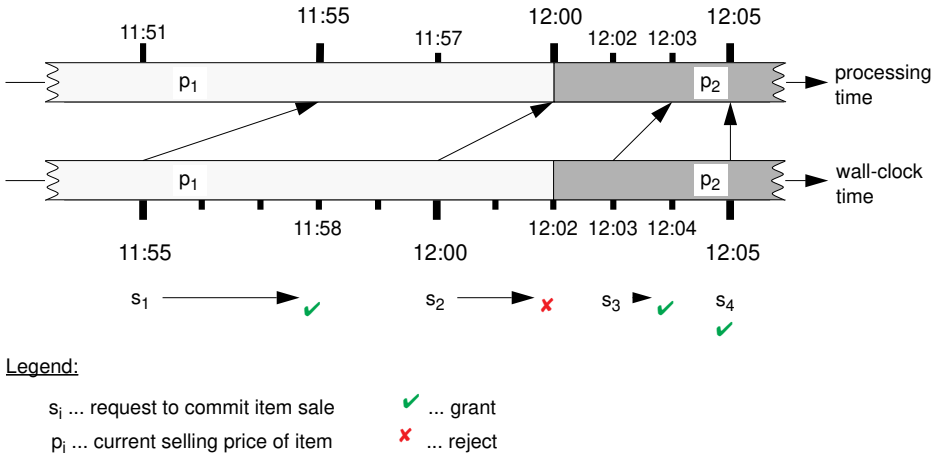


Figure 6.4: Schematic representation of temporally faithful execution

2. *Priority observation:* Transactions with the same time-stamp are serialized according to the transaction priorities defined under (1) for the ideal model.

3. *Completeness:* The transaction log satisfies the requirements defined under (2–5) for the ideal model.

As activation diagrams provide a graphical language for specifying business rules during conceptual design, we have intentionally described their execution semantics declaratively and not by an operational model. Different strategies will be applied when a rule manager is implemented for activation diagrams or when activation diagrams are mapped to the rule language of some active database management system. Aborts of transactions need to be monitored and transactions aborted due to concurrency conflicts need to be restarted.

Rule consideration and rule execution may lag behing wall-clock time because of concurrency conflicts or because of a too heavy workload of the rule manager. The time currently processed by the rule manager is called *processing time*. If rule consideration and rule execution of do-rules lag behind wall-clock time, the commit of user transactions needs to be delayed, unless conflicts with do-rule transactions can be excluded. Younger user transactions that are in conflict with older do-rule transactions must be aborted, similarly as transactions with a lower time-stamp abort transactions with a higher time-stamp in the wound-wait protocol. Finally, if do-rule transactions preclaim all needed resources, conflicts with younger user transactions can be excluded and, thus, younger user transactions may commit despite that older do-rule transactions have not yet committed. See Chapter 10 for more details of implementation strategies.

Example: Assume a business rule that requires that selling prices are adjusted at 12:00 noon, that sales transactions performed before 12:00 noon (i.e., a customer has decided to buy and thus requested to commit a sales transaction before 12:00 noon)

sales request	DB action	price	time-stamp
s_1	grant	p_1	11:55
s_2	reject		
s_3	grant	p_2	12:03
s_4	grant	p_2	12:05

Table 6.1: Statistics of temporally faithful executions

should be serialized before the transaction adjusting the selling prices, and that sales transactions performed after 12:00 noon should be serialized after the transaction adjusting the selling prices. This business rule is modeled by means of a do-rule that changes the price upon the 12:00 clock-tick.

Figure 6.4 illustrates this setting. The upper time axis denotes the proccessing time, the lower time axis denotes the wall-clock time. The price change is denoted by two prices p_1 and p_2, the four requests to commit a sales transaction are represented by s_1 to s_4. Table 6.1 shows the statistics of the sales transactions. E.g., sales transaction s_1 requests to commit at 11:55 wall-clock time (11:51 processing time), it is granted at 11:58 wall-clock time (11:55 processing time) with price p_1 and time-stamp 11:55. Sales transaction s_2 requests to commit at 12:00 wall-clock time (11:57 processing time), but is rejected at 12:02 wall-clock time (12:00 processing time) because otherwise it would be granted with a time-stamp $\geq$ 12:00 but with the old price p_1.

6.4 Summary

Activation diagrams and situation diagrams together constitute Situation/Activation Diagrams, which fill the need for a high-level graphical language to specify business rules in conceptual active object-oriented database design. In the sequel, the main contributions of activation diagrams are summarized and related to the requirements on conceptual languages for the specification of business rules as defined in Section 3.3.

1. Activation diagrams represent a graphical language to represent business rules following an ECA structure. The event class, the rule condition, and the action activity are grouped around the activation arrow, which symbolizes the rule category (start-rule or do-rule).

2. The rules described by means of activation diagrams may be global or local to object types. A local rule of some type T may be defined only upon an event class local to T and may perform only an activity defined for T. A local rule of some type T

is handled for every instance of T. A global rule is local to the global object type, whose only instance represents the database.

3. The underlying execution model provides a common occurrence time-stamp for all activities performed in one transaction in order to formulate business rules from the external observer's perspective.

4. Rules are executed reliably because the transactions performing rule executions are restarted automatically if rules are aborted due to serialization failures.

   Rules are executed controllably because appropriate exception events are raised if specific exceptional situations (e.g., the precondition of the rule action is violated) occur during rule execution. Exception events can again be handled by business rules.

5. Depending on the rule category, rules either are executed asynchronously to the event occurred until a certain maximum delay (start-rules) or are executed immediately upon temporal events (do-rules). The execution model guarantees a temporally faithful execution of business rules (modeled by means of do-rules) and user transactions even if the processing time lags behind the wall-clock time.

Chapter 7

Mapping Environment

Contents

7.1 Target Model 133

7.1.1 Rationale 134

7.1.2 TriGS 134

7.2 Mapping Strategy 140

7.3 Related Work 146

The goal of this chapter is to present an environment for mapping a conceptual model specified with Active Object/Behavior Diagrams to a logical model supported by an active object-oriented database system. The chapter is divided into three sections: Section 7.1 selects a target model for the mapping. It motivates the selection and outlines the selected system. Section 7.2 discusses different mapping strategies and presents the adopted approach. Finally, Section 7.3 concludes the chapter by giving a survey of related work in the area of design transformations.

7.1 Target Model

When selecting a target model for mapping Active Object/Behavior Diagrams, the question arises, whether mapping should be done directly to the data model of a particular active object-oriented database system or to an intermediate model. This question is considered and a target system is selected in Section 7.1.1. In Section 7.1.2, the selected target system is presented.

7.1.1 Rationale

The goal of this thesis is not to present a mapping tailored to the peculiarities of a particular system. Rather, insights should be gained for mapping Active Object/Behavior Diagrams to the concepts provided by active object-oriented database systems (cf. Section 2.2) in general. An intermediate model reduces the number of necessary considerations substantially if mappings from the intermediate model to particular active object-oriented database systems already exist. Thus, whether transformation should be done directly to a particular system or to an intermediate model, depends on the availability of an appropriate intermediate model.

In the area of object-oriented database systems, there are standardization efforts. With *The Object Database Standard: ODMG 2.0* (cf. [CB97]) there is a well-accepted standard of a data model and a query language for object-oriented database systems. By contrast, there is no such standard in the area of active object-oriented database systems. A first attempt to establish a common understanding of an active database system is made by a manifesto (cf. [Con96]). This manifesto, however, is much too informal to serve as a standard. The different research prototypes diverge from each other in their knowledge and execution models considerably (cf. Section 2.2).

Since no appropriate intermediate model has emerged so far, we decided to show how Active Object/Behavior Diagrams can be mapped to an existing system. The immediate follow-up question is: Which system out of a series of research prototypes should be selected as the target system? We selected TriGS (cf. [KRSRV94, Hie96, Ret96, KRSR98]), an active extension to the commercial object-oriented database system GemStone (GemStone Systems, Inc.). A major reason to select TriGS (TriGS=Trigger System for GemStone) has been that TriGS is developed as a prototype at the University of Linz. Source code, documentation, and support by developers are available for TriGS. TriGS provides a sophisticated knowledge and execution model for active rules that is comparable with the knowledge model and execution model provided by other prototypes of active object-oriented database systems. Thus, we assume that lessons learned from mapping Active Object/Behavior Diagrams to TriGS can be transferred to mapping them to other active object-oriented database systems.

7.1.2 TriGS

GemStone with its active extension TriGS has been selected as the target model for mapping Active Object/Behavior Diagrams. While information on GemStone can be found in the available manuals (especially in [Gem96c]), the features of TriGS that are necessary for understanding this and the following chapters will be outlined briefly in the following. The focus will be on the knowledge model and the execution model of TriGS. For details on the architecture and the implementation of TriGS, the interested reader is referred to [KRSRV94, Hie96, Ret96, KRSR98].

```
<rule_definition> :: =
      DEFINE RULE <rule_name>
        ON <eventSelector>
        IF <condition> THEN
        [[WAIT UNTIL] ON <eventSelector>]
        EXECUTE [INSTEAD] <action>
        [WITH PRIORITY <number>]
        [TRANSACTION MODES ([C: SERIAL|PARALLEL ,] [A: SERIAL|PARALLEL])]
      END RULE <rule_name>.
```

Figure 7.1: Syntax of a TriGS rule ([KRSR98])

Figure 7.1 shows the basic syntax of a rule in TriGS. In the Backus-Naur Form (BNF) used for specifying the syntax, the symbols "`::=`", "`[ ]`", "`|`", and "`.`" are meta symbols, and angle brackets delimit non-terminal symbols.

Every TriGS rule has a user-defined name. Like most active database systems, a TriGS rule consists of an event part, a condition part, and an action part. Unlike other systems, however, TriGS extends the ECA paradigm by allowing an additional event component between the condition and the action, resulting in an ECEA structure. The two event parts are called event selectors. A *condition event selector* determines the events that trigger the evaluation of a rule's condition, an *action event selector* determines the events that trigger the execution of a rule's action. Upon signaling of an event that corresponds to the condition event selector of a rule, the rule's condition is evaluated. If the condition evaluates to true, and if afterwards an event that corresponds to the action event selector is signaled, the rule's action is executed. In the case that no action event selector is specified, the action is executed immediately after the condition has evaluated to true. If several condition evaluations and action executions are triggered simultaneously, rule priorities determine the order according to which the condition evaluations and action executions are started. The transactional relationship between event signaling and condition evaluation on the one hand and between event signaling and action execution on the other is determined by transaction modes.

Event selectors, conditions, actions, rule priorities, and transaction modes are considered in more detail now:

Event Selector. An event selector represents either a primitive event or a composite event. TriGS supports five kinds of primitive events: message events, value events, transaction events, time events, and explicit events.

1. *Message events:* An event selector representing a message event refers to a *triggering class* and to a *triggering method*. The triggering class denotes the class for which event occurrences are signaled. Keyword `IS_KIND_OF` indicates that event occurrences

are signaled for the triggering class and its subclasses, keyword `IS_MEMBER_OF` indicates that event occurrences are signaled for the triggering class without its subclasses. If no keyword is provided, the former is assumed. The triggering method denotes the method that raises the event when the corresponding message is received. If the triggering method is a class method, keyword `CLASSMETHOD` has to be indicated with the event selector. Additionally, it has to be indicated whether the message event has to be signaled before or after execution of the triggering method (keywords `PRE` and `POST`).

Example: Assume a class `Item` that defines method `recordSale:` for its instances, and consider the following event selector (referred to as `E`):

```
PRE(Item, recordSale:  noOfItems)
```

Event `E` is signaled every time message `recordSale` is sent to a direct or indirect instance of class `Item`, more precisely, before the method bound to `recordSale:` is executed.

2. *Transaction events:* Since in GemStone transactions are controlled by sending messages to class `System`, transaction events correspond to special message events. TriGS provides predefined transaction events that are signaled before/after the begin of a transaction (selectors `Before_BoT` and `After_BoT`), before/after the abort of a transaction (selectors `Before_Ab` and `After_Ab`), and before/after the commit of a transaction (selectors `Before_Cm` and `After_Cm`).

3. *Time events:* Absolute time events are specified in the form `TIME(yy-mm-dd hh:mm:ss)`. If a part of the specification is omitted, the omitted part is filled automatically with the lowest possible value. Unlike other active object-oriented database systems, TriGS considers relative time events and periodical time events as composite events (see below).

4. *Explicit events:* The event selector for an explicit event defines the name of the explicit event and, optionally, a set of parameters. An explicit event is signaled by sending message `raiseEvent` to the TriGS system.

Table 7.1 summarizes the parameters that are bound when primitive TriGS events occur. These parameters can be accessed within the condition part or action part of a rule via their names (given in parenthesis in Table 7.1).

For any kind of event, a guard may be specified that restricts the events able to trigger the condition or action of a rule. In the presence of a guard, the evaluation of a rule's condition or the execution of a rule's action is triggered only if the signaled event occurrence satisfies the guard. A guard represents a predicate over the parameters of an event. From within a guard expression, the event occurrence for which the guard has to be checked can be referred to as `event`.

parameter/event	message event	transaction event	time event	explicit event
occurrence time (`time`)	+	+	+	+
session identifier (`sid`)	+	+	−	+
transaction identifier (`tid`)	+	+	−	+
name of event (`name`)	−	−	−	+
optional parameters	−	−	−	+
triggering object (`trgObj`)	+	−	−	−
triggering method (`trgMeth`)	+	−	−	−
parameters of method	+	−	−	−
return value (`retVal`)	+	−	−	−

Table 7.1: Parameters of primitive TriGS events ([Ret96])

Example: The event selector introduced in the above example could be refined to:

```
PRE(Item, recordSale: noOfItems)
   [event trgObj qnttyInStock < 100]
```

Now, the event is signaled only if the quantity in stock of the sold item is below 100. The quantity in stock is obtained by performing method `qnttyInStock` on an item.

TriGS provides event operators to construct composite events from two or more component events. The component events can be primitive events or again composite ones. The composite events supported by TriGS are disjunction events, conjunction events, sequence events, cumulation events, relative time events, and periodic time events:

1. *Disjunction:* A disjunctive event ($\texttt{E}_1$ `OR` $\texttt{E}_2$) occurs when $\texttt{E}_1$ or $\texttt{E}_2$ occurs.

2. *Conjunction:* A conjunctive event ($\texttt{E}_1$ `AND` $\texttt{E}_2$) occurs when $\texttt{E}_1$ and $\texttt{E}_2$ have occurred, independent of their order. A conjunction event is allowed to contain a negated event expression. For example, ($\texttt{E}_1$ `AND` $\texttt{E}_2$ `AND NOT` $\texttt{E}_3$) requires that $\texttt{E}_1$ and $\texttt{E}_2$ have occurred, without an occurrence of $\texttt{E}_3$ in between.

3. *Sequence:* A sequential event ($\texttt{E}_1;\texttt{E}_2$) occurs when $\texttt{E}_1$ and $\texttt{E}_2$ have occurred in the specified order. Like a conjunctive event, also a sequential event is allowed to contain a negated event expression.

4. *Cumulation:* A cumulative event `COLL`($\texttt{E}_1, \texttt{E}_2, \texttt{E}_3$) cumulates all occurrences of event $\texttt{E}_2$ that have occurred within the closed time interval $[\texttt{E}_1 \ldots \texttt{E}_3]$. The cumulative event occurs only once when event $\texttt{E}_3$ is signaled.

5. *Relative Time:* A relative time event `REL`($\texttt{E}_1$,`distance`) occurs at the point in time that is obtained by adding the value of `distance` to the occurrence time of $\texttt{E}_1$.

6. *Periodic Time:* A periodic time event `PER(`$\mathtt{E}_1$`,distance,maxPeriods,maxTime)` occurs for the first time, like a relative time event, at the point in time that is obtained by adding the value of `distance` to the occurrence time of $\mathtt{E}_1$. But then, the periodic time event occurs again and again with the specified distance. Signaling of the periodic time event either terminates after a specific number of periods (`maxPeriods`), at a specific point in time (`maxTime`), or never if non of the both terminating bounds is specified.

Composite events in TriGS are internally represented as trees. Within a guard expression, within the condition, or within the action of a rule the components of a composite event can be accessed by means of the keywords `left` and `right`.

TriGS supports different event contexts according to which event occurrences are combined to composite events. TriGS borrows the basic principles of event contexts from [CKAK94], where contexts have been first introduced as a mechanism for implementing different consumption modes. In the following, the contexts supported by TriGS are briefly introduced. The term *initiating event* is used to refer to an event that according to the event selector may start the detection of a composite event, and the term *terminating event* is used to refer to an event that according to the event selector may complete the detection of a composite event. If an event occurrence is *consumed*, it cannot contribute to a future instance of a composite event. More details on event contexts in TriGS can be found in [Ret96].

1. *General context* ("`/GEN`"): In the general context, there are no restrictions on the detection of composite events. Any combination of the occurred events is allowed. Events are not consumed.

2. *Continuous context* ("`/CONT`"): In the continuous context, each occurrence of an initiating event starts the detection of an own composite event. Upon occurrence of a terminating event, all the composite events whose detection has been started occur simultaneously. All the initiating event occurrences and the terminating event occurrence are consumed.

3. *Chronicle context* ("`/CHR`"): In the chronicle context, each occurrence of an initiating event starts the detection of an own composite event. Upon occurrence of a terminating event, the composite event whose detection has been started first occurs. The initiating event occurrence and the terminating event occurrence are consumed.

4. *Reverse chronicle context* ("`/REV`"): In the reverse chronicle context, each occurrence of an initiating event starts the detection of an own composite event. Upon occurrence of a terminating event, the composite event whose detection has been started most recently occurs. The initiating event occurrence and the terminating event occurrence are consumed.

5. *Recent context* ("`/REC`"): In the recent context, only the most recent occurrence of an initiating event can contribute to the occurrence of a composite event. Thus, upon occurrence of an initiating event, an existing older occurrence is discarded. When a terminating event occurs, it is paired with the most recent occurrence of the initiating event. The initiating event occurrence is *not* consumed.

6. *Reverse recent context* ("`/FIRST`"): In the reverse recent context, only the oldest occurrence of an initiating event can contribute to the occurrence of a composite event. Subsequent occurrences of an initiating event are discarded. When a terminating event occurs, it is paired with the oldest occurrence of the initiating event. The initiating event occurrence is *not* consumed.

In the specification of a composite event selector, the keyword denoting the applied context is appended to the selector. The keywords that are used to denote the different contexts are given in parenthesis in the above enumeration. If no keyword is indicated, the general context is applied. In the case that a guard is part of a composite event selector, the keyword denoting the applied context is given before the guard. By means of the "guard/context modification flag" (GCM flag), TriGS allows to specify that an indicated guard is considered already when events are composed. The GCM flag can be set by adding the suffix `GCM` to the keyword that denotes the applied context.

Example: Assume a library system containing class `Book`, and further assume that class `Book` defines method `reserveFor: aUser` and method `return` for its instances. The following event selector expresses the scenario that a reserved book is returned and that the book is now available for the user who has first reserved the book:

```
(POST(Book, reserveFor:);POST(Book, return))
   /CHRGCM [event left trgObj == event right trgObj]
```

The event selector specifies a sequence event. An initiating event is signaled after method `reserveFor:` is performed on a book, a terminating event is signaled after method `return` is performed on a book. The chronicle context is applied for detecting composite events. Since the GCM flag is set, the indicated guard is considered already during event composition. This means that an occurrence of the terminating event is paired with the oldest occurrence of the initiating event that satisfies the indicated guard. In the example, the return of a book is paired with the oldest reservation of the same book.

Condition. The condition part of a rule is specified either as a simple condition or as a result condition. A *simple condition* represents a boolean expression. Thus, a simple condition is satisfied if the boolean condition evaluates to true. A *result condition* represents a database query, which is specified as a method invocation selecting objects. A result condition is satisfied if the result of the database query is not empty. In the condition part of a rule, the condition triggering event can be referred to as `event`.

Action. The action part of a rule is specified as a sequence of method invocations. Within the action part, keywords are predefined to refer to different elements of rule triggering: `conditionEvent` refers to the condition triggering event, `conditionResult` refers to the result of the condition evaluation (meaningful only in the case of a result condition), and `actionEvent` refers to the action triggering event (meaningful only if an action event selector has been indicated). By default, a session that signals an event triggering the condition of a rule is not blocked while the triggered rule is waiting for an event that triggers its action. The signaling session can be blocked, however, if keyord `WAIT UNTIL` is indicated. In the case of a rule that is triggered by a message event, keyword `INSTEAD` allows to specify that the action is executed instead of the method signaling the message event.

Transaction Mode. Most existing active database systems use coupling modes to specify both the temporal and the transactional relationships between event occurrence, condition evaluation, and action execution. TriGS, in contrast, clearly distinguishes between the specification of the time semantics and the specification of the transaction semantics. The time semantics is specified using an event-based approach, which allows to specify arbitrary events that trigger the condition evaluation and the action execution of a rule. The transaction semantics are specified by so-called *transaction modes*. The transaction mode specified for the condition is relative to the condition triggering event, whereas the transaction mode specified for the action is relative to the action triggering event. The supported transaction modes are `SERIAL` and `PARALLEL`. Transaction mode `SERIAL` specifies that the condition evaluation/action execution is done directly within the triggering transaction. Transaction mode `PARALLEL` specifies that a new independent top-level transaction is generated within which the condition evaluation/action execution is done.

Two restrictions apply to the specification of transaction modes: First, the action of a rule cannot be executed serially if the condition of the rule is evaluated in parallel. Second, condition evaluation/action execution cannot be done serially if the triggering event occurs outside of a transaction (time events) or if the triggering event is a composite event and the partial events originate from different transactions.

Rule Priority. If several rules are triggered simultaneously, i.e., if conditions and actions of different rules are triggered at the same time, the rules are scheduled according to their priorities. Due to inter-rule parallelism, however, for parallel condition evaluations and action executions it is not guaranteed that they are finished in scheduling order.

7.2 Mapping Strategy

After we have selected GemStone/TriGS as the target model for mapping Active Object/Behavior Diagrams, we will now discuss the basic mapping strategy. Throughout

the rest of this chapter and also in the following chapters, we will follow the typographic convention to show names of elements that are specified with Active Object/Behavior Diagrams in sans serif font (e.g., object type PERSON, property age, activity retire) and names of GemStone/TriGS elements in monospace typeface (e.g., class `Person`, instance variable `age`, method `retire`). Important characteristics of an operational system are determined during logical design. Among others, it is determined whether the operational system follows an interpreter approach or a translator approach:

- *Interpreter approach:* In an interpreter approach, the kernel of the logical model comprises a small set of generic methods and generic data structures. The behavior of the generic methods relies on information stored in the conceptual model (serving as data dictionary). Therefore, the conceptual model must either be part of the logical model, or information from the conceptual model has to be replicated in the logical model. The implementation of the generic methods often resembles decision tables. Different actions are taken depending on the information retrieved from the conceptual model. In other words, the generic methods *interpret* the conceptual model.

  Example: In an interpreter approach, the generic behavior for storing and manipulating property values may be factored out in the abstract GemStone/TriGS class `AOBDObject`. This class serves as direct superclass for every GemStone/-TriGS class that represents an object type specified in the conceptual model. (Note: If all classes that represent object types specified in the conceptual model are mapped to direct subclasses of `AOBDObject`, there is no isomorphic correspondence between the object type hierarchy in the conceptual model and the class hierarchy in the logical model.) Figure 7.2 shows parts of class `AOBDObject` in a simplified form. By means of class instance variable `description`, a class refers to its description. The description may be part of the stored conceptual model or a replicate thereof in the logical model. Instances, which are created only of subclasses of `AOBDObject`, store the value of all properties in instance variable `properties`, a dictionary holding pairs of property name and property value. When a new instance is created, the new instance is initialized by means of method `initialize`. Method `initialize` interpretes the conceptual model in that it retrieves meta information about all properties (`self class description allProperties`) and initializes instance variable `properties` accordingly. The sketched methods `getProperty:`, `assign:toProperty:`, `add:toProperty:`, and `remove:fromProperty:` represent a subset of the generic methods available to access and manipulate properties.

- *Translator approach:* In a translator approach, the conceptual model is not interpreted by generic methods, but the semantics specified in the conceptual model is directly coded into methods of the logical model. In other words, the conceptual

```
Object subclass: 'AOBDObject'
    instVarNames: #('properties')
    classVars: #()
    classInstVars: #('description')
    poolDictionaries: #()
    inDictionary: AOBD
    constraints: #[#[#properties, PropertyDictionary]]
    instancesInvariant: false
    isModifiable: false.

AOBDObject methodsFor: 'initialization'

initialize
    properties := PropertyDictionary new.
    self class description allProperties do:
        [ :p | p isSingleValued
            ifTrue: [properties at: p name put: nil]
            ifFalse: [properties at: p name put: Set new].].

AOBDObject methodsFor: 'propertyAccess'

getProperty: aPropertyName
    ...
assign: anObject toProperty: aPropertyName
    ...
add: anObject toProperty: aPropertyName
    ...
remove: anObject fromProperty: aPropertyName
    ...
```

Figure 7.2: Interpreting approach—class `AOBDObject` (selection)

```
AOBDObject subclass: 'Item'
    instVarNames: #('description' 'suppliers')
    classVars: #()
    classInstVars: #()
    poolDictionaries: #()
    inDictionary: AOBD
    constraints: #[#[#description, String],
                   #[#suppliers, SetOfSuppliers]]
    instancesInvariant: false
    isModifiable: false.

Item methodsFor: 'initialization'

initialize
    description := nil.
    suppliers := SetOfSuppliers new.

Item methodsFor: 'propertyAccess'

description
    ...
description: aString
    ...
suppliers
    ...
addSupplier: aSupplier
    ...
removeSupplier: aSupplier
```

Figure 7.3: Translator approach—class `Item` (selection)

model is *translated*. Compared to an interpreter approach, the resulting logical model comprises an extended set of non-generic methods.

Example: In a translator approach, an object-type hierarchy in the conceptual model may be mapped to an isomorphic class hierarchy in the logical model. Figure 7.3 shows parts of an exemplary class `Item` representing an object type with properties description and suppliers. Also here, class `Item` is subclass of a class `AOBDObject`. Class `AOBDObject` as it is used here, however, cannot be compared with class `AOBDObject` as it has been presented in the interpreter approach (cf. Figure 7.2). In the interpreter approach, class `AOBDObject` provided the functionality to interpret the conceptual model, in the translator approach, class `AOBDObject` factors out the common features of all classes representing object types (e.g., the maintenance of the class's extension). Class `Item` defines a properly constrained instance variable for each property and an extended set of non-generic methods. When an instance is created, no meta information is re-

trieved from the conceptual model, but the necessary initializations are directly coded into method `initialize`.

In the following, the interpreter approach and the translator approach are compared to each other with respect to usability, efficiency, and adaptability of the generated logical model:

- *Usability:* An automatically generated logical model provides an external interface through which a user can work with the system, an activity-implementation interface through which the activities of the conceptual model can be implemented, and a physical-design interface through which storage structures can be specified. The usability of the logical model is determined by the shape of these three interfaces:

  1. *External interface:* The external interface of a model designed with Active Object/Behavior Diagrams consists of properties and public activities. Thus, in a logical model, there has to be a retrieval method for each property and an update method for each public activity. Whether an interpreter approach or a translator approach is adopted, does not affect the way a user has to invoke the generated methods. In an interpreter approach, the invocation of generic methods for reading properties can be hidden to users by introducing non-generic wrapping methods.

     Example: Assume an object type Item defining property suppliers. If a translator approach is adopted, the suppliers of an item can be retrieved by method `suppliers` (cf. Figure 7.3). If an interpreter approach is adopted, object type Item is mapped to a subclass of `AOBDObject` (cf. Figure 7.2). Then, the suppliers of an item can be retrieved by generic method `getProperty: 'suppliers'`. The invocation of the generic method can be hidden to users if a method `suppliers` is introduced, which internally invokes the generic counterpart.

  2. *Activity-implementation interface:* Although Active Object/Behavior Diagrams provide a graphical notation for specifying activity implementations (cf. 4.4.3), implementation of activities is often done only within the logical model. At the conceptual level, the operations at hand for a designer specifying an activity are other activities and accessors to properties of the object type for which the activity is specified. Whereas generic methods serve as accessors to properties in an interpreter approach, there are non-generic accessors in a translator approach. However, if wrappers are provided for the generic methods in the case of an interpreter approach, the adopted approach does not affect the way activity realization can be done (see the example given for the external interface).

  3. *Physical-design interface:* Physical design is done in terms of the logical model. By providing generic methods, an interpreter approach often bypasses intrinsic

features of a system. If features are bypassed that are crucial for physical design, physical design may become impossible.

Example: An intrinsic feature of GemStone/TriGS is that the instance variables of an object o can be accessed directly from within methods performed on o. Indexing, the main mechanism of GemStone/TriGS for doing physical design, requires that this intrinsic feature is exploited, i.e., that instance variables are accessed directly from within queries. Unless there are instance variables for properties (cf. above example where properties are stored in a dictionary and generic methods are the only way to access their values), indexing cannot be used.

In a translator approach, intrinsic features of a system usually are used in a more natural way.

- *Efficiency:* Generic methods that retrieve information from the conceptual model, that run through decision trees, that indirectly invoke other methods, and that compile and evaluate expressions during runtime, usually make an interpreter approach rather expensive. In a translator architecture, most "interpretative work" is done during the mapping process. Mapping to a non-interpreting system is more complex and more time consuming than mapping to an interpreting system, but with the effect that the resulting operational system runs more efficiently. An additional aspect affecting efficiency has already been outlined above: By using intrinsic features, non-interpreting systems allow to exploit physical-design mechanisms that improve efficiency.

- *Adaptability:* In an interpreter approach, common structural and behavioral aspects are localized in generic methods accessing generic data structures. If an adaption of common structure or behavior is necessary, the methods or data structures that have to be changed can be localized easily.

  Example: Assume that the representation of all multi-valued properties with an object type as domain should be changed from `Set` to `IdentitySet`. In an interpreter approach as sketched in Figure 7.2, this can be achieved by a simple adaption of method `initialize`. If a multi-valued property's domain is an object type—meta information that can be retrieved from the conceptual model—the corresponding entry in dictionary `properties` is initialized with an instance of `IdentitySet` instead of an instance of `Set`.

  In an interpreter approach, however, it may reveal difficult to deal with adaptions due to special cases. If a special case requires a treatment that deviates from the standard mapping, severe workarounds will become necessary in the logical model.

  Example: Assume that property suppliers of an object type Item should be represented by a sorted list instead of a set. In an interpreter approach as sketched

in Figure 7.2, the corresponding class `Item` would be a subclass of the common root class `AOBDObject`. To cover the special case of property suppliers, generic-method `initialize` has to be overridden at class `Item` and—since the inheritance mechanism is circumvented—at all classes representing subtypes of object type Item. Additionally, the generic access methods tailored to a set-representation of multi-valued properties cannot be used to access property suppliers properly.

In a translator approach, common structural and behavioral aspects are distributed throughout the generated logical model. An adaption to common structure or behavior cannot be made within the logical model, but has to be reflected in the mapping component. Special cases, however, can be incorporated easily into the logical model. Single non-generic methods or structures can be adapted without side effects.

Example: Assume that instance variable `suppliers` of class `Item` (cf. Figure 7.3) should represent a sorted list instead of a set. To cope with this deviation from the standard mapping, method `initialize` and the non-generic access methods for instance variable `suppliers` have to be manually adapted. Doing this, the adaptions automatically become effective for all direct or indirect subclasses of `Item`—provided that instance variable `suppliers` is not redefined.

In our opinion, the enumerated criteria favour the adoption of a translator approach. The first decisive argument is that a translator approach allows to exploit intrinsic features of GemStone/TriGS. This is crucial for achieving an acceptable performance. The second decisive argument is that in a translator approach it is much easier to incorporate adaptions to special cases. A principal goal in the development of Active Object/Behavior Diagrams has been to enable a smooth transformation from a conceptual model to an operational system. A conceptual model designed with Active Object/Behavior Diagrams is detailed enough, so that its mapping yields a practicable logical model and not only a skeleton. Nevertheless, it cannot be assumed that all peculiarities of the system to be developed can already be captured in the conceptual model. Adaptions caused by special cases will be necessary. Concerning adaptability to special cases, a translator approach seems to be more flexible.

7.3 Related Work

The work described in this and the following chapters belongs to the research field of design transformation in forward engineering. Almost all work done so far in the area of translating high-level schemas into operational object-oriented models is restricted to object structure and passive object behavior. Active capabilities generally are not considered.

Much work has been done in the area of mapping Entity-Relationship diagrams to object-oriented models. A survey of database design transformations based on the (Extended)

Entity-Relationship model, including transformations to object-oriented models, is given in [FV95]. A major aspect in transforming an (Extended) Entity-Relationship schema into an object-oriented model is the incorporation of integrity constraints that cannot be represented explicitly via structural concepts in object-oriented models. A common approach to either actively enforce or passively check integrity constraints is to use methods. The work of Mok and Embley [ME96] can be subsumed under this category. They show how a conceptual object-oriented model (OSA, [EKW92]) incorporating integrity constraints but disregarding behavioral elements can be transformed to an object-oriented database design.

Several mapping approaches have also been presented for semantic data models enriched with language constructs for behavioral modeling. Exponents of semantic data models for which a mapping to object-oriented models has been proposed are ERC+ [Tar92] or Morse [BM91b]. (For an overview of semantic data models in general cf. [HK87]). Both models allow dynamic modeling with logic. Formulas which are used to model integrity constraints and general production rules are mapped to methods of an object-oriented model.

In Section 3.6, different approaches for the conceptual design of active object-oriented databases have been presented. Among the described approaches, Silva and Carlson (A/OODBMT, [SC96]), Mück (ERSC monitors, [Müc94]), and Herst (BROCOM, [Her97]), do not discuss how conceptual schemas developed with the respective approach can be mapped to the operational model of an existing system. Kappel et al. [KRSRS96] present rule patterns. Rule patterns are macros that allow the parameterization of business rules but that do not introduce a higher level concept for modeling business rules. Thus, mapping rule patterns does not have to be considered.

With Active Object/Behavior Diagrams, as they are presented by Bichler and Schrefl (cf. [BS94]), the impulse for the research presented in this thesis has been given. Bichler and Schrefl extended Object/Behavior Diagrams with *Activation Scripts* for modeling the active behavior of a system at the conceptual level. The authors also exemplified a mapping to Ode (cf. [LGA96]). The mapping of Activation Scripts into Ode rules is rather straight since both follow an operational approach for composing events. Different to Activation Scripts, Situation/Activation Diagrams, as they have been introduced in Chapter 5 and Chapter 6, follow a declarative approach and require a controllable, reliable, and temporally faithful execution of rules. Thus, the approach adopted for mapping Activation Scripts cannot be transferred to Situation/Activation Diagrams.

Mapping of rules has also been part of the IDEA project (cf. [CF97]). In Chimera (cf. [CFPT96]), the design language of the IDEA methodology, both passive and active rules can be expressed. Rule specifications are the input for a tool named *Argonaut*, which generates active rules from the declarative specification of integrity constraints and views. Another tool, named *Pandora*, takes a Chimera schema including the output produced by *Argonaut* and translates it into Oracle v.7.2. *Pandora* requires user interaction for optimization, since the translation to Oracle leaves open several options and requires re-design.

Chapter 8

Mapping Object Structure

Contents

8.1 Types and Properties 150

8.2 Derived Properties 153

8.2.1 Rationale 154

8.2.2 GemStone Smalltalk/OQL Binding 154

8.3 Query diagrams and OQL 157

8.3.1 Elementary Structure 158

8.3.2 Expressions 160

8.3.3 Domain Box 162

8.3.4 Condition Box 162

8.3.5 Result Box 164

8.3.6 Advanced Concepts 166

The goal of this chapter is to show how the concepts of Active Object/Behavior Diagrams for modeling the structure of objects can be mapped to GemStone/TriGS. The chapter is divided into three sections: Section 8.1 describes how data types, object types, and properties—the core structural concepts of Active Object/Behavior Diagrams—can be mapped. Section 8.2 discusses different approaches of mapping query diagrams associated with derived properties to GemStone/TriGS and motivates the selection of OQL (cf. [CB97]) as intermediate language. Finally, Section 8.3 shows how query diagrams can be mapped to OQL queries.

Object/Behavior Diagrams	GemStone/TriGS class
INTEGER	`Integer`
REAL	`Float`
STRING	`String`
BOOLEAN	`Boolean`
DATE	`Date`
TIME	`Time`
TIMESTAMP	`DateTime`

Table 8.1: Mapping of data types to GemStone/TriGS kernel classes

8.1 Types and Properties

Many authors point out that operational object-oriented data models lack expressiveness in modeling structural aspects of information systems (e.g., Bouzeghoub and Métais [BM91b], Elmasri et al. [EJK93], Tari [Tar92]). Among the identified deficiencies is that relationships are represented uniformly by stored object references in operational object-oriented data models. Different relationship semantics do not become apparent.

Although Active Object/Behavior Diagrams provide different relationship semantics, they strictly obey the principle of object autonomy. It has already been described in Section 4.4 that neither inverse relationships are maintained automatically, nor compliance with suggested deletion rules is enforced by Active Object/Behavior Diagrams. Activities have to be implemented and invoked explicitly so that inverse relationships are maintained correctly and deletion rules are followed. Even role switches have to be performed explicitly. This principle of object autonomy is adhered to when Active Object/Behavior Diagrams are mapped to GemStone/TriGS. Compliance with deletion rules and maintenance of inverse relationships are not enforced actively. But, since activity implementations may violate inverse declarations between relationships, the maintenance of inverse relationships is checked passively as an integrity constraint. Checking of integrity constraints requires a basic understanding of the activity execution model and, thus, its discussion is postponed to Section 9.2.

The approach respecting object autonomy followed within this thesis is contrary to the approach presented in [AA91]: There, a mapping of high-level graphical schemata to GemStone is shown. Operations to maintain inverse relationships and to realize different relationship semantics are not modeled explicitly in the high-level graphical schema, but corresponding methods are generated automatically when the schema is mapped to GemStone.

In the following it is shown how the structural concepts of Active Object/Behavior Diagrams are mapped to GemStone/TriGS:

Data type. Every data type of Active Object/Behavior Diagrams is represented by a kernel class in GemStone/TriGS. The mapping of data types to kernel classes is shown in Table 8.1.

Object type. Every object type specified with Active Object/Behavior Diagrams is represented by a user-defined class in GemStone/TriGS. The user-defined class is named after the object type it represents. If an object type does not have a supertype, the corresponding user-defined class is subclass of the predefined class `AOBDObject`, the common (direct or indirect) superclass for all classes representing object types.

Property. Every property, irrespective whether attribute or relationship, is represented by a retrieval method of the same name. The implementation of the retrieval method depends on whether the property is stored or derived:

- *Stored property:* For every stored property, a constrained instance variable of the same name is defined. The retrieval method returns the value of the instance variable. An instance variable's constraint is determined by the domain and the multiplicity of the stored property the variable represents. A single-valued attribute is mapped to an instance variable with a kernel class as constraint. A single-valued relationship, independent of its semantics, is mapped to an instance variable with a user-defined class as constraint. Contrary to Active Object/Behavior Diagrams, instance variables cannot be constrained directly to a set of values or objects of a particular type. Sets are first-class objects in GemStone/TriGS, and an instance variable can be constrained only to instances of a specific set class. So, for each kernel class and user-defined class, a subclass of GemStone/TriGS class `Set` (or `IdentitySet`) is defined, whose elements are constrained to be instances of the respective kernel or user-defined class. A multi-valued attribute is mapped to an instance variable with the appropriate subclass of class `Set` as constraint. Subclasses of `Set` guarantee that their instances do not contain equal elements. A multi-valued relationship is mapped to an instance variable with the appropriate subclass of class `IdentitySet` as constraint. Subclasses of `IdentitySet` guarantee that their instances do not contain identical elements. Both, sets and identity sets allow the definition of indices during physical design. In the case of a multi-valued property, the generated retrieval method returns only a shallow copy of the set or identity set held by the instance variable. This ensures encapsulation since no elements can be added or removed outside the object for which the instance variable is defined.

- *Derived property:* No instance variable is defined for a derived property. Rather, the public retrieval method computes the property value by issuing a query (cf. Section 8.2).

Type hierarchy and inherited properties. Subtype relationships specified with Active Object/Behavior Diagrams are mapped to subclass relationships in GemStone/TriGS. As already mentioned above, an object type for which no supertype is specified is mapped to a subclass of the root class `AOBDObject`. An object type for which a supertype is indicated is mapped to a subclass of the class representing its supertype. According to the represented object type hierarchy, a subclass may redefine the constraint of an inherited instance variable.

So far, methods have been introduced to read instance variables that represent stored properties. The question whether instance variables should be updated directly or only via methods remains to be discussed.

Arguments favouring a direct manipulation of instance variables are:

1. *Privacy:* In Active Object/Behavior Diagrams, properties of an object type can be updated only through activities defined for the object type. Likewise, it is desirable that instance variables of a GemStone/TriGS class cannot be manipulated outside methods representing activities (cf. Section 9.1). In GemStone/TriGS, there is no mechanism for restricting the visibility of methods. All methods belong to the public interface of a class. Thus, providing methods that update instance variables would contradict the principle that properties of an object type can be updated only through activities defined for the object type. Providing no methods, however, guarantees that privacy is obeyed.

2. *Uniformity:* Activities can be implemented at the level of Active Object/Behavior Diagrams by means of activity-realization diagrams (cf. 4.4.3). Containers, properties of the object *self*, and ports are subsumed by the term *storage repository* in activity-realization diagrams. Storage repositories can be uniformly manipulated by different kinds of directed arcs (cf. [KS96b, page 128]). If activity implementation is mapped to GemStone/Trigs or directly done at the level of GemStone/TriGS, the "storage repositories" of the corresponding method are the temporary variables, the instance variables of the class for which the method is defined, and the method arguments. It is desirable that these repositories can be manipulated in a uniform way—like in activity-realization diagrams. Temporary variables and method arguments can be manipulated only directly. If instance variables are manipulated directly, too, uniformity is preserved.

Arguments favouring the definition of methods are:

1. *Laissez faire initialization:* Laissez faire initialization (often referred to as *lazy initialization*) means that instance variables are not fully initialized at instance creation but when they are accessed (read or manipulated) for the first time (cf. [Lor95]). This strategy requires the definition of methods checking whether an instance variable is initialized or not. Laissez faire initialization mainly increases the robustness

of a system under development, it is of minor importance in automatically generated systems.

2. *Facilitation of rules:* In an active database system, rules may have to be specified that initiate actions upon changes of instance variables. In absence of value events, it is difficult to identify the events upon which to react. If methods are defined for manipulating instance variables and if circumvention of these methods is strictly avoided, the methods can serve as anchors for such rules.

We will use rules based on updates to instance variables when we check integrity constraints (cf. Section 9.2). In its current implementation, TriGS supports the detection of value events by utilizing the GemStone stone process (cf. [Hie96]). However, changes of instance variables are detected and value events are signaled only after the commit of the transaction performing the changes. This is too late for checking integrity constraints. Only methods can be the anchors for rules that are triggered when instance variables are updated. Thus, the facilitation of rules is the deciding factor to introduce methods for updating instance variables. It is a matter of discipline not to invoke the provided methods from outside but to use them in activity implementation.

For a single-valued instance variable `s`, the only private method defined is the assignment method `s: anObject`. For a multi-valued instance variable `m`, the private methods defined are (1) the assignment method `m: aSetOfObjects`, (2) the insertion methods `addToM: anObject` and `addAllToM: aSetOfObjects`, and (3) the removal methods `removeFromM: anObject` and `removeAllFromM: aSetOfObjects`. The update methods provided for multi-valued instance variables are internally reduced to `addToM: anObject` and to `removeFromM: anObject`.

Summarizing, an instance variable is read by means of a retrieval method and updated by means of private update methods.

8.2 Derived Properties

In Active Object/Behavior Diagrams, a query diagram is associated with a derived property. A query diagram represents a derivation expression following the select-from-where paradigm. Section 8.2.1 briefly outlines alternatives for mapping query diagrams to GemStone/TriGS, motivates the introduction of an intermediate language, and selects OQL ([CB97]) as the intermediate language to which query diagrams are mapped. Section 8.2.2 shows how GemStone/TriGS and OQL can be coupled.

8.2.1 Rationale

Efficient computation of derivation expressions following the select-from-where paradigm requires sophisticated query processing mechanisms. Graefe (cf. [Gra93]) describes the general steps in query processing as: (1) translation of a user language into some logical algebra, (2) logical query optimization (algebraic rewriting), (3) generation of an access plan (translation into the physical algebra of a concrete system), and (4) query execution. Within every query processing step, optimization and incorporation of heuristics to improve performance are central issues.

Due to the absence of a query language and of sophisticated query processing capabilities in GemStone/TriGS, the question arises, how to map query diagrams to an operational model. Mapping query diagrams to some logical algebra (e.g., ADL [SS86]) or directly to a set of operations implementing a physical algebra for GemStone/TriGS would be superficial and ad-hoc. On the one hand, regarding different aspects of optimization during the translation steps would exceed the scope of this thesis. On the other hand, disregarding optimization would result in a system that lacks satisfactory performance. For that reason, query diagrams will be mapped to an intermediate "user" language leaving open the different translation and optimization tasks. It will be the responsibility of future work to extend GemStone/TriGS with appropriate capabilities for processing queries that are specified in the intermediate language.

With this decision, mapping query diagrams to GemStone/TriGS has been reduced to the selection of a suitable intermediate language. Regarding its structure and expressiveness, a query diagram largely resembles an OQL query (cf. [CB97]). OQL is part of *The Object Database Standard: ODMG 2.0* and, thus, a proposal for querying object-oriented databases that is supported by a great number of database vendors. Since GemStone Systems, Inc., itself is participant of the ODMG, it can be expected that GemStone will support an OQL compliant query language in the future. A Smalltalk binding to OQL is already part of the ODMG proposal. The similarities in structure and expressiveness between query diagrams and OQL and the role of OQL as de-facto standard qualify OQL as the intermediate language used within this thesis.

8.2.2 GemStone Smalltalk/OQL Binding

There are generally two options for binding a query sublanguage to a (database) programming language: loosely coupled or tightly coupled (cf. [CB97]). In a *loosely coupled* approach, query operations are introduced that take strings containing queries as their arguments. These operations parse, optimize, and execute a query at runtime. In a *tightly coupled* approach, the query sublanguage is integrated directly into the programming language by extending its grammar. A tightly coupled approach allows queries to be parsed and optimized at compile time. Like the Smalltalk/OQL binding currently proposed by the ODMG standard, the integration of an OQL compliant query language into GemStone

Smalltalk is expected to follow a loosely coupled approach. In the following, a minimal model of a query facility is introduced.

The query facility is represented by class `Query`. Disregarding exception handling and error management, class `Query` provides the functionality for parsing, optimizing, and executing queries. Class `Query` provides class method

```
create: aQueryString params: aParameterArray
```

which parses a textual OQL query passed as `aQueryString`, optimizes the query, and returns the parsed and optimized query in some internal representation (as an instance of class `Query`). OQL queries may be parameterized. The passed query string `aQueryString` may contain parameter references of the form `$i`. When executed, parameter `$i` is bound to the object at position `i` in `aParameterArray`.

For instances of class `Query`, method

```
execute: aParameterArray
```

is defined. Parameterization of queries makes it possible that a query can be executed several times (with different parameters) without the overhead of repeated parsing and optimization.

The introduced query facility allows to create and execute queries within any method. Thus, a derived property can be easily mapped to a retrieval method issuing a query. In Active Object/Behavior Diagrams, a query diagram is evaluated for every instance of the object type for which it is defined. The instance for which a query diagram is evaluated, referred to as *self*, represents the query diagram's name context: all names are resolved starting at *self*. In this sense, *self* is an implicitly defined parameter of every query diagram. If a query diagram is mapped to an OQL query, the object for which the OQL query has to be evaluated has to be passed as parameter explicitly.

Generally, every time a derived property is computed, a new query instance could be created and executed with *self* as parameter. Such an approach would enforce a textual query to be parsed and optimized every time it is issued. In a more efficient approach, a query is parsed and optimized once and then stored in an instance variable. Storing a query for every instance of a class separately would mean an overhead, too. Parameterization allows that all instances of a class share a common parsed and optimized query instance by storing it in a class instance variable.

Figure 8.1 shows by means of an example how a query can be shared by the instances of a class. In the example, an object type AnObjectType is assumed for which a derived property prop is defined. When mapped to GemStone/TriGS, two class instance variables are defined for class `AnObjectType`: variable `queryStringForProp` holding the textual

```
AOBDObject subclass: AnObjectType
    instVarNames: #()
    classVars: #()
    classInstVars: #(queryStringForProp, optimizedQueryForProp)
    poolDictionaries: #()
    inDictionary: AOBD
    constraints: #[]
    instancesInvariant: false
    isModifiable: false.

AnObjectType classMethodsFor: 'initialization'

initializeQueryForProp: aQueryString
    queryStringForProp := aQueryString.
    optimizedQueryForProp := nil.

AnObjectType classMethodsFor: 'queryExecution'

executeQueryForPropWithParams: aParamArray
    optimizedQueryForProp isNil
        ifTrue: [optimizedQueryForProp :=
                    Query create: queryStringForProp params: aParamArray].
    ^optimizedQueryForProp execute: aParamArray.

AnObjectType methodsFor: 'retrieving'

prop
    ^self class executeQueryForPropWithParams: (Array with: self).
```

Figure 8.1: Query shared by the instances of a class

OQL query to derive `prop` and variable `optimizedQueryForProp` holding the parsed and optimized counterpart. At class definition, the textual representation is set by means of class method `initializeQueryForProp:`. Parsing and optimization cannot be done at class definition since actual parameters are required for that. Rather, parsing and optimization is done when the value of property `prop` is retrieved for the first time by performing method `prop` on an instance of class `AnObjectType`. Method `prop` invokes class method `executeQueryForPropWithParams:` passing *self* as parameter.

8.3 Query diagrams and OQL

Since the design of query diagrams as a graphical language has been influenced by OQL, there is large semantic correspondence between the expressions used in query diagrams and the expressions used in OQL queries. Regarding the syntactic representation, however, there are considerable differences between expression used in query diagrams and in expression used in OQL queries. This is obvious, since query diagrams represent a graphical and an OQL a textual language to query object bases.

The following sections show how query diagrams can be mapped to OQL queries. For that, skeletons of query diagrams are presented that abstract from the actual representation of expressions used therein. The expressions are placed within the skeletons, they are named, and the variables that can be used within the expressions are indicated. In a next step, it is shown how those abstract expressions find their pendants in OQL queries. Following the general typographic conventions, expressions used in query diagrams are set in sans-serif typeface and expressions used in the the target system (OQL) are set in monospace typeface. The correspondence between an expression used in a query diagram and an OQL expression is established by a common name. The following conventions are obeyed in naming expressions:

- x denotes a variable,
- $\mathsf{e}(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ denotes an expression on variables $\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n$ that evaluates to an arbitrary type,
- $\mathsf{c}(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ denotes an expression on variables $\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n$ that evaluates to a collection of values/objects,
- $\mathsf{f}(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ denotes an expression on variables $\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n$ that evaluates to a single value/object, and
- $\mathsf{p}(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ denotes an expression on variables $\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n$ that evaluates to a boolean value.

```
select [distinct]  f(x1, x2, ..., xn, xn+1, xn+2, ..., xn+p)
from   x1 in   c1(xn+1, xn+2, ..., xn+p),
       x2 in   c2(x1, xn+1, xn+2, ..., xn+p),
       ⋮
       xn in   cn(x1, x2, ..., xn−1, xn+1, xn+2, ..., xn+p)
[where  p(x1, x2, ..., xn, xn+1, xn+2, ..., xn+p)]
```

Figure 8.2: General form of an OQL select statement

8.3.1 Elementary Structure

A query diagram may be primitive or it may follow the *select-from-where* paradigm. A primitive query diagram is a selecting query diagram which does not iterate over collections of objects but simply selects the value of an expression. A query diagram following the select-from-where paradigm is a selecting or generating query diagram that defines variables iterating over collections of objects and that imposes conditions on these variables. A select-from-where query diagram may consist of a single branch or it may consider several alternatives.

Mapping a primitive query diagram to an OQL query corresponds to mapping the expression whose value is selected. As already introduced above, OQL queries that represent query diagrams are parameterized. Even for primitive query diagrams, the object for which the corresponding OQL query has to be evaluated is passed as parameter. Within the OQL query, this parameter is referred to as `$1`. How expressions other than query diagrams can be mapped to OQL queries will be shown in Section 8.3.2.

A query diagram that follows the select-from-where paradigm can be mapped to an OQL *select statement*, a special kind of an OQL query. The general form of an OQL select statement is shown in Figure 8.2. Whereas variables $x_1, x_2, \ldots, x_n$ are defined in the from clause of the OQL select statement and bound during the evaluation of the select statement, variables $x_{n+1}, x_{n+2}, \ldots, x_{n+p}$ are variables that are bound in the query's environment (in the case of a subquery) or that are bound to arguments passed to the query evaluation as parameters.

The semantics of a select statement can be described procedurally by the following steps (assuming that $x_{n+1}, x_{n+2}, \ldots, x_{n+p}$ are bound to $X_{n+1}, X_{n+2}, \ldots, X_{n+p}$):

1. *From clause evaluation:* The result of the from clause evaluation is a collection of tuples that is computed as follows:

```
FORALL x1 IN bagOf(c1(Xn+1, Xn+2, ..., Xn+p))
    FORALL x2 IN bagOf(c2(x1, Xn+1, Xn+2, ..., Xn+p))
      ⋮
        FORALL xn IN bagOf(cn(x1, x2, ..., xn−1, Xn+1, Xn+2, ..., Xn+p))
            tuple(x1, x2, ..., xn−1)
        END
      ⋮
    END
END
```

In OQL, different kinds of collections are provided: sets, bags, lists, and arrays. Independent of its type, a collection c_i is converted to a bag (possibly containing duplicates) if it is used as range of a variable.

2. *Where clause evaluation:* The result of the from clause is filtered by retaining only those tuples $(X_1, X_2, \ldots, X_n)$ that satisfy the boolean expression

   $$p(X_1, X_2, \ldots, X_{n-1}, X_{n+1}, X_{n+2}, \ldots, X_{n+p})$$

3. *Select clause evaluation:* To each of the filtered tuples the expression

   $$f(X_1, X_2, \ldots, X_{n-1}, X_{n+1}, X_{n+2}, \ldots, X_{n+p})$$

   is applied. The whole set of tuples filtered by step (2) can be referred to by the special symbol $*$.

4. *Duplicate elimination:* If keyword `distinct` is indicated, duplicates are eliminated and a set without duplicates is obtained. If the keyword is missing, a bag containing duplicates is obtained. If e_1 and e_2 are expressions which denote objects (with identity) of compatible object types, e_1 and e_2 are equal if they designate the same object. If e_1 and e_2 are expressions which denote values (without identity) of compatible types, equality of e_1 and e_2 is computed as follows:

   - e_1 and e_2 are scalars: e_1 and e_2 have to be the same value.
   - e_1 and e_2 are tuple structures: each of the attributes of e_1 and e_2 must be equal.
   - e_1 and e_2 are sets: e_1 and e_2 must have the same number of elements, and for each element in e_1, there has to be an equal element in e_2, and vice versa.

The structure as well as the semantics of a query diagram largely correspond to those of an OQL select statement. Informally, a query diagram following the select-from-where paradigm is mapped to an OQL select statement as follows (details will be presented in Sections 8.3.3 to 8.3.5): The domain box is mapped to the from clause, the condition box is mapped to the where clause, and the result box is mapped to the select clause of the OQL select statement.

8.3.2 Expressions

Expressions that can be used in query diagrams are literals, functions, navigation paths, variables, and other query diagrams. In the following it is discussed how these different kinds of expressions can be mapped to OQL expressions.

Literals. In query diagrams, there are scalar literals for every data type and set literals. For scalar literals of types INTEGER, REAL, STRING, and BOOLEAN, there are direct pendants in OQL. By contrast, literals of types DATE, TIME, and TIMESTMP are not directly supported in OQL. In these cases, date/time-construction functions have to be used instead of literals (see next paragraph). Set literals can be constructed by using operator `set()`, which takes an arbitrary number of scalar literals—all of the same type—as arguments.

Functions. For almost all functions that can be used in query diagrams there are appropriate counterparts in OQL. The only functions that have to be considered more closely are date/time-construction functions, date/time-manipulation functions, and state functions. These functions lack a direct mapping to OQL. DATE values, TIME values, and TIMESTAMP values are represented by instances of GemStone/TriGS classes `Date`, `Time`, and `DateTime` respectively. These classes provide the necessary functionality for creating instances and manipulating them. Since OQL allows to call a method at any position where the result type of the method matches the expected type in the surrounding expression, date/time-construction functions and date/time-manipulation functions can be mapped to invocations of methods provided by classes `Date`, `Time`, and `DateTime`.

Every GemStone/TriGS class that represents an object type defines an instance variable holding the states in which an instance resides. This instance variable can be accessed by a set of methods (cf. 9.1). So, state functions can be mapped to invocations of methods accessing state information.

Navigation Paths. Although OQL provides the concept of path expressions to follow relationships, there is no direct correspondence between navigation paths in query diagrams and path expressions in OQL. Path expressions allow to navigate only along single-valued relationships. In contrast to navigation paths, path expressions do not iterate automatically over all elements of a traversed multi-valued relationship. If all traversed relationships are single-valued, a navigation path can be directly mapped to an OQL path expression. If a navigation path traverses at least one multi-valued relationship, however, the navigation path has to be mapped to an OQL select statement.

Figure 8.3 shows the general structure of a navigation path containing at least one multi-valued relationship. A path starts at an object o, follows an arbitrary number of single-valued properties (denoted by s_j^i) and iterates over n multi-valued properties ($m^1, m^2, \ldots, m^n$).

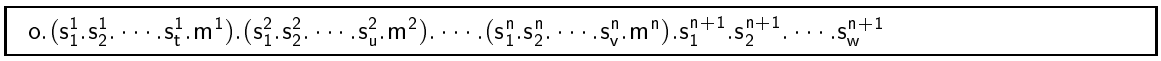


Figure 8.3: Structure of a navigation path

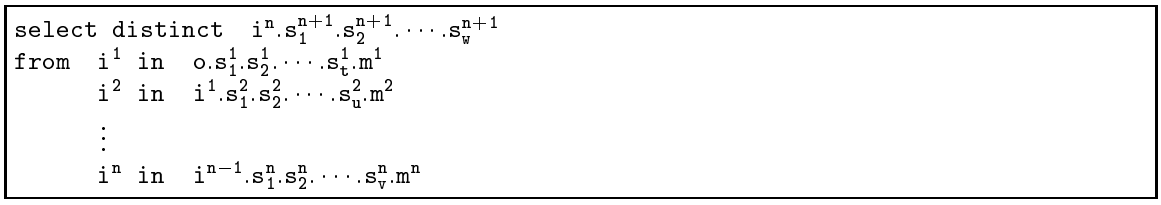


Figure 8.4: Mapping a navigation path to a select statement

When evaluated, every multi-valued property opens an iteration level. Parts of the navigation path that denote collections to be iterated at evaluation are delimited by parenthesis. At iteration level $i (1 \leq i \leq n)$, the collection of objects to be iterated is represented by navigation path $s^i_1.s^i_2.\cdots.s^i_x.m^i$ where $s^i_1.s^i_2.\cdots.s^i_x$ denotes a sequence of single-valued properties that are followed starting at the object currently selected at interation level $i-1$ (or starting at o if $i = 1$). Expression $s^{n+1}_1.s^{n+1}_2.\cdots.s^{n+1}_w$ denotes an optional sequence of single-valued properties that are followed starting at the object currently selected at iteration level n.

Figure 8.4 shows how a navigation path traversing at least one multi-valued relationship can be mapped to an OQL select statement. In the from clause, a variable `i` is defined for every iteration level. In the select clause, either variable i^n or the value of instance variable s^{n+1}_w obtained by following the instance variables $s^{n+1}_1.s^{n+1}_2.\cdots.s^{n+1}_{w-1}$ is selected.

Navigation along derived properties does not differ from navigation along stored properties. Both, stored properties and derived properties can be accessed by public methods when mapped to GemStone/TriGS. OQL allows to call a method anywhere the result type of the method matches the expected type in the surrounding expression. Whether an instance variable representing a stored property is accessed directly or via the public access method of the same name, is a matter of optimization and is not further discussed here. The notation for calling a method is exactly the same as for accessing an instance variable.

Variables. In the domain box of a query diagram, iterator variables and reference variables can be defined. Iterator variables have a direct counterpart in OQL, their usage has been shown above. By contrast, OQL does not provide reference variables. Since reference variables have been introduced only as shorthand for long expressions, their absence in OQL can be compensated easily by textual substitution of expressions. It is a matter of query optimization to avoid that same expressions are computed several times when a query is evaluated. Visibility of variables is equivalent in query diagrams and OQL queries.

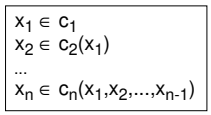


Figure 8.5: Skeleton of a domain box

Query Diagrams. Mapping of nested query diagrams to OQL will be discussed in Section 8.3.6.

8.3.3 Domain Box

Figure 8.5 shows the skeleton of a query diagram's domain box after removing reference variables. The semantics of evaluating a domain box of a query diagram (cf. Section 4.5.3) corresponds to the semantics of evaluating the from clause of an OQL select statement (cf. Section 8.3.1). Therefore, a domain box can be mapped to the select clause of an OQL select statement in the form:

```
⋮
from   x1 in  c1($1),
       x2 in  c2($1, x1),
       ⋮
       xn in  cn($1, x1, x2, ..., xn−1)
⋮
```

8.3.4 Condition Box

The condition box of a query diagram consists of a set of expanded variables. It shows the application of graphical comparison predicates, the graphical navigation along relationships, and the specification of shorthand variables for properties. In the following it is shown how comparison predicates, the graphical navigation along relationships, and shorthand variables can be reflected in OQL.

Comparison Predicates. The comparison predicates supported in query diagrams are scalar-comparison predicates, scalar/set-comparison predicates, and set-comparison predicates. OQL provides a set of operators to which scalar-comparison predicates can be mapped. Besides equality of objects and equality of values (cf. Section 8.3.1), diverse other comparison operators are available for values.

The two groups of scalar/set-comparison predicates in query diagrams, namely (1) predicates that require a multi-valued property to contain a single value/object and (2) predicates that require a single-valued property to be contained in a set of values/objects,

can be mapped to a single OQL operator. By means of boolean expression `f in c`, the membership of an object/value `f` in a collection `c` is tested.

Regarding set-comparison predicates, only equality of sets is directly supported in OQL (cf. Section 8.3.1). Subset ($\subset$) and subset-or-equal ($\subseteq$) predicates—to which superset ($\supset$) and superset-or-equal ($\supseteq$) predicates can obviously be transformed by exchanging the operands—can be expressed in OQL by universal quantification and membership testing. So, an expression $c_1 \subseteq c_2$ can be mapped to

```
for all i in c1:  i in c2
```

and an expression $c_1 \subset c_2$ can be mapped to

```
(for all i in c1:  i in c2) and not (c1 = c2)
```

As shown in the last OQL expression, an expression can be negated by prefixing it with the boolean operator `not`. All comparison predicates that are derived from a single condition box are combined by a logical `and` in the where clause of an OQL select statement.

Navigation along Relationships. Semantically, there is no difference between defining a variable textually in the domain box and defining a variable implicitly by navigating along a relationship in the condition box. When a query diagram is evaluated, the two kinds of variable definitions are treated equally. All the variables implicitly defined by navigating along relationships of an expanded variable `v` are considered to be defined in an arbitrary order after `v` in the domain box.

When a variable implicitly defined by navigating along a relationship `r` of an expanded variable `v` is mapped to OQL, treatment of the variable depends on whether it is a reference variable or an iterator variable. If `v` is a reference variable, no variable is introduced for the referenced object in the corresponding OQL select statement, but the referenced object is referred to as `v.r` (see above). If `v` is an iterator variable, an auxiliary variable is introduced for it in the corresponding OQL select statement. The auxiliary variable is defined behind `v` in the from clause and is specified to range over `v.r`.

Shorthand Variables. Semantically, there is no difference between defining a variable in the domain box and defining a shorthand variable in the condition box. When a query diagram is evaluated, the two kinds of variable definitions are treated equally. All the shorthand variables that are defined for properties of an expanded variable `v` are considered to be defined in an arbitrary order after `v` in the domain box. The mapping of a shorthand variable to OQL corresponds to the mapping of an implicitly defined variable as it has been described in the previous paragraph.

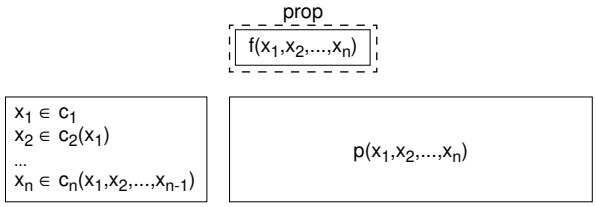


Figure 8.6: Selecting query diagram

```
select distinct  f($1, x1, x2, ..., xn)
from   x1 in  c1($1),
       x2 in  c2($1, x1),
       ⋮
       xn in  cn($1, x1, x2, ..., xn−1)
where  p($1, x1, x2, ..., xn)
```

Figure 8.7: Select statement for a selecting query diagram

8.3.5 Result Box

Figure 8.6 shows the skeleton of a selecting query diagram. In the domain box, variables $x_1, x_2, \ldots, x_n$ are specified to iterate over collections obtained by expressions $c_1, c_2, \ldots, c_n$. The condition box, which actually consists of a set of expanded variables, is sketched out by a single rectangle in the skeleton. All the conditions that are applied to the expanded variables are represented together by the boolean expression p. In the result box, expression f is specified that is applied to every tuple of variables satisfying expression p.

Figure 8.7 shows how a selecting query diagram can be mapped to an OQL select statement. Parameter `$1` refers to the object for which the OQL query is evaluated. Keyword `distinct` is indicated with every OQL query that represents a selecting query diagram. Duplicates are eliminated because of the set semantics of multi-valued properties in Active Object/Behavior Diagrams.

Generating query diagrams differ from selecting query diagrams only in their result box. Figure 8.8 shows the skeleton of a generating query diagram. An instance of type OBJECT_TYPE is generated for every tuple $(x_1, x_2, \ldots, x_n)$ satisfying expression p. The properties of the newly created instances $(prop_1, prop_2, \ldots, prop_m)$ are initialized with expressions $e_1, e_2, \ldots, e_m$.

In the select clause of an OQL select statement that represents a generating query diagram, an object has to be constructed for each tuple filtered by the where clause. Thus, in an object-generating select statement, expression f has to be an expression constructing new objects. Object construction can be expressed in OQL as follows: if t is a type name and $prop_1, prop_2, \ldots, prop_m$ are properties of this type then $t(prop_1 : e_1, prop_2 : e_2, \ldots, prop_m : e_m)$

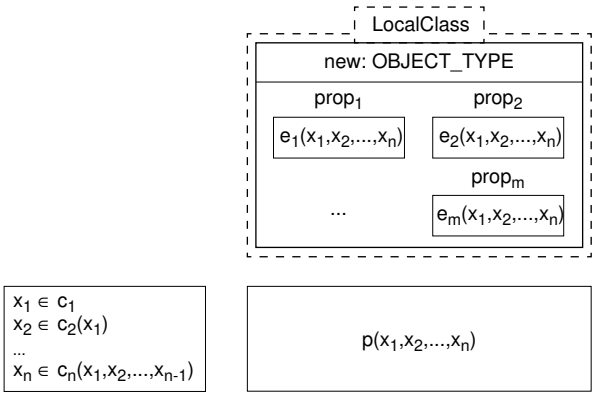


Figure 8.8: Generating query diagram

```
select  OBJECT_TYPE (
        componentOf : $1,
        prop1 : e1($1, x1, x2, ..., xn),
        prop2 : e2($1, x1, x2, ..., xn),
        ⋮
        propm : em($1, x1, x2, ..., xn))
from    x1 in  c1($1),
        x2 in  c2($1, x1),
        ⋮
        xn in  cn($1, x1, x2, ..., xn−1)
where   p($1, x1, x2, ..., xn)
```

Figure 8.9: Select statement for a generating query diagram

is an expression that returns a new object of type `t` whose properties $\texttt{prop}_1, \texttt{prop}_2, \ldots, \texttt{prop}_m$ are initialized with the expressions $\texttt{e}_1, \texttt{e}_2, \ldots, \texttt{e}_m$.

Figure 8.9 shows how a generating query diagram can be mapped to an OQL select statement. In a generating query diagram, the component-of relationship of a generated instance is implicitly set to the object for which the query diagram is evaluated. In the corresponding OQL select statement, however, this assignment has to be done explicitly. Thus, the object for which the select statement is evaluated (passed as parameter `$1`) is assigned to the instance variable that represents the component-of relationship (by default named `componentOf`).

For a generating query diagram whose result is grouped, mapping has to be reconsidered. Figure 8.10 shows the skeleton of a generating query diagram with a grouped result. If the result of a query is grouped, variables are divided into two disjunctive sets: (1) variables $\texttt{x}^g_1, \texttt{x}^g_2, \ldots, \texttt{x}^g_k$ according to which the tuples satisfying `p` are grouped, and (2) variables $\texttt{x}^n_1, \texttt{x}^n_2, \ldots, \texttt{x}^n_l$ that are nested. A variable is qualified as a variable according to which the

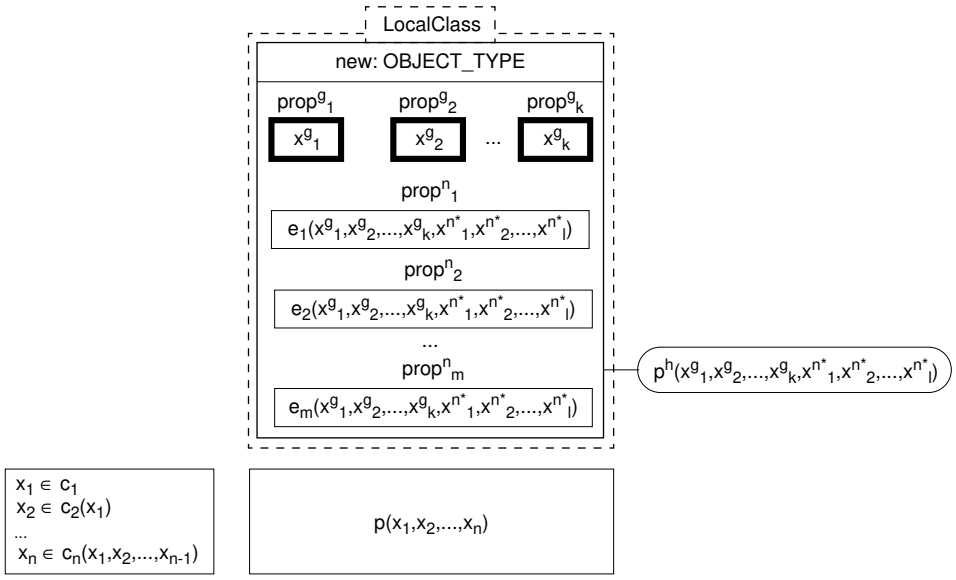


Figure 8.10: Generating query diagram with grouped result

result is grouped by assigning it to a property marked with a boldface border. All the variables that are not assigned to a property with a boldface border are nested. After grouping, a nested variable x_i^n can be used as x_i^{n*} in expressions $e_1, e_2, \ldots, e_m$ and in having clause p^h. An instance of type OBJECT_TYPE is created for each group satisfying expression p^h.

OQL select statements can be extended by a group-by clause and a having clause. In the group-by clause, so-called "partition attributes" are defined, according to which the tuples filtered by the where clause are grouped. All tuples that have the same values according to the given partition attributes belong to the same partition. After grouping, each group has the valued partition attributes completed by an attribute named `partition`. Attribute `partition` contains all tuples matching the particular partition attributes. The partitioned result may be filtered by a having clause.

Figure 8.11 shows how a generating query diagram with a grouped result can be mapped to an OQL select statement. In the group-by clause, partition attributes $\mathtt{att}_1, \mathtt{att}_2, \ldots, \mathtt{att}_k$ have to be introduced for variables $x_1^g, x_2^g, \ldots, x_k^g$. These partition attributes can be used in the select clause and in the having clause of the select statement. If a nested variable x_i^n should be used in the select clause or in the having clause, it cannot be used directly, but its value has to be derived from the nested `partition` attribute.

8.3.6 Advanced Concepts

Alternatives. A single query diagram may be satisfied by several scenarios, each described by a separate branch. Figure 8.12 shows the skeleton of a query diagram with m

```
select  OBJECT_TYPE (
        componentOf : $1,
        prop_1^g : att_1,
        prop_2^g : att_2,
        ⋮
        prop_k^g : att_k,
        prop_1^n : e_1($1, att_1, att_2, ..., att_k,  x_1^n*, x_2^n*, ..., x_1^n*)
        prop_2^n : e_2($1, att_1, att_2, ..., att_k,  x_1^n*, x_2^n*, ..., x_1^n*)
        ⋮
        prop_m^n : e_m($1, att_1, att_2, ..., att_k,  x_1^n*, x_2^n*, ..., x_1^n*))
from  x_1 in  c_1($1),
      x_2 in  c_2($1, x_1),
      ⋮
      x_n in  c_n($1, x_1, x_2, ..., x_n-1)
where  p($1, x_1, x_2, ..., x_n)
group by  att_1 : x_1^g,
          att_2 : x_2^g,
          ⋮
          att_k : x_k^g
having  p^h($1, att_1, att_2, ..., att_k,  x_1^n*, x_2^n*, ..., x_1^n*)

 x_i^n* = select distinct  p.x_i^n from  p in partition
```

Figure 8.11: Select statement for a generating query diagram with grouped result

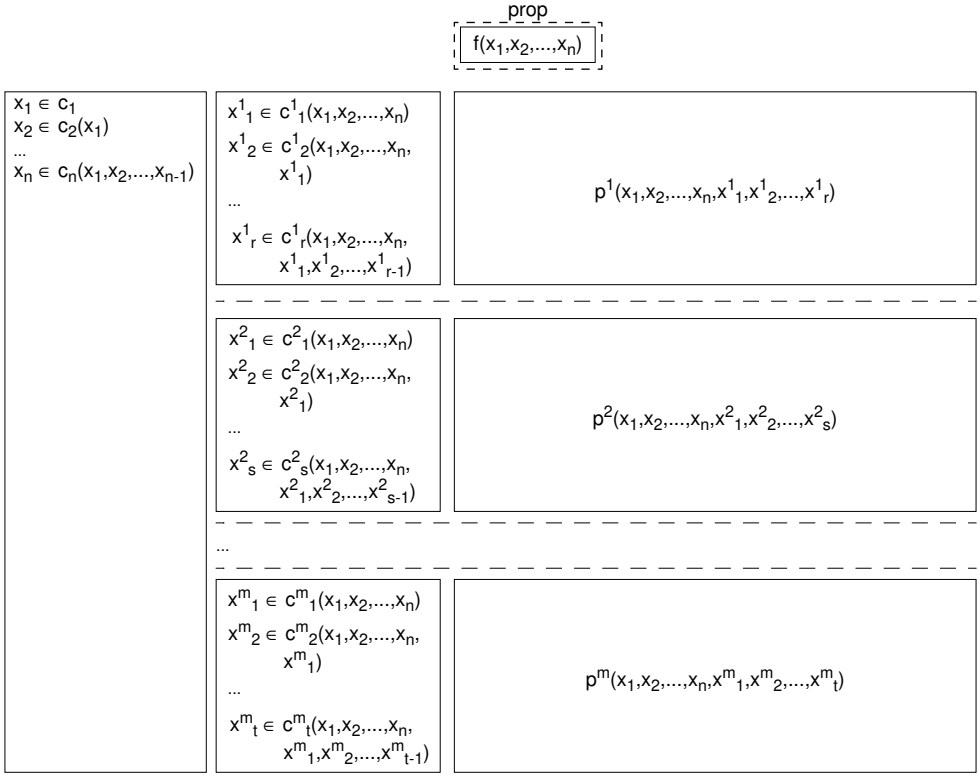


Figure 8.12: Query diagram with alternatives

```
select distinct  f($1, x_1, x_2, ..., x_n)
from   x_1 in  c_1($1),
       x_2 in  c_2($1, x_1),
       ⋮
       x_n in  c_n($1, x_1, x_2, ..., x_{n-1})
where exists (select *
              from  x_1^1 in  c_1^1($1, x_1, x_2, ..., x_n),
                    x_2^1 in  c_2^1($1, x_1, x_2, ..., x_n, x_1^1),
                    ⋮
                    x_r^1 in  c_r^1($1, x_1, x_2, ..., x_n, x_1^1, x_2^1, ..., x_{r-1}^1)
              where  p^1($1, x_1, x_2, ..., x_n, x_1^1, x_2^1, ..., x_r^1))
     or
     exists (select *
              from  x_1^2 in  c_1^2($1, x_1, x_2, ..., x_n),
                    x_2^2 in  c_2^2($1, x_1, x_2, ..., x_n, x_1^2),
                    ⋮
                    x_s^2 in  c_s^2($1, x_1, x_2, ..., x_n, x_1^2, x_2^2, ..., x_{s-1}^2)
              where  p^2($1, x_1, x_2, ..., x_n, x_1^2, x_2^2, ..., x_s^2))
     or
      ⋮
     or
     exists (select *
              from  x_1^m in  c_1^n($1, x_1, x_2, ..., x_n),
                    x_2^m in  c_2^n($1, x_1, x_2, ..., x_n, x_1^m),
                    ⋮
                    x_t^m in  c_t^m($1, x_1, x_2, ..., x_n, x_1^m, x_2^m, ..., x_{t-1}^m)
              where  p^m($1, x_1, x_2, ..., x_n, x_1^m, x_2^m, ..., x_t^m))
```

Figure 8.13: Select statement for a query diagram with alternatives

branches. Each branch i defines local variables $x_1^i, x_2^i, \ldots, x_u^i$. In the local condition box of a branch i, conditions (represented all together as p^i) are imposed on global variables and on variables local to i.

Figure 8.13 shows how a query diagram with alternatives can be mapped to an OQL select statement. Each branch of the query diagram is mapped to a select statement nested in the where clause of the main select statement. Since only global variables can be used in the select clause of the main statement, actual bindings to local variables are not of interest for computing the result. Rather, bindings that satisfy the local where clauses are required only to *exist*.

Nested Query Diagrams. For better readability, nested query diagrams are not depicted within the enclosing query diagram, placeholders are introduced. In OQL select

statements, no placeholders are introduced for nested select statements, but they are specified in place. Nested select statements can be used at any position the result matches the expected type. The visibility of variables bound by an enclosing environment is equivalend in nested query diagrams and in nested select statements.

Chapter 9

Mapping Passive Object Behavior

Contents

9.1 Activities and States 172
9.2 Integrity Maintenance 175
9.2.1 Preconditions and Postconditions 177
9.2.2 State Changes 179
9.2.3 Invariant Properties 180
9.2.4 Obligatory Properties 180
9.2.5 Unique Keys 180
9.2.6 Subclass/Superclass Relationship 183
9.2.7 Inverse Relationships 184
9.2.8 Referential Integrity 186
9.2.9 Local Referential Integrity 188
9.2.10 Summary 196
9.3 Task Model 199
9.3.1 Tasks and Transactions 201
9.3.2 Execution Model of GemStone/TriGS 205
9.3.3 Tasks in GemStone/TriGS 208
9.3.4 Temporal Faithfulness 211
9.3.5 Completeness 215
9.3.6 Priority Observation 220
9.3.7 The Task Manager—Class `TaskMngr` 221
9.3.8 Internal Tasks 223

The goal of this chapter is to show how the concepts of Active Object/Behavior Diagrams for modeling the passive behavior of objects can be mapped to GemStone/TriGS. The chapter is divided into three sections: Section 9.1 describes how activities and states—the core behavioral concepts of Active Object/Behavior Diagrams—can be mapped. Section 9.2 discusses integrity constraints and shows how they can be maintained in GemStone/TriGS by means of rules. Finally, Section 9.3 introduces an execution model that extends the conventional way a user works with GemStone/TriGS.

9.1 Activities and States

During conceptual design, the passive behavior of objects is designed by means life cycles and activity specifications.[1] In this section, it is shown how the elements of life cycles—states and activities—are mapped to GemStone/TriGS.

State

Every GemStone/TriGS class that represents an object type in Active Object/Behavior Diagrams is direct or indirect subclass of the predefined class `AOBDObject`. Class `AOBDObject` defines a set-valued instance variable `states`. In this instance variable, every instance records the names of the basic states in which it resides. Besides this instance variable, two access methods are defined: method `enterState: aStateName` for entering a state and method `leaveState: aStateName` for leaving a state. Further, class `AOBDObject` overrides class method `new` so that instance variable `states` of a new instance of `AOBDObject` is initialized with an empty set. Section 9.2.2 will show how state changes are effected.

Activity

An activity defined for an object type `O`, irrespective whether public or restrictedly public, is mapped to an instance method of the class representing `O`. The method is named after the activity it represents.

For a creation activity, a class method is generated additionally. The class method gets the name of the activity it represents, i.e., class method and instance method generated for a creation activity carry the same name. The class method, whose implementation is generated, creates a new instance of the class and performs the instance method that carries the same name as the class method on the new instance. The reasons for introducing both a class method and an instance method for a creation activity are: First, the class

[1] Note, we assume that activities are implemented at the level of the logical model, i.e. at the level of GemStone/TriGS, and not at the level of Active Object/Behavior Diagrams.

method has been introduced to hide the creation of a new instance. It is not necessary to create a new instance separately and then to perform a creation activity on it. Second, the instance method has been introduced to provide for a uniform treatment of creation activities, inner activities, and removal activities with respect to integrity maintenance (cf. Section 9.2).

The elements of an activity specification, i.e., activity ports, prestates, poststates, precondition, and postcondition, are mapped as follows:

Activity Port. The input ports of an activity, apart from the primary input port, are mapped to arguments of the method representing the activity. In the case of a creation activity, the input ports are mapped to arguments of both, the class method and the instance method representing the activity. The selector of the method representing the activity is formed by the name derived from the activity together with the port names. The primary input port of an activity represents the object on which the activity is performed. The object on which a method is performed can be referenced as *self* in the implementation of the method. So, the primary input port can be disregarded when activity ports are mapped to GemStone/TriGS. The output ports of an activity show effects of the activity on participating objects. Output ports do not have a pendant in GemStone/TriGS, there is no intrinsic feature to make side effects of a method explicit in GemStone/TriGS. Even an existing return port is not considered at method definition since no formal return value is specified for a GemStone/TriGS method.

Summarizing, an activity a with the (non-primary) input ports $p_1, p_2, \ldots, p_n$ is mapped to an instance method (and a class method in the case of a creation activity) with selector:

$$\texttt{aWithP}_1\texttt{:}\ p_1\ \texttt{withP}_2\texttt{:}\ p_2\ \ldots\ \texttt{withP}_n\texttt{:}\ p_n$$

Precondition. The precondition of an activity represents a restriction on the activity's input parameters and on the object on which the activity is performed. Intuitively, checking of a precondition could be done by a single method that evaluates to true or false. The execution model that will be introduced later in this section (cf. Section 9.3) requires to distinguish between two kinds of precondition violations: The first kind of violation occurs if the object on which an activity should be executed does not reside in the required prestate(s). If an object on which an activity should be executed does not reside in the required prestate(s), the object is said to be *not enabled*. The second kind of violation occurs if the object on which an activity should be executed is enabled, but some other part of the precondition is violated. Respecting this distinction, the checking of a precondition is separated into two methods.

Consider an activity a with the (non-primary) input ports $p_1, p_2, \ldots, p_n$. The check whether an object is enabled is implemented by method

```
enabled_a.
```

The check whether some other part of the precondition is violated is implemented by method

```
preWithP1 : p1 withP2 : p2 ... withPn : pn.
```

The second method checks whether all objects assigned to input ports reside in appropriate prestates and satisfy all other conditions imposed on the them.

Both methods are instance methods. They are not introduced as class methods for creation activities. Section 9.2.1 will show when the two methods are executed.

Postcondition. The postcondition of an activity expresses a transitional constraint on (i) the object on which the activity is performed, on (ii) objects that are passed to activity execution via input ports, on (iii) the formal return value of the activity, and on (iv) other objects created or manipulated during activity execution.

Generally, it is desireable to decouple the checking of a postcondition from the method representing an activity, i.e., checking of the postcondition should not be part of the activity implementation, but should be done in a separate method. If the postcondition of an activity is checked by a separate method, the following problem arises: Only the interface of the method representing the activity is visible, and the only objects upon which the postcondition can be checked are (i) the object on which the method is performed, (ii) objects that have been passed as input parameters when the method has been invoked, and (iii) the return value of the method. Other objects created or manipulated by the method are not available outside the implementation for checking the postcondition. Possible approaches to cope with this problem are:

1. *Extending the return value:* The return value of a method representing an activity can be extended to a structure, so that all objects that are created or manipulated by the method are returned. The postcondition then can be checked upon the elements of the result structure. The most important weakness of this approach is that the enforced way of implementing a method contradicts the "natural" way of implementing methods in GemStone/TriGS. Not the object representing the semantic return value of the method has to be returned but a complex structure of objects not bearing the semantics of a return value. Moreover, GemStone/TriGS lacks appropriate concepts to make the structure of a method's result explicit.

2. *Restricting the set of output ports:* Another approach would be to take measures already at the level of conceptual design and to restrict the set of allowed output ports. If only output ports are allowed for an activity that are (i) the primary output port, (ii) an output port which at the same time is an input port, or (iii) the

return port, the postcondition check can be decoupled when the activity is mapped to GemStone/TriGS. Allowing only output ports that fulfil these requirements is not a severe restriction: Usually, objects that are created or manipulated by an activity realization are related directly or indirectly to the object for which the activity is defined. This allows to impose a postcondition on created or manipulated objects by navigating to them.

Within this thesis, we adopt the latter approach and restrict the set of allowed output ports at the level of Active Object/Behavior Diagrams. This does not represent a severe restriction and saves a developer from composing bulky result structures.

The postcondition of an activity may refer to an object as it has been before activity execution. Placeholders have to be defined in the preamble of a postcondition to refer to values as they have before activity execution. Before an activity is executed, the preamble is evaluated and current values are bound to the placeholders. The evaluation of the preamble is done by a separate method. For an activity a with the (non-primary) input ports $p_1, p_2, \ldots, p_n$ that defines placeholders $ph_1, ph_2, \ldots, ph_m$, a method with selector

```
preamble_aWithP1 : p1 withP2 : p2 ... withPn : pn.
```

is introduced. This method returns a dictionary that contains for each placeholder ph_i in $ph_1, ph_2, \ldots, ph_m$ its current value. The values of the placeholders can be used when the postcondition of the activity is checked after activity execution. Section 9.2.1 will show when method `preamble_a` is performed and how the values of placeholders can be passed to the postcondition check.

The postcondition check of an activity a with the (non-primary) input ports $p_1, p_2, \ldots, p_n$, with the return port p_{ret}, and with the placeholders $ph_1, ph_2, \ldots, ph_m$ is performed by a separate method with selector

```
post_aWithP1 : p1 withP2 : p2 ... withPn : pn withPret : pret
    withPh1 : ph1 withPh2 : ph2 ... withPhm : phm
```

Both, the method evaluating the preamble of the postcondition and the method checking the postcondition of an activity are instance methods. They are not introduced as class methods for creation activities.

9.2 Integrity Maintenance

This section discusses integrity constraints related to object behavior and object structure and shows how they can be maintained in GemStone/TriGS. An activity execution

preserves the integrity of a database if the precondition holds before activity execution, if the postconditon holds after activity execution, if state changes are effected correctly, and if no integrity constraints related to object structure are violated. Integrity constraints related to object structure, which have been introduced in the course of Section 8.1, are invariant properties, obligatory properties, unique keys, subclass/superclass relationships, inverse relationships, referential integrity, and local referential integrity.

In Active Object/Behavior Diagrams, public activities are the "units of integrity". This means that public activities have to preserve the integrity of a database by transforming it from one database state satisfying integrity constraints to another. Active Object/Behavior Diagrams follow the principle of object autonomy. It is the responsibility of a developer to maintain integrity constraints. This means that a violation of an integrity constraint is not countered actively by the system (e.g., by means of an action compensating the violation) but that it is countered passively by preventing a violating operation to be completed. An attempt to violate an integrity constraint represents a non-recoverable error that has to be signaled to the user causing it.

Besides object autonomy, the following goals have affected the way we check integrity constraints in GemStone/TriGS:

- *Efficiency:* It would be sufficient to check integrity constraints generally at the end of methods representing public activities. Such an approach would result in rather time-consuming checks: Any operation performed by a method representing a public activity may have caused a violation and, therefore, the set of potentially violated constraints may be extensive. To avoid this, we try to couple the checking of an integrity constraint closely with operations that potentially violate it. As far as possible, checking of integrity constraints is done immediately before or after the execution of operations.

- *Autonomy of activity implementation:* As already mentioned above, we assume that activities are likely to be implemented at the logical level rather than at the conceptual level. The developer implementing activities should be freed from checking integrity constraints. To achieve the desired autonomy of activity implementation, rules are engaged to check integrity constraints and to signal errors in the caise of violations. Methods representing public activities and methods for accessing properties are the main anchors upon which rules for checking integrity constraints are defined. Checking integrity constraints by means of rules additionally brings the additional benefit that checking can be deactivated and reactivated at any time without side effects. Deactivation increases the performance of the system and may make sense when a system behaves properly after a testing phase. Reactivation may become necessary when changes are made to activity implementations.

In the following, it is shown how the different kinds of integrity constraints can be maintained. For every kind of integrity constraint, the set of critical operations is identified

```
DEFINE RULE O_a_PreconditionCheck
  ON PRE(O, aWithP1: p1 withP2: p2 ... withPn: pn)
  IF ((event trgObj enabled_a &
      event trgObj pre_aWithP1: p1 withP2: p2 ... withPn: pn) not) THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #precondViol)
                 args: #[conditionEvent trgMeth,
                         conditionEvent trgObj,
                         conditionEvent params]
                 signalDictionary: AOBDErrors.
  WITH PRIORITY 1
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_a_PreconditionCheck
```

Figure 9.1: Rule for checking the precondition of an activity

and rules for checking the constraint are presented. If a constraint violation is detected, an error is reported to the user issuing the violating operation. GemStone/TriGS provides a mechanism for signaling errors to users (cf. [Gem96c, Chapter 11]). When a signal is sent, GemStone/TriGS halts execution and reports the error to the user. A user may either take provisions to handle a signal automatically or he/she may terminate the execution of the violating activity and abort the running transaction manually. In Section 9.2.10, the different types of errors, together with their associated parameters, are summarized.

The execution of a single method may trigger multiple rules. The triggered rules may be interdependent in the sense that the changes effected by one rule may influence the execution of another rule. To determine an execution order for rules, different priorities are assigned to them. A summary of the dependencies among rules and the priorities asssigned to resolve these dependencies is also given in Section 9.2.10.

9.2.1 Preconditions and Postconditions

The execution of an activity is mapped to the execution of a method with preceding precondition check and succeeding postcondition check. The precondition of an activity represents a necessary condition. If the precondition fails, an activity cannot be executed.

Figure 9.1 shows the rule for checking the precondition of an activity `a` of object type `O` with the (non-primary) input ports $p_1, p_2, \ldots, p_n$ and the return port p_{ret}. Rule `O_a_PreconditionCheck` checks whether the receiver of activity `a` is enabled and whether all other parts of the precondition are satisfied. If the check fails, error `#precondViol` is signaled.

The postcondition of an activity represents an assertion on the database state after activity execution. If the postcondition fails, an inconsistency between the implementation of an activity (specified procedurally) and the asserted database state (specified declaratively)

```
DEFINE RULE O_a_PreambleEvaluation
  ON PRE(O, aWithP1: p1 withP2: p2 ... withPn: pn)
  EXECUTE  conditionEvent trgObj preamble_aWithP1: p1 withP2: p2 ... withPn: pn.
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O_a_PreambleEvaluation


DEFINE RULE O_a_PostconditionCheck
  ON (POST(O, preamble_aWithP1:withP2:... withPn:);
      POST(O, aWithP1: p1 withP2: p2 ... withPn: pn))
      /RECGCM [(event left tid = event right tid) &
               (event left trgObj == event right trgObj)]
  IF ((event right trgObj post_aWithP1: p1 withP2: p2 ... withPn: pn
       withPret: event right retVal
       withPh1: event left retVal at: ph1
       withPh2: event left retVal at: ph2
       ⋮
       withPhm: event left retVal at: phm) not) THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #postcondViol)
                 args: #[conditionEvent right trgMeth,
                         conditionEvent right trgObj,
                         conditionEvent right params,
                         conditionEvent right retVal,
                         conditionEvent left retVal]
                 signalDictionary: AOBDErrors.
  WITH PRIORITY 2
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_a_PostconditionCheck
```

Figure 9.2: Rules for checking the postcondition of an activity

becomes apparant.

Figure 9.2 shows the rules for checking the postcondition of an activity a of object type O with the (non-primary) input ports $p_1, p_2, \ldots, p_n$, with the return port p_{ret}, and with the placeholders $ph_1, ph_2, \ldots, ph_m$. Rule `O_a_PreambleEvaluation` is triggered before activity a is executed. In the rule's action, the current value of every placeholder ph_i in $ph_1, ph_2, \ldots, ph_m$ is computed.

The event part of rule `O_a_PostconditionCheck` has to be considered more closely: Rule `O_a_PostconditionCheck` is triggered by a sequence event. Detection of the sequence event is initiated when the preamble of activity a is evaluated before activity a is executed (see rule `O_a_PreambleEvaluation`). Detection of the sequence event is terminated after activity a has been executed. Parameter context "recent" (`/REC`)[2], the specified guard, and the set

[2]Since the initiating event and the terminating event always occur alternately, also parameter context

```
DEFINE RULE O_a_StateChangesAtBegin
  ON PRE(O, a)
  EXECUTE conditionEvent trgObj leaveState: pre_1.
          conditionEvent trgObj leaveState: pre_2.
           ⋮
          conditionEvent trgObj leaveState: pre_m.
          conditionEvent trgObj enterState: a.
  WITH PRIORITY 3
  TRANSACTION MODES (A:SERIAL)
END RULE O_a_StateChangesAtBegin


DEFINE RULE O_a_StateChangesAtEnd
  ON POST(O, a)
  EXECUTE conditionEvent trgObj leaveState: a.
          conditionEvent trgObj enterState: post_1.
          conditionEvent trgObj enterState: post_2.
           ⋮
          conditionEvent trgObj enterState: post_n.
  WITH PRIORITY 1
  TRANSACTION MODES (A:SERIAL)
END RULE O_a_StateChangesAtEnd
```

Figure 9.3: Rules performing state changes

guard/context modification flag (`GCM`) guarantee that a terminating event is paired with the last initiating event that has occurred on the same object within the same transaction (cf. Section 7.1.2). The incorporation of the preamble evaluation into the triggering event allows to access the return value of the evaluation and, thus, the values of the placeholders as they have been bound before activity execution. Rule `O_a_PostconditionCheck` checks the postcondition of activity `a` and signals error `#postcondViol` if the check fails.

9.2.2 State Changes

When an activity is performed on an object, the set of basic states in which the object resides changes. State changes have to be performed at the begin and at the end of an activity execution. In order to save an activity implementor from coding state changes as part of activity implementations, state changes are effected by rules. Assume an object-type O for which an activity a with prestates $\mathsf{pre}_1, \mathsf{pre}_2, \ldots, \mathsf{pre}_m$ and poststates $\mathsf{post}_1, \mathsf{post}_2, \ldots, \mathsf{post}_n$ is defined. Figure 9.3 shows rules `O_a_StateChangesAtBegin` and `O_a_StateChangesAtEnd`, which perform state changes at the begin and at the end of activity `a`.

"chronicle" could be used.

```
DEFINE RULE O_p_InvariantPropCheck
  ON PRE(O, p: anObject)
  IF (event trgObj p notNil) THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #invariantPropViol)
                 args: #[conditionEvent trgObj, p]
                 signalDictionary: AOBDErrors.
  WITH PRIORITY 1
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_p_InvariantPropCheck
```

Figure 9.4: Rule ensuring property invariance

9.2.3 Invariant Properties

Key properties and object-defining relationships are invariant, once initialized their value must not be changed throughout the lifetime of an object. Key properties and object-defining relationships are always single-valued.

Consider invariant property `p` of object type `O`: Figure 9.4 shows rule `O_p_InvariantPropCheck`, which signals error `#invariantPropViol` if an attempt is made to manipulate property `p` after it has been initialized.

9.2.4 Obligatory Properties

Key properties and object-defining relationships are obligatory, i.e., after creation of a new instance, they must not be *nil*. To guarantee this, after every creation activity it has to be checked whether all key properties and object-defining relationships have been initialized with a value not *nil*. Obligatory properties at the same time are invariant properties (cf. Section 9.2.3). Invariance guarantees that obligatory properties cannot take *nil* after they have been initialized.

Consider an object type `O` with the obligatory properties $p_1, p_2, \ldots, p_n$ and with the creation activities $c_1, c_2, \ldots, c_m$: Obligatory properties have to be checked after every activity c_i. Figure 9.5 shows rule `O_`c_i`_ObligatoryPropCheck`, which signals error `#obligatoryPropViol` if the obligatory properties of object type `O` have not been initialized correctly by activity c_i.

9.2.5 Unique Keys

A property of an object type may be specified to be a key property. The key of an object type serves to identify a direct or indirect instance of the object type externally. A system

```
DEFINE RULE O_ci_ObligatoryPropCheck
  ON POST(O, ci)
  IF (event trgObj p1 isNil |
      event trgObj p2 isNil |
      ⋮
      event trgObj pn isNil) THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #obligatoryPropViol)
                args: #[conditionEvent trgObj]
                signalDictionary: AOBDErrors.
  WITH PRIORITY 3
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_ci_ObligatoryPropCheck
```

Figure 9.5: Rule checking obligatory properties

providing keys has to ensure that key values are unique. To perform the uniqueness check efficiently, it is necessary to have all direct and indirect instances of an object type at hand. Since GemStone/TriGS does not maintain class extensions automatically, classes that represent object types are extended to maintain their direct and indirect instances.

Class `AOBDObject`, the root class of every GemStone/TriGS class that represents an object type, defines a class instance variable `extension`. Instances can be added to the extension by means of class method `addInstance: anInstance`, they can be removed from the extension by means of class method `removeInstance: anInstance`.

Adding instances to and removing instances from the extension of an object type is effected by rules. Figure 9.6 shows the rules necessary for maintaining the extension of an object type O_0 with the subclasses $O_1, O_2, \ldots, O_n$: Rule `O0_InstanceCreation` inserts every new (direct or indirect) instance of O_0 into the extension of O_0.[3] Class method `new` can serve as anchor for object creation since it is performed by every creation activity. Contrary to object creation, there is no particular method that can serve as an anchor for object deletion. So, every activity removing an object semantically has to be monitored separately. Rule `Oi_rji_InstanceRemoval` removes an object from the extension of O_0 after removal activity r_j^i of object type O_i has been performed on the object.

To guarantee that keys are unique, key values have to be checked after every creation activity. Key properties are obligatory (cf. Section 9.2.4) as well as invariant (cf. Section 9.2.3). To be obligatory ensures that a key property is not *nil* after object creation. To be invariant guarantees that a key property cannot be changed after it has been initialized.

Consider an object type O_0 with the subtypes $O_1, O_2, \ldots, O_n$ and with the key properties $p_1, p_2, \ldots, p_m$: Uniqueness of the key properties has to be checked after every creation

[3] Note, the rule inheritance mechanism of GemStone/TriGS guarantees that the rule-triggering event occurs not only with class O_0 but also with the subclasses of O_0.

```
DEFINE RULE O0_InstanceCreation
  ON POST (O0, CLASSMETHOD new)
  EXECUTE O0 addInstance: (conditonEvent retVal).
  WITH PRIORITY 1
  TRANSACTION MODES (A:SERIAL)
END RULE O0_InstanceCreation


DEFINE RULE Oi_rji_InstanceRemoval
  ON POST (Oi, rji)
  EXECUTE O0 removeInstance: (conditionEvent trgObj).
  WITH PRIORITY 5
  TRANSACTION MODES (A:SERIAL)
END RULE Oi_rji_InstanceRemoval
```

Figure 9.6: Rules for extension management

```
DEFINE RULE Oi_cji_UniqueKeyCheck
  ON POST(Oi, cji)
  IF (O0 extension select: [:e |
     (event trgObj p1 = e p1) &
     (event trgObj p2 = e p2) &
     ⋮
     (event trgObj pn = e pn)]) size ~= 1 THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #uniqueKeyViol)
                 args: #[conditionEvent trgObj]
                 signalDictionary: AOBDErrors.
  WITH PRIORITY 4
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE Oi_cji_UniqueKeyCheck
```

Figure 9.7: Rule checking uniqueness of key

```
DEFINE RULE O_c_i^sub_SubclassCheck
  ON PRE (O, addToC_i^sub: element)
  IF (event trgObj c^sup includes: element) not THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #subclassViol)
                 args: #[conditionEvent trgObj, element, C_i^sub]
                 signalDictionary: AOBDErrors.
  WITH PRIORITY 1
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_c_i^sub_SubclassCheck


DEFINE RULE O_c^sup_SuperclassCheck
  ON PRE (O, removeFromC^sup: element)
  IF ((event trgObj c_1^sub includes: element) |
     (event trgObj c_2^sub includes: element) |
     ⋮
     (event trgObj c_n^sub includes: element)) THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #superclassViol)
                 args: #[conditionEvent trgObj, element, C^sup]
                 signalDictionary: AOBDErrors.
  WITH PRIORITY 1
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_c^sup_SuperclassCheck
```

Figure 9.8: Rules checking subclass/superclass relationships

activity c_j^i of object type O_i. Figure 9.7 shows rule `O_i_c_j^i_UniqueKeyCheck`, which signals error `#uniqueKeyViol` if the key is not unique after performing creation activity c_j^i.

9.2.6 Subclass/Superclass Relationship

Local object classes of an object type may be organized in a class hierarchy, where the set of members of a subclass must be a subset of the set of members of its superclass. The subset constraint may be violated by two operations: (1) An object is added to a subclass without adding it to the superclass previously. (2) An object is removed from the superclass without removing it from all subclasses in which it is contained previously.

Consider a subclass hierarchy between a superclass c^{sup} and its subclasses $c_1^{sub}, c_2^{sub}, \ldots, c_n^{sub}$ at object type O: Figure 9.8 shows the rules checking subclass/superclass relationships. Rule `O_c_i^sub_SubclassCheck` signals error `#subclassViol` if an element not contained in superclass c^{sup} is added to a subclass c_i^{sub}. Rule `O_c^sup_SuperclassCheck` if an element contained in a subclass c_i^{sub} is removed from superclass c^{sup}.

9.2.7 Inverse Relationships

Two directed relationships may be declared to be inverses. Contrary to the integrity constraints presented so far, checking of inverse relationships cannot be done immediately upon operations performed within a public activity but has to be deferred until the end of a public activity. The correct maintenance of inverse relationships can be checked efficiently only if a record of updates to relationships is available. If no such record is maintained, all relationships for which an inverse is specified have to be checked.

For that purpose, GemStone/TriGS is extended by an *Inverse Relationship Manager (IRM)*, which maintains a record of updates to relationships. For each session, the IRM maintains a session record. A session record keeps track of missing references, i.e., of references that have to be established in order to satisfy inverse declarations. Since a session represents a thread of control within which public activities are processed sequentially, the missing references contained in a session record correspond to references not maintained correctly by the currently processed public activity. Consider a relationship r of object type O and its inverse relationship r' of object type O'. If object o (of type O) references object o' (of type O') via relationship r but o' does not reference o via relationship r', there is a missing reference from o' to o via relationship r'. This missing reference is reflected by the session record of the activity that has established the reference from o to o' via r. A public activity can be completed only if the session record of the session within which the activity is executed is empty, i.e., if there is no object for which an inverse relationship is not maintained correctly.

The IRM provides the following methods:

- `addRefFrom: obj1 to: obj2 via: rel1 inverse: rel2`
  Tells the IRM that a reference from object obj_1 to object obj_2 via relationship with name rel_1 has been established. If the session record of the performing session includes a missing reference from obj_1 to obj_2 via rel_1, this missing reference is deleted. If the session record does not include a missing reference from obj_1 to obj_2 via rel_1, a missing reference from obj_2 to obj_1 via rel_2 is added.

- `removeRefFrom: obj1 to: obj2 via: rel1 inverse: rel2`
  Tells the IRM that the reference from object obj_1 to object obj_2 via relationship with name rel_1 has been removed. If the session record of the performing session includes a missing reference from obj_2 to obj_1 via rel_2, this missing reference is deleted. If the session record does not include a missing reference from obj_2 to obj_1 via rel_2, a missing reference from obj_1 to obj_2 via rel_1 is added.

- `sessionRecord`
  Returns the session record of the inquiring session. If the session record is empty, i.e., if there is no missing reference, the method returns *nil*.

```
DEFINE RULE O_r_IRMAssignment
  ON PRE (O, r: anObject)
  EXECUTE (conditionEvent trgObj r notNil)
             ifTrue: [IRM removeRefFrom: conditionEvent trgObj
                      to: conditionEvent trgObj r
                      via: r inverse: r'].
          (anObject notNil)
             ifTrue: [IRM addRefFrom: conditionEvent trgObj
                      to: anObject via: r inverse: r'].
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O_r_IRMAssignment
```

Figure 9.9: Rule for IRM notification (single-valued relationship)

```
DEFINE RULE O_r_IRMInsertion
  ON PRE (O, addToR: anObject)
  IF ((event trgObj r includes: anObject) not) THEN
  EXECUTE IRM addRefFrom: conditionEvent trgObj
              to: anObject
              via: r inverse: r'.
  WITH PRIORITY 2
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_r_IRMInsertion


DEFINE RULE O_r_IRMRemoval
  ON PRE (O, removeFromR: anObject)
  IF (event trgObj r includes: anObject) THEN
  EXECUTE IRM removeRefFrom: conditionEvent trgObj
              to: anObject
              via: r inverse: r'.
  WITH PRIORITY 2
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_r_IRMRemoval
```

Figure 9.10: Rules for IRM notification (multi-valued relationship)

```
DEFINE RULE O_a_InverseRelationshipCheck
  ON POST (O, a)
  IF (IRM sessionRecord) THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #inverseRelViol)
                 args: #[conditionResult]
                 signalDictionary: AOBDErrors.
  WITH PRIORITY 4
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_a_InverseRelationshipCheck
```

Figure 9.11: Rule for checking inverse relationships

The IRM is notified of updates to relationships by means of rules. Consider a relationship r of object-type O with inverse relationship r' of object-type O'. What rules are defined for notifying the IRM depends on whether relationship r is single-valued or multi-valued. Figure 9.9 shows the rule generated if r is a single-valued relationship. Rule `O_r_IRMAssignment` informs the IRM when an object is assigned to r. Figure 9.10 shows the rules generated if r is a multi-valued relationship. Rule `O_r_IRMInsertion` informs the IRM when an object not already contained in r is added to r, rule `O_r_IRMRemoval` informs the IRM when an object contained in r is removed from r. (As already mentioned in Section 8.1, all methods for updating multi-valued properties are reduced to adding an element and removing an element.)

At the end of every public activity it has to be checked whether the session record of the session performing the activity is empty. Consider a public activity a defined for an object type O: Figure 9.11 shows rule `O_a_InverseRelationshipCheck`, which signals error `#inverseRelViol` if the session record is not empty after performing activity a.

9.2.8 Referential Integrity

In the common understanding, referential integrity requires that there are no "dangling pointers", i.e., that there are no references to non-existing objects. In GemStone/TriGS, referential integrity violations in the sense of "dangling pointers" cannot occur: objects cannot be deleted explicitly by programs, but they are deleted by a garbage collector if they are not referenced from other objects any more.

In the context of Active Object/Behavior Diagrams, referential integrity has to be considered rather on a semantical level: objects that have been removed must not be referenced. Generally, a removal activity is expected to remove all references to the object on which the activity is performed. If a removal activity does not meet this expectation, an object remains referenced after its removal and, thus, violates referential integrity.

GemStone/TriGS provides method `findReferences` as instance method of class `Object` to

```
DEFINE RULE O_r_Assignment
  ON PRE (O, r: anObject)
  EXECUTE (conditionEvent trgObj r notNil)
             ifTrue: [conditionEvent trgObj r
                          removeReference: conditionEvent trgObj].
          (anObject notNil)
             ifTrue: [anObject addReference: conditionEvent trgObj].
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O_r_Assignment


DEFINE RULE O_r_Insertion
  ON PRE (O, addToR: anObject)
  IF ((event trgObj r includes: anObject) not) THEN
  EXECUTE anObject addReference: conditionEvent trgObj.
  WITH PRIORITY 2
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_r_Insertion


DEFINE RULE O_r_Removal
  ON PRE (O, removeFromR: anObject)
  IF (event trgObj r includes: anObject) THEN
  EXECUTE anObject removeReference: conditionEvent trgObj.
  WITH PRIORITY 2
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_r_Removal
```

Figure 9.12: Rules for maintaining references

search for objects that reference the receiver of the method. Since the whole repository has to be searched, execution of method `findReferences` is very time-consuming and, therefore, not the appropriate way to determine whether an object has been removed correctly. To maintain referential integrity efficiently, every object keeps track of the objects referencing it. Class `AOBDObject`, which serves as direct or indirect superclass for all GemStone/TriGS classes that represent object types, defines variable `references` for its instances.

Updates to instance variable `references` are performed by means of rules. Every time an object `o` is assigned to a single-valued relationship, added to a multi-valued relationship, or removed from a multi-valued relationship, rules are triggered that update the set of references maintained at `o`. Figure 9.12 shows rules monitoring relationship `r` of object type `O`. If relationship `r` is single-valued, rule `O_r_Assignment` informs the old value of `r` that a reference is removed and the new value of `r` that a reference is added. If relationship `r` is multi-valued, rule `O_r_Insertion` adds a reference to the assigned object, and rule `O_r_Removal` removes a reference to the assigned object.

```
DEFINE RULE O_r_i_ReferentialIntegrityCheck
  ON POST (O, r_i)
  IF (event trgObj references notEmpty) THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #refIntViol)
                args: #[conditionEvent trgObj,
                        conditionEvent trgObj references]
                signalDictionary: AOBDErrors.
  WITH PRIORITY 4
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_r_i_ReferentialIntegrityCheck
```

Figure 9.13: Rule for checking referential integrity

At the end of every removal activity it has to be checked whether all references to a removed object have been deleted. Consider a removal activity r_i of an object type O: Figure 9.13 shows rule `O_r_i_ReferentialIntegrityCheck`, which signals error `#refIntViol` if there are references to an instance of O after removal activity r_i has been performed on it.

9.2.9 Local Referential Integrity

The definition of a local referential integrity constraint that applies to the instances of an object type O consists of (cf. Section 4.3):

1. a locality path P_{source} (source path): $O^s_0.c^s_1[O^s_1].c^s_2[O^s_2].\cdots.c^s_n[O^s_n]$, where $O^s_0 = O$, where c^s_i denotes a local object class maintained at the objects of type O^s_{i-1}, and where O^s_i represents the type of elements in c^s_i to which the path is restricted;
2. a directed relationship r of object type O^s_n;
3. a locality path P_{target} (target path): $O^t_0.c^t_1[O^t_1].c^t_2[O^t_2].\cdots.c^t_m[O^t_m]$, where $O^t_0 = O$, where c^t_i denotes a local object class maintained at the objects of type O^t_{i-1}, and where O^t_i represents the type of elements in c^t_i to which the path is restricted.

Like the checking of inverse relationships, also the checking of local referential integrity cannot be done immediately upon operations performed within a public activity, but has to be deferred until the end of a public activity. The correct maintenance of local referential integrity can be checked efficiently only if updates to local object classes and relationships that are used in the specification of a local referential integrity constraint are recorded, i.e., if objects participating in local referential integrity constraints are monitored.

There are several ways an object can *participate* in a local referential integrity constraint. Figure 9.14 shows these different ways by means of an exemplary local referential integrity constraint and an exemplary object structure. The definition of the shown local referential

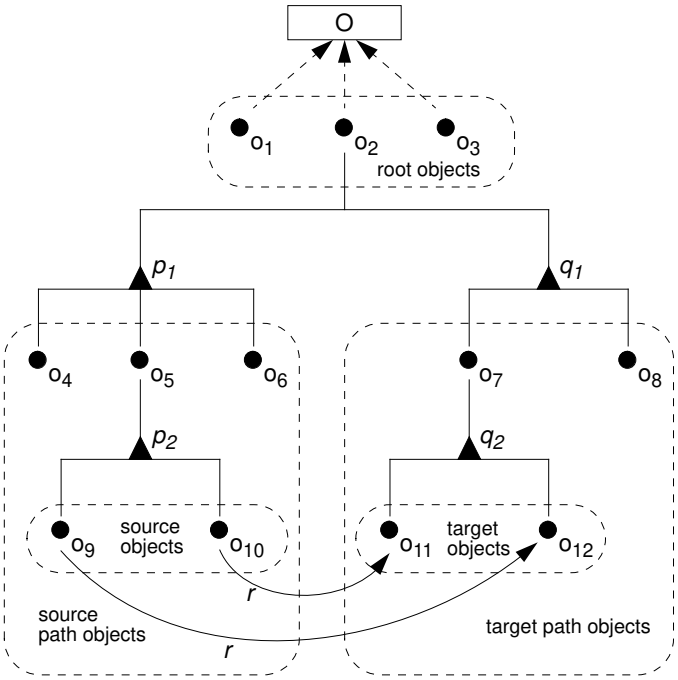


Figure 9.14: Participants in a local referential integrity constraint

integrity constraint consists of source path $O.p_1.p_2$, directed (general) relationship r, and target path $O.q_1.q_2$. The symbols used in Figure 9.14 have the following meaning: An object type is depicted by a rectangle showing the name of the object type inside. An object is symbolized by a filled bullet, where the identifier of the object is associated with the bullet. A relationship is depicted by a line to which the name of the relationship is associated. A line that forks at a filled triangle represents a local object class. A solid arrow represents a general relationship. A dashed arrow represents an instance-of relationship.

The instances of the object type for which a local referential integrity constraint is defined participate in the constraint as *root objects*. In the exemplary object structure, the instances of O (o_1, o_2, and o_3) are root objects. All the objects that can be reached by navigating along the source path, starting the navigation at a root object, participate in the local referential integrity constraint as *source path objects*. The sets of source path objects spanned by two different root objects of a local referential integrity constraint are disjoint. The subset of source path objects that reside in the leaf class of the source path are referred to as *source objects*. In the example, the source path objects spanned by root object o_2 are the members of local object class p_1 maintained at o_2 (i.e., o_4, o_5, and o_6) and the members of local object class p_2 maintained at o_5 (i.e., the source objects o_9 and o_{10}). Correspondingly, all the objects that can be reached by navigating along the target path, starting the navigation at a root object, participate in the local referential integrity constraint as *target path objects*. Also here, the sets of target path objects spanned by two different root objects are disjoint. The objects that reside in the leaf class of the target

path are referred to as *target objects*. In the example, the target path objects spanned by root object o_2 are the members of local object class q_1 maintained at o_2 (i.e., o_7 and o_8) and the members of local object class q_2 maintained at o_7 (i.e., the target objects o_{11} and o_{12}).

GemStone/TriGS is extended by a *Local Referential Integrity Manager (LRIM)* to maintain local referential integrity constraints. For each session, the LRIM maintains a session record. A session record keeps for every LRIC a set of violating source objects. A *violating source object* is a source object that violates a local referential integrity constraint by referencing an object not contained in the target objects of its root object. Since a session represents a thread of control within which public activities are processed sequentially, the violations contained in a session record correspond to violations caused by the currently processed public activity. Local referential integrity is maintained correctly by a public activity if there are no violating source objects after its completion.

Three different scenarios may lead to violating source objects:

- *Scenario 1:* An object o is assigned or added to a relationship of object o', where the domain of the relationship is restricted by a local referential integrity constraint. If o' is a source object of the local referential integrity constraint, and if o is not contained in the target objects of the root object to which o' belongs, o' becomes a violating source object.

- *Scenario 2:* An object o is added to a local class of object o', where the local class is element of the source path of a local referential integrity constraint. By adding o, also the direct and indirect components of o that are located along the source path are added to the source path objects or source objects of the local referential integrity constraint. If an added source object references an object that is not contained in the target objects of its root object, the added source object becomes a violating source object.

- *Scenario 3:* An object o is removed from a local class of object o', where the local class is element of the target path of a local referential integrity constraint. By removing o, also the direct and indirect components of o that are located along the target path are removed from the target path objects or target objects of the local referential integrity constraint. A source object that references a removed target object becomes a violating source object.

The LRIM monitors the source path and the target path of each LRIC to detect violating source objects. Changes to the source path and the target path of an LRIC are signaled to the LRIM by means of rules (see below). Unique identifiers are assigned to LRICs so that they can be identified easily.

The LRIM provides the following methods:

- `addRootObject: obj forLRIC: id`
  Informs the LRIM that a new instance `obj` of the object type for which LRIC `id` is defined has been created.

- `removeRootObject: obj forLRIC: id`
  Informs the LRIM that instance `obj` of the object type for which LRIC `id` is defined has been removed.

- `addPotentialSourcePathObject: obj1 to: obj2 forLRIC: id`
  Informs the LRIM that object `obj1` has been added to the local object class of `obj2` that is contained in the source path of LRIC `id`.

- `removePotentialSourcePathObject: obj1 from: obj2 forLRIC: id`
  Informs the LRIM that object `obj1` has been removed from the local object class of `obj2` that is contained in the source path of LRIC `id`.

- `addPotentialTargetPathObject: obj1 to: obj2 forLRIC: id`
  Informs the LRIM that object `obj1` has been added to the local object class of `obj2` that is contained in the target path of LRIC `id`.

- `removePotentialTargetPathObject: obj1 from: obj2 forLRIC: id`
  Informs the LRIM that object `obj1` has been removed from the local object class of `obj2` that is contained in the target path of LRIC `id`.

- `addReferenceFrom: obj1 to: obj2 forLRIC: id`
  Informs the LRIM that a reference has been established from object `obj1` to `obj2` via the relationship constrained by LRIC `id`.

- `removeReferenceFrom: obj1 to: obj2 forLRIC: id`
  Informs the LRIM that a reference from object `obj1` to `obj2` via the relationship constrained by LRIC `id` has been removed.

- `sessionRecord`
  Returns the session record of the inquiring session. If the session record is empty, i.e., if there are no violating source objects, the method returns *nil*.

Changes to the root objects, to the source path, and to the target path are signaled to the LRIM by means of rules. Consider a local referential integrity constraint with the source path $O_0^s.c_1^s[O_1^s].c_2^s[O_2^s].\cdots.c_n^s[O_n^s]$, where $O_0^s = O$, the constrained relationship r, the target path $O_0^t.c_1^t[O_1^t].c_2^t[O_2^t].\cdots.c_m^t[O_m^t]$, where $O_0^t = O$, and the internal identifier id:

Figure 9.15 shows the rules generated for inserting and removing root objects. After a new instance of object type O has been created, rule `O_LRIMRootObjectInsertion` informs the LRIM that a new instance of object type O (for which LRIC `id` is defined) has been created. After a removal activity r_i has been performed on an instance of O, rule `O_`r_i`_LRIMRootObjectRemoval` informs the LRIM about the removal.

```
DEFINE RULE O_LRIMRootObjectInsertion
  ON POST (O, CLASSMETHOD new)
  EXECUTE LRIM addRootObject: (conditionEvent retVal) forLRIC: id.
  WITH PRIORITY 1
  TRANSACTION MODES (A:SERIAL)
END RULE O_LRIMRootObjectInsertion


DEFINE RULE O_r_i_LRIMRootObjectRemoval
  ON POST (O, r_i)
  EXECUTE LRIM removeRootObject: (conditionEvent trgObj) forLRIC: id.
  WITH PRIORITY 5
  TRANSACTION MODES (A:SERIAL)
END RULE O_r_i_LRIMRootObjectRemoval
```

Figure 9.15: Rules for LRIM root object maintenance

Figure 9.16 shows the rules generated for informing the LRIM about (potential) changes to the source path of LRIC id. For every local class c_i^s rules are generated: If c_i^s is single-valued, rule $O_{i-1}^s_c_i^s$_LRIMSourceAssignment informs the LRIM about the deletion of the old value of c_i^s and about the assignment of a new value to c_i^s. If c_i^s is multi-valued, two rules are engaged: Rule $O_{i-1}^s_c_i^s$_LRIMSourceInsertion informs the LRIM that an object has been added to c_i^s and rule $O_{i-1}^s_c_i^s$_LRIMSourceRemoval informs the LRIM that an object has been removed from c_i^s.

Figure 9.17 shows the rules generated for informing the LRIM about (potential) changes to the target path of LRIC id. For every local class c_i^t rules are generated: If c_i^t is single-valued, rule $O_{i-1}^t_c_i^t$_LRIMTargetAssignment informs the LRIM about the deletion of the old value of c_i^t and about the assignment of a new value to c_i^t. If c_i^t is multi-valued, two rules are engaged: Rule $O_{i-1}^t_c_i^t$_LRIMTargetInsertion informs the LRIM that an object has been added to c_i^t and rule $O_{i-1}^t_c_i^t$_LRIMTargetRemoval informs the LRIM that an object has been removed from c_i^t.

Figure 9.18 shows the rules generated for informing the LRIM about changes to relationship r, which is constrained by LRIC id. If relationship r is single-valued, rule O_n^s_r_LRIMReferenceAssignment informs the LRIM that an old reference has been removed and that a new reference has been established. If relationship r is multi-valued, rule O_n^s_r_LRIMReferenceInsertion informs the LRIM about adding of a new reference to r, and rule O_n^s_r_LRIMReferenceRemoval informs the LRIM about removing of a reference from r.

At the end of every public activity it has to be checked whether the session record of the session performing the activity contains violating source objects. Consider a public activity a defined for an object type O: Figure 9.19 shows rule O_a_LocalReferentialIntegrityCheck, which signals error #localRefIntViol if the session record contains violating source objects after performing activity a.

```
DEFINE RULE O^s_{i-1}_c^s_i_LRIMSourceAssignment
  ON PRE (O^s_{i-1}, c^s_i: anObject)
  EXECUTE (conditionEvent trgObj c^s_i notNil)
             ifTrue: [LRIM
                         removePotentialSourcePathObject: conditionEvent trgObj c^s_i
                         from: conditionEvent trgObj
                         forLRIC: id].
          (anObject notNil)
             ifTrue: [LRIM
                         addPotentialSourcePathObject: anObject
                         to: conditionEvent trgObj
                         forLRIC: id].
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^s_{i-1}_c^s_i_LRIMSourceAssignment


DEFINE RULE O^s_{i-1}_c^s_i_LRIMSourceInsertion
  ON PRE (O^s_{i-1}, addToC^s_i: anObject)
  EXECUTE LRIM
            addPotentialSourcePathObject: anObject
            to: conditionEvent trgObj
            forLRIC: id.
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^s_{i-1}_c^s_i_LRIMSourceInsertion


DEFINE RULE O^s_{i-1}_c^s_i_LRIMSourceRemoval
  ON PRE (O^s_{i-1}, removeFromC^s_i: anObject)
  EXECUTE LRIM
            removePotentialSourcePathObject: anObject
            from: conditionEvent trgObj
            forLRIC: id.
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^s_{i-1}_c^s_i_LRIMSourceRemoval
```

Figure 9.16: Rules for LRIM source path maintenance

```
DEFINE RULE O^t_{i-1}_c^t_i_LRIMTargetAssignment
  ON PRE (O^t_{i-1}, c^t_i: anObject)
  EXECUTE (conditionEvent trgObj c^t_i notNil)
             ifTrue: [LRIM
                         removePotentialTargetPathObject: conditionEvent trgObj c^t_i
                         from: conditionEvent trgObj
                         forLRIC: id].
          (anObject notNil)
             ifTrue: [LRIM
                         addPotentialTargetPathObject: anObject
                         to: conditionEvent trgObj
                         forLRIC: id].
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^t_{i-1}_c^t_i_LRIMTargetAssignment


DEFINE RULE O^t_{i-1}_c^t_i_LRIMTargetInsertion
  ON PRE (O^t_{i-1}, addToC^t_i: anObject)
  EXECUTE LRIM
            addPotentialTargetPathObject: anObject
            to: conditionEvent trgObj
            forLRIC: id.
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^t_{i-1}_c^t_i_LRIMTargetInsertion


DEFINE RULE O^t_{i-1}_c^t_i_LRIMTargetRemoval
  ON POST (O^t_{i-1}, removeFromC^t_i: anObject)
  EXECUTE LRIM
            removePotentialTargetPathObject: anObject
            from: conditionEvent trgObj
            forLRIC: id.
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^t_{i-1}_c^t_i_LRIMTargetRemoval
```

Figure 9.17: Rules for LRIM target path maintenance

```
DEFINE RULE O^s_n_r_LRIMReferenceAssignment
  ON PRE (O^s_n, r: anObject)
  EXECUTE (conditionEvent trgObj r notNil)
            ifTrue: [LRIM
                        removeReferenceFrom: conditionEvent trgObj
                        to: conditionEvent trgObj r
                        forLRIC: id].
          (anObject notNil)
            ifTrue: [LRIM
                        addReferenceFrom: conditionEvent trgObj
                        to: anObject
                        forLRIC: id].
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^s_n_r_LRIMReferenceAssignment


DEFINE RULE O^s_n_r_LRIMReferenceInsertion
  ON PRE (O^s_n, addToR: anObject)
  EXECUTE LRIM
            addReferenceFrom: conditionEvent trgObj
            to: anObject
            forLRIC: id.
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^s_n_r_LRIMReferenceInsertion


DEFINE RULE O^s_n_r_LRIMReferenceRemoval
  ON PRE (O^s_n, removeFromR: anObject)
  EXECUTE LRIM
            removeReferenceFrom: conditionEvent trgObj
            to: anObject
            forLRIC: id.
  WITH PRIORITY 2
  TRANSACTION MODES (A:SERIAL)
END RULE O^s_n_r_LRIMReferenceRemoval
```

Figure 9.18: Rules for LRIM reference maintenance

```
DEFINE RULE O_a_LocalReferentialIntegrityCheck
  ON POST (O, a)
  IF (LRIM sessionRecord) THEN
  EXECUTE System signal: (AOBDErrorSymbols at: #localRefIntViol)
                args: #[conditionResult]
                signalDictionary: AOBDErrors.
  WITH PRIORITY 4
  TRANSACTION MODES (C:SERIAL,A:SERIAL)
END RULE O_a_LocalReferentialIntegrityCheck
```

Figure 9.19: Rule for checking local referential integrity

9.2.10 Summary

In this section, rules have been introduced to effect state changes, to check preconditons and postconditions of activities, and to check integrity constraints related to object structure. All these rules are triggered by a small set of events. The triggering events are:

- `PRE <activity>`: An event of this kind occurs when a method representing an activity is invoked. The rules that are triggered simultaneously check the preconditon of the activity, perform the necessary state changes, and bind values to the placeholders needed to refer to old values in the postcondition of the activity.

- `POST <activity>`: An event of this kind occurs after a method representing an activity has been completed. The rules that are triggered simultaneously perform the necessary state changes, check the postcondition of the activity, check whether inverse relationships are maintained correctly, and check whether local referential integrity is satisfied. In the case of a creation activity, additional rules may be triggered that check whether obligatory properties have been initialized properly and whether the object's key is unique. In the case of a removal activity, other rules check whether all references to the removed object have been deleted, remove the triggering object from auxiliary structures maintained for checking the uniqueness of keys, and inform the local referential integrity manager about the removal of a root object.

- `POST <new>`: An event of this kind occurs after class method `new` has been performed on a class representing an object type. The rules that are triggered simultaneously register the created object with the auxiliary structures maintained for checking the uniqueness of keys and inform the local referential integrity manager about the creation of a root object.

- `PRE <assignToProp>`: An event of this kind occurs before an object is assigned to a single-valued property. The rules that are triggered simultaneously guarantee that an invariant property cannot be changed, update the auxiliary structures maintained for

checking inverse relationships and referential integrity, and inform the local referential integrity manager about changes to the source path, to the target path, and to the relationship constrained by a local referential integrity constraint.

- `PRE <addToProp>`: An event of this kind occurs before an object is added to a multi-valued property. The rules that are triggered simultaneously check compliance with an existing subclass/superclass relationship, insert the added object into the auxiliary structures maintained for checking inverse relationships and referential integrity, and inform the local referential integrity manager about changes to the source path, to the target path, and to the relationship constrained by a local referential integrity constraint.

- `PRE <removeFromProp>`: An event of this kind occurs before an object is removed from a multi-valued property. The rules that are triggered simultaneously check compliance with an existing subclass/superclass relationship, take the removed object from the auxiliary structures maintained for checking inverse relationships, and referential integrity, and and inform the local referential integrity manager about changes to the source path, to the target path, and to the relationship constrained by a local referential integrity constraint.

Rules that are triggered simultaneously by a single event generally are executed in a non-deterministic order. But there are both, cases in which a non-deterministic execution order is incorrect, and cases in which it is inefficient. An execution order is incorrect if dependencies between rules are not obeyed.

Example: After completion of an activity, a rule for effecting the necessary state changes and a rule for checking the postcondition of the activity are triggered simultaneously. Since the postcondition of an activity is satisfied only if the participating objects reside in the appropriate post states, it is vital that the rule effecting the state changes is executed before the rule checking the postcondition.

An execution order may be inefficient if updates to auxiliary structures that cannot fail are made before checks.

Example: Before an object is removed from a local class that is superclass of one or more subclasses, a rule is triggered that checks whether the removed object is not included in a subclass. An additional rule is triggered that updates the auxiliary structure used to check inverse relationships. Generally, there are no dependencies among these two rules triggered simultaneously. Nevertheless, efficiency may differ with different execution orders: Efficiency may be increased if the update to the auxiliary structure for inverse relationships is postponed behind the check of subclass inclusion. If the check of subclass inclusion fails, the update to the auxiliary structure for inverse relationships, which cannot fail, needs not be done.

event/priority	1	2	3	4	5
PRE <activity>	PreconditionCheck	PreambleEvaluation	StateChangesAtBegin		
POST <activity>	StateChangesAtEnd	PostconditionCheck	ObligatoryPropCheck	UniqueKeyCheck, InverseRelationshipCheck, ReferentialIntegrityCheck, LocalReferentialIntegrityCheck	InstanceRemoval, LRIMRootObjectRemoval
POST <new>	InstanceCreation, LRIMRootObjectInsertion				

Table 9.1: Rules and priorities (Part 1)

event/priority	1	2
PRE <assignToProp>	InvariantPropCheck	Assignment, IRMAssignment, LRIMReferenceAssignment, LRIMSourceAssignment, LRIMTargetAssignment
PRE <addToProp>	SubclassCheck	Insertion, IRMInsertion, LRIMSourceInsertion, LRIMReferenceInsertion, LRIMTargetInsertion
PRE <removeFromProp>	SuperclassCheck	Removal, IRMRemoval, LRIMTargetRemoval, LRIMSourceRemoval, LRIMReferenceRemoval

Table 9.2: Rules and priorities (Part 2)

All rules introduced so far are executed in serial mode (i.e., as part of the transaction within which the triggering event occurred). GemStone/TriGS provides the concept of priorities to specify an execution order among several rules that are triggered simultaneously and that have to be executed in serial mode. When mapping Active Object/Behavior Diagrams to GemStone/TriGS, priorities can be used to achieve that dependencies among rules are adhered to, that light checks are performed before heavy ones, and that checks are made before updates to auxiliary structures. Table 9.1 and Table 9.2 summarize for each kind of triggering event the priorities that are assigned to rules.

The rules employed to check integrity constraints signal errors to users if integrity violations occur. The GemStone/TriGS mechanism for signaling errors to users is based on two dictionaries: a dictionary mapping error symbols to internal error numbers and a dictionary containing language specific error texts for these internal error numbers. Therefore, we assume two dictionaries to be defined for every user that works with a GemStone/TriGS database designed with Active Object/Behavior Diagrams: a dictionary `AOBDErrorSymbols` mapping error symbols (e.g. `#precondViol`) to internal numbers and a dictionary `AOBDErrors` containing error texts. A summary of possible integrity violations showing their associated error symbols and arguments is given in Table 9.3.

9.3 Task Model

In Chapter 6, an execution model has been introduced that provides for a reliable, controllable, and temporally faithful execution of start-rules and do-rules. Like all other existing active object-oriented database systems, GemStone/TriGS does not support the proposed execution semantics directly. A module has to be built that implements a reliable, controllable, and temporally faithful execution of rules on top of GemStone/TriGS. When we sketched such a module, we identified different kinds of transactions and a coordinator for the different transactions as the core contributors to a reliable, controllable, and temporally faithful rule execution. During the design process, the idea emerged with us that the different kinds of transactions could also be used to enrich a user's way of interacting with a database system.

The conventional way a user works with a database system follows the principle of *synchronous interaction*. The user starts a transaction, alternately issues operations and takes action upon the delivered results, and finally commits or aborts the transaction. With rules, operations are executed temporally faithfully at a specific point in time (do-rule) or not later than a specific point in time (start-rule). We consider it useful to enable also a normal user to work with a database system in a rule-like manner. A user should be enabled to register an operation with a database system and to determine that the operation has to be executed temporally faithfully at a specific point in time or not later than a specific point in time. It should be the duty of the database system to pick up the registered operations and to execute them correctly. In such a setting, the way the user works with a database

error symbol	*description and arguments*	
`#precondViol`	The precondition of an activity is violated.	
	1	name of activity
	2	object on which activity is invoked
	3	parameters of invocation
`#postcondViol`	The postcondition of an activity is violated.	
	1	name of activity
	2	object on which activity is invoked
	3	parameters of invocation
	4	return value of activity
	5	old values bound to placeholders
`#invariantPropViol`	An invariant property is modified.	
	1	modified object
	2	name of modified property
`#obligatoryPropViol`	The obligatory properties of a new object are not initialized correctly.	
	1	new object
`#uniqueKeyViol`	The key of a new object is not unique.	
	1	new object
`#subclassViol`	An object added to a subclass is not contained in the superclass.	
	1	object holding superclass and subclass
	2	object added to subclass
	3	name of subclass
`#superclassViol`	An object removed from a superclass is still contained in a subclass.	
	1	object holding superclass and subclass
	2	object removed from superclass
	3	name of superclass
`#inverseRelViol`	Inverse relationships are not maintained correctly.	
	1	session record of violating session
`#refIntViol`	A removed object is still referenced.	
	1	removed object
	2	objects referencing the removed object
`#localRefIntViol`	A local referential integrity constraint is violated.	
	1	session record of violating session

Table 9.3: Error signals and their arguments

system follows the principle of *asynchronous interaction*.

In this section, we will show how the way a user works with GemStone/TriGS can be extended by asynchronous interaction. In the presented model, a user interacts with GemStone/TriGS by registering tasks with a task manager. We use the term *task* to refer to both a synchronous and an asynchronous of a user with GemStone/TriGS. The task manager plays the central role in task management. It coordinates the reliable, controllable, and temporally faithful execution of registered tasks.

The rest of this section is organized as follows: In Section 9.3.1, the different kinds of supported tasks are presented. Their execution semantics is defined declaratively and requirements on a module performing task management are derived. In Section 9.3.2, selected aspects of the execution model of GemStone/TriGS are discussed. In Sections 9.3.3 to 9.3.7, it is shown how the requirements on a module performing task management can be met by extending GemStone/TriGS.

9.3.1 Tasks and Transactions

The introduced task model supports both, synchronous and asynchronous interaction with GemStone/TriGS. Synchronous interaction with GemStone/TriGS is provided by *interactive tasks*, asynchronous interaction is provided by *start-tasks* and *do-tasks*. As the names suggest, the execution semantics of start-tasks and do-tasks is very close to the execution semantics of start-rules and do-rules. A start-task is executed some time after it has been placed. If the system does not succeed to complete the execution of a start-task until a specified maximum-delay time, an exception is reported. A do-task is executed temporally faithfully at the point in time that is indicated explicitly when the task is placed.

The operations a user can perform by tasks on a GemStone/TriGS database designed with Active Object/Behavior Diagrams are public activities. To distinguish start-tasks and do-tasks that are available externally to users from start-tasks and do-tasks that are introduced only for internal purposes (cf. Section 9.3.8), the former are referred to as activity-start-tasks and activity-do-tasks:

1. *Activity-start-task:* An activity-start-task is a start-task that executes a single activity if the condition associated with the task is satisfied. The activity is not executed later than an indicated maximum-delay time.

2. *Activity-do-task:* An activtity-do-task is a do-task that executes a single activity if the condition associated with the task is satisfied. The activity is executed at the indicated do-time.

Tasks may be placed with GemStone/TriGS only by registered agents. An *agent* represents a user of the database ("human agent") or represents a database object that implements

exception/task	activity-start-task	activity-do-task	interactive-task
receiver not enabled	definite	definite	−
precondition violated	+	+	−
constraint violated	+	+	+
activity terminated	+	+	+
task canceled	+	+	−
max. delay exceeded	+	−	−

Table 9.4: Exceptions in the different kinds of tasks

some functionality by means of tasks ("software agent", e.g. a rule manager that detects situations and triggers actions by issuing tasks).

Different kinds of exceptions may arise during the execution of a task. Table 9.4 shows which exceptions are relevant for which kinds of tasks. In detail, the exceptions have the following semantics:

- *Receiver not enabled:* The receiver of the activity that has to be performed by an activity-start-task or an activity-do-task is not in the appropriate prestate. Like a rule, activity-start-tasks and activity-do-tasks may be specified to be experimental or definite. The exception "receiver not enabled" is reported only in the case of a definite task. In the case of an experimental task, no exception is reported, and the task is considered to be executed successfully.

- *Precondition violated:* The precondition of the activity that has to be performed by an activity-start-task or an activity-do-task is violated.

- *Constraint violated:* An integrity constraint is violated during or immediately after the execution of the activity that has to be performed by an activity-start-task or an activity-do-task. Constraint violations are signaled to the system as summarized in Section 9.2.10.

- *Activity terminated:* This exception represents the "programmed abort" of an activity. An activity may be implemented to signal error `#activityTerminated` if an exceptional situation is encountered that prevents the activity from being completed. Error `#activityTerminated` is contained in dictionary `AOBDErrorSymbols` and thus can be used in the implementation of every activity designed with Active Object/Behavior Diagrams. This emergency exit frees the developer from extending every activity with parameters that reflect whether the activity could be executed successfully. Note, signaling error `#activityTerminated` is different from aborting the running transaction: After a transaction abort, control remains with the activity that performs the

abort. After signaling the error, control is given back to the user, who then may abort the running transaction.

- *Task canceled:* A task is canceled externally by the agent who initiated the task.
- *Maximum delay exceeded:* A start-task has not been executed until its maximum-delay time.

Regarding the description of task semantics, we follow the approach adopted in Section 6.3. Tasks are executed by performing different kinds of transactions. At first, we specify the semantics of tasks by using an ideal execution model in which transactions are performed instantaneously, i.e., in zero time, but not simultaneously. Then we relax these assumptions and define which schedule of transaction executions in a realistic execution model correctly reflects the semantics of a schedule of transaction executions in the ideal model.

We define the semantics of the different kinds of tasks declaratively by considering the sequence of database states as they are observed at chronon changes and by identifying which kind of transactions must be present in which order in a log of committed transactions.

The following kinds of transactions may appear in the transaction log: *user transactions* (U), which are initiated by users as interactive tasks; *activity-do-task transactions* (D), which comprise the condition/activity pair of an activity-do-task; *activity-start-task transactions* (S), which comprise the condition/activity pair of an activity-start-task; *halt transactions* (H), which report the exceptions of a do-task; *repair transactions* (R), which are initiated by the user upon a halt transaction; and *exception-reporting transactions* (E), which report the exceptions of start-tasks. Further, we consider a subset E^d of E, the set of transactions reporting a delay-exceeded exception.

An interactive task t^i that is completed successfully is reflected in the log of committed transactions by a user transaction U_{t^i}. An aborted interactive task does not appear in the log. An activity-start-task t^s is reflected in the log either by an activity-start-task transaction S_{t^s} or by an exception-reporting transaction E_{t^s}. (If the activity-start-task transaction S_{t^s} is aborted due to some exception, it will not appear in the log of committed transactions. But then, a transaction E_{t^s} reporting the exception will be present in the log.) An activity-do-task t^d is reflected in the log either by an activity-do-task transaction D_{t^d} or by a halt transaction H_{t^d} together with a repair transaction R_{t^d}. Transactions are time-stamped: All transactions that are performed during a chronon i receive time-stamp i (cf. Section 6.3).

A set of tasks is executed correctly up to some database state db_i (the database state at change from chronon i to chronon $i+1$) if the log of committed transactions satisfies the following three criteria (*ideal model*):

1. *Transaction priorities:* Within transactions carrying the same time-stamp, every transaction with a higher priority is executed before any transaction with a lower

priority. Any transaction from the set $D \cup H \cup R$ has a higher priority than any transaction from the set $U \cup S \cup (E \setminus E^d)$, and any transaction from the set $U \cup S \cup (E \setminus E^d)$ has a higher priority than any E^d-transaction.

2. *Activity-do-task execution:* Assume that some activity-do-task t^d has been registered with the explicit execution time j ($j \leq i$) and that the condition of the task is satisfied. The log contains exactly one H_{t^d}-transaction and exactly one R_{t^d}-transaction if (i) the task is definite and the object on which the activity has to be performed is not enabled or (ii) the object is enabled, but the precondition of the activity is violated. The log contains either exactly one H_{t^d}-transaction together with exactly one R_{t^d}-transaction or exactly one D_{t^d}-transaction if the object is enabled and the precondition of the activity is satisfied. The transaction(s) must carry time-stamp j.

3. *Activity-start-task execution:* Assume that some activity-start-task t^s has been registered with the maximum-delay time k and that the condition of the task is satisfied. If $i < k$, the log contains at most one E_{t^s}-transaction, and if $i \geq k$, the log contains exactly one E_{t^s}-transaction in the following cases: (i) the task is definite and the object on which the activity has to be performed is not enabled, (ii) the object is enabled, but the precondition of the activity is violated. If $i < k$, the log contains at most one S_{t^s}-transaction or at most one E_{t^s}-transaction, and if $i \geq k$, the log contains either exactly one S_{t^s}-transaction or exactly one E_{t^s}-transaction if the object is enabled and the precondition of the activity is satisfied. The time-stamp of the transaction contained in the log must be $\leq k$. If the log contains an $E^d_{t^s}$-transaction, its time-stamp must be k.

We assumed for our ideal execution model that transactions are executed instantaneously. We now drop this assumption but we still assign unique time-stamps to them as follows: A D_t-transaction, a H_t-transaction, and a R_t-transaction receives as time-stamp the explicit execution time indicated with task t. An E^d_t-transaction is time-stamped by the maximum-delay time indicated with task t. All remaining kinds of transactions receive as time-stamp the current reading of the system clock when the commit-request is issued by the transaction. We also drop the assumption that transactions are not executed concurrently. To provide a reliable task execution, all transactions (except user transactions and repair transactions) are restarted automatically in the case of a concurrency conflict. Now, we can give the execution semantics for a realistic execution model.

A set of tasks has been executed correctly up to database state db_i if the following requirements are met after all transactions with a time stamp $\leq i$ have been aborted or committed (*realistic model*):

1. *Temporal faithfulness:* Transactions with different time-stamps are serialized according to their time-stamps, i.e., the effect of executing any two transactions T and T' such that the time-stamp of T is less than the time-stamp of T' must be the same as if transaction T' had been executed after T in serial order.

2. *Completeness:* The transaction log satisfies the requirements defined under (2-5) for the ideal model.

3. *Priority observation:* Transactions with the same time-stamp are serialized according to the transaction priorities defined under (1) for the ideal model.

Temporal faithfulness, completeness, and priority observation are the basic requirements on the module performing task management on top of GemStone/TriGS.

9.3.2 Execution Model of GemStone/TriGS

In the following paragraphs, selected aspects of the GemStone/TriGS execution model are presented:

Sessions. The GemStone system provides an *object server*, which consists of a single database monitor process called *Stone*. Access to a GemStone object server is accomplished by establishing a *session*. A process that represents a session and that communicates with the Stone is called *Gem*. Once a session is established, all GemStone activity is carried out in the context of that session. A Session represents a single thread of control. Sessions begin, commit, and abort transactions and, thus, control what changes are made to the repository. Generally, sessions have to be established manually through interface libraries that are available for different programming environments (e.g., GemBuilder for VisualWorks [Gem96b], or GemBuilder for C [Gem96a]). The active capabilities of GemStone/TriGS provide the functionality to spawn new sessions by means of rules. If the transaction mode of a rule's condition and/or action is specified to be `PARALLEL`, the condition is evaluated and/or the action is execution in its own session.

Concurrency control. Transactions are the core concept in concurrency control. A transaction endures from its begin until it is committed successfully or it is aborted. Throughout a transaction, a single view of the database is presented to a session. The view is created and frozen at the begin of the transaction, and modifications to database objects are done only to the private view. At the commit of a transaction, the view of the committing session is merged with the committed data contained in the repository. The concurrency control mechanism of GemStone/TriGS performs optimistic concurrency control with backward certification (cf. [VGH93]). A session is allowed to commit a transaction if there are no concurrency conflicts, otherwise the session has to abort the transaction. When a transaction t requests to commit, GemStone/TriGS checks whether there are concurrency conflicts with concurrent transactions. A transaction t' is *concurrent* to transaction t if t' has committed between the begin of t and the point in time when t requests to commit. Two kinds of concurrency conflicts may be detected:

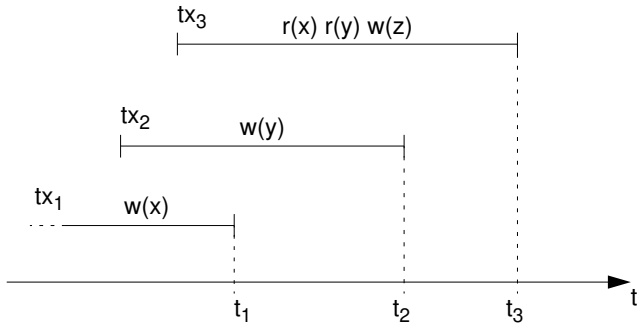


Figure 9.20: Serialization of transactions in GemStone/TriGS

- *Write/write conflict:* An object that has been written by the committing transaction has also been written by a concurrent transaction.
- *Read/write conflict:* An object that has been written by the committing transaction has been read by a concurrent transaction, and another object that has been read by the committing transaction has been written by a concurrent transaction.

The concurrency control mechanism of GemStone/TriGS guarantees that a concurrent execution of transactions corresponds to some serial execution, but does not guarantee that the concurrent execution corresponds to the serial execution following commit points. Figure 9.20 shows the concurrent execution of the three transactions tx_1, tx_2, and tx_3. Transaction tx_1 writes object x and commits at point in time t_1. Transaction tx_2 writes object y and commits at point in time t_2. Transaction tx_3 reads objects x and y, writes object z, and commits at point in time t_3. All three transactions are allowed to commit. The shown concurrent execution is equivalent to the serial executions $tx_3 \prec tx_1 \prec tx_2$ and $tx_3 \prec tx_2 \prec tx_1$, but it is not equivalent to the serial execution following commit points $tx_1 \prec tx_2 \prec tx_3$.

Besides optimistic concurrency control, concurrent access to objects can be controlled also by means of locks (pessimistic concurrency control). GemStone supports three kinds of locks: read locks, write locks, and exclusive locks. GemStone grants a lock-request on an object if no other session holds an incompatible lock. A granted lock is dirty if the locked object is not up to date in the current view. GemStone provides method `System continueTransaction` to bring the current view up to date without aborting or committing a running transaction. Method `continueTransaction` returns true if the running transaction could be committed, i.e., if there are no concurrency conflicts between the running transaction and its concurrent transactions. Once `continueTransaction` has been used within a transaction, a subsequent commit of that transaction ignores read/write conflicts. For more details on concurrency control in GemStone, refer to [Gem96c, Chapter 6].

Exception Handling. Unless an error is fatal to GemStone, it can be handled in applications. To handle errors, a category of errors has to be defined, exception handlers have to

be installed, and errors have to be signaled under appropriate circumstances. Note, an error is signaled to a user via the user interface if no exception handler has been installed for the error. Exception handlers are instances of the kernel class `Exception`. GemStone provides two kinds of exception handlers: activation exception handlers and static-exception handlers. An *activation exception handler* is associated with a method. The handler catches only errors that are signaled while the method is being executed. A *static exception handler* is independent of a method. It takes control in the case of an error for which no other handler has been defined. Static exception handlers are useful for handling errors that appear at unpredictable times. Regarding the flow of control, an activation exception handler may behave in two ways: First, it may behave like an interrupt handler by returning control to the point in method execution where the exception has been raised. Second, it may behave like an emergency exit by terminating the execution of the method and by giving control back to the invoking method. Static exception handlers always behave like interrup handlers. For more details on exception handling in GemStone, refer to [Gem96c, Chapter 11].

Inter-session Communication. GemStone provides two mechanisms that allow several sessions to communicate with each other:

1. *Object change notification:* Object change notifications are signals generated by the object server to inform sessions that specified objects have changed. If a session places an object in its *notify set*, the session receives notification of all committed changes to the object. A session can receive notification of committed changes to the objects in its notify set either by enabling automatic notification or by polling for changes. To trap change notifications automatically, a session has to install an exception handler for the predefined GemStone error (actually an event) `#rtErrSignalCommit`. There is no way to bring a session into a state in which it waits for a change notification passively.

2. *Gem-to-Gem signals:* GemStone enables a session to send a signal to any other current session. Unlike object change notification, signaling happens immediately, without waiting for a commit. A session can receive signals from other session by receiving automatic notification or by polling for signals. To trap signals from other sessions automatically, a session has to install an exception handler for the predefined GemStone error (actually an event) `#rtErrSignalGemStoneSession`. Like for object change notification, there is no way to bring a session into a state in which it waits for a signal passively.

For more details on inter-session communication in GemStone, refer to [Gem96c, Chapter 10].

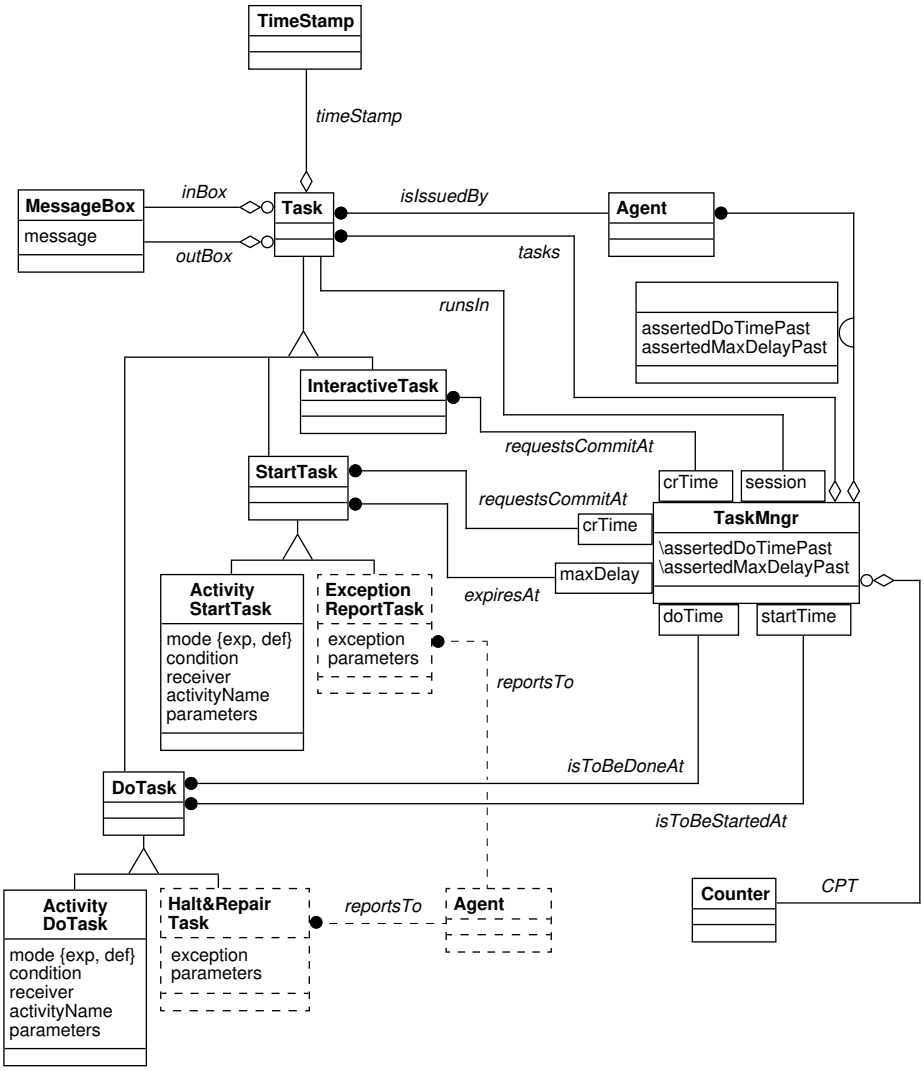


Figure 9.21: Classes implementing task management

9.3.3 Tasks in GemStone/TriGS

In the following, general characteristics of the module performing task management on top of GemStone/TriGS are discussed. The module consists of a set of GemStone/TriGS classes whose structure and behavior have to be specified. To specify object structure, we adopt the object model notation of OMT (cf. [RBP+91]). To specify object behavior, we use an adapted form of life-cycle diagrams (cf. Section 4.4.1). States are represented by rounded rectangles, methods are shown by vertical bars, preconditions and postconditions are depicted directly in the life cycle, and a GemStone/TriGS rule that invokes a method is indicated by an arrow and the rule name above the triggered method.

Figure 9.21 shows the classes performing task management on top of GemStone/TriGS.

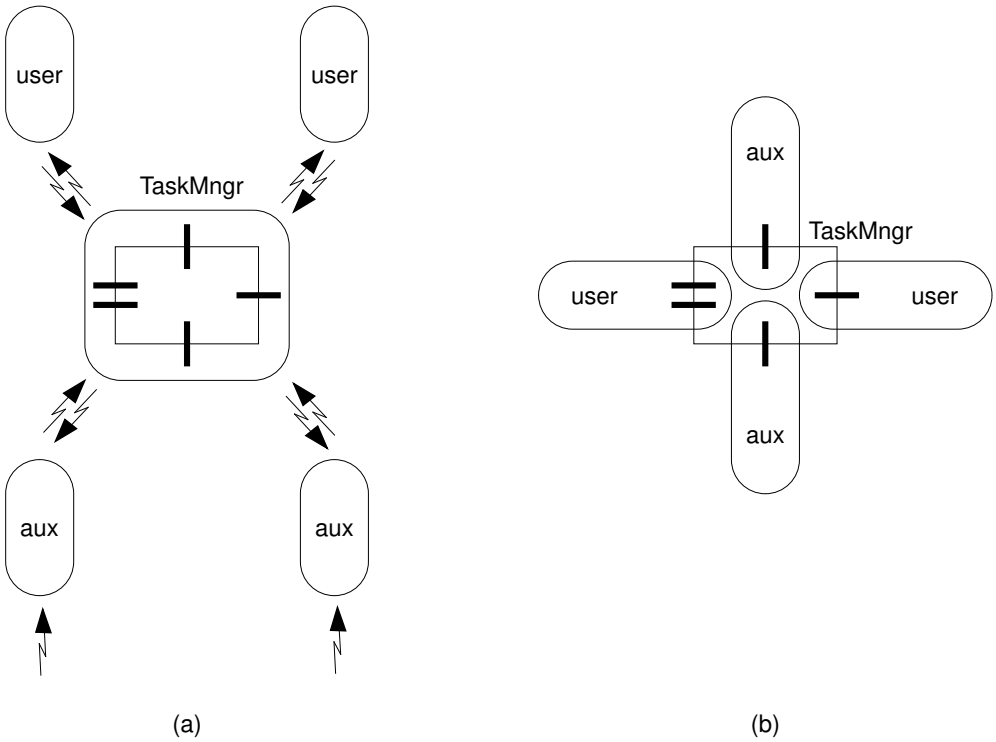


Figure 9.22: Task manager: session or active object?

For the moment ignore the classes shown with dashed borders and their associated relationships. They will be introduced in Section 9.3.8. Although large parts of Figure 9.21 are self-explanatory, some elements will become clear only during Sections 9.3.4–9.3.7 when details of task management are discussed.

Class `TaskMngr` plays the central role in the task management module. It represents the task manager that coordinates the execution of tasks. When designing the task management module, a fundamental question has been whether to incorporate the task manager as (a) a special GemStone/TriGS session continuously running in a loop and receiving signals or as (b) an "active" GemStone/TriGS object.

Figure 9.22 compares the two approaches for incorporating the task manager. The symbols in Figure 9.22 have the following meaning: A rounded rectangle symbolizes a session. An arrow represents an inter-session signal. Finally, a rectangle with slots represents an object and its method interface. Part (a) of Figure 9.22 outlines the task manager as a special GemStone/TriGS session that communicates with other sessions by sending and receiving signals. This approach relies on the mechanism of Gem-to-Gem signaling (cf. Section 9.3.2). The task manager receives signals from users (e.g., task registrations), performs some operations on database objects (e.g., creation of new task objects), and starts auxiliary sessions (e.g., task executions) with which it communicates through signals, too. Part (b) of Figure 9.22 outlines the task manager as a GemStone/TriGS object providing methods

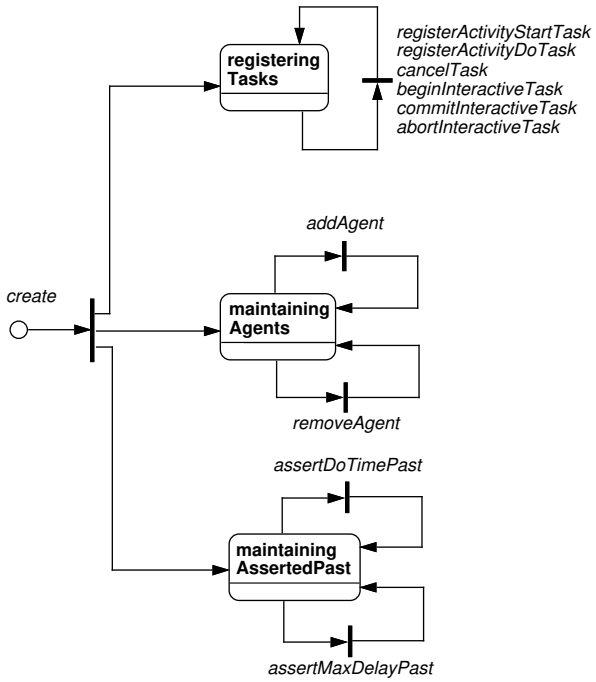


Figure 9.23: External interface of task manager

that are performed within rather short sessions. The methods of the task manager are invoked either within user sessions (e.g., task registrations) or within auxiliary sessions (e.g., task executions) that are spawned by GemStone/TriGS rules (cf. Section 9.3.2). In our opinion, the second approach is more favourable: A session is active only when changes to the task manager have to be actually done. Contrary to the first approach, where the task manager is polling for signals continuously, there are no active waits. The second approach involves a great number of methods that are executed in independent sessions. The execution of a method in an independent session does not require a new session to be established every time. GemStone/TriGS maintains a number of idle sessions waiting for operations to be executed. These considerations motivated the decision to design the task manager as an active GemStone/TriGS object.

Figure 9.23 summarizes the external interface of the task manager, i.e., the interface available for agents that work with the task manager. The task manager provides methods for registering and canceling different kinds of tasks, for adding and removing agents, and for maintaining the asserted past.

Method `registerActivityStartTask` registers an activity-start-task. The properties of the new activity-start-task are passed as parameters, and a task object representing the new activity-start-task is returned. The properties of an activity-start-task are the mode of the task (i.e., whether the task is experimential or definite), the condition under which the task should be executed, the activity that should be invoked, the receiver of the activity, the parameters with which the activity should be invoked, and the deadline for executing

the task (the so-called *maximum-delay-time*). Method `registerActivityDoTask` registers an activity-do-task. Also here, the properties of the new activity-do-task are passed as parameters, and a task object representing the new activity-do-task is returned. Instead of the maximum-delay-time, the properties of an activity-do-task embrace the point in time at which the task should be temporally faithfully executed (the so-called *do-time*) and the point in time at which the execution of the task should be started (the so called *start-time*, which can be earlier than the do-time). Method `cancelTask` cancels a registered task. The object representing the task that should be canceled has to be passed as parameter. Of course, a task can be canceled only if its execution has not already been completed. Method `beginInteractiveTask` starts a new interactive task. No parameters have to be passed, the task object representing the new interactive task is returned. Methods `commitInteractiveTask` and `abortInteractiveTask` commit/abort an interactive task. The task object representing the interactive task that should be committed/aborted has to be passed as parameter. As mentioned above, tasks can be registered only by agents that are known to the task manager. Method `addAgent` adds an agent to and method `removeAgent` removes an agent from the set of agents that are allowed to register tasks. The methods provided for maintaining the asserted past will be explained in Section 9.3.4.

Informally, the task manager executes tasks that are registered for an asynchronous processing by spawning sessions. A session executes a task by issuing one or more transactions. The session is responsible for restarting transactions in the case of concurrency conflicts, for handling exceptions that occur during the execution of a task, and for reporting exceptions to agents.

The three requirements on a module performing task management that have been formulated in Section 9.3.1 are temporal faithfulness, completeness, and priority observation. The following Sections 9.3.4–9.3.6 will show how these requirements are met in GemStone/TriGS. The functionality provided by the task manager supports all three requirements—its discussion cannot be done in the scope of a single requirement. Therefore, Section 9.3.7 will give a detailed presentation of the task manager.

9.3.4 Temporal Faithfulness

Temporal faithfulness requires that transactions that are issued in the execution of tasks are serialized according to their time stamps. Our approach to fulfill this requirement is to assure that transactions are serialized according to their commits and that transactions are committed in time-stamp order.

As it has been shown in Section 9.3.2, GemStone/TriGS guarantees that a concurrent execution of transactions corresponds to some serial execution, but does not guarantee that the concurrent execution corresponds to the serial execution following commit points. A serialization of transactions following commit points is guaranteed if no transaction reads from a concurrent transaction. A transaction t *reads from* a transaction t' if t

```
runningTxReadsFromConcurrentTxs

| objectsRead conflict |

conflict := false.
(System transactionConflicts includesKey: #'Read-Write')
   ifTrue: [objectsRead := System transactionConflicts at: #'Read-Write'.
            (objectsRead size = 1)
               ifTrue:  [conflict := ((objectsRead at: 1) != self inBox)]
               ifFalse: [conflict := true].]
^conflict
```

Figure 9.24: Testing conflicts with concurrent transactions

reads or writes an object that has been written by t'. Whether a transaction reads from concurrent transactions can be checked in GemStone/TriGS by issuing command `System continueTransaction` before the actual commit and by checking the reported transaction conflicts (by means of method `System transactionConflicts`). To achieve a serialization of transactions according to commit points, a transaction must not be committed if the reported transaction conflicts contain read/write conflicts.

A session executes a task by issuing transactions. We require that every transaction that is issued in the execution of a task is committed only if it does not read from concurrent transactions. To check whether the running transaction reads from concurrent transactions, we introduce method `runningTxReadsFromConcurrentTxs` defined at class `Task` (cf. Figure 9.24). Method `runningTxReadsFromConcurrentTxs` has to be invoked immediately after method `continueTransaction`. It checks whether read-write conflicts have been reported. Since methods `continueTransaction` and `runningTxReadsFromConcurrentTxs` are executed sequentially, it may be argued that the conflicts analysed by the second method do not reflect the current state. This is basically true since another transaction may commit between the two methods and, thus, enlarge the conflict set. As it will be shown below, however, this cannot happen since transactions are not committed autonomously. Commits are controlled by a coordinator, which ensures that the commit phases of transactions do not overlap. Method `runningTxReadsFromConcurrentTxs` additionally handles the special case that the only object changed by a concurrent transaction is the task's message box for incoming messages (message boxes will be introduced below). Changes to the task's message box for incoming messages are not considered as conflicts.

If transactions are serialized according to their commit points, a temporally faithful execution of transactions can be achieved by imposing an order on these commit points. An order can be imposed on transaction commits only if transactions are not committed autonomously but under the supervision of a coordinator. The role of a coordinator is taken over by the task manager (cf. Section 9.3.7). This means that every time a transaction issued in the execution of task should be committed, a commit request is placed at the task

```
commitTx: kindOfTx

| result |

[(System signaledObjects) size = 0]
   whileTrue: [System sleep: 2].

(System continueTransaction)
   ifTrue:  [(self inBox message = 'granted')
                ifTrue:  [(self runningTxReadsFromConcurrentTxs)
                             ifTrue:  [System abortTransaction.
                                       result := 'cConflict'.]
                             ifFalse: [result := 'done'.
                                       self timeStamp value :=
                                            TaskMngr timeStampOf: self.].]
                ifFalse: [System abortTransaction.
                          result := self inBox message.].]
   ifFalse: [System abortTransaction.
             result := 'cConflict'.].
self outBox message: result.
System commitTransaction.
^result
```

Figure 9.25: Commit phase of transactions

manager. The task manager handles the commit request by either granting or rejecting it. In this setting, the commit of a transaction extends to a *commit phase*. A session that executes a task runs through such a commit phase every time a transaction should be committed.

A commit request is generally not handled immediatly by the task manager, but it is registered for future handling. I.e., handling is not done withing the requesting session, but in a separate one. Therefore, the need arises that a session issuing the commit request and a session handling the commit request communicate. The required inter-session communication is supported by the object notification mechanism of GemStone/TriGS (cf. Section 9.3.2). For the purpose of inter-session communcation, every task holds two message boxes: an in-box for incoming messages and an out-box for outgoing messages (cf. Figure 9.21). A session that executes a task makes use of the task's message boxes.

Figure 9.25 shows method `commitTx: kindOfTx` of class `Task`, which implements the commit phase of a transaction. A session that executes a task runs through this commit phase every time a transaction should be committed. As will be shown in Section 9.3.7, invocation of method `commitTx` triggers the task manager to handle the commit request in an independent session. In a first step, method `commitTx` uses the executed task's in-box as a sensor and waits for an answer to the commit request. The answer of the task manager to a commit request may be `'granted'`, `'canceled'`, or `'late'`. The semantics of

the answers `'canceled'` and `'late'` will be explained later in this section. If the answer is `'canceled'` or `'late'`, the requesting session aborts the running transaction, acknowledges receipt of the answer by writing it to the executed task's out-box, and returns the answer of the task manager as the result of method `commitTx`. If the answer is `'granted'`, the requesting session checks for concurrency conflicts. If there are concurrency conflicts preventing a commit (method `System continueTransaction`) or if the running transaction has read from concurrent transactions (method `Task runningTxReadsFromConcurrentTxs`), the requesting session aborts the running transaction, sets the result of the commit request to `'cConflict'`, and reports the concurrency conflict to the task manager by writing the result to the out-box. If there are no concurrency conflicts, the requesting session sets the result of the commit request to `'done'`, reports the success to the task manager, and commits the running transaction.

So far, it has been shown how sessions that execute tasks contribute to a temporally faithful execution of transactions by requesting commits at the task manager. Now, the task manager's contribution to a temporal faithful execution of transactions is discussed. The task manager is responsible for granting commit requests in time-stamp order. To remember: a D_t-transaction, a H_t-transaction, and a R_t-transaction receives as time-stamp the explicit execution time indicated with task t, an E_t^d-transaction is time-stamped by the maximum-delay time indicated with task t, and all remaining kinds of transactions receive as time-stamp the current reading of the system clock when the commit-request is issued.

Enforcing a commit order on transactions is unproblematic if only transactions are considered that are time-stamped according to their commit-requests. The order in which commit-requests arrive corresponds to the order in which the requests are granted. The order that has to be imposed on D-transactions, H-transactions, R-transactions, and E^d-transactions, however, does not correspond to the order of their commit-requests. In absence of restrictions, a start-task with a maximum-delay-time k or a do-task with an execution time j could be registered with the task manager when the current reading of the system clock is later than k or j. This would violate temporal faithfulness since commit-requests that have been granted since k or j could not be revoked. To avoid such conflicts, every agent has to assert a *do-time past* and a *maximum-delay past* (cf. Figure 9.21). By asserting time-stamp t as the maximum-delay-past (do-time-past), an agent assures not to place a start-task (do-task) with a maximum-delay time (do-time) $\leq t$.

To ensure temporal faithfulness, the task manager keeps track of the chronon it currently processes (cf. Figure 9.21). This chronon, which is referred to as *current processing time (CPT)*, may lag behind the current reading of the system clock reduced to chronon granularity. The current reading of the system clock reduced to chronon granularity is referred to as *wall-clock time (WCT)*. The task manager ensures that the CPT is not higher than the WCT and not higher than the minimum of the do-time-past and the minimum of the maximum-delay-past asserted by an agent. The commit-request of a transaction t is granted immediately only if the transaction's time-stamp coincides with the CPT and if the task manager is not waiting for the commit of a transaction t' where the time-stamp

of t' is the CPT, but t' has a higher priority than t. In all other cases, the commit-request of t is recorded for future handling. (Remember that transactions have different priorities and that transactions with the same time-stamp have to be ordered according to their priorities.) The point in time for which a commit-request of transaction t is recorded corresponds to the time-stamp of t. If several commit requests have been recorded for the same point in time, they are handled according to their priorities. Even commit requests with the same priority are handled sequentially. After handling the commit request of a transaction, the task manager waits for an acknowledgment from the requesting session and only then continues with the next request. This guarantees that no transaction is committed while a session is analysing concurrency conflicts in the commit phase. Details on priority observation in the commit of transactions are given in Section 9.3.6.

9.3.5 Completeness

Completeness means that the requirement of activity-start-task execution and the requirement of activity-do-task execution are satisfied (cf. Section 9.3.1).

Every activity-start-task and every activity-do-task is executed in its own session. This session, which is spawned by means of a rule, controls the execution of the task by performing method `execute` on it. Method `execute` is responsible for issuing transactions, for restarting transactions whose commit request fails because of concurrency control conflicts, and for dealing with exceptions. The exceptions that have to be handled (cf. Section 9.3.1) can be divided into three groups with respect to their potential occurrence points:

1. *Exceptions related to activity consideration:* The exception that the receiver of an activity is not enabled and the exception that the precondition of an activity is not satisfied are detected by method `execute` before the activity associated with an activity-start-task or an activity-do-task is performed.

2. *Exceptions related to activity execution:* Exceptions that may arise during or immediately after the execution of an activity are integrity constraint violations and a programmed abort. Integrity constraint violations are signaled by rules as summarized in Section 9.2.10. A programmed abort is implemented by signaling error `#activityTerminated` (cf. Section 9.3.1).

3. *Exceptions related to tasks:* This group of exceptions embraces the exception that a task is canceled and—in the case of an activity-start-task—the exception that the maximum-delay time has exceeded. An activity-start-task whose maximum-delay time has exceeded is said to be *late*. Exceptions related to tasks may occur before the execution of a task is started but also when the task is running. Of course, a task cannot be canceled and an activity-start-task cannot be late after its execution has been completed.

The way method `execute` deals with an exception depends on the group to which the exception belongs.

Exceptions related to activity consideration are detected by method `execute` directly when it checks whether the receiver of the activity is enabled and whether the precondition of the activity is satisfied. If a check fails, execution of the activity is abandoned and the encountered exception is reported.

Exceptions related to activity execution are signaled at any position during activity execution. Method `execute` deals with this group of exceptions by establishing an exception handler for the errors that may be signaled. This exception handler is designed as an activation exception handler (cf. Section 9.3.2). It catches all signals normally sent to the user who issued the erroneous operation. Upon receipt of a signal, the exception handler aborts the causing transaction and reports the encountered exception. The handler does not give back control to the method signaling the error.

Exceptions related to tasks occur asynchronously to the execution of method `execute`. A task may be canceled and the maximum-delay time of a start-task may exceed before the execution of the task is started, while the task is executing its associated activity, or while the task is waiting for an answer on a commit request. The three different situations are handled differently: The task manager starts the execution of every task regardless whether the task has been canceled or is already late. If the task has been canceled or is late, it is the responsibility of method `execute` not to perform the regular task but to report the task-related exception. If a task is canceled or becomes late while it is executing its associated activity, the cancellation or delay is signaled from the task manager to the session executing the task by means of the GemStone/TriGS Gem-to-Gem signaling mechanism (cf. Section 9.3.2). To catch inter-session signals, method `execute` installs a static exception handler for the predefined GemStone/TriGS error `#rtErrSignalGemStoneSession`. Method `execute` reacts on a signal by aborting the running transaction and reporting the task-related exception. If a task is canceled or becomes late while the task is waiting for an answer on its commit-request, the task manager answers the commit-request with `'canceled'` or `'late'` (cf. Figure 9.25). Also in this case, method `execute` aborts the running transaction and reports the encountered task-related exception.

Method `execute` is implemented differently for activity-start-tasks and activity-do-tasks. In the following paragraphs, the execution of activtity-start-tasks and activity-do-tasks is discussed. The discussion is concluded by a separate paragraph on interactive tasks and their interplay with synchronous tasks.

Activity-Start-Task Execution. Figure 9.26 shows the life cycle of an activity-start-task. The task manager creates a new task t^s when an activity-start-task with the maximum-delay time k is registered. Immediately after registration of the task, method `execute` is performed on t^s in an independent session. Method `execute` opens a new transaction (representing S_{t^s}) and checks whether the task's condition is satisfied, whether the

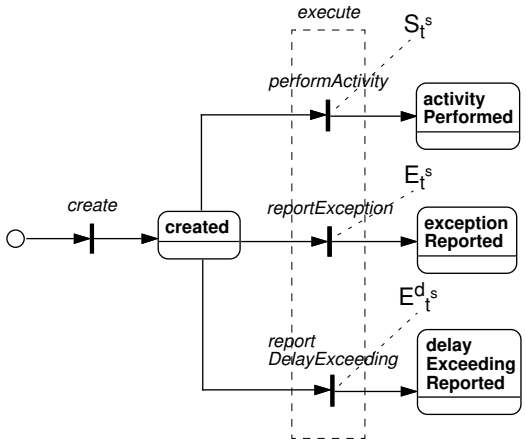


Figure 9.26: Life cycle of an activity-start-task

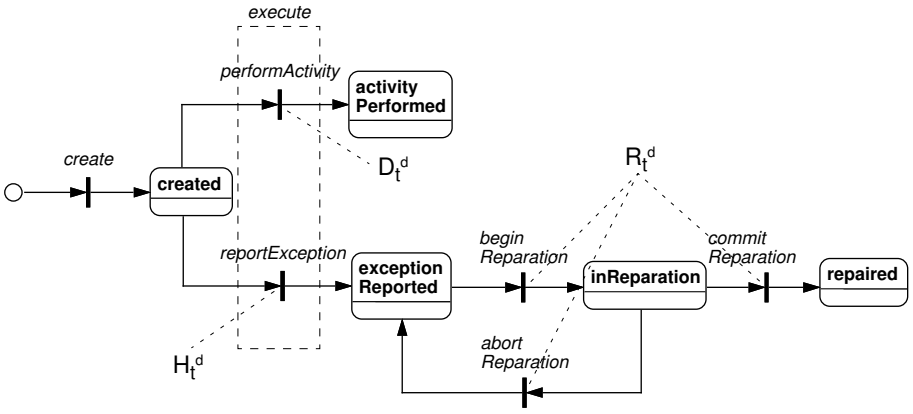


Figure 9.27: Life cycle of an activity-do-task

object on which the task's activity has to be performed is enabled, and whether the precondition of the task's activity is met. If the checks hold, the task's activity is performed in the same transaction by performing method `performActivity`. If a check fails or if another exception occurs, transaction S_{t^s} is aborted and a new transaction is started. This new transaction (representing transaction E_{t^s}) reports the encountered exception to the initiator of t^s by invoking method `reportException`. In the case that the task is late, the new transaction (then representing $E^d_{t^s}$) performs method `reportDelayExceeding`. When method `execute` requests to commit transaction S_{t^s} or transaction E_{t^s}, the task manager sets the time-stamp of t^s to the WCT. In the case of transaction $E^d_{t^s}$, the task manager sets the time-stamp of t^s to the maximum-delay time k. If method `execute` fails to commit transaction S_{t^s} or transaction E_{t^s} (also $E^d_{t^s}$) because of a concurrency conflict, it starts a new transaction automatically and reinvokes the respective operation. The task manager considers task t^s to be completed if there is either a committed transaction S_{t^s} or if there is a committed transaction E_{t^s} (also $E^d_{t^s}$).

Activity-Do-Task Execution. Figure 9.27 depicts the life cycle of an activity-do-task. The task manager creates a new task t^d when an activity-do-task with start-time i and do-time $j (i \leq j)$ is registered. The task manager assigns time-stamp j to task t^d and invokes method `execute` on t^d in an independent session at start-time i. Method `execute` is said to be *on time* if it currently works on db_j (the task's do-time), it is said to be *in advance* if it currently works on db_k where $i \leq k < j$. If the task's start time is lower than its do-time, method `execute` initially is in advance, otherwise it is on time. First, method `execute` opens a new transaction representing D_{t^d}. In D_{t^d} the following steps are performed:

1. *Check condition:* The task's condition is checked. If the condition is not satisified and if method `execute` is on time, execution of the task's activity is abandoned and task execution is complete. If method `execute` is in advance, however, execution of the task's activity cannot be abandoned immediately. The execution semantics of a do-task with do-time j requires that the task is executed semantically on db_j, i.e., on the database as it is after changing to chronon j. Even if the execution of the do-task is started earlier, this execution semantics have to be obeyed. For that reason, method `execute` places a commit-request with the task manager. The task manager handles the commit-request at point in time j. If the result of the commit-request is `'granted'`, the objects read during the condition check have not changed in the meantime, and the task's condition still is not satisfied. In this case, execution of the task's activity is abandoned. If the result of the commit-request is `'cConflict'`, however, the objects read during the condition check have been modified in the meantime. Then, method `execute` restarts transaction D_{t^d}. By restarting the transaction, method `execute` is on time.

2. *Check enabling:* If the condition is satisfied, method `execute` checks whether the object on which the task's activity has to be performed is enabled. If the object is not enabled and method `execute` is on time, depending on the task mode, either transaction D_{t^d} is aborted and a halt-transaction D_{t^d} reporting the exception is started (see below), or transaction D_{t^d} is aborted and execution of the task is abandoned. If method `execute` is in advance, however, a commit-request has to be placed for the reason stated for the condition check above. If the result of the commit-request is `'granted'`, depending on the task mode, either a halt-transaction D_{t^d} reporting the exception is started or execution of the task is abandoned. If the result of the commit-request is `'cConflict'`, however, method `execute` restarts transaction D_{t^d}. By restarting the transaction, method `execute` is on time.

3. *Check precondition:* If the object on which the task's activity has to be performed is enabled, method `execute` checks the precondition of the task's activity. If the precondition is not satisfied and method `execute` is on time, transaction D_{t^d} is aborted and a halt-transaction D_{t^d} reporting the exception is started (see below). If method `execute` is in advance, however, a commit-request has to be placed for the reason stated for

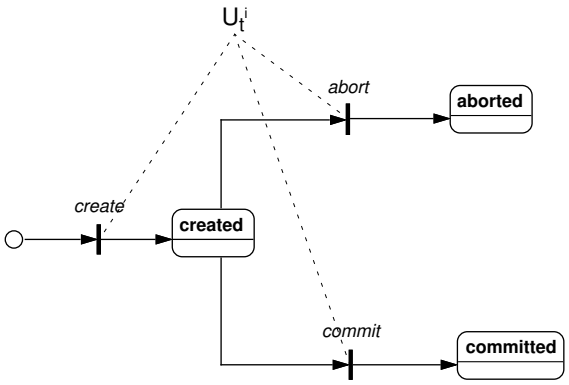


Figure 9.28: Life cycle of an interactive task

the condition check above. If the result of the commit-request is `'granted'`, a halt-transaction D_{t^d} reporting the exception is started. If the result of the commit-request is `'cConflict'`, however, method `execute` restarts transaction D_{t^d}. By restarting the transaction, method `execute` is on time.

4. *Perform activity:* If the precondition of the task's activity is satisfied, method `execute` performs the task's activity by invoking method `performActivity`. If an exception is signaled during activity execution, transaction D_{t^d} is aborted and a halt-transaction D_{t^d} reporting the exception is started (see below). After performing the activity, method `execute` places a commit-request. If the result of the commit-request is `'granted'`, execution of the task is complete. If the result of the commit-request is `'cConflict'`, however, method `execute` restarts transaction D_{t^d}. By restarting the transaction, method `execute` is on time.

If an exception occurs during D_{t^d}, method `execute` starts a halt-transaction by invoking method `reportException`. This transaction represents transaction H_{t^d} and reports the encountered exception to the initiator of t^d. After completing the exception report, method `execute` places a commit-request. If the result of the commit-request is `'granted'`, the execution of method `execute` is complete, and the halted task has to be repaired. If the result of the commit-request is `'cConflict'`, however, method `execute` restarts transaction H_{t^d}.

Reparation of a halted task is done by means of a transaction that circumvents the task manager. The begin, the commit, and the abort of a repair transaction are recorded with the repaired task. The repairing transaction represents transaction R_{t^d}. The task manager considers task t^d to be completed if there is either a committed transaction D_{t^d} or if there is a committed transaction H_{t^d} followed by a committed repair transaction R_{t^d}.

Interactive-Task Execution. Figure 9.28 depicts the life cycle of an interactive task. The task manager creates a new task t^i when an agent performs method `beginInterac-`

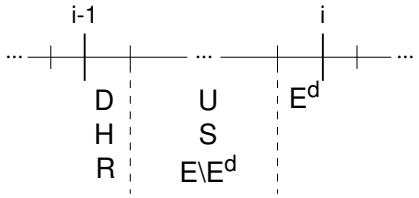


Figure 9.29: Transactions in chronon i

`tiveTask` on it. The task manager starts a new transaction (representing U_{t^i}) within which the agent performs activities. If a constraint is violated during the execution of an activity or if an activity terminates by signaling error `#activityTerminated`, this is signaled to the agent through the user interface. An interactive task is aborted by means of method `abortInteractiveTask` provided by the task manager. When an interactive task should be committed, method `commitInteractiveTask` is performed on the task manager. This method places a commit request and runs through the commit phase like any other task.

9.3.6 Priority Observation

To remember, within transactions carrying the same time-stamp, any transaction from the set $D \cup H \cup R$ has a higher priority than any transaction from the set $U \cup S \cup (E \setminus E^d)$, and any transaction from the set $U \cup S \cup (E \setminus E^d)$ has a higher priority than any E^d-transaction. A set of committed transactions satisfies the requirement of priority observation if transactions carrying the same time-stamp are serialized according to their priorities.

Figure 9.29 shows chronon i—lasting from clock-tick $i-1$ to clock-tick i—and the kinds of transactions that are committed in the different parts of chronon i. In the head of chronon i, transactions form the set $D \cup H \cup R$ are committed; in the body of chronon i, transactions from the set $U \cup S \cup (E \setminus E^d)$ are committed; and in the tail of chronon i, E^d-transactions are committed. Following the names of the parts in which transactions are committed, transactions from the set $D \cup H \cup R$ are referred to as *head-transactions*, transactions from the set $U \cup S \cup (E \setminus E^d)$ are called *body-transactions*, and E^d-transactions are termed *tail-transactions*.

The task manager obeys priorities by controlling when commit-requests of transactions are granted. After changing the CPT to i, the task manager runs through the following steps:

1. *Wait for do-time past to be $\geq i$:* In the head-phase of chronon i, all do-tasks with the explicit execution time i have to be executed. Since not all do-tasks with the explicit execution time i may be registered when the task manager changes the CPT to i, the task manager waits until the minimal do-time past asserted by an agent is $\geq i$.

2. *Start do-tasks:* The task manager starts all registered do-tasks with start-time i by

performing method `execute` on them in an independent session (cf. Section 9.3.5).

3. *Handle waiting head-transactions:* The task manager grants the commit-requests of all waiting head-transactions with time-stamp i.

4. *Wait for all do-tasks to be completed:* The task manager waits until all do-tasks with the explicit execution time i have been completed. In this phase, the task manager grants the arriving commit-requests of head-transactions with time-stamp i.

5. *Handle waiting body-transactions:* The task manager grants the commit-requests of all waiting body-transactions with time-stamp i.

6. *Handle arriving commit-requests of body-transactions:* As long as the WCT corresponds to i, the task manager grants the arriving commit-requests of body-transactions.

7. *Wait for maximum-delay past to be $\geq i$:* In the tail-phase of chronon i, the delay exceedings of all start-tasks with the maximum-delay time i have to be reported. Since not all start-tasks with the maximum-delay time i may be registered when the task manager has handled the body-phase of chronon i, the task manager waits until every agent has asserted a maximum-delay past $\geq i$.

8. *Signal delay exceedings:* The task manager signals the delay exceeding to all start-tasks with the maximum-delay time i that have not been completed yet.

9. *Wait for all start-tasks to be completed:* The task manager waits until all start-tasks with the maximum-delay time i have been completed, i.e., until all start-tasks with the maximum-delay time i have reported their delay exceedings. In this phase, the task manager grants the arriving commit-requests of tail-transactions with time-stamp i.

After completing these steps, the task manager has finished the processing of chronon i and changes the CPT to $i + 1$.

9.3.7 The Task Manager—Class `TaskMngr`

The task manager as coordinator for the execution of tasks and transactions contributes to all three requirements imposed on a module performing tasks: The task manager time-stamps transactions and grants commit-requests of transactions in time-stamp order (temporal faithfulness). It signals exceptions to sessions performing tasks and waits until task executions are completed (completeness). And, finally, the task manager commits transactions sharing the same time-stamp according to their priorities (priority observation).

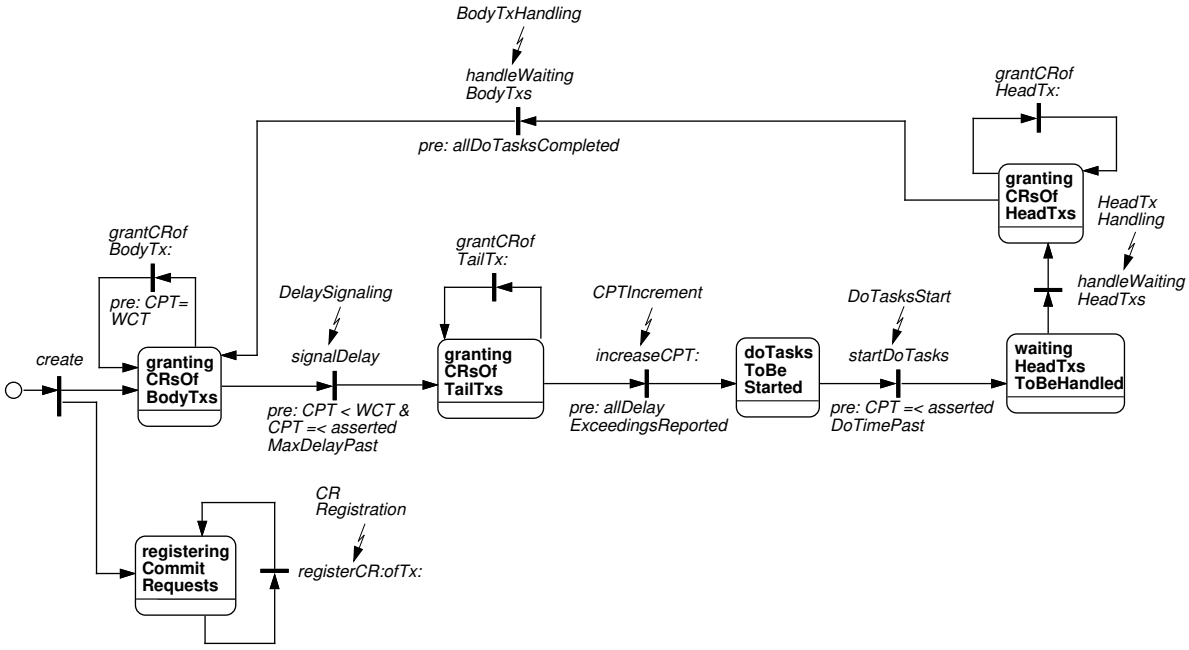


Figure 9.30: The task manager: controlling tasks and transactions

Figure 9.30 shows the part of class `TaskMngr` responsible for controlling the execution of tasks and transactions. Since there is only one task manager in a GemStone/TriGS database, the methods provided by class `ClassMngr` are class methods. It is the class object that actually represents the task manager. In the life-cycle of the task manager, parallelism is shown explicitly. For example, tasks may be registered (cf. Figure 9.23) and commit-requests may be registered (cf. Figure 9.30) concurrently. The implementation of the task manager has to ensure that two methods cannot be executed on the task manager concurrently unless they belong to two parallel branches. This can be achieved by implementing the task manager as an ensemble of objects and by locking objects exclusively. An object representing a branch then could be locked exclusively by every method belonging to the branch. Class `TaskMngr` serves as interface and provides the following methods:

- `registerCR: aTask ofTx: kindOfTx`
  Registers the commit-request of a transaction of kind `kindOfTx` that is performed on task `aTask`. Depending on `aTask`, on `kindOfTx`, on the state of the task manager, on the CPT, and on the WCT, the arriving commit-request is rejected immediately, granted immediately (by invoking method `grantCRofHeadTx:`, method `grantCRofBodyTx:`, or method `grantCRofTailTx:`), or recorded for a future handling.

- `grantCRofBodyTx: aTask`
  Grants the arriving commit-request of a body-transaction and waits for acknowledgement.

- `signalDelay`

Signals an exception to all sessions that perform tasks whose maximum-delay time corresponds to the CPT.

- `grantCRofTailTx: aTask`
  Grants the arriving commit-request of a tail-transaction and waits for acknowledgement.

- `increaseCPT: aChronon`
  Increases the CPT to `aChronon`.

- `startDoTasks`
  Starts the execution of all do-tasks whose start-time corresponds to the CPT. For a do-task that has to be started, an event is raised upon which method `execute` is performed on the task in an independent session.

- `handleWaitingHeadTxs`
  Handles the pending commit-requests of every head-transaction whose time-stamp corresponds to the CPT. For every head-transaction, the task manager grants the commit-request and waits for acknowledgement.

- `grantCRofHeadTx: aTask`
  Grants the arriving commit-request of a tail-transaction and waits for acknowledgement.

- `handleWaitingBodyTxs`
  Handles the pending commit-requests of every body-transaction whose time-stamp corresponds to the CPT. For every body-transaction, the task manager grants the commit-request and waits for acknowledgement.

Following the active object approach (cf. Figure 9.22), these methods are executed in independent sessions, which are spawned by means of rules. The rules that spawn independent sessions and execute the methods provided by the taks manager are summarized in Figures 9.31 and 9.32. Rule `TaskMngr_DelaySignaling` refers to configuration parameters `SYSTEM_START` and `CHRONON_DURATION`. The first parameter specifies the point in time at the task manager starts its work. The second parameter specifies the duration of a chronon and thus determines the granularity of time.

9.3.8 Internal Tasks

The tasks introduced so far—activity-start-tasks, activity-do-tasks, and interactive tasks—are tasks available to the external user. The rule execution model presented in the chapter will require tasks by means of which exceptions can be reported to agents in a controlable, reliable, and temporally faithful manner. Therefore, two additional kinds of tasks are introduced (cf. Figure 9.21):

```
DEFINE RULE TaskMngr_CRRegistration
  ON PRE (Task, commitTx: kindOfTx)
  EXECUTE TaskMngr registerCR: (conditionEvent trgObj) ofTx: kindOfTx.
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE TaskMngr_CRRegistration


DEFINE RULE TaskMngr_DelaySignaling
  ON PER(TIME(SYSTEM_START), CHRONON_DURATION) OR
     (POST(TaskMngr, CLASSMETHOD assertMaxDelayPast:); After_Cm)
         [event left tid = event right tid] OR
     (POST(TaskMngr, CLASSMETHOD handleWaitingBodyTxs); After_Cm)
         [event left tid = event right tid]
  IF (TaskMngr inState: 'grantingCRsOfBodyTxs') &
     (TaskMngr cpt < TaskMngr wct) &
     (TaskMngr cpt <= TaskMngr assertedMaxDelayPast) THEN
  EXECUTE TaskMngr signalDelay.
  WITH PRIORITY 1
  TRANSACTION MODES (C:PARALLEL,A:PARALLEL)
END RULE TaskMngr_DelaySignaling


DEFINE RULE TaskMngr_CPTIncrement
  ON (POST(TaskMngr, CLASSMETHOD grantCRofTailTx); After_Cm)
     [event left tid = event right tid]
  IF (TaskMngr allDelayExceedingsReported) THEN
  EXECUTE TaskMngr increaseCPT: (TaskMngr cpt + 1).
  WITH PRIORITY 1
  TRANSACTION MODES (C:PARALLEL,A:PARALLEL)
END RULE TaskMngr_CPTIncrement
```

Figure 9.31: Rules controlling the task manager (Part 1)

```
DEFINE RULE TaskMngr_DoTasksStart
  ON (POST(TaskMngr, CLASSMETHOD increaseCPT:); After_Cm)
         [event left tid = event right tid] OR
     (POST(TaskMngr, CLASSMETHOD assertDoTimePast:); After_Cm)
         [event left tid = event right tid]
  IF (TaskMngr inState: 'doTasksToBeStarted') &
     (TaskMngr cpt <= TaskMngr assertedDoTimePast) THEN
  EXECUTE TaskMngr startDoTasks.
  WITH PRIORITY 1
  TRANSACTION MODES (C:PARALLEL,A:PARALLEL)
END RULE TaskMngr_DoTasksStart


DEFINE RULE TaskMngr_HeadTxHandling
  ON (POST(TaskMngr, CLASSMETHOD startDoTasks); After_Cm)
     [event left tid = event right tid]
  EXECUTE TaskMngr handleWaitingHeadTxs.
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE TaskMngr_HeadTxHandling


DEFINE RULE TaskMngr_BodyTxHandling
  ON (POST(TaskMngr, CLASSMETHOD handleWaitingHeadTxs); After_Cm)
         [event left tid = event right tid] OR
     (POST(TaskMngr, CLASSMETHOD grantCRofHeadTx:); After_Cm)
         [event left tid = event right tid]
  IF (TaskMngr alldoTasksCompleted) THEN
  EXECUTE TaskMngr handleWaitingBodyTxs.
  WITH PRIORITY 1
  TRANSACTION MODES (C:PARALLEL,A:PARALLEL)
END RULE TaskMngr_BodyTxHandling
```

Figure 9.32: Rules controlling the task manager (Part 2)

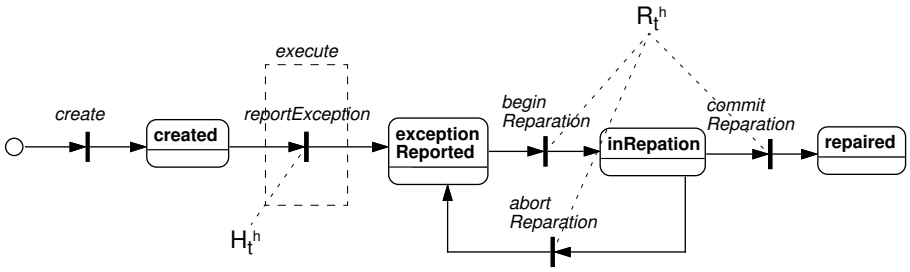


Figure 9.33: Life cycle of a halt-and-repair-task

1. *Exception-report-task:* An exception-report-task is a start-task that reports an exception to an agent. The exception report is not executed later than an indicated maximum-delay time. An exception-report-task can be seen as a special form of an activity-start-task, namely an activity-start-task performing an "empty" activity and reporting a specific exception.

2. *Halt-and-repair-task:* A halt-and-repair task is a do-task that reports an exception to an agent and additionally requires the exceptional situation to be repaired. The exception is reported at the indicated do-time. A halt-and-repair-task can be seen as a special form of an activity-do-task, namely an activty-do-task performing an "empty" activity and reporting a specific exception.

Like activity-start-tasks and activity-do-tasks, exception-report-tasks and halt-and-repair-tasks are registered with the task manager, and their execution is controlled by the task manager. In the following paragraphs, the execution of exception-report-tasks as a special form of activity-start-tasks and of halt-and-repair tasks as a special form of activity-do-tasks is shown:

Halt-And-Repair-Task Execution. Figure 9.33 depicts the life cycle of a halt-and-repair-task. The task manager creates a new task t^h when an a halt-and-repair-task with the start-time i and the explicit execution time $j (i \leq j)$ is registered. The task manager assigns time-stamp j to task t^h and invokes method `execute` on t^h in an independent session at start-time i. Method `execute` starts a new transaction (representing H_{t^h}) and invokes method `reportException`. If task t^h is canceled, transaction H_{t^h} is aborted and a new transaction is started. This new transaction (also representing a transaction H_{t^h}) reports the cancellation to the initiator of t^h. If method `execute` fails to commit transaction H_{t^h} because of a concurrency conflict, it starts a new transaction automatically and reinvokes the exception report. The reparation of a halt-and-repair-task corresponds to the reparation of an activity-do-task. The task manager considers task t^h to be completed if there is a committed transaction H_{t^h} followed by a committed repair transaction R_{t^h}.

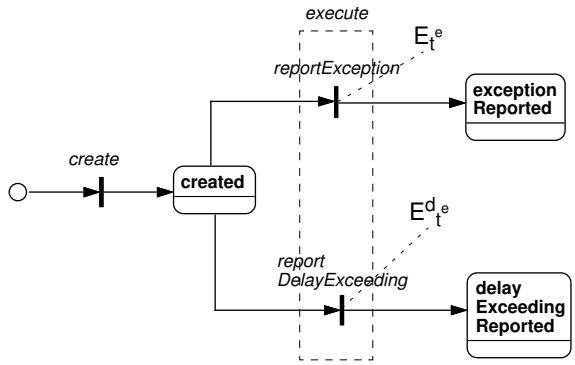


Figure 9.34: Life cycle of an exception-report-task

Exception-Report-Task Execution. Figure 9.34 depicts the life cycle of an exception-report-task. The task manager creates a new task t^e when an exception-report-task with the maximum-delay time k is registered. Immediately after registration of the task, method `execute` is performed on t^e in an independent session. Method `execute` opens a new transaction (representing E_{t^e}) and reports the exception by invoking method `reportException`. If the task is canceled, transaction E_{t^e} is aborted and a new transaction (also representing a transaction E_{t^e}) is started to report the cancellation to the initiator of t^e (by means of method `reportException`). If the task is late, transaction E_{t^e} is aborted and a new transaction (representing transaction $E^d_{t^e}$) is started to report the delay to the initiator of t^e (by means of method `reportDelayExceeding`). When transaction $E^d_{t^e}$ requests to commit, the task manager sets the time-stamp of t^e to the WCT. In the case of transaction $E^d_{t^e}$, the task manager sets the time-stamp of t^e to the maximum-delay time k. If method `execute` fails to commit transaction E_{t^e} (also $E^d_{t^e}$) because of a concurrency conflict, it starts a new transaction automatically and reinvokes the exception report. The task manager considers task t^e to be completed if there is a committed transaction E_{t^e} (also $E^d_{t^e}$).

Chapter 10

Mapping Active Object Behavior

Contents

10.1 Rationale 230
10.2 Situation Types and Situation Classes 235
10.3 Detecting Basic Database Situations 238
10.4 Detecting Logical Situations 240
10.4.1 Detecting Potential Occurrences 241
10.4.2 Ex-post Evaluation 250
10.5 Handling Rules 255
10.5.1 Rules 256
10.5.2 Rule Handlers 264

The goal of this chapter is to show how the concepts of Active Object/Behavior Diagrams for modeling the active behavior of objects—situations and rules—can be mapped to GemStone/TriGS. When the mapping of active behavior is discussed, the term *rule* is ambiguous. Rule may mean either a rule that is modeled with Active Object/Behavior Diagrams or a rule that is defined as part of a GemStone/TriGS schema. To avoid confusion, a rule in the former sense is referred to as an AOBD rule, whereas a rule in the latter sense is referred to as a GemStone/TriGS rule. This convention is dropped only if the meaning becomes clear from the context. Correspondingly, we distinguish between AOBD events and GemStone/TriGS events. The remainder of this chapter is divided into four sections. Section 10.1 outlines the principles of mapping active object behavior. Section 10.2 discusses how situation types and situation classes can be mapped to classes of a GemStone/TriGS schema. Section 10.3 describes the detection of basic database situations and basic calendar situations. Section 10.4 continues with a discussion on the detection of logical situations. Finally, Section 10.5 introduces rule handlers that provide for a reliable, controllable, and temporally faithful execution of rules.

```
DEFINE RULE <BasicAOBDRule>
  ON ((POST(<basicDBEvent>); After_Cm)
     /CONTGCM [event left tid = event right tid]);
     POST(TaskMngr, CLASSMETHOD increaseCPT:))
     /CONT
  EXECUTE <ruleConsideration>
          <taskRegistration>
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE <BasicAOBDRule>
```

Figure 10.1: Mapping basic rules: a first approach

10.1 Rationale

When we first sketched the topic of mapping active object behavior to GemStone/TriGS, we had in mind a rather straight mapping. The same terminology—the term rule is used in both, Active Object Behavior Diagrams and GemStone/TriGS—suggested similar semantics. Intuitively, we intended to map an AOBD rule directly to a single GemStone/TriGS rule. In the following, we will motivate deviations from this intuitive mapping. We will restrict our discussion initially to basic AOBD rules (i.e., rules that are triggered by a basic database event) and later will extend it to AOBD rules in general.

Figure 10.1 shows a first approach of mapping a basic AOBD rule to GemStone/TriGS. In this first approach, a basic AOBD rule is mapped to a single GemStone/TriGS rule. The GemStone/TriGS rule implements rule handling, i.e., rule consideration and rule execution. This first approach reflects the main characteristics of rule handling in Active Object Behavior Diagrams:

1. *External observation perspective:* To adhere to the external observation semantics of AOBD rules, rule handling must not be triggered by the occurrence of a basic database event alone. AOBD rules are triggered by events that are committed and they are handled at changes from one chronon to the next. Thus, the event selector of the GemStone/TriGS rule is extended to a sequence event. The first partial event `(POST(<basicDBEvent>); After_Cm) /CONTGCM [event left tid = event right tid]`, which is again a sequence event, is raised for every basic database event after the transaction within which the basic database event occurred is committed (context `continuous` with guard/context modification). A chronon change is signaled when the task manager changes from one chronon to the next (`POST(TaskMngr, CLASSMETHOD increaseCPT:)`). After the chronon change, the GemStone/TriGS rule is triggered for every committed basic database event (context `continuous`). Since the partial events triggering the GemStone/TriGS rule originate from transactions in different sessions, handling of the AOBD rule has to be done in a separate transaction

(coupling mode `PARALLEL`).

2. *Reliable, controllable and temporally faithful rule execution:* AOBD rules require a reliable, controllable, and temporally faithful execution. These requirements cannot be met by mapping the condition evaluation and the action execution of an AOBD rule to the invocation of a GemStone/TriGS method. Rather, the services provided by the task manager (cf. Section 9.3.7) are used. Rule handling is done on top of task management. This means that, depending on the result of rule consideration, tasks are registered with the task manager.

The main characteristic of this first approach is that AOBD rules are mapped to units that act autonomously. Every AOBD rule is handled autonomously by a GemStone/TriGS rule, there is no central coordinator. On the one hand, the advantage of a high degree of autonomy is that there is no coordination overhead. On the other hand, the high degree of autonomy makes it difficult to assure that all the rules that are triggered at the same chronon change have been handled before the task manager continues with processing the next chronon.

Further, the lack of a coordinator makes it difficult to adopt strategies that involve several rules. Possible strategies that involve several rules are:

- *Joint rule handling:* The overhead of starting an independent transaction for every rule handling can be considerably reduced if several rules are handled jointly within a single transaction.

- *Asynchronous handling of start-rules:* In the first approach, work of the task manager is suspended until all rule handlings triggered by a chronon change have been completed. Although such a suspension is essential for do-rules, start-rules could be handled asynchronously, too.

- *Prescheduling of do-rules:* If a do-task related with the execution of a do-rule is registered only when the do-rule is actually triggered, the execution of the do-rule may delay the whole task management. Thus, it is desireable to recognize situations that will trigger do-rules in advance and to handle them and register do-tasks before their actual occurrence time. Prescheduling seems especially promising for do-rules that are triggered by events whose occurrence is exclusively determined by the calendar and not by operations performed on the database.

To remain open for strategies that involve several rules, we introduce agents that coordinate rule handling. Such agents are called *rule handlers*. Rule handlers interact closely with the task manager to handle rules. In the presence of rule handlers, basic database events that are detected by GemStone/TriGS do not trigger the handling of AOBD rules directly but only trigger rules that register the event occurrences for further processing.

```
DEFINE RULE <BasicAOBDRule>
  ON (POST(<basicDBEvent>); After_Cm)
     /CONTGCM [event left tid = event right tid]
  EXECUTE <registerOccurrence>
  WITH PRIORITY 1
  TRANSACTION MODES (A:SERIAL)
END RULE <BasicAOBDRule>
```

Figure 10.2: Mapping basic rules: using rule handlers

Figure 10.2 shows a GemStone/TriGS rule generated for a basic AOBD rule. The generated rule registers a committed occurrence of a basic database event for further handling. Informally, rule handlers are activated by the task manager at chronon changes. Then, the rule handlers pick up the registered event occurrences and perform rule consideration and execution. Rule handlers will be presented in detail in Section 10.5.2.

So far, only basic AOBD rules have been considered. Now, the considerations are extended to cover also AOBD rules that are triggered by logical events. Active Object/Behavior Diagrams provide logical events to describe scenarios upon which actions have to be taken automatically. In active object-oriented database systems such scenarios are described by composite events. In a first approach, the idea may arise to represent logical AOBD events isomorphically by composite GemStone/TriGS events.

But, logical AOBD events and composite events differ in their semantics considerably. The differences concern mainly the database state on which logical AOBD events on the one hand and composite GemStone/TriGS events on the other are evaluated, the way a calendar system is incorporated, and the way intervals are represented. In the following, these three aspects are considered in detail:

1. *Database State:* A logical AOBD event represents a view that is evaluated on the current database state. The parameters of a logical AOBD event reflect the current database state. By contrast, a composite GemStone/TriGS event occurs when its terminating partial event occurs, i.e., when the partial event completing the detection of the composite event occurs. The parameters of a partial event that contributes to a composite event reflect the database state as it has been at occurrence of the partial event. (This is not true for parameters that are object identifiers; object identifiers are always resolved in the current database state.)

   Figure 10.3 considers a logical AOBD event (part (a)) and a composite GemStone/-TriGS event (part (b)) along the time axis at three different points in time (`10:00`, `11:00`, and `12:00`). Both, the logical AOBD event and the composite GemStone/TriGS event bind parameters $\texttt{par}_1$, $\texttt{par}_2$, and $\texttt{par}_3$. Instances of the logical AOBD event and the composite GemStone/TriGS event are represented by horizontal slices showing the occurrence time at the front side. The values assigned to parameter $\texttt{par}_1$ are

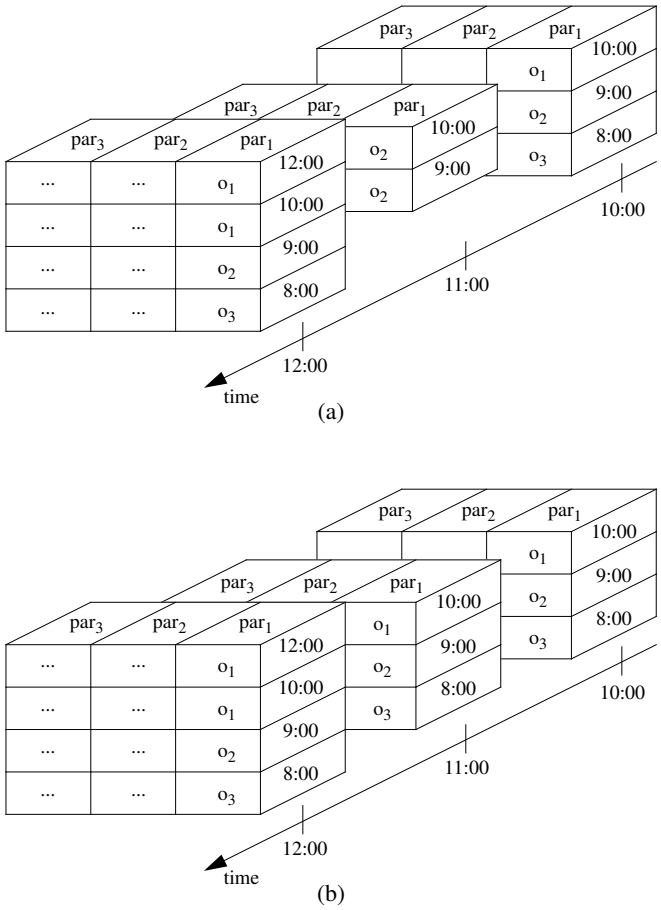


Figure 10.3: Logical situation vs. composite event

shown, parameters par_2 and par_3 are ignored.

Consider part (a) of Figure 10.3: A bar depicted above a point in time represents the set of occurrences that are obtained when the logical event is evaluated at the point in time. When the logical situation is evaluated at 10:00, three occurrences of the situation are derived. The logical situation occurs at 8:00, 9:00, and 10:00 binding value o_3, value o_2, and value o_1, respectively. When the logical situation is reevaluated at 11:00, due to changed circumstances, the situation does not occur at 8:00 and binds another value when it occurs at 10:00. Finally, at 12:00 changed circumstances may reintroduce an occurrence at 8:00, may cause the occurrence at 10:00 to bind value o_1, and may cause a new occurrence at 12:00.

Consider part (b) of Figure 10.3. A bar depicted above a point in time represents the set of occurrences contained in the history of the composite event at the point in time. The history of the composite event contains three occurrences when it is considered at 10:00. Changes to the database between 10:00 and 11:00 do not affect the history of the composite event. The history of a composite event changes only if new occurrences are detected or if occurrences are consumed. Different consumption modes are not considered here (cf. Section 2.2.5). A new occurrence of the composite event is detected at 12:00 and, thus, added to the event's history.

2. *Calendar System:* Active Object/Behavior Diagrams allow to refer to future calendar situations. The calendar is assumed to be preinstantiated. In GemStone/TriGS, temporal events occur as time proceeds and are available only for the past.

3. *Representation of Intervals:* Active Object/Behavior Diagrams treat intervals like events as first-class objects. Intervals are not part of the event language of GemStone/TriGS. Rather, intervals are spanned by pairs of events. For example, collection event $\texttt{COLL}(E_1, E_2, E_3)$ cumulates all occurrences of event E_2 within the closed time interval $[E_1, E_3]$ (cf. Section 7.1.2). No parameters can be bound by intervals in GemStone/TriGS.

This comparison shows that logical AOBD events cannot be compared with composite GemStone/TriGS events directly. The only events in Active Object/Behavior Diagrams with a close correspondence to events in GemStone/TriGS are basic database events. As shown in Figure 10.2, the detection of basic database events and the registration of occurrences can be supported fully by the active capabilities of GemStone/TriGS. For the detection of logical AOBD events, however, no such straight mapping can be presented. Logical AOBD events cannot be detected incrementally, but have to be evaluated in an ex-post manner at chronon changes. Evaluating logical AOBD events at chronon changes is the only way to ensure that the obtained occurrences reflect the current database state.

The evaluation of a logical AOBD event is done by means of a query. A query evaluating a logical AOBD event imposes conditions on basic events and on the result of evaluating other logical events. Conditions on basic events can be imposed only if the basic events

are stored in the database or if they can be computed. Basic events as they are detected by GemStone/TriGS are neither stored in the database nor can they be computed; they are part of transient structures built by rule schedulers. For that reason, GemStone/TriGS events cannot be used in evaluating logical AOBD events. Rather, occurrences of basic AOBD events have to be detected and inserted in collections. The insertion of basic AOBD events into collections, however, can be triggered by GemStone/TriGS rules. The detection of basic AOBD events will be presented in detail in Section 10.3.

Every time the task manager changes from one chronon to the next, the logical AOBD events on which rules are based have to be evaluated. Of course, evaluating *all* the events on which rules are based at every chronon change, will overload the system. A way to reduce the system load is to restrict the set of events that have to be evaluated to those events that may have *potentially* occurred. Consider for example a logical AOBD event S whose occurrence time is determined by a basic database event E. At change from chronon i to chronon $i+1$, logical event S may have potentially occurred only if basic database event E has occurred during i. This means that logical event S can be disregarded at change from i to $i+1$ unless basic database event E has occurred during i. Information about which logical events potentially occur at the change from i to $i+1$ can be collected and aggregated already during i. An approach to determine the set of events that have potentially occurred at a chronon change will be presented in Section 10.4.

10.2 Situation Types and Situation Classes

The mapping of situation types and situation classes to GemStone/TriGS follows the principles for mapping object types and their properties (cf. Chapter 8).

Contrary to object types, situation types are not mapped to subclasses of the root class `AOBDObject`. Two classes are introduced that serve as root classes for all situation types: event types are mapped to direct or indirect subclasses of class `AOBDEvent`, interval types are mapped to direct or indirect subclasses of class `AOBDInterval`. Class `AOBDEvent` implements occTime, the implicit property of every event type, and class `AOBDInterval` implements start and end, the two implicit properties of every interval type. To provide for a uniform treatment of events and intervals in situation expressions, class `AOBDEvent` provides methods `start` and `end`, too.

The predefined basic situation types in Active Object/Behavior Diagrams are:

- *Basic database event types (activity-event types):* An activity-event type is predefined for every activity in an object type's life cycle. The name of an activity-event type is constructed from the name of the object type for which the activity is defined, from the keyword _ACTIVITY_, and from the name of the activity, i.e., the name is of the form ⟨*ObjectTypeName*⟩_ACTIVITY_⟨*ActivityName*⟩. An activity-event type defines a property for every port of the activity for which it is defined. The name

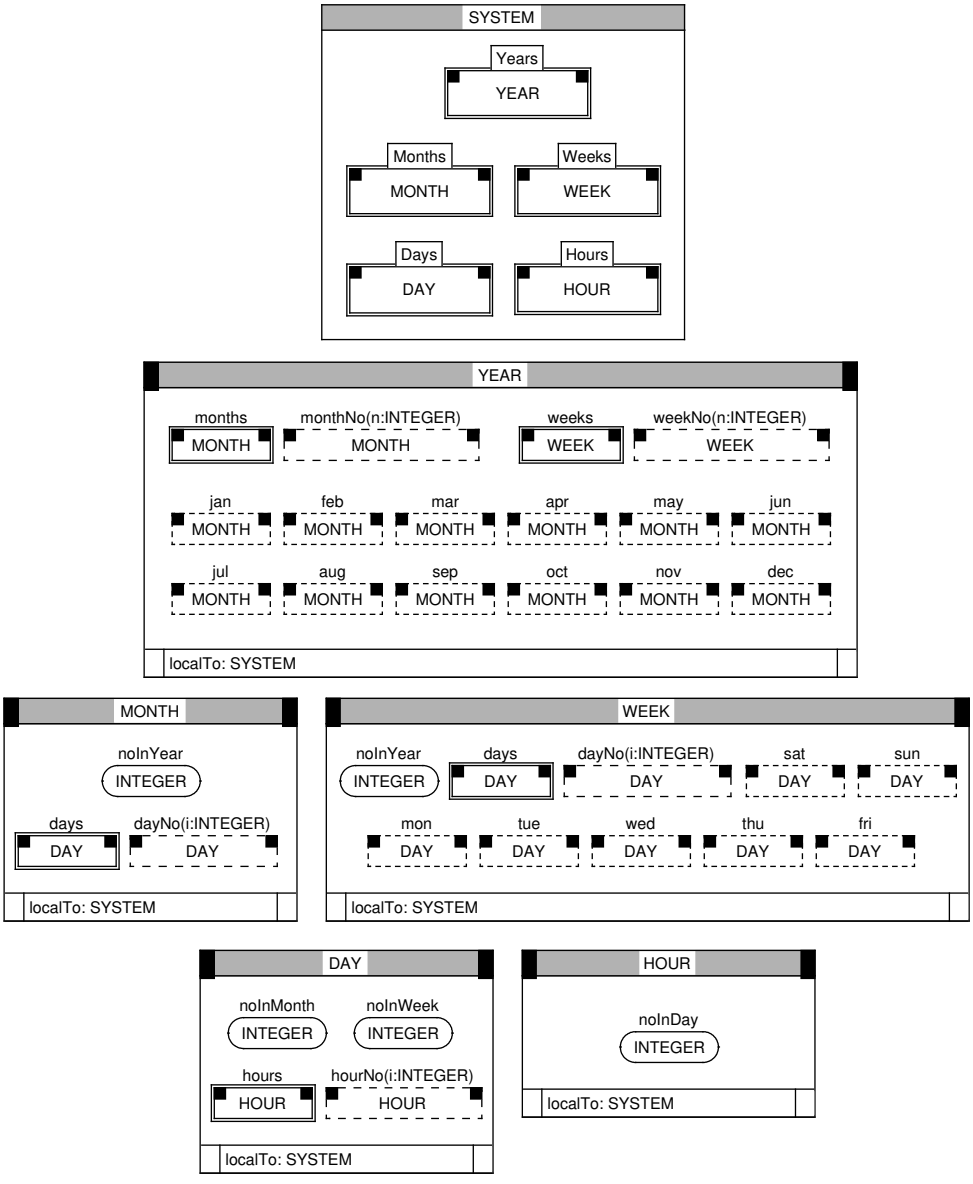


Figure 10.4: Basic calendar interval types

and the domain of a property correspond to the name and the domain of the port the property represents. The property defined for the primary port represents a reference from an activity event to the object for which the activity event occurs. An activity-event type is mapped to a subclass of class `AOBDEvent`. The properties of an activity-event type are mapped like properties of an object type.

- *Basic database interval types (state-interval types):* A state-interval type is predefined for every state in an object type's life cycle. The name of a state-interval type is constructed from the name of the object type for which the state is defined, from the keyword _STATE_, and from the name of the state, i.e., the name is of the form ⟨*ObjectTypeName*⟩_STATE_⟨*StateName*⟩. A state-interval type defines a single property named localTo. This property represents a reference from a state interval to the object for which the state interval occurs. A state-interval type is mapped to a subclass of class `AOBDInterval`. Property localTo is mapped like the properties of an object type.

- *Basic calendar event type:* The only predefined basic calendar event type is CLOCK_TICK. Event type CLOCK_TICK does not define additional properties. It is mapped to a subclass of class `AOBDEvent`.

- *Basic calendar interval types:* The predefined calendar interval types and their properties are summarized in Figure 10.4. A basic calendar interval type is mapped to a subclass of class `AOBDInterval`. The properties are mapped like properties of an object type.

User-defined situation types and their properties (cf. Section 5.1) are treated like object types and their properties when they are mapped to GemStone/TriGS.

Situations are organized in situation classes, which are defined with object types. Situation classes can be classified into situation classes with local members, situation classes with indirectly local members, and situation classes with non-local members (cf. Section 5.2). When situation classes are mapped to GemStone/TriGS, they are uniformly treated like relationship properties of object types.

The predefined basic situation classes in Active Object/Behavior Diagrams are:

- *Basic database event classes (activity-event classes):* For an object type, an activity-event class is predefined for every activity in the object type's life cycle. The name of an activity-event class is constructed from the activity name (starting with a capital letter) followed by postfix _Events, i.e., the name is of the form ⟨*ActivityName*⟩_Events. Activity-event classes are mapped like stored relationships.

- *Basic database interval classes (state-interval classes):* For an object type, a state-interval class is predefined for every state in the object type's life cycle. The name of a state-interval class is constructed from the state name (starting with a capital letter)

followed by postfix _Intervals, i.e., the name is of the form ⟨*StateName*⟩_Intervals. State-interval classes are mapped like stored relationships.

- *Basic calendar event class:* The basic calendar events are clock ticks. In Active Object/Behavior Diagrams, there is no class collecting all clock ticks. Event-retrieving function Clockticks returns a set of clock-tick events according to a given time-stamp or time-stamp pattern. When mapped to GemStone/TriGS, no collection containing all clock ticks is introduced. Rather, GemStone/TriGS is extended by a function retrieving clock ticks. Since GemStone/TriGS does not allow to define global functions, the function retrieving clock ticks is associated with class `CLOCK_TICK`.

- *Basic calendar interval classes:* The basic calendar interval classes are Years, Months, Weeks, Days, and Hours. These classes are "global", i.e., they are local to the global object type. When the global object type is mapped to GemStone/TriGS, the basic calendar interval classes are treated like stored relationships. Basic calendar intervals are preinstantiated. This means that basic calendar interval classes are filled once with all intervals lying within the considered calendar window.

User-defined logical situation classes are mapped like derived relationships. The mapping of situation expressions to OQL queries corresponds largely to the mapping of derivation expressions for properties. Peculiarities of mapping situation expressions will be shown in Section 10.4.

10.3 Detecting Basic Database Situations

Unlike basic calendar situations, which are created when a database is intialized, basic database situations have to be detected at runtime. A basic database situation that has been detected is inserted into the proper situation class. To preserve autonomy of activity implementation, detected database situations are inserted into situation classes by means of rules (cf. Section 9.2). The basic database situations that have to be detected are activity events and state intervals:

Activity Events. Consider an object type O for which an activity a with the primary port primary, the non-primary input ports $p_1, p_2, \ldots, p_n$, and the return port p_{ret} is specified. Activity-event class A_Events is populated as follows: When an execution of activity a on an object o is completed, an instance of basic event type O_ACTIVITY_A is created, the occurrence time of the instance is set, the properties of the instance are set to the values assigned to the ports, and the instance is inserted into class A_Events of o.

Figure 10.5 shows rule `O_a_Detection`, responsible for populating activity-event class A_Events. The rule is triggered after the execution of activity a. Since the rule's priority is set to 5, the rule is triggered only after all integrity constraints related with activity a have been

```
DEFINE RULE O_a_Detection
  ON POST(O, aWithP1: p1 withP2: p2 ... withPn: pn)
  EXECUTE | newEvent |
          newEvent := O_ACTIVITY_A new.
          newEvent primary: conditionEvent trgObj.
          newEvent p1:  p1.
          newEvent p2:  p2.
          ⋮
          newEvent pn:  pn.
          newEvent pret:  pret.
          newEvent occTime: TaskMngr myTask timeStamp.
          conditionEvent trgObj addToA_Events: newEvent.
  WITH PRIORITY 5
  TRANSACTION MODES (A:SERIAL)
END RULE O_a_Detection
```

Figure 10.5: Rule populating an activity-event class

checked (cf. Section 9.2.10). Most parts of the presented rule are self-explanatory. Only setting the occurrence time of the new event instance has to be explained: Activities are executed within tasks (cf. Section 9.3). Every task has an associated time-stamp. Events receive as occurrence time the time-stamp of the task within which they occur. Method `myTask` provided by class `TaskMngr` returns the task object representing the task currently executed by the inquiring session. Unless a task is a do-task, the task's time-stamp is not known during task execution. Therefore, the object returned by method `timeStamp` is not the actual time-stamp of the task but an object wrapping the time-stamp. This allows to refer to a temporarily unknown time-stamp, which is set when the execution of the task is completed.

State Intervals. Consider an object type O for which a state s is specified. State-interval class S_Intervals is populated as follows: When an object o enters state s, a new instance of state-interval type O_STATE_S is created, the start time of the instance is set, the instance's end time is set to the special value "open", and the instance is inserted into class S_Intervals of o. When object o leaves state s, the end time of the corresponding instance is set.

Figure 10.6 shows rules `O_s_Opening` and `O_s_Closing`, responsible for populating state-interval class S_Intervals. Rule `O_s_Opening`, which creates a new interval instance, sets the end time of the created instance to `OPEN`. Special value `OPEN` represents a GemStone/TriGS class of which no instances are created and which provides the functionality so that no distinction has to be made between closed and open intervals when conditions are imposed on the temporal order of situations (cf. Section 10.4). Rule `O_s_Closing` closes an open interval. Thereby, it assumes that an object never enters a state for a second time without leaving the state previously, in other words, that an object never resides in a state more

```
DEFINE RULE O_s_Opening
  ON POST(O,enterState: aState) [aState = s]
  EXECUTE | newInterval |
          newInterval := O_STATE_S new.
          newInterval localTo: conditionEvent trgObj.
          newInterval start: TaskMngr myTask timeStamp.
          newInterval end: OPEN.
          conditionEvent trgObj addToS_Intervals: newInterval.
  WITH PRIORITY 1
  TRANSACTION MODES (A:SERIAL)
END RULE O_s_Opening

DEFINE RULE O_s_Closing
  ON POST(O,leaveState: aState) [aState = s]
  EXECUTE | openInterval |
          openInterval := conditionEvent trgObj S_Intervals
              detect: [ :i | i end == OPEN].
          openInterval end: TaskMngr myTask timeStamp.
  WITH PRIORITY 1
  TRANSACTION MODES (A:SERIAL)
END RULE O_s_Closing
```

Figure 10.6: Rules populating a state-interval class

than once at a single point in time. The start time and the end time of a state interval are set to the time-stamp of the task within which the state interval is opened and/or closed. As it has been explained for activity events above, the time-stamps associated with state intervals are actually wrappers.

Both, rules populating activity-event classes and rules populating state-interval classes adhere to the external observation perspective. The external observation perspective requires that only those event and interval instances become visible that belong to successfully committed transactions. The actions of the presented rules are executed as part of the transactions triggering the rules (transaction mode `SERIAL`). This means that a generated event or interval instance becomes part of the database only if the triggering transaction commits.

10.4 Detecting Logical Situations

Two phases have to be distinguished in the detection of logical situations. During a chronon, potential occurrences of logical situations are detected incrementally. At a chronon change, the situation expressions associated with logical situations are evaluated in an ex-post manner to compute the actual situation occurrences. Each of the following two

subsections discusses one phase in detecting logical situations.

10.4.1 Detecting Potential Occurrences

Although Active Object/Behavior Diagrams treat events and intervals homogeneously, rules may be defined exclusively on events. Detecting potential occurrences of events is crucial for deciding which rules have to be evaluated at a chronon change. For that reason, the following discussion is initially restricted to logical events. Logical intervals, as elements in the specification of logical events, will be introduced later.

Logical event classes (like basic event classes) are defined for object types. A logical event class E defined for an object type O is maintained by each instance of O separately. Event E occurs on an instance o of O at point in time t if event class E maintained at o contains an event instance with occurrence time t. Intuitively, it would be desireable to detect potential occurrences per instance, i.e., for each instance separately.

To determine whether a logical event potentially occurs, the event expression associated with the logical event has to be considered. An event expression associated with event class E of object type O may refer not only to situation classes that are defined for O but also to situation classes that are defined for other object types. From the perspective of an instance o of O this means: When the event expression associated with E is evaluated, not only situation classes of o may be accessed but also an event class E' that is maintained by an object o' of type O', assumed that o' can be reached by navigating along relationships, starting the navigation at o. Depending on the usage of event class E' in the event expression associated with E, the occurrence of event E' on object o' may trigger event E on object o (see below).

To determine whether an occurrence of event E' on an instance o' of O' actually may trigger event E on object o, it would be necessary to check whether o is affected by an occurrence on object o'. Object o is affected only if it can be reached from o' by traversing back the navigation path along which event class E' is referenced in E. However, relationships are directed in Active Object/Behavior Diagrams and inverse relationships are not necessarily available, or relationships may even be derived.

Since navigation paths not always can be traversed backwards, it is impossible to detect potential occurrences per instance. The level at which potential occurrence is discussed has to be shifted from an individual instance to an object type. Considered at this level, the occurrence of event E' on any instance of O' represents a potential occurrence of event E of object type O. This potential occurrence of event E expresses only that E may occur on *some* instance of O.

The potential occurrence of a logical event depends on whether the expression associated with the logical event is selecting or generating:

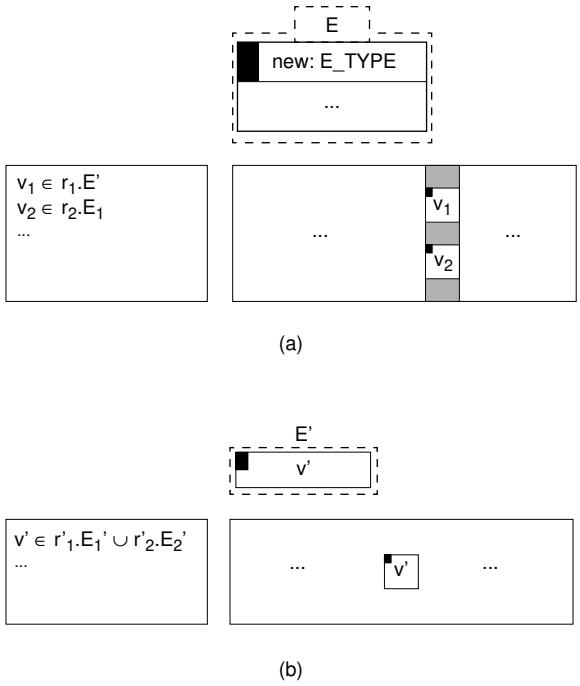


Figure 10.7: Example for a selecting and a generating event expression

- *Selecting event expression:* Let a selecting event expression for event class E select event variable v in its result box, and let variable v iterate over events contained in event classes $\mathsf{E}_1, \mathsf{E}_2, \ldots, \mathsf{E}_n$. Event E potentially occurs at a point in time t if an event E_i in $\mathsf{E}_1, \mathsf{E}_2, \ldots, \mathsf{E}_n$ occurs or potentially occurs at t. (Note that a basic event is said to occur instead of to potentially occur.)

- *Generating event expression:* Let the occurrence bar of a generating event expression for event class E cover event variables $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n$, and let variable v_i iterate over events contained in event classes $\mathsf{E}^i_1, \mathsf{E}^i_2, \ldots, \mathsf{E}^i_m$. Event E potentially occurs at a point in time t if for every variable v_i there is an event E^i_j in $\mathsf{E}^i_1, \mathsf{E}^i_2, \ldots, \mathsf{E}^i_m$ that occurs or potentially occurs at t.

Example: Figure 10.7 sketches out a generating event expression for logical event E of object type O and a selecting event expression for logical event E' of object type O'. The event expression associated with E' selects event variable v' in its result box (cf. Figure 10.7 (b)). Variable v' iterates over the union of the basic event classes E'_1 (of an object type O'_1) and E'_2 (of an object type O'_2). Relationships r'_1 and r'_2 are defined for O'. Logical event E' potentially occurs at a point in time t, if E'_1 or E'_2 occurs at t.

The occurrence bar of the event expression associated with E covers event variables v_1 and v_2 (cf. Figure 10.7 (a)). Variable v_1 iterates over logical event class E' of object type O' (see above). Variable v_2 iterates over basic event class E_1 (of an object type

```
DEFINE EVENT SELECTOR O_E_Committed AS
  (POST(O,addToE_Events: newEvent);After_Cm)
  /CONTGCM [left tid = right tid]
END EVENT SELECTOR O_E_Committed


DEFINE EVENT SELECTOR CT_P_Occ AS
  PRE(TaskMngr, CLASSMETHOD increaseCPT: newCPT)
     [CLOCK_TICK chronon: newCPT matches: P]
END EVENT SELECTOR CT_P_Occ
```

Figure 10.8: Starting points for detecting potential occurrences

O_1). Relationships r_1 and r_2 are defined for O. Logical event E potentially occurs at a point in time t, if logical event E' potentially occurs at t and if basic event E_1 occurs at t.

So far, Active Object/Behavior Diagrams have been analyzed with a view to the detection of potential occurrences of logical events. In the following, it will be shown how potential occurrences of logical events can be detected in GemStone/TriGS.

The starting point for the detection of potential occurrences of logical events is the detection of basic events:

- *Activity events:* The detection of activity events has already been described in Section 10.3. For the detection of potential occurrences only those activity events are relevant that have occurred in committed transactions. Thus, a GemStone/TriGS sequence event is defined for every activity event. Figure 10.8 (event 1) shows the sequence event `O_E_Committed` defined for activity event E of object type O. After the commit of a transaction, the sequence event is raised for every instance that has been added to the corresponding activity-event class within the transaction.

- *Clock ticks:* As already mentioned above, clock ticks are neither preinstantiated nor are they instantiated as time proceeds. Clock ticks are obtained dynamically by performing the event-retrieving function ClockTicks. In an event expression, a pattern can be passed to function ClockTicks. If function ClockTicks is used without a restricting pattern, a clock tick "occurs" at every chronon change. If a pattern is passed, however, the occurrence of clock ticks can be restricted. For an invocation of ClockTicks with a pattern in an event expression, a GemStone/TriGS event with a guard is specified. Figure 10.8 (event 2) shows the GemStone/TriGS event `CT_P_OCC` defined for an invocation with pattern P. Event `CT_P_OCC` is raised before the task manager changes to the next chronon if the next chronon matches with pattern P. Of course, a GemStone/TriGS event can be defined only for static patterns, i.e., for

```
DEFINE EVENT SELECTOR O_E_PotOcc AS
   O1_E1_PotOcc OR
   O2_E2_PotOcc OR
   ⋮
   On_En_PotOcc OR
END EVENT SELECTOR O_E_PotOcc
```

Figure 10.9: Potential occurrence of a selected logical event

```
DEFINE EVENT SELECTOR O_E_PotOcc AS
  ( O¹1_E¹1_PotOcc OR O¹2_E¹2_PotOcc OR ... O¹m_E¹m_PotOcc) AND
  ( O²1_E²1_PotOcc OR O²2_E²2_PotOcc OR ... O²m_E²m_PotOcc) AND
   ⋮
  ( Oⁿ1_Eⁿ1_PotOcc OR Oⁿ2_Eⁿ2_PotOcc OR ... Oⁿm_Eⁿm_PotOcc) AND
NOT (PRE(TaskMngr, CLASSMETHOD increaseCPT))
END EVENT SELECTOR O_E_PotOcc
```

Figure 10.10: Potential occurrence of a generated logical event

patterns that do not depend on values that change as time proceeds (e.g., property values).

The potential occurrence of a logical event is represented by a composite GemStone/TriGS event. The composite event reflects the structure of the event expression associated with the logical event it represents.

- *Selecting event expression:* Let a selecting event expression for event class E of object type O select event variable v in its result box, and let variable v iterate over events contained in event classes $\mathsf{E}_1, \mathsf{E}_2, \ldots, \mathsf{E}_n$ of object types $\mathsf{O}_1, \mathsf{O}_2, \ldots, \mathsf{O}_n$. Figure 10.9 shows the composite event representing the potential occurrence of logical event E. If E_i is a basic event, the corresponding event selector $\mathtt{O}_i\mathtt{_E}_i\mathtt{_PotOcc}$ is replaced by the event selector defined for the basic event.

- *Generating event expression:* Let the occurrence bar of a generating event expression for event class E of object type O cover event variables $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n$, and let variable v_i iterate over events contained in event classes $\mathsf{E}^i_1, \mathsf{E}^i_2, \ldots, \mathsf{E}^i_m$ of object types $\mathsf{O}^i_1, \mathsf{O}^i_2, \ldots, \mathsf{O}^i_m$. Figure 10.10 shows the composite event representing the potential occurrence of logical event E. If E^i_j is a basic event, the corresponding event selector $\mathtt{O}^i_j\mathtt{_E}^i_j\mathtt{_PotOcc}$ is replaced by the event selector defined for the basic event. The composite event represents a conjunction event. The composite event requires that there is no chronon change between the event initializing its detection and the event terminat-

```
DEFINE RULE O_E_PotOccRegistration
  ON COLL(POST(TaskMngr, CLASSMETHOD increaseCPT),
          O_E_PotOcc,
          PRE(TaskMngr, CLASSMETHOD increaseCPT))
     /CHR
  EXECUTE <register> registerPotOcc
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE O_E_PotOccRegistration

DEFINE RULE O_E_NoOccRegistration
  ON (POST(TaskMngr, CLASSMETHOD increaseCPT);
      NOT O_E_PotOcc;
      PRE(TaskMngr, CLASSMETHOD increaseCPT))
     /CHR
  EXECUTE <register> registerNoOcc
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE O_E_NoOccRegistration
```

Figure 10.11: Rules registering potential occurrences

ing its detection. All partial events have to occur within the same chronon. Therefore, event selector `O_E_PotOcc` contains a negated event expression `PRE(TaskMngr, CLASSMETHOD increaseCPT)`, which represents a chronon change (cf. Section 9.3).

GemStone/TriGS rules are used to register potential occurrences of logical events on which rules are defined. As shown in Figure 10.11, two GemStone/TriGS rules are generated for a logical event `E` of object type `O`: Rule `O_E_PotOccRegistration` registers that event `E` potentially has occurred, and rule `O_E_NoOccRegistration` registers that event `E` certainly has not occurred within a chronon. Rule `O_E_PotOccRegistration` is not triggered directly by the composite event `O_E_PotOcc` representing the potential occurrence of event `E`. All occurrences of composite event `O_E_PotOcc` within a chronon are collected. The collected occurrences together trigger a registration before the task manager changes to the next chronon. If there is no occurrence of composite event `O_E_PotOcc` within a chronon, rule `O_E_NoOccRegistration` is triggered. The presented rules follow a coordinated approach as it has been motivated in Section 10.1. They assume the presence of rule handlers. A potential occurrence of a logical event does not trigger a rule handling, but only a registration of the potential occurrence, which has to be picked up by a rule handler later. Which objects can serve as registers and how the registered occurrences can be picked up by rule handlers will be shown in Section 10.5.1. Transaction mode `PARALLEL` is specified for the actions of the presented rules. This is necessary since the partial events triggering the rules originate from different transactions, and GemStone/TriGS does not support the serial mode for such rules. Performing the registrations in independent transactions, however, requires

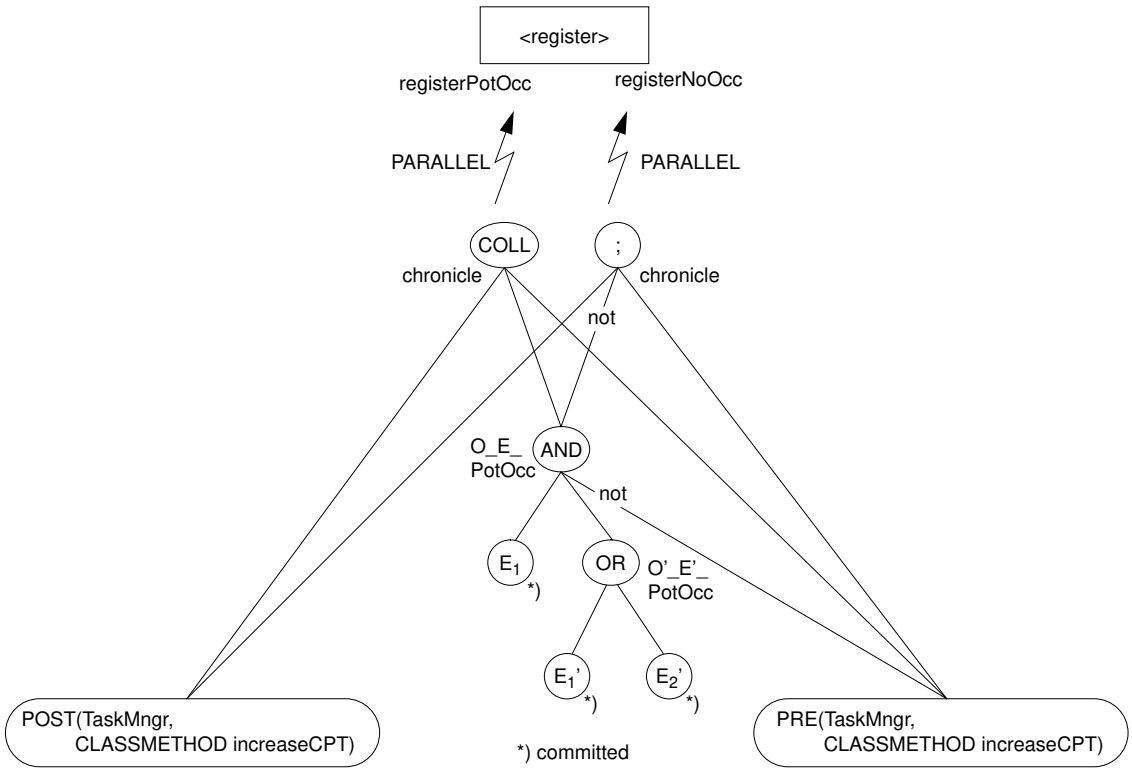


Figure 10.12: Registration of potential event occurrences triggering rules

additional coordination: Before a rule can be handled for a specific chronon, it has to be ensured that registration of potential occurrences for the considered chronon has been completed. How this can be done, will be shown in Section 10.5.1.

Example: Figure 10.12 summarizes how potential occurrences of logical events can be detected in GemStone/TriGS by means of the example introduced above. All the used graphical symbols represent elements of GemStone/TriGS: A rectangle represents a plain database object. Circles and ovals depict events. In the case of a composite event, the event operator is given within the circle or oval. Connecting lines between events denote the usage of an event in a superordinate composite event. An arrow represents a rule. The event depicted at the bottom of an arrow triggers the rule. The method name depicted at the head of an arrow is performed on the object against which the arrow is directed when the rule is triggered.

An AOBD rule is assumed to be defined on logical event `E` presented in Figure 10.7 and, thus, potential occurrences of `E` have to be registered. Logical event `E` potentially occurs if basic event $\mathtt{E}_1$ occurs and logical event `E'` potentially occurs within a chronon. Therefore, composite event `O_E_PotOcc`, representing the potential occurrence of `E`, is defined as a conjunction of $\mathtt{O}_1\mathtt{_E}_1\mathtt{_Committed}$ and `O'_E'_PotOcc` (with the default restriction that there is no chronon change between the contributing events). Logical

event E' potentially occurs if basic event E'_1 or basic event E'_2 occurs. Therefore, composite event O'_E'_PotOcc, representing the potential occurrence of E', is defined as a disjunction of O'_1_E'_1_Committed and O'_1_E'_1_Committed

The bottleneck of the strategy for detecting and registering potential occurrences of events presented in the previous paragraphs is obvious: At every chronon change, a separate transaction has to be issued for every AOBD rule. A transaction has to be started not only in the case that the event associated with an AOBD rule has potentially occurred, but also in the contrary case that the event certainly has not occurred. The rule handler responsible for handling a rule has to wait until one of the two possible transactions has been completed before it can consider the rule. Thus, a lot of independent transactions have to be started and coordinated. As has been explained above, separate transactions are necessary since complex event expressions are employed to compose the triggering events from partial events that may stem from different transactions.

A major reason to employ complex event expressions to detect potential event occurrences has been the goal to keep the number of registered potential occurrences low that later turn out to be useless. If this goal is dropped, potential occurrences can be registered also by means of rules that are executed within triggering transactions, i.e., by means of rules for which transaction mode SERIAL is indicated. This can be achieved, if *every* committed basic event that may somehow contribute to the triggering of a logical event triggers the registration of a potential occurrence of the logical event independently.

In this alternative approach—with a flat set of basic events instead of a complex event expression—potential occurrences of one and the same logical event may be registered repeatedly within a single chronon. Additionally, potential occurrences may be registered that would have been excluded by a complex event expression. Nevertheless, there may be settings in which a flat approach is more efficient than an approach using complex event expressions. Aspects that influence whether a flat approach or an approach using complex event expressions is more efficient are the event load and the complexity of a logical event:

1. *Event load:* Event load means the number of events that occur during a chronon. If the event load is high, it is more important to determine potential occurrences of logical events selectively. If the event load is low, however, the overhead for registering potential occurrences in independent transactions for coordinating the independent transactions may exceed the overhead for registering potential occurrences that later turn out to be useless and for registering the potential occurrence of one and the same logical event repeatedly within a single chronon. Note that the event load increases with the size of a chronon.

2. *Complexity of a logical event:* The complexity of a logical event means the number of different basic events that have to occur within a chronon to trigger the logical event. If the complexity of a logical event is high, potential occurrences of the logical event are better detected using complex event expressions. Using conjunctive events keeps

```
DEFINE EVENT SELECTOR O_I_Start AS
  (POST(O,addToI_Intervals: newInterval);After_Cm)
  /CONTGCM [event left tid = event right tid]
END EVENT SELECTOR O_I_Start


DEFINE EVENT SELECTOR O_I_End AS
  (POST(O_STATE_I, end: timeStamp);After_Cm)
  /CONTGCM [event left tid = event right tid]
END EVENT SELECTOR O_I_End
```

Figure 10.13: Starts and ends of state intervals

the set of registered potential occurrences small. In a flat approach, the number of registered potential occurrences increases with the complexity of a logical event.

We believe that a flat approach can be incorporated in a system easily and, therefore, do not further elaborate on it. A system should provide both alternatives for detecting and registering potential occurrences, so that during physical design a designer could decide for each rule which strategy to adopt.

So far, intervals have been disregarded. Now, the detection of potential occurrences of logical events is extended to cover also intervals. Starts and ends of intervals can be used in generating event expressions in the same way as events to determine the occurrence time of generated events. Everywhere occurrences of basic events and potential occurrences of logical events have been used so far, also starts/ends of basic intervals and potential starts/ends of logical intervals can be placed.

At first, the starts and ends of basic intervals are considered. The start/end of a basic interval is represented by a GemStone/TriGS event:

- *State intervals:* The detection of state intervals has already been described in Section 10.3. For the detection of potential occurrences of logical events only those state intervals are relevant that have been started/ended in committed transactions. Thus, two GemStone/TriGS sequence events are defined for every state interval. Figure 10.13 show the sequence events `O_I_Start` and `O_I_End` defined for state interval `I` of object type `O`. After the commit of a transaction, event `O_I_Start` is raised for every interval instance that has been added to the corresponding state-interval class within the transaction, and event `O_I_End` is raised for every interval instance whose end time has been set within the transaction.

- *Calendar intervals:* As already mentioned above, calendar intervals are preinstantiated, their starts and ends are not detected as time proceeds. In an ad-hoc approach, it would be possible to evaluate logical events that may potentially be triggered by

```
DEFINE EVENT SELECTOR O_I_PotStart AS
   O1_I1_PotStart OR
   O2_I2_PotStart OR
   ⋮
   On_In_PotStart OR
END EVENT SELECTOR O_I_PotStart


DEFINE EVENT SELECTOR O_I_PotEnd AS
   O1_I1_PotEnd OR
   O2_I2_PotEnd OR
   ⋮
   On_In_PotEnd OR
END EVENT SELECTOR O_I_PotEnd
```

Figure 10.14: Potential start/end of selected logical interval

starts/ends of calendar intervals at every chronon. More efficiently, at a chronon change it could be checked whether the chronon change coincides with the start or end of a calendar interval, and an event representing the start or end of the calendar interval could be raised only then. This second approach is adopted here. For every calendar interval, two explicit events are defined: one representing the calendar interval's start and one representing its end. At every chronon change a rule is triggered that checks whether the chronon change starts or ends a calendar interval and that raises the defined explicit events. The implementation of such a rule is trivial and, thus, not illustrated.

The potential start/end of a logical interval is represented by a composite GemStone/TriGS event. The composite event reflects the structure of the interval expression associated with the logical interval. The composite event depends on whether the associated interval expression is selecting or generating:

- *Selecting interval expression:* Let a selecting interval expression for interval class I of object type O select interval variable v in its result box, and let variable v iterate over intervals contained in interval classes $\mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_n$ of object types $\mathsf{O}_1, \mathsf{O}_2, \ldots, \mathsf{O}_n$. Interval I potentially starts (ends) at a point in time t if an interval I_i in $\mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_n$ starts (ends) or potentially starts (ends) at t. Event 1 in Figure 10.14 shows the composite event representing the potential start of logical interval I, event 2 shows the composite event representing its potential end. If I_i is a basic interval, the corresponding event selector $\mathtt{O_i_I_i_PotStart}$ (or $\mathtt{O_i_I_i_PotEnd}$) is replaced by the event selector defined for the basic interval.

- *Generating interval expression:* Let event variables $\mathsf{v}_1^S, \mathsf{v}_2^S, \ldots, \mathsf{v}_n^S$ delimit the start of

```
DEFINE EVENT SELECTOR O_I_PotStart AS
  ( O_1^S1_E_1^S1_PotOcc OR O_2^S1_E_2^S1_PotOcc OR ... O_m^S1_E_m^S1_PotOcc) AND
  ( O_1^S2_E_1^S2_PotOcc OR O_2^S2_E_2^S2_PotOcc OR ... O_m^S2_E_m^S2_PotOcc) AND
   ⋮
  ( O_1^Sn_E_1^Sn_PotOcc OR O_2^Sn_E_2^Sn_PotOcc OR ... O_m^Sn_E_m^Sn_PotOcc) AND
NOT (PRE(TaskMngr, CLASSMETHOD increaseCPT))
END EVENT SELECTOR O_I_PotStart


DEFINE EVENT SELECTOR O_I_PotEnd AS
  ( O_1^E1_E_1^E1_PotOcc OR O_2^E1_E_2^E1_PotOcc OR ... O_m^E1_E_m^E1_PotOcc) AND
  ( O_1^S2_E_1^S2_PotOcc OR O_2^E2_E_2^E2_PotOcc OR ... O_m^E2_E_m^E2_PotOcc) AND
   ⋮
  ( O_1^En_E_1^En_PotOcc OR O_2^En_E_2^En_PotOcc OR ... O_m^En_E_m^En_PotOcc) AND
NOT (PRE(TaskMngr, CLASSMETHOD increaseCPT))
END EVENT SELECTOR O_I_PotEnd
```

Figure 10.15: Potential start/end of a generated logical interval

the occurrence bar of a generating interval expression for interval class I of object type O, let event variables $\mathsf{v}_1^{\mathsf{E}}, \mathsf{v}_2^{\mathsf{E}}, \ldots, \mathsf{v}_n^{\mathsf{E}}$ delimit the end of the occurrence bar, let variable $\mathsf{v}_i^{\mathsf{S}}$ iterate over events contained in event classes $\mathsf{E}_1^{\mathsf{Si}}, \mathsf{E}_2^{\mathsf{Si}}, \ldots, \mathsf{E}_m^{\mathsf{Si}}$ of object types $\mathsf{O}_1^{\mathsf{Si}}, \mathsf{O}_2^{\mathsf{Si}}, \ldots, \mathsf{O}_m^{\mathsf{Si}}$, and let variable $\mathsf{v}_i^{\mathsf{E}}$ iterate over events contained in event classes $\mathsf{E}_1^{\mathsf{Ei}}, \mathsf{E}_2^{\mathsf{Ei}}, \ldots, \mathsf{E}_m^{\mathsf{Ei}}$ of object types $\mathsf{O}_1^{\mathsf{Ei}}, \mathsf{O}_2^{\mathsf{Ei}}, \ldots, \mathsf{O}_m^{\mathsf{Ei}}$. Interval I potentially starts at a point in time t if for every variable $\mathsf{v}_i^{\mathsf{S}}$ there is an event $\mathsf{E}_j^{\mathsf{Si}}$ in $\mathsf{E}_1^{\mathsf{Si}}, \mathsf{E}_2^{\mathsf{Si}}, \ldots, \mathsf{E}_m^{\mathsf{Si}}$ that occurs or potentially occurs at t. Interval I potentially ends at a point in time t if for every variable $\mathsf{v}_i^{\mathsf{E}}$ there is an event $\mathsf{E}_j^{\mathsf{Ei}}$ in $\mathsf{E}_1^{\mathsf{Ei}}, \mathsf{E}_2^{\mathsf{Ei}}, \ldots, \mathsf{E}_m^{\mathsf{Ei}}$ that occurs or potentially occurs at t. Of course, the occurrence bar of a logical interval may be delimited also by the start or end of intervals. In this case the (potential) event occurrences are replaced by (potential) interval starts and ends. Event 1 in Figure 10.15 shows the composite event representing the potential start of logical interval I, event 2 shows the composite event representing its potential end. If E_j^i is a basic event, the corresponding event selector $\mathsf{O}_j^i_\mathsf{E}_j^i_\mathtt{PotOcc}$ is replaced by the event selector defined for the basic event. The composite events represent conjunction events. They require that there is no chronon change between the event initializing a detection and the event terminating the detection. Thus, all partial events have to occur within the same chronon.

10.4.2 Ex-post Evaluation

When a logical situation is evaluated at a chronon change, the situation expression associated with the logical situation has to be evaluated. Mapping of situation expressions to GemStone/TriGS follows largely the mapping presented for derivation expressions of de-

rived properties: every situation expression is mapped to an OQL query. In the following, only peculiarities of mapping situation expressions are discussed. For the integration of a query mechanism into GemStone/TriGS and for mapping of derivation expressions to OQL queries in general, the reader is referred to Section 8.2 and Section 8.3.

Query Parameters. At a chronon change, the OQL query associated with a logical event is evaluated if rules are defined on the logical event and if the logical event has potentially occurred. The goal of evaluating the expression is to compute event instances whose occurrence time corresponds to the considered chronon. Event instances whose occurrence time corresponds to the considered chronon are referred to as *present* event instances, in contrast to *past* and *future* event instances. If a logical event expression is evaluated like a derivation expression for a derived property, the result of the evaluation contains not only the present event instances, but all the past, present, and—if instantiated in advance—future event instances. To obtain the present event instances, they have to be selected from the result of evaluating the event expression. Present event instances can be computed more efficiently if the chronon for which the event expression has to be evaluated is passed as parameter to expression evaluation and then can be used directly in the condition part of the expression. Therefore, every event expression has the chronon for which it has to be evaluated as parameter. When evaluated, an event expression returns only those instances that occur at the passed chronon. This keeps the set of returned instances small and allows a query optimizer to optimize the query evaluation.

When a logical event E based on another logical event E' is evaluated, not only present instances of E' may be of interest. For example, assume that E requires E' only to have occurred in the past. No specific chronon can be passed when E' is evaluated as part of E. In this case, special object `SOMETIMES` is passed as parameter, which means that no filtering according to a passed chronon is done. When a logical event E is evaluated for a chronon t, also the evaluation of specific logical events that are used in the expression associated with E can be restricted to t, i.e., t can be passed as parameter when the used logical event is evaluated. Which logical events can be restricted depends on whether the expression associated with E is selecting or generating:

- *Selecting event expression:* Let a selecting event expression for event class E select event variable v in its result box, and let variable v iterate over events contained in event classes $E_1, E_2, \ldots, E_n$. If E_i in $E_1, E_2, \ldots, E_n$ is a logical event, then the evaluation of E_i can be restricted to t.

- *Generating event expression:* Let the occurrence bar of a generating event expression for event class E cover event variables $v_1, v_2, \ldots, v_n$, and let variable v_i iterate over events contained in event classes $E^i_1, E^i_2, \ldots, E^i_m$. If E^i_j in $E^i_1, E^i_2, \ldots, E^i_m$ is a logical event, then the evaluation of E^i_j can be restricted to t.

Likewise, the evaluation of logical intervals can be performed more efficiently by passing

parameters. Every logical interval defines two parameters: one that restricts the result of evaluating a logical interval to interval instances that start at the passed chronon, and one that restricts the result of evaluating a logical interval to interval instances that end at the passed chronon. When a generating event expression associated with logical event E is evaluated for a chronon t, the evaluation of specific logical intervals that are used in the generating expression can be restricted to t: If the start/end of a variable ranging over a logical interval class I coincides with the occurrence bar in the event expression for E, then t can be passed to the evaluation of I to obtain only those interval instances that start/end at t. If the start or end (or both) of obtained interval instances should remain unrestricted, special object SOMETIMES is passed. When the evaluation of a logical interval I is restricted to interval instances that start/end at a specific point in time, also the evaluation of specific logical events that are used in the expression associated with I can be restricted. Which logical events can be restricted depends on whether the expression associated with I is selecting or generating:

- *Selecting interval expression:* Let a selecting interval expression for interval class I select interval variable v in its result box, and let variable v iterate over intervals contained in interval classes $\mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_n$. If the evaluation of I is restricted to select interval instances that start at t_s (end at t_e) and if I_i in $\mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_n$ is a logical interval, then the evaluation of I_i can be restricted to interval instances that start at t_s (end at t_e).

- *Generating interval expression:* Let event variables $\mathsf{v}_1^{\mathsf{S}}, \mathsf{v}_2^{\mathsf{S}}, \ldots, \mathsf{v}_n^{\mathsf{S}}$ delimit the start of the occurrence bar of a generating interval expression for interval class I, let event variables $\mathsf{v}_1^{\mathsf{E}}, \mathsf{v}_2^{\mathsf{E}}, \ldots, \mathsf{v}_n^{\mathsf{E}}$ delimit the end of the occurrence bar, let variable $\mathsf{v}_i^{\mathsf{S}}$ iterate over events contained in event classes $\mathsf{E}_1^{\mathsf{Si}}, \mathsf{E}_2^{\mathsf{Si}}, \ldots, \mathsf{E}_m^{\mathsf{Si}}$, and let variable $\mathsf{v}_i^{\mathsf{E}}$ iterate over events contained in event classes $\mathsf{E}_1^{\mathsf{Ei}}, \mathsf{E}_2^{\mathsf{Ei}}, \ldots, \mathsf{E}_m^{\mathsf{Ei}}$. If the evaluation of I is restricted to generate interval instances that start at t_s and if $\mathsf{E}_j^{\mathsf{Si}}$ in $\mathsf{E}_1^{\mathsf{Si}}, \mathsf{E}_2^{\mathsf{Si}}, \ldots, \mathsf{E}_m^{\mathsf{Si}}$ is a logical event, then the evaluation of $\mathsf{E}_j^{\mathsf{Si}}$ can be restricted to t_s. If the evaluation of I is restricted to generate interval instances that end at t_e and if $\mathsf{E}_j^{\mathsf{Ei}}$ in $\mathsf{E}_1^{\mathsf{Ei}}, \mathsf{E}_2^{\mathsf{Ei}}, \ldots, \mathsf{E}_m^{\mathsf{Ei}}$ is a logical event, then the evaluation of $\mathsf{E}_j^{\mathsf{Ei}}$ can be restricted to t_e. Of course, the occurrence bar of a logical interval may be delimited also by the start (end) of another logical interval. In this case, the evaluation of the used logical interval is restricted to return only interval instances that start with t_s or t_e (end with t_s or t_e).

Properties of Generated Situations. Every event type defines property occTime and every interval type defines properties start and end for its instances. When a new event instance or interval instance is generated, the corresponding properties have to be set:

- *Generating event expression:* Let the occurrence bar of a generating event expression cover event variables $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n$. Then, the occurrence time of a new event instance is set to the occurrence time of the binding to an arbitrary variable v_i in $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n$.

Of course, the occurrence bar of a generating event expression may be covered also by the start (end) of an interval. In this case, the occurrence time of a new event instance is set to the start (end) of the covering interval.

- *Generating interval expression:* Let event variables $v_1^S, v_2^S, \ldots, v_n^S$ delimit the start of the occurrence bar of a generating interval expression, and let event variables $v_1^E, v_2^E, \ldots, v_n^E$ delimit the end of the occurrence bar. Then, the start time of a new interval instance is set to the occurrence time of the binding to an arbitrary variable v_i^S in $v_1^S, v_2^S, \ldots, v_n^S$. The end time of a new interval instance is set to the occurrence time of the binding to an arbitrary variable v_i^E in $v_1^E, v_2^E, \ldots, v_n^E$. Of course, the occurrence bar of a generating interval expression may be delimited also by the start (end) of an interval. In this case, the start time or end time of a new interval instance is set to the start (end) of the delimiting interval.

Temporal Conditions. Situation diagrams are tailored to express temporal conditions on situations in a way that is easy to comprehend. When a situation diagram is mapped to an OQL query, the temporal conditions imposed by the situation diagram have to be expressed in the *where* clause of the OQL query.

Figure 10.16 shows how temporal precedence, temporal distance, temporal synchronization, and temporal adjacency can be expressed in the *where* clauses of OQL queries. The middle column of Figure 10.16 presents graphical templates for the different temporal conditions, the right column shows the OQL counterparts to the presented templates. A small square used in a graphical template represents a situation variable. The mapping of temporal conditions does not require to distinguish between event variables and interval variables. Therefore, contrary to the condition box of situation diagrams, the small filled squares depicted in the upper left or both upper corners to distinguish events and intervals are omitted. In detail, the different temporal conditions are mapped as follows:

- *Temporal precedence:* Part (a) of Figure 10.16 specifies temporal precedence between v_1 and v_2 by arranging them on the same horizontal line. In the corresponding OQL expression, the end of v_1 is required to be less or equal to the start of v_2.

- *Temporal distance:* Part (b) of Figure 10.16 specifies a condition on the temporal distance between v_1 and v_2 by arranging them on the same horizontal line and by associating a comparison predicate with the connecting time line. In the corresponding OQL expression, the temporal distance between v_1 and v_2 is represented as the difference between the start of v_2 and the end of v_1. The comparison predicate associated with the time line in the graphical template is applied to this difference.

- *Temporal synchronization:* Part (c) of Figure 10.16 shows how the starts of $v_1^s, v_2^s, \ldots, v_n^s$ and the ends of $v_1^e, v_2^e, \ldots, v_m^e$ can be synchronized by attaching them to a synchronization line. Temporal synchronization can be expressed in an OQL expression by

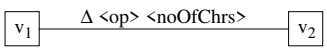
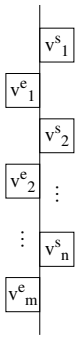
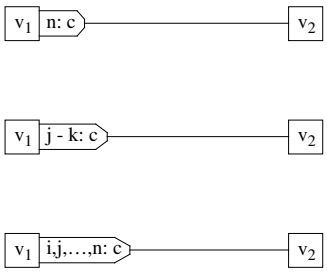
(a)		v_1.end ≤ v_2.start
(b)		(v_2.start-v_1.end) <op> <noOfChrs>
(c)		(v^s_1=v^s_2) AND … AND (v^s_1=v^s_n) AND (v^s_1=v^e_1) AND (v^s_1=v^e_2) AND … AND (v^s_1=v^e_m)
(d)		COUNT([X]) = (n-1) (j-1) ≤ COUNT([X]) AND COUNT([X]) ≤ (k-1) COUNT([X]) IN {(i-1),(j-1),…,(n-1)} [X]: select * from <DomainOfv_1> v_1' where NOT(v_1=v_1') AND v_1.end ≤ v_1'.start AND v_1'.end ≤ v_2.start AND c

Figure 10.16: Expressing temporal conditions in OQL queries

taking the start of an arbitrary v_i^s in $v_1^s, v_2^s, \ldots, v_n^s$ or the end of an arbitrary v_i^e in $v_1^e, v_2^e, \ldots, v_n^e$ and by requiring all other starts and ends to coincide with the selected start or end. In the expression presented in Figure 10.16, the start of v_1^s is selected.

- *Temporal adjacency:* Part (d) of Figure 10.16 shows three variants of conditions on the temporal adjacency of v_1 and v_2. The variants require v_1 to be the

  1. n^{th}
  2. j^{th} to k^{th}
  3. i^{th} or j^{th} or ... or n^{th}

  occurrence in the domain of v_1 that occurs before v_2 and that satisfies condition c. The OQL expressions representing the three variants are all based on counting the occurrences in the domain of v_1 that occur between v_1 and v_2 and that satisfy condition c. Counting is done in a subquery.

10.5 Handling Rules

After it has been shown in the previous sections how basic situations and logical situations can be detected in GemStone/TriGS, the point that remains to be discussed in this section is how AOBD rules can be handled at chronon changes.

Rule handling is done on top of task management as it has been presented in Section 9.3. When rules are handled, no transactions are issued directly on GemStone/TriGS, but tasks are registered with the task manager. The tasks that can be registered with the task manager and that are relevant for handling rules are start-tasks and do-tasks. The provided start-tasks are activity-start-tasks for executing an activity and exception-report-tasks for reporting an exception not later than a maximum-delay-time. The provided do-tasks are activity-do-tasks for executing an activity and halt-and-repair tasks for reporting an exception temporally faithfully at a specific point in time. Informally, the execution of a start-rule is reflected by the registration of an activity-start-task or of an exception-report-task, and the execution of a do-rule is reflected by the registration of an activity-do-task or of a halt-and-repair task.

The remainder of this section is organized as follows: Section 10.5.1 distinguishes different categories of AOBD rules with respect to their handling and shows how AOBD rule data can be incorporated into a GemStone/TriGS schema. Section 10.5.2 introduces rule handlers as software agents responsible for handling rules.

10.5.1 Rules

The decision to introduce coordinating agents for handling rules has been motivated by the different requirements of different kinds of rules (cf. Section 10.1). It is not possible to handle all rules in a uniform way *and* efficiently. Different strategies have to be adopted. Aspects that determine the way a rule can be handled efficiently are the rule's mode, the complexity of its triggering event, and the origin of its triggering event:

1. *Rule mode:* According to the rule mode, rules are classified into do-rules and start-rules (cf. Section 6.1). To conform with the execution semantics defined for do-rules, do-rules have to be considered and executed immediately at a chronon change. Therefore, do-rules have to be handled by registering do-tasks. After a change to chronon i, commit-requests of start-tasks and interactive tasks are delayed until the execution of all do-rules that have to be executed at i is completed. If start-rules were handled the same way, parallel tasks would be delayed unnecessarily. The execution semantics of start rules allows that start rules are considered asynchronously to other tasks, which are allowed to commit. Only two requirements have to be fulfilled: First, it has to be ensured that the start-rules are considered on the database state as it is at the chronon change. Second, it has to be ensured that the consideration of start rules initiated at change from chronon i to chronon $i+1$ is completed before the change from chronon $i+1$ to chronon $i+2$ is effected. This is necessary to avoid that the exceeding of the maximum delay of a start-rule is missed. The maximum delay of a start rule triggered at the change from i to $i+1$ can be $i+1$. The first requirement can be met, for example, by conserving the database state in some way. The second requirement can be met, for example, by postponing the change from $i+1$ to $i+2$ until the consideration of start-rules is completed.

2. *Complexity of triggering event:* According to the complexity of their triggering events, rules can be classified into basic rules and logical rules. A *basic rule* is defined on a basic database event, a *logical rule* is defined on some logical event. The main difference has already been worked out above: In the case of a basic event on which a basic rule is defined, an occurrence of the basic event is registered directly. A rule handler only has to pick up the registered occurrences and knows for which objects the triggered rule has to be considered. In the case of a logical event on which a logical rule is defined, only potential occurrences are registered. A rule handler that knows that a rule has potentially occurred, has to evaluate the event for every instance of the object type for which the triggered rule is defined. By means of the evaluation, the rule handler determines for which instances the triggered rule has to be considered.

3. *Origin of triggering event:* According to the origin of their triggering events, rules can be classified into calendar rules and database rules. A *calendar rule* is defined on an event whose occurrence relies exclusively on preinstantiated calendar knowledge.

A *database rule* is defined on an event whose occurrence depends on the database state (and, optionally, also the calendar). In the case of a calendar rule, triggerings of the rule are known throughout the future. Rule executions may be prescheduled. In the case of a database rule, triggerings are detected as time proceeds. Prescheduling is possible only for rules that are finally triggered by a calendar event. But even in those cases, changes to the database may invalidate a prescheduled execution.

The distinction of rules according to their mode into start-rules and do-rules is not new, it has already been introduced at the conceptual level (cf. Section 6.1). The other two dimensions—the complexity and the origin of the triggering event—become meaningful when the concrete handling of rules in a system is considered. According to the discussed dimensions, eight different rule categories could be distinguished. However, the discussed dimensions are not fully orthogonal:

First, do-rules cannot be basic rules. Do-rules may be based only on temporal event classes, whose extension is known already at the start of a chronon. Since basic database event classes are not temporal event classes, a distinction between basic rules and logical rules does not make sense for do-rules. Do-rules are classified only into database-do-rules and calendar-do-rules.

Second, a distinction between calendar rules and database rules is not promising for start-rules. The distinction between calendar rules and database rules has been made with a view to prescheduling. Prescheduling, however, is not necessary for start-rules. When executed asynchronously, start-rules do not delay task management. Start-rules are classified only into basic start-rules and logical start-rules.

Rules specified with Active Object/Behavior Diagrams are not mapped directly to GemStone/TriGS rules. Nevertheless, rule data has to be stored in GemStone/TriGS. Rule data embraces both, rule-definition data and rule-triggering data. Rule-definition data is data that is provided when a rule is defined—for example, the condition that has to be checked and the action that has to be executed when a rule is triggered. Rule-triggering data is data that is bound when a rule is triggered—for example, the object on which a rule is triggered or the event by which a rule is triggered. The distinction between rule-definition data and rule-triggering data can be preserved by mapping every rule specified with Active Object/Behavior Diagrams to a class in GemStone/TriGS. Then, rule-definition data is captured by a rule class, and rule-triggering data is captured by instances of the rule class.

Rule classes contribute to rule handling by providing the following functionality:

1. *Relating a rule handler:* Every rule class is related with the rule handler that is responsible for handling triggerings of the rule class.

2. *Registering (potential) occurrences:* During a chronon, occurrences of activity events and potential occurrences of logical events are signaled by means of GemStone/TriGS

rules (cf. Section 10.1 and Section 10.4). Rule classes serve as registers that keep track of signaled occurrences and potential occurrences.

3. *Computing rule triggerings:* At a chronon change, the triggerings of a rule class have to be computed. In the case of a basic rule, computing the rule triggerings means to create an instance of the rule class for every registered event occurrence. In the case of a logical rule, the actual event occurrences have to be computed by evaluating the logical event on which the rule is based, and an instance of the rule class is created for every computed event occurrence.

4. *Considering rule triggerings:* After computing the triggerings of a rule at a chronon change, the triggerings have to be considered. Informally, considering a rule triggering means to check whether the condition of the rule is satisfied, to check whether the action of the rule can be executed, and to register an appropriate task with the task manager.

Figure 10.17 shows GemStone/TriGS classes that serve as superclasses of classes to which AOBD rules are mapped. The used notation is borrowed from OMT. An attribute represents an instance variable constrained to a GemStone/TriGS kernel class, a relationship represents an instance variable constrained to a user-defined GemStone/TriGS class, and an operation represents a GemStone/TriGS instance method. Different to OMT, however, also class objects are shown. Like instances, class objects are described by their attributes, relationships and operations. An attribute of a class object represents a class instance variable constrained to a GemStone/TriGS kernel class, a relationship of a class object represents a class instance variable constrained to a user-defined GemStone/TriGS class, and an operation of a class object represents a GemStone/TriGS class method. A class box describing the class object of a class `A` is marked with a `C` and is placed at the right upper corner of the class box describing the instances of `A`.

Root class `Rule` defines the common features of all rule instances and of all rule classes. Every rule instance is triggered by an instance of a subclass of `AOBDEvent` and is triggered on an instance of a subclass of `AOBDObject`. Every rule class holds its instances. Whether a rule is definite or experimental is stored in class instance variable `mode`. Additionally, rule classes are responsible for creating exception events if exceptions arise during the execution of tasks that are related to rule triggerings. During the execution of a task, an arising exception is reported to the agent that has registered the task. In the case of a task that is related to a rule triggering, the informed agent is a rule handler. By means of the affected task, which is passed as parameter of the exception report, the rule handler identifies the rule triggering that has caused the exception. For every object type for which an AOBD rule is defined, a set of exception event classes is available (cf. Section 6.1). When the object type is mapped to GemStone/TriGS, these exception event classes are mapped like relationship properties. Upon an exception report, a rule handler invokes method `handleExceptn` on the concerned triggering. This method creates an exception event and inserts it into the proper exception event class of the triggering object.

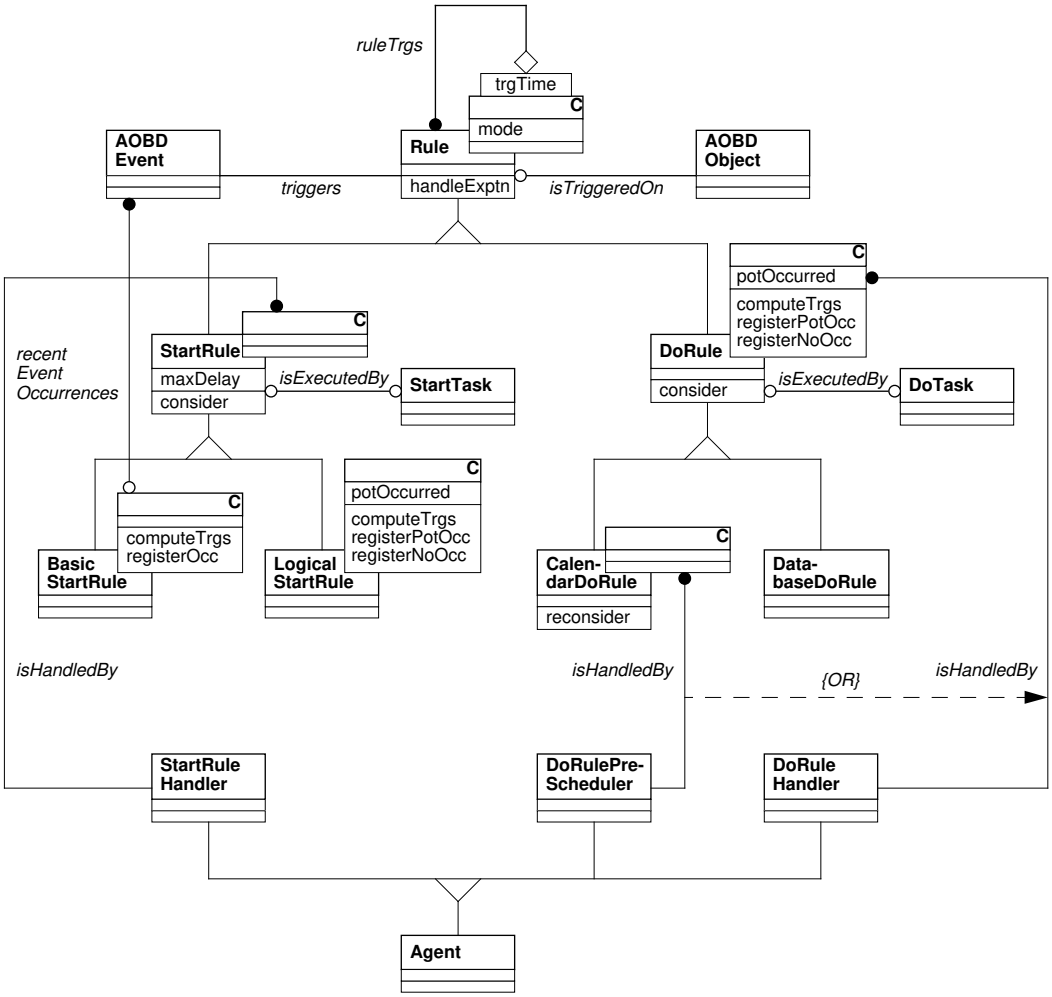


Figure 10.17: Rule classes

Class `StartRule` defines the common features of all start-rule instances and of all start-rule classes. Basic start-rules and logical start-rules have been identified as two different categories of start-rules. The two categories differ in the way rule triggerings are computed. They do not differ in the way they are handled. Thus, every start-rule—irrespective whether basic or logical—is handled by an instance of class `StartRuleHandler`. Handlers for start-rules will be presented in detail in Section 10.5.2. Class `StartRule` defines method `consider` for its instances. This method considers a rule triggering by performing the following steps:

1. *Check of condition:* The rule's condition is checked for the rule triggering. If the check fails, the consideration of the triggering is complete.

2. *Check of enabling:* If the condition is satisfied, it is checked whether the object on which the rule's activity should be executed is enabled. If the object is not enabled and the rule is experimental, the consideration of the triggering is complete.

3. *Check of precondition:* If the object on which the rule's activity should be executed is enabled, the precondition of the rule's activity is checked.

4. *Computation of maximum delay:* The point in time until which the triggering has to be handled is computed.

5. *Registration of task:* A start-rule triggering is handled by the registration of a start-task with the task manager. The maximum-delay-time of the registered task is set to the point in time computed in step (4). Three different scenarios have to be distinguished:

   (a) The considered rule is definite, and the object on which the rule's activity should be executed is not enabled: An exception-report-task reporting exception `objectNotEnabledDuringConsid` is registered.

   (b) The object on which the rule's activity should be executed is enabled, but the precondition of the rule's activity is violated: An exception-report-task reporting exception `precondViolatedDuringConsid` is registered.

   (c) The object on which the rule's activity should be executed is enabled, and the precondition of the rule's activity is satisfied: An activity-start-task performing the rule's activity (i.e., the rule's action) is registered.

The enumerated steps are performed by calling auxiliary methods. These auxiliary methods are overridden by the rule classes to which rules specified with Active Object/Behavior Diagrams are mapped. For example, consider the method that checks the condition of a rule (i.e., that issues a corresponding query): For every rule class representing a rule specified with Active Object/Behavior Diagram, the method checking the condition is overridden to issue the query that represents the rule's condition.

Class `BasicStartRule` defines the common features of all basic start-rule classes. When an event on which a basic start-rule is based occurs, the event occurrence is signaled to the start-rule class by means of a GemStone/TriGS rule (cf. Section 10.1). The invoked method is `registerOcc`. A signaled event occurrence is stored with the rule class. Class `BasicStartRule` additionally defines class method `computeTrgs`. This method computes the triggerings of the rule class. It creates an instance of the rule class for every signaled event occurrence for which no instance has been created so far, i.e., for every event occurrence that has been signaled since the last evaluation.

Class `LogicalStartRule` defines the common features of all logical start-rule classes. Potential occurrences of logical events that trigger logical start-rules are detected incrementally during a chronon. At the end of a chronon, a GemStone/TriGS rule registers with the rule class either that the logical event has potentially occurred or that the logical event certainly has not occurred (cf. Section 10.4). The invoked methods are `registerPotOcc` and `registerNoOcc`, respectively. Whether the logical event has potentially occurred or not is stored in class instance variable `potOccurred`. Also class `LogicalStartRule` defines class method `computeTrgs`. The triggerings of a logical rule-class are computed by performing the following steps:

1. *Wait for registration:* Since the registration of potential occurrences is done asynchronously (i.e., within an independent transaction), method `computeTrgs` has to wait until it is registered for the considered chronon that the logical event either potentially occurred or not occurred. If it is registered that the logical event has not occurred, the computation of the triggerings is complete.

2. *Evaluate logical events:* If it is registered that the logical event has potentially occurred, the logical event is evaluated for every instance of the object type for which the rule is defined. If no event occurrences are detected, the computation of the triggerings is complete.

3. *Create instances:* If event occurrences are detected, an instance of the rule class is created for every occurrence.

Like method `consider` defined for instances of start-rules, also class method `computeTrgs` of logical start-rule classes calls auxiliary methods. The auxiliary methods are overridden by the logical start-rule classes to which rules specified with Active Object/Behavior Diagrams are mapped.

Class `DoRule` defines the common features of all do-rule instances and of all do-rule classes. Calendar-do-rules and database-do-rules have been identified as two different categories of do-rules. Although calendar-do-rules are promising candidates for prescheduling, their handling has not necessarily to be prescheduled. Both, calendar-do-rules and database-do-rules can be handled by a do-rule handler (an instance of class `DoRuleHandler`). Do-rule handlers will be presented in detail in Section 10.5.2. Potential occurrences of events that trigger

do-rules are signaled like potential occurrences of events that trigger logical start-rules. Thus, like logical start-rule classes, do-rule classes define class methods `registerPotOcc` and `registerNoOcc`, and class instance variable `potOccurred`. Class `DoRule` also defines class method `computeTrgs`. The steps performed by this method correspond to the steps performed by the equally named class method defined by class `LogicalStartRule`. For its instances, class `DoRule` defines method `consider`. The consideration of a do-rule triggering corresponds largely to the consideration of a start-rule triggering, but some steps are different:

1. *Check of condition:* The rule's condition is checked for the rule triggering. If the check fails, the consideration of the triggering is complete.

2. *Check of enabling:* If the condition is satisfied, it is checked whether the object on which the rule's activity should be executed is enabled. If the object is not enabled and the rule is experimental, the consideration of the triggering is complete.

3. *Check of precondition:* If the object on which the rule's activity should be executed is enabled, the precondition of the rule's activity is checked.

4. *Registration of task:* A do-rule triggering is handled by the registration of a do-task with the task manager. The do-time of a registered task is set to the point in time currently handled. Three different scenarios have to be distinguished:

   (a) The considered rule is definite, and the object on which the rule's activity should be executed is not enabled: A halt-and-repair-task reporting exception `objectNotEnabledDuringConsid` is registered.

   (b) The object on which the rule's activity should be executed is enabled, but the precondition of the rule's activity is violated: A halt-and-repair-task reporting exception `precondViolatedDuringConsid` is registered.

   (c) The object on which the rule's activity should be executed is enabled, and the precondition of the rule's activity is satisfied: An activity-do-task performing the rule's activity (i.e., the rule's action) is registered.

Do-rules are classified into calendar-do-rules and database-do-rules with a view to prescheduling. Prescheduling of rules is supported by the underlying task model by do-taks for which a start-time earlier than the do-time is indicated. Class `CalendarDoRule` defines the common features of all calendar-do-rule instances and of all calendar-do-rule classes. A calendar-do-rule can be handled alternatively by a do-rule handler or a do-rule prescheduler (an instance of class `DoRulePrescheduler`). Do-rule preschedulers will be presented in detail in Section 10.5.2. If a calendar-do-rule is handled by a do-rule prescheduler, triggerings of the calendar-do-rule are computed and considered before their occurrence time. Generally, consideration of a triggering of a calendar-do-rule before its occurrence time does not differ from the consideration of a triggering of a do-rule as it has been presented above.

However, if a do-task is registered, the do-task's start-time is set to the point in time when the triggering is prescheduled, and the do-task's do-time is set to the (generally later) occurrence time of the triggering. Considering a rule triggering before its occurrence time means to check the triggering's condition, enabling, and precondition on a database state that may be different from the database state as it is at the triggering's occurrence time. This means that the registered tasks may be different at the point in time when triggerings are computed and considered and the point in time when the triggerings actually occur. Since there must not be a difference between the prescheduled and non-prescheduled execution of rules, triggerings have to be reconsidered at their occurrence time (method `reconsider`). Reconsideration is done by performing the following steps:

1. *Check of condition:* The rule's condition is checked for the rule triggering. If the check fails, any task that has been placed when the triggering has been prescheduled is canceled, and reconsideration of the triggering is complete.

2. *Check of enabling:* If the condition is satisfied, it is checked whether the object on which the rule's activity should be executed is enabled. If the object is not enabled and the triggered rule is definite, a task other than a halt-and-repair task reporting exception `objectNotEnabledDuringConsid` that has been placed when the triggering has been prescheduled is canceled. If the object is not enabled and the triggered rule is experimental, any task that has been placed when the triggering has been prescheduled is canceled, and reconsideration of the triggering is complete.

3. *Check of precondition:* If the object on which the rule's activity should be executed is enabled, the precondition of the rule's activity is checked. If the precondition fails, a task other than a halt-and-repair task reporting exception `precondViolatedDuring-Consid` that has been placed when the triggering has been prescheduled is canceled.

4. *Registration of task:* Three different scenarios have to be distinguished:

   (a) The considered rule is definite, and the object on which the rule's activity should be executed is not enabled: If not already done when the triggering has been prescheduled, a halt-and-repair-task reporting exception `objectNotEnabledDu-ringConsid` is registered.

   (b) The object on which the rule' activity should be executed is enabled, but the precondition of the rule's activity is violated: If not already done when the triggering has been prescheduled, a halt-and-repair-task reporting exception `pre-condViolatedDuringConsid` is registered.

   (c) The object on which the rule's activity should be executed is enabled, and the precondition of the rule's activity is satisfied: If not already done when the triggering has been prescheduled, an activity-do-task performing the rule's activity (i.e., the rule's action) is registered.

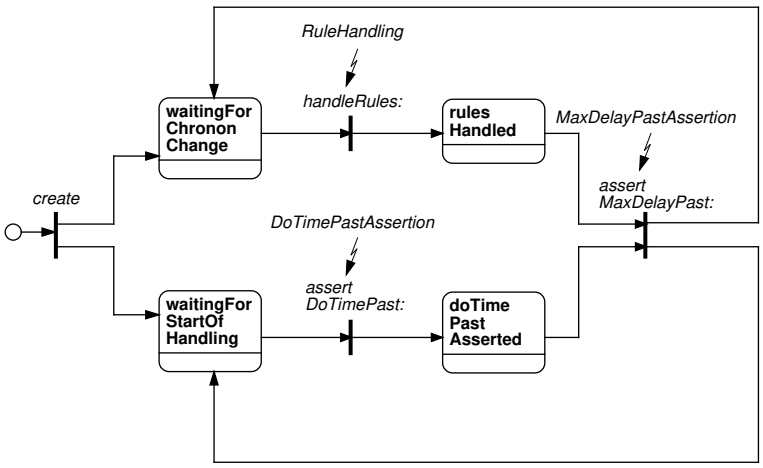


Figure 10.18: Handler for start-rules

Class `DatabaseDoRule` does not add further functionality for database-do-rules. It has been introduced so that do-rules can be mapped uniformly to subclasses of leaf-classes in the presented hierarchy.

10.5.2 Rule Handlers

Rules are handled by rule handlers. In a single GemStone/TriGS database there may be an arbitrary number of rule handlers. By default, a single start-rule handler responsible for all start-rules and a single do-rule handler responsible for all do-rules is assumed to exist. To make the handling of rules more efficient and scalable, additional rule handlers can be created and the responsibilities can be shared among them. An additional way to improve the efficiency of rule handling is to create do-rule preschedulers and to assign calendar-do-rules to them. Rule handlers (and do-rule preschedulers) are instances of subclasses of class `Agent` (cf. Figure 10.17). As agents, they are registered with the task manager, and they are allowed to place tasks.

The notation used to describe the life cycle of rule handlers in the following corresponds to the notation introduced in Section 9.3 to describe the life cycle of tasks and of the task manager.

Start-Rule Handlers. A start-rule handler is responsible for handling a set of basic or logical start-rules.

Figure 10.18 shows the life cycle of a start-rule handler. Class `StartRuleHandler` defines the following methods:

- `handleRules: aChronon`

```
DEFINE RULE StartRuleHandler_RuleHandling
  ON ((POST(StartRuleHandler, create); After_Cm)
     [event left tid = event right tid]);
     ((POST(TaskMngr, CLASSMETHOD increaseCPT: aChronon); After_Cm)
     [event left tid = event right tid])
  EXECUTE conditionEvent left left trgObj handleRules: (aChronon-1).
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE StartRuleHandler_RuleHandling


DEFINE RULE StartRuleHandler_DoTimePastAssertion
  ON PRE(StartRuleHandler, handleRules: aChronon)
  EXECUTE conditionEvent trgObj assertDoTimePast: (aChronon+1).
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE StartRuleHandler_DoTimePastAssertion


DEFINE RULE StartRuleHandler_MaxDelayPastAssertion
  ON ((POST(StartRuleHandler, handleRules: aChronon1); After_Cm)
          [event left tid = event right tid]) AND
     ((POST(StartRuleHandler, assertDoTimePast: aChronon2); After_Cm)
          [event left tid = event right tid])
     /CHRGCM [event left left trgObj == event right left trgObj]
  EXECUTE conditionEvent left left trgObj assertMaxDelayPast: aChronon2.
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE StartRuleHandler_MaxDelayPastAssertion
```

Figure 10.19: Rules controlling a start-rule handler

The start-rule handler handles the rules for which it is responsible. First, the start-rule handler computes rule triggerings by performing method `computeTrgs` on every associated rule class. Second, the start-rule handler performs method `consider` on every computed triggering.

- `assertDoTimePast: aChronon`
  The start-rule handler asserts the task manager that it will not register any do-task with a do-time less than or equal to `aChronon`.

- `assertMaxDelayPast: aChronon`
  The start-rule handler asserts the task manager that it will not register any start-task with a maximum-delay-time less than or equal to `aChronon`.

Figure 10.19 shows the GemStone/TriGS rules controlling start-rule handlers. Rule `StartRuleHandler_RuleHandling` is triggered on a start-rule handler when the task manager

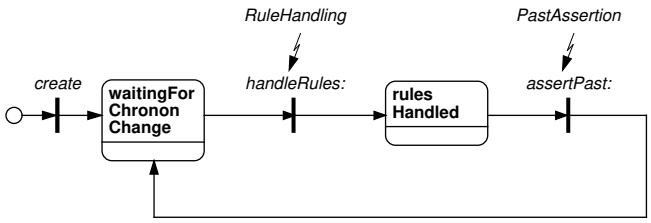


Figure 10.20: Handler for do-rules

increases the current processing time (CPT). When the task manager increases the CPT to chronon i, a start-rule handler has to handle chronon $i-1$. After increasing the CPT to i, the task manager waits until the do-time past asserted by every agent is $\geq i$. A start-rule handler cannot assert do-time past i immediately since this would allow the task manager to commit head-transactions with time-stamp i. Although the execution semantics of start-rules allows that start-rules are considered asynchronously, it has to be ensured that they are considered on db_{i-1} (i.e., the database state as it is at the change from chronon $i-1$ to chronon i). Do-time past i can be asserted only after a transaction handling the rules triggered on db_{i-1} has been started. This transaction obtains a private view of the database when it starts, and the database state relevant for the rule handling is preserved for the transaction (cf. Section 9.3.2). Rule `StartRuleHandler_DoTimePastAssertion` is triggered on a start-rule handler after the transaction performing method `handleRules` has been started. Since the identifier of the started transaction is not known, the transaction start cannot be referred to in an event expression. Therefore, rule `StartRuleHandler_DoTime-PastAssertion` is actually triggered before the execution of method `handleRules` is started in the new transaction. After the do-time past has been asserted to the task manager, the task manager is allowed to commit head-transactions and body-transactions with time-stamp i. The task manager, however, is not allowed to continue with a next chronon change. Before a change to chronon $i+1$ can be initiated, it has to be ensured that the rule handling for chronon $i-1$ is completed. This restriction has to be respected since the maximum-delay time of a start rule triggered on db_{i-1} may be i. It would not be possible to react on an exceeding of maximum delay i appropriately if the current processing time had already been increased to $i+1$. Thus, rule `StartRuleHandler_MaxDelayPastAssertion` is triggered and maximum-delay past i is asserted only after the rule handling for $i-1$ has been finished (and of course after do-time past i has been asserted, too).

Do-Rule Handlers. A do-rule handler is responsible for handling a set of database-do-rules and calendar-do-rules.

Figure 10.20 shows the life cycle of a do-rule handler. Class `DoRuleHandler` defines the following methods:

- `handleRules: aChronon`
  The do-rule handler handles the rules for which it is responsible. First, the do-rule

```
DEFINE RULE DoRuleHandler_RuleHandling
  ON ((POST(DoRuleHandler, create); After_Cm)
     [event left tid = event right tid]);
     ((POST(TaskMngr, CLASSMETHOD increaseCPT: aChronon); After_Cm)
     [event left tid = event right tid])
  EXECUTE conditionEvent left left trgObj handleRules: aChronon.
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE DoRuleHandler_RuleHandling


DEFINE RULE DoRuleHandler_PastAssertion
  ON (POST(DoRuleHandler, handleRules: aChronon); After_Cm)
          [event left tid = event right tid]
  EXECUTE conditionEvent left trgObj assertPast: aChronon.
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE DoRuleHandler_PastAssertion
```

Figure 10.21: Rules controlling a do-rule handler

handler computes rule triggerings by performing method `computeTrgs` on every associated rule class. Second, it performs method `consider` on every computed triggering.

- `assertPast: aChronon`
  The start-rule handler asserts the task manager that it will register neither a do-task with a do-time less than or equal to `aChronon` nor a start-task with a maximum-delay time less than or equal to `aChronon`.

Figure 10.21 shows the GemStone/TriGS rules controlling do-rule handlers. Rule `DoRuleHandler_RuleHandling` is triggered on a do-rule handler when the task manager increases the current processing time (CPT). When the task manager increases the CPT to chronon i, a do-rule handler has to handle chronon i. After increasing the CPT to i, the task manager waits until the minimal do-time past asserted by an agent is $\geq i$. The execution semantics of do-rules requires that do-rules are considered and the resulting tasks are registered immediately at a chronon change. Thus, rule `DoRuleHandler_PastAssertion` is triggered to assert do-time past i and maximum-delay time past i only after handling of chronon i has been completed.

Do-Rule Preschedulers. A do-rule prescheduler is responsible for handling a set of calendar-do-rules. Generally, a do-rule prescheduler handles rules like a do-rule handler. If a do-rule prescheduler is not busy, however, it looks ahead and preschedules future rule triggerings. Like handling, prescheduling always concerns the associated rules as a whole,

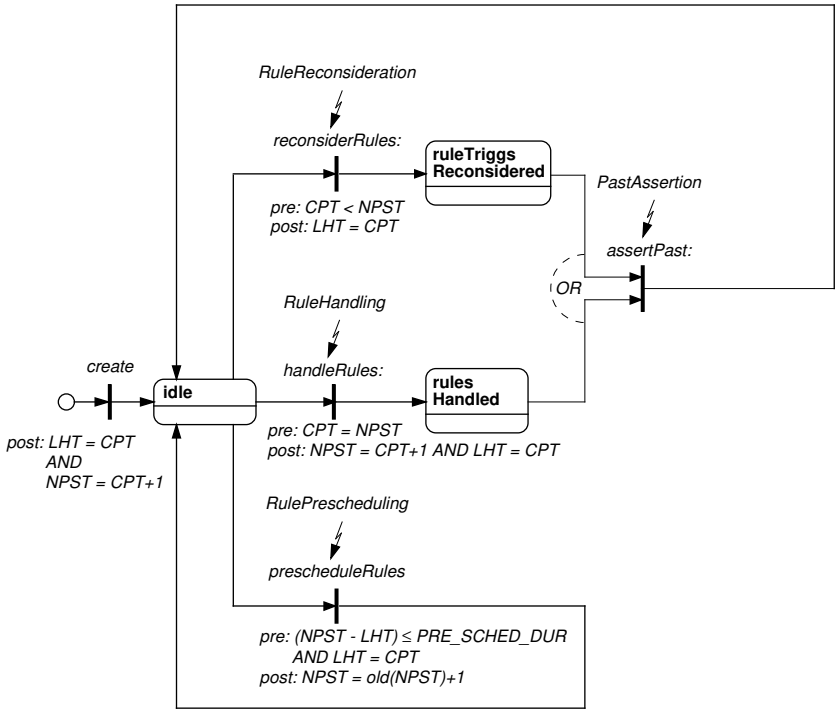


Figure 10.22: Prescheduler for do-rules

i.e., either all the associated rules are prescheduled for a chronon or non of them. How a do-rule prescheduler handles its rules after a chronon change depends on whether the do-rule prescheduler has already prescheduled the chronon to which it changes or not. If it has prescheduled the chronon, the prescheduled triggerings are only reconsidered. If it has not prescheduled the chronon, the rules are handled conventionally like by a do-rule handler. For every do-rule prescheduler, parameter `PRE_SCHED_DUR` specifies the maximum duration (i.e., the maximum number of chronons) the do-rule prescheduler looks ahead. The chronon that has to be prescheduled next by a do-rule prescheduler is stored in `NPST` (next prescheduling time). The chronon that has been handled last by a do-rule prescheduler is stored in `LHT` (last handling time). Additionally, every do-rule prescheduler maintains a log of prescheduled triggerings: for every chronon that has been prescheduled but not handled, the prescheduled triggerings are stored.

Figure 10.22 shows the life cycle of a do-rule prescheduler. Class `DoRulePrescheduler` defines the following methods:

- `prescheduleRules: aChronon`
  The do-rule prescheduler preschedules its associated rules for future chronon `aChronon`: The prescheduler computes future rule triggerings by performing method `computeTrgs` (with `aChronon` as parameter) on every associated rule class. The computed triggerings are stored in the prescheduler's log, and method `consider` is performed on every triggering. The `NPST` is incremented by one.

```
DEFINE RULE DoRulePreSched_RuleReconsideration
  ON ((POST(DoRulePreSched, create); After_Cm)
     [event left tid = event right tid]);
     ((POST(TaskMngr, CLASSMETHOD increaseCPT: aChronon); After_Cm)
     [event left tid = event right tid])
  IF aChronon < event left left trgObj npst THEN
  EXECUTE conditionEvent left left trgObj reconsiderRules: aChronon.
  WITH PRIORITY 1
  TRANSACTION MODES (C:PARALLEL,A:PARALLEL)
END RULE DoRulePreSched_RuleReconsideration


DEFINE RULE DoRulePreSched_RuleHandling
  ON ((POST(DoRulePreSched, create); After_Cm)
     [event left tid = event right tid]);
     ((POST(TaskMngr, CLASSMETHOD increaseCPT: aChronon); After_Cm)
     [event left tid = event right tid])
  IF aChronon = event left left trgObj npst THEN
  EXECUTE conditionEvent left left trgObj handleRules: aChronon.
  WITH PRIORITY 1
  TRANSACTION MODES (C:PARALLEL,A:PARALLEL)
END RULE DoRulePreSched_RuleHandling
```

Figure 10.23: Rules controlling a do-rule prescheduler (part 1)

- `handleRules: aChronon`
  The do-rule prescheduler handles its associated rules for chronon `aChronon` (representing the task manager's current processing time): The prescheduler computes rule triggerings by performing method `computeTrgs` on every associated rule class. Then, it performs method `consider` on every computed triggering. The `LHT` is set to `aChronon` and the `NPST` is set to `aChronon + 1` (the chronon following the task manager's current processing time).

- `reconsiderRules: aChronon`
  The do-rule prescheduler reconsiders its associated and prescheduled rules for chronon `aChronon` (representing the task manager's current processing time): The prescheduler performs method `reconsider` on every rule triggering that has been stored in the log when chronon `aChronon` has been prescheduled. Then, the `LHT` is set to `aChronon`, and all the reconsidered triggerings are removed from the log.

- `assertPast: aChronon`
  The do-rule prescheduler asserts the task manager that it will register neither a do-task with a do-time less than or equal to `aChronon` nor a start-task with a maximum-delay time less than or equal to `aChronon`.

```
DEFINE RULE DoRulePreSched_PastAssertion
  ON (POST(DoRuleHandler, reconsiderRules: aChronon); After_Cm)
          [event left tid = event right tid] OR
     (POST(DoRuleHandler, handleRules: aChronon); After_Cm)
          [event left tid = event right tid]
  EXECUTE conditionEvent left trgObj assertPast: aChronon.
  WITH PRIORITY 1
  TRANSACTION MODES (A:PARALLEL)
END RULE DoRulePreSched_PastAssertion


DEFINE RULE DoRulePreSched_RulePrescheduling
  ON (POST(DoRulePreSched, create); After_Cm)
          [event left tid = event right tid] OR
     (POST(DoRulePreSched, assertPast:); After_Cm)
          [event left tid = event right tid] OR
     (POST(DoRulePreSched, prescheduleRules); After_Cm)
          [event left tid = event right tid]
  IF (TaskMngr cpt = event left trgObj lht) AND
     ((event left trgObj npst - event left trgObj lht) <=
     event left trgObj pre_sched_dur) THEN
  EXECUTE conditionEvent left trgObj prescheduleRules.
  WITH PRIORITY 1
  TRANSACTION MODES (C:PARALLEL,A:PARALLEL)
END RULE DoRulePreSched_RulePrescheduling
```

Figure 10.24: Rules controlling a do-rule prescheduler (part 2)

Figures 10.23 and 10.24 show the GemStone/TriGS rules controlling do-rule preschedulers. When the task manager increases its current processing time (CPT) to chronon i, rule `DoRulePreSched_RuleReconsideration` and rule `DoRulePreSched_RuleHandling` are triggered on a do-rule prescheduler. The condition of which rule is actually satisfied depends on whether chronon i has already been prescheduled by the do-rule prescheduler. If chronon i has already been prescheduled, the triggerings recorded in the log for i are reconsidered. If chronon i has not yet been prescheduled, the rules associated with the do-rule prescheduler have to be completely handled for i. After increasing the CPT to i, the task manager waits until the do-time past asserted by every agent is $\geq i$. Rule `DoRulePreSched_PastAssertion` is triggered (i.e., do-time past i and maximum-delay time past i are asserted) only after the reconsideration or the complete handling is finished. This guarantees that calendar-do-rules are considered and that the resulting tasks are registered immediately at a chronon change. Every time a do-rule prescheduler enters state `idle` (i.e., after the prescheduler is created, after the prescheduler has asserted the handling of a chronon to the task manager, and after the prescheduler has prescheduled a chronon) rule `DoRulePreSched_RulePrescheduling` is triggered on the prescheduler. The rule's condition requires that the maximum duration the prescheduler looks ahead is not exceeded and that there is no pending chronon change the prescheduler has to handle. There is no pending chronon change if the task manager's current processing time corresponds to the last handling time (LHT) of the prescheduler. If the condition is satisfied for the rule triggering, the chronon stored in the prescheduler's next prescheduling time (NPST) is prescheduled.

Chapter 11

Conclusion

Contents

11.1 Summary 273
11.1.1 Conceptual Design 274
11.1.2 Logical Design 275
11.2 Tool Support 278
11.3 Outlook 279

This chapter gives a brief overview of the work presented in this thesis. Section 11.1 summarizes both the major contributions of Active Object/Behavior Diagrams to the conceptual design of active object-oriented databases and the cornerstones of mapping a conceptual schema specified with Active Object/Behavior Diagrams to a logical schema supported by an active object-oriented database system. Section 11.2 outlines an editor that has been developed for Active Object/Behavior Diagrams. Section 11.3 enumerates some points left open in this thesis and outlines ideas for further work.

11.1 Summary

Modeling business rules at the conceptual level requires a different set of concepts than currently provided by active object-oriented database systems (cf. Chapter 2) or by recent approaches to active object-oriented database design (cf. Chapter 3). A first goal of this thesis has been to identify required concepts and to develop a design notation meeting the identified requirements. With Active Object/Behavior Diagrams we introduce such a design notation. Besides the development of Active Object/Behavior Diagrams, a second goal of this thesis has been to study how a conceptual schema specified with Active Object/Behavior Diagrams could be mapped to a logical schema supported by an active object-oriented database system.

11.1.1 Conceptual Design

In a first step, we have identified a set of concepts that are required for modeling business rules at the conceptual level. The requirements are (cf. Section 3.3):

1. *External user's perspective:* Database changes should be observed only at the level of transactions that are not further decomposable. In the realm of ACID-transactions, which are assumed, all actions performed and all events raised within a transaction occur instantaneously for the external observer.

2. *Future calendar events and intervals:* Business rules should be able to refer to calender time, including future calendar events as well as future calendar intervals.

3. *Events and intervals as first-class objects:* Events and intervals should be represented in the data model of a system as first-class objects. This means that events and intervals should be described by types and collected in sets like conventional database objects.

4. *Abstracting from granularity of time:* The time model lying behind the conceptual model should allow to abstract from the granularity of the time axis defined by an internal system clock.

5. *Declarative event language:* Similarly as QBE and SQL are better accepted by end users than Relational Algebra, end users are assumed to prefer a declarative event language over an operational one.

6. *Reliable and controllable rule execution:* Modelers and system users expect business rules to be executed reliably, i.e., transactions performing rule executions should be restarted automatically if they are aborted due to serialization failures, and controllably, i.e., specific types of exceptional situations occurring during rule execution should be reported by events, which can again be handled by business rules.

7. *Asynchronous and immediate rule execution:* Two rule categories are needed to formulate business policies: (1) rules that are considered and executed asynchronously to user transactions and (2) rules that are considered and executed immediately upon temporal events. This feature does not necessarily require the capabilities of real-time database systems, but temporally faithful rule execution (as presented in Section 6.3) is sufficient for many business applications.

Based on these requirements, we have introduced Active Object/Behavior Diagrams, a high-level graphical language for the conceptual design of active object-oriented databases. Active Object/Behavior Diagrams extend Object/Behavior Diagrams [KS91a] with Situation/Activation Diagrams [LOS96, LOS97].

Object/Behavior Diagrams. Object/Behavior Diagrams are a graphical object-oriented design notation for modeling the structure and the passive behavior of objects. Within this thesis, we have extended the original definition of Object/Behavior Diagrams by introducing object-class diagrams and query diagrams (cf. Chapter 4). An object-class diagram depicts local referential integrity constraints that apply to the instances of the object type with which the object-class diagram is associated. A query diagram allows to specify a graphical derivation expression for a derived property in an OQL-like structure ([CB97]).

Situation/Activation Diagrams. Situation/Activation Diagrams are integrated seamlessly into Object/Behavior Diagrams for modeling the active behavior of objects. They consist of situation diagrams (cf. Chapter 5) and of activation diagrams (cf. Chapter 6).

A situation diagram specifies declaratively when a situation (i.e., an event or interval) occurs. Like the structure of a query diagram, also the structure of a situation diagram is OQL-like. Situation diagrams visualize the temporal order of situations such that temporal conditions, which are typically used in the definition of situations, are easy to specify and easy to recognize. Situation diagrams incorporate a calendar system that allows to refer also to future calendar situations. Situations are viewed from the external user's perspective. The underlying time model abstracts from the granularity of the time axis defined by the internal system clock. Similar to objects, which are instances of an object type and are collected in object classes, situation occurrences are instances of a situation type and are collected in situation classes.

An activation diagram specifies graphically which activity has to be performed upon some triggering event provided that an associated condition is satisfied. Rules specified by means of activation diagrams are executed reliably and controllably. Two rule categories are provided: (1) start-rules, which are executed asynchronously to the triggering event, and (2) do-rules, which are executed temporally faithfully at the point in time when they are triggered by temporal events. Both start-rules and do-rules are executed in top-level transactions.

Summarizing, Situation/Activation Diagrams meet the requirements we have identified for modeling business rules at the conceptual level.

11.1.2 Logical Design

Since there is currently no accepted standard in the area of active object-oriented database systems, mapping of Active Object/Behavior Diagrams has been discussed for an existing prototype system. The selected system is the commercial object-oriented database system GemStone (GemStone Systems, Inc.) with its active extension TriGS ([KRSRV94, Hie96, Ret96, KRSR98]).

The basic strategy we have adopted for mapping Active Object/Behavior Diagrams to

GemStone/TriGS follows a translator (cf. Chapter 7). It is shown how the concepts of Active Object/Behavior Diagrams can be translated to concepts of GemStone/TriGS. We favoured a translator approach over an interpreter approach. Following an interpreter approach, GemStone/TriGS would have been extended by a small set of generic methods that interpret the concepts of Active Object/Behavior Diagrams. We argue that a translator approach is more appropriate for achieving an acceptable performance and additionally is more flexible concerning the adaptability of the generated schema to special cases.

The presented mapping covers all three dimensions of a schema specified with Active Object/Behavior Diagrams: object structure (cf. Chapter 8), passive object behavior (cf. Chapter 9), and active object behavior (cf. Chapter 10).

Object Structure. Object types, the core concept of Active Object/Behavior Diagrams for modeling object structure, are mapped to GemStone/TriGS classes rather straight. An object type for which a supertype is indicated is mapped to a subclass of the class representing its supertype. A property of an object type is mapped to an instance variable of the class that represents the object type and to a set of methods for manipulating the instance variable. Derived properties and the derivation expressions associated with them in the form of query diagrams are considered separately. Since GemStone/TriGS currently does not provide sophisticated query processing capabilities, query diagrams are mapped to an intermediate language. The selected intermediate language is OQL ([CB97]). OQL has been selected because of the similarities in structure and expressiveness between query diagrams and OQL and because of the role of OQL as de-facto standard that can be expected to be supported by GemStone in the future.

Passive Object Behavior. Activities and states are the core concepts of Active Object/Behavior Diagrams for modeling passive object behavior. Activities are mapped to methods. The states in which an instance of an object type resides are recorded by the instance in an instance variable.

The maintenance of integrity constraints receives special attention in mapping passive object behavior. In Active Object/Behavior Diagrams, public activities are the "units of integrity". This means that public activities have to preserve the integrity of a database by transforming it from one database state satisfying integrity constraints to another. An activity execution preserves the integrity of a database if the precondition holds before activity execution, if the postconditon holds after activity execution, if state changes are effected correctly, and if no integrity constraints related to object structure are violated. Integrity constraints related to object structure are invariant properties, obligatory properties, unique keys, subclass/superclass relationships, inverse relationships, referential integrity, and local referential integrity. A violation of an integrity constraint is not countered actively by the system (e.g., by means of an action compensating the violation) but it is countered passively by preventing a violating activity to be completed. An attempt to

violate an integrity constraint represents a non-recoverable error that is signaled to the user causing it. To free a developer implementing activities from checking integrity constraints, GemStone/TriGS rules are engaged to check integrity constraints and to signal errors in the case of violations.

The way start-rules and do-rules specified with Active Object/Behavior Diagrams are executed gave us the idea to enrich also a user's way of working with GemStone/TriGS. Therefore, we introduced a task model, in which a user can work with GemStone/TriGS in the conventional synchronous manner by issuing interactive tasks and additionally in a rule-like asynchronous manner by issuing activity-start-tasks and activity-do-tasks. Activity-start-tasks and activity-do-task represent activities that are registered with the task manager for execution. An activity-start-task is executed by performing its associated activity temporally faithfully some time after the task has been placed, but not later than a specified maximum-delay time. An activity-do-task is executed by performing its associated activity temporally faithfully at the point in time that is indicated explicitly when the task is placed.

The task manager, a module built on top of GemStone/TriGS, is designed as an "active object". It provides methods that are executed within rather short GemStone/TriGS sessions spawned by means of GemStone/TriGS rules. Every activity-start-task and every activity-do-task is executed in its own GemStone/TriGS session. The task manager guarantees a temporally faithful, controllable, and reliable execution of tasks. Temporal faithfulness is achieved in that every GemStone/TriGS session that executes a task places a commit-request with the task manager when a transaction should be committed, and in that the task manager grants the commit-requests according to task semantics. Controllable task execution is achieved in that every GemStone/TriGS session that executes a task reports exceptions that arise during the execution of the task to the agent who has issued the task. Reliable task execution is achieved in that every GemStone/TriGS session that executes a task restarts a transaction automatically if it has been aborted due to a concurrency conflict.

Active Object Behavior. A business rule specified with Active Object/Behavior Diagrams cannot be mapped to a single GemStone/TriGS rule since logical events specified with Active Object/Behavior Diagrams and composite events of GemStone/TriGS differ in their semantics considerably. The differences concern mainly the database state on which logical events and composite events are evaluated, the way a calendar system is incorporated, and the way intervals are represented. Logical events cannot be detected incrementally, but have to be evaluated in an ex-post manner at chronon changes. Evaluating them at chronon changes is the only way to ensure that the obtained occurrences reflect the correct database state.

Evaluating a logical event means evaluating its associated situation diagram. For the same reason as query diagrams, situation diagrams are mapped to OQL. A query evaluating

a logical event imposes conditions on basic events and on the result of evaluating other logical events. Conditions on basic events can be imposed only if the basic events are stored in the database or if they can be computed. Basic events as they are detected by GemStone/TriGS are neither stored in the database nor can they be computed; they are part of transient structures built by rule schedulers. Therefore, GemStone/TriGS rules are employed to detect occurrences of basic events and to insert them into collections. To avoid that all the events on which business rules are based have to be evaluated at every chronon change, potential occurrences of logical events are detected incrementally during a chronon and registered for further processing. Detection and registration is done by means of GemStone/TriGS rules.

The registered potential occurrences of logical events and the registered occurrences of basic events are picked up by rule handlers at chronon changes. A rule handler is a software agent designed as an active GemStone/TriGS object. It is responsible for handling a set of rules. Rules are executed on top of task management. No transactions are issued directly on GemStone/TriGS when rules are executed, but tasks are registered with the task manager. Different rule handlers are introduced for different kinds of rules. In a single GemStone/TriGS database there may be an arbitrary number of rule handlers. By default, a single start-rule handler responsible for all start-rules and a single do-rule handler responsible for all do-rules is assumed to exist. To make the handling of rules more efficient and scalable, additional rule handlers can be created and the responsibilities can be shared among them. A subset of do-rules—calendar-do-rules—can be handled by do-rule preschedulers. A calendar-do-rule is a do-rule defined on an event whose occurrence relies exclusively on preinstantiated calendar knowledge. Triggerings of a calendar-do-rule are known throughout the future. If a do-rule prescheduler is not busy, it looks ahead and preschedules future rule triggerings. A do-rule prescheduler reduces the number of do-tasks that are registered at chronon changes for immediate execution and, thus, reduces the number of tasks thar are likely to delay task execution.

11.2 Tool Support

Generally, the diagrams provided by Active Object/Behavior Diagrams could be drawn by hand. However, the different diagrams are highly interrelated and drawing them by hand may be cumbersome and error-prone. Therefore, using Active Object/Behavior Diagrams is better supported by a graphical editor, which (1) facilitates drawing the diagrams, (2) helps the user to ensure the consistency of the resulting conceptual schema, and (3) enables the automatic translation of the conceptual database schema into a logical database schema.

A prototype of a graphical editor for Active Object/Behavior Diagrams, the so-called *AOBD editor*, is available ([LOKT98]). It has been developed by Kraus and Thalhammer [KT97] using MetaDoME (Honeywell, Inc.), an editor for building diagram editors. The editor generated upon the specification in MetaDoME is based (like the MetaDoME

editor itself) on the graphical framework DoME, "Domain Modeling Environment" (Honeywell, Inc.)[1], which extends the class library of VisualWorks (ParcPlace-Digitalk, Inc.).

The editor distinguishes three kinds of objects, each kind is associated with a single layer: Objects providing interaction facilities with the designer ("DoME layer"), objects describing the visual representation of the modeling concepts ("graphical layer"), and objects holding the conceptual database schema by means of instances of object types of the meta model ("semantic layer"). The consistency of the conceptual schema is guaranteed during the interactive design of diagrams, i.e., the designer is prevented from modifying the schema in a way that violates its consistency. The syntactic correctness of the diagrams is ensured by not allowing to apply the components of diagrams in an invalid way. The semantic correctness is ensured by checking the consistency of the incrementally developed conceptual schema. Textual elements (e.g., pre- and postconditions of activities) are so far analyzed only syntactically. The expressions are parsed and the names of objects are resolved, but the semantic correctness of expressions is not checked in the current version of the editor.

The conceptual database schema (the graphical data as well as the semantic data) modeled by means of the AOBD editor may be stored either into the file system using the binary object streaming service (BOSS) provided by VisualWorks or into the object-oriented database system GemStone (GemStone Systems, Inc.).

11.3 Outlook

There are some aspects and open points of our approach that are worth to be mentioned and to be dealt with in the future.

- So far, situation diagrams cannot be defined recursively as it is the case for some query languages such as Datalog.
- Parametrizing situation diagrams (by specifying input parameters, which can be used to formulate conditions within the diagram) would allow to specify rule patterns [KRSRS96] in Active Object/Behavior Diagrams (cf. Section 3.6).
- Temporal databases use temporal elements to describe several segments of the time axis at once. A temporal element is a finite union of intervals [GN93]. Intervals can be considered as a special case of temporal elements. Currently, Active Object/Behavior Diagrams do not provide temporal elements but only intervals; a situation class containing several intervals may be used instead. However, a single temporal element defines a different semantics than a set of intervals and, therefore, providing temporal elements should be considered in the future.

[1]We thank Honeywell for making a DoME license available for us at no cost.

- Situations occurring within a single chronon are recognized to occur simultaneously because they are time-stamped with the same chronon. Considering also the processing time of situations (the time at which they occur within a transaction) would allow to recognize the sequence of situations that occur in the same transaction or in different transactions that are committed in the same chronon. However, in absence of a bi-temporal model covering the "effective time" as well as the "internal processing time," our approach to consider only the effective time (the time when the transaction requests to commit) is meaningful in practice because business rules are viewed from the external user's perspective.

- The mapping of Active Object/Behavior Diagrams to GemStone/TriGS as it has been presented in this thesis, supports "only" the concepts presented in Chapters 4, 5, and 6. The specialization of object structure, passive behavior, and active behavior as described in [Lan97] is not yet supported. A full-fledged mapping should incorporate the specialization concepts and should provide for checking the consistency of specialization at run-time.

- The presented mapping of Active Object/Behavior Diagrams to GemStone/TriGS disregarded the recovery of a GemStone/TriGS database after a system crash. This has been done since the recovery after a system crash is open in TriGS. In TriGS, where event occurrences are not stored in the database but are part of transient structures built by rule schedulers, event occurrences are lost after a system crash. Intuitively, loosing event occurrences is less critical in our setting. Basic events raised in committed transactions are stored in the database, and logical events can be reconstructed by issuing queries. Also the active objects (task manager and rule handlers) store the state in which they reside after the last successfull transaction and thus can continue their work after a recovery. More critical, however, would be the loss of database views. Consider for example a transaction that considers start-rules in its private (possibly outdated) view when the system crashes. After a recovery only the current database state is reconstructed. Private views of transactions running when the system crashed are lost.

- It has been a goal of this thesis to show how Active Object/Behavior Diagrams can be mapped to active object-oriented database systems. In absence of an accepted standard, GemStone/TriGS has been selected as the target system. To gain broader insights in mapping Active Object/Behavior Diagrams it may be interesting to consider also other target systems. Generally, we believe that the lessons learned from mapping Active Object/Behavior Diagrams to GemStone/TriGS can be transferred to other active object-oriented database systems. The only feature of GemStone that has been exploited in the presented mapping and that is not a common feature of active object-oriented database systems is optimistic concurrency control. Throughout a transaction, a single view of the database is presented to a session. The view is created and frozen at the begin of the transaction. Thus, optimistic concurrency

control together with private views are excellent mechanisms for preserving the database state as it has been at the begin of a transaction. Time-consuming read-only operations that have to be performed on a particular database state (e.g., the consideration of start-rules) can be executed in a transaction without causing concurrency conflicts with other transactions. If a system does not support optimistic concurrency control, workarounds have to be developed.

List of Figures

2.1 From immediate checking to active object-oriented database systems 12

2.2 Comparison of immediate checking, polling, and active database systems . 14

3.1 Elements of the business area 35

3.2 Database design process 39

3.3 Where the elements of the business area are reflected in information-system design 41

4.1 The global object type COMMUNICATION_SYSTEM 64

4.2 Object types representing users and communications 65

4.3 Object-class diagram of object type NOTICE_BOARD_ADMIN 73

4.4 Life-cycle diagram of object type COMMUNICATION_SYSTEM (selection) . . 75

4.5 Life-cycle diagram of object type CIRCULAR 75

4.6 Specification of activity createCircular 77

4.7 Specification of activity create 77

4.8 Specification of activity deliverToNextReceiver 78

4.9 Specification of compound activity recordStepCompletion 79

4.10 Query diagram for property noOfNotices 81

4.11 Query diagram for property AllNoticeBoardsInUse 82

4.12 Skeleton of a domain box 84

4.13 Tuple creation 84

4.14 Scalar-comparison predicates 85

4.15 Scalar/set-comparison predicates 85

4.16 Set-comparison predicates 85
4.17 Query diagram for property AllNoticeBoardsInUse 87
4.18 Query diagram for property AllNoticeBoardsInUse 88
4.19 Query diagram for property MailReports 90
4.20 Query diagram for property LateUsers 92
4.21 Query diagram for property UnreadMails 93

5.1 Situation-type diagrams 97
5.2 Object-type diagrams including situation classes 100
5.3 The global object type including situation classes 101
5.4 Life-cycle diagram of object type NOTICE 102
5.5 Situation diagram for situation class OwnNoticeExpiries 105
5.6 Situation diagram for situation class NoticeExpiries 106
5.7 Situation diagram of situation class RepeatedUnexpCommHandlings 106
5.8 Clock ticks and chronons 108
5.9 Situation diagram for situation class expiry 110
5.10 Situation diagram for situation class commentDelay 112
5.11 Situation diagram for situation class RepeatedLateMailReadings 113
5.12 Situation diagram for situation class RepeatedCommentLacks 114
5.13 The declarative semantics of Sentinel's sequence operator 116

6.1 Rule Expiration of object type NOTICE 124
6.2 Rule NoticeRemoval of object type NOTICE_BOARD 124
6.3 Rule DelayRecording of object type CIRC_STEP_DELIVERY 125
6.4 Schematic representation of temporally faithful execution 130

7.1 Syntax of a TriGS rule ([KRSR98]) 135
7.2 Interpreting approach—class `AOBDObject` (selection) 142
7.3 Translator approach—class `Item` (selection) 143

8.1 Query shared by the instances of a class 156

8.2 General form of an OQL select statement 158
8.3 Structure of a navigation path 161
8.4 Mapping a navigation path to a select statement 161
8.5 Skeleton of a domain box 162
8.6 Selecting query diagram 164
8.7 Select statement for a selecting query diagram 164
8.8 Generating query diagram 165
8.9 Select statement for a generating query diagram 165
8.10 Generating query diagram with grouped result 166
8.11 Select statement for a generating query diagram with grouped result 167
8.12 Query diagram with alternatives 168
8.13 Select statement for a query diagram with alternatives 169

9.1 Rule for checking the precondition of an activity 177
9.2 Rules for checking the postcondition of an activity 178
9.3 Rules performing state changes 179
9.4 Rule ensuring property invariance 180
9.5 Rule checking obligatory properties 181
9.6 Rules for extension management 182
9.7 Rule checking uniqueness of key 182
9.8 Rules checking subclass/superclass relationships 183
9.9 Rule for IRM notification (single-valued relationship) 185
9.10 Rules for IRM notification (multi-valued relationship) 185
9.11 Rule for checking inverse relationships 186
9.12 Rules for maintaining references 187
9.13 Rule for checking referential integrity 188
9.14 Participants in a local referential integrity constraint 189
9.15 Rules for LRIM root object maintenance 192
9.16 Rules for LRIM source path maintenance 193

9.17 Rules for LRIM target path maintenance 194
9.18 Rules for LRIM reference maintenance 195
9.19 Rule for checking local referential integrity 196
9.20 Serialization of transactions in GemStone/TriGS 206
9.21 Classes implementing task management 208
9.22 Task manager: session or active object? 209
9.23 External interface of task manager 210
9.24 Testing conflicts with concurrent transactions 212
9.25 Commit phase of transactions 213
9.26 Life cycle of an activity-start-task 217
9.27 Life cycle of an activity-do-task 217
9.28 Life cycle of an interactive task 219
9.29 Transactions in chronon i 220
9.30 The task manager: controlling tasks and transactions 222
9.31 Rules controlling the task manager (Part 1) 224
9.32 Rules controlling the task manager (Part 2) 225
9.33 Life cycle of a halt-and-repair-task 226
9.34 Life cycle of an exception-report-task 227

10.1 Mapping basic rules: a first approach 230
10.2 Mapping basic rules: using rule handlers 232
10.3 Logical situation vs. composite event 233
10.4 Basic calendar interval types 236
10.5 Rule populating an activity-event class 239
10.6 Rules populating a state-interval class 240
10.7 Example for a selecting and a generating event expression 242
10.8 Starting points for detecting potential occurrences 243
10.9 Potential occurrence of a selected logical event 244
10.10 Potential occurrence of a generated logical event 244

10.11Rules registering potential occurrences 245
10.12Registration of potential event occurrences triggering rules 246
10.13Starts and ends of state intervals 248
10.14Potential start/end of selected logical interval 249
10.15Potential start/end of a generated logical interval 250
10.16Expressing temporal conditions in OQL queries 254
10.17Rule classes 259
10.18Handler for start-rules 264
10.19Rules controlling a start-rule handler 265
10.20Handler for do-rules 266
10.21Rules controlling a do-rule handler 267
10.22Prescheduler for do-rules 268
10.23Rules controlling a do-rule prescheduler (part 1) 269
10.24Rules controlling a do-rule prescheduler (part 2) 270

List of Tables

3.1 Examples of conceptual and logical models in database design 43

6.1 Statistics of temporally faithful executions 131

7.1 Parameters of primitive TriGS events ([Ret96]) 137

8.1 Mapping of data types to GemStone/TriGS kernel classes 150

9.1 Rules and priorities (Part 1) 198

9.2 Rules and priorities (Part 2) 198

9.3 Error signals and their arguments 200

9.4 Exceptions in the different kinds of tasks 202

Bibliography

[AA91] J. Almarode and T.L. Anderson. GemStone Visual Schema Designer: A Tool for Object-Oriented Database Design. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Proc. of the IFIP TC2/WG 2.6 Working Conf. on Object-Oriented Databases: Analysis, Design & Construction, Windermere, UK*. North-Holland, 1991.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K.R. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. of the 1st Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 223–240. Elsevier Science Publishers, December 1989.

[ABJZ96] H.-J. Appelrath, H. Behrends, H. Jasper, and O. Zukunft. Case Studies on Active Database Applications: Lessons Learned. In R.R. Wagner and H. Thoma, editors, *Proc. of the 7th Intl. Conf. on Database and Expert Systems Applications (DEXA)*, LNCS 1134, pages 69–78. Springer, September 1996.

[ACL91] R. Agrawal, R. Cochrane, and B. Lindsay. On Maintaining Priorities in a Production Rule System. In Lohman et al. [LSC91], pages 479–487.

[AG89] R. Agrawal and N. H. Gehani. Ode (Object Database and Environment): The Language and the Data Model. In Clifford et al. [CLM89], pages 36–45.

[AMC93] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. volume 22 of *SIGMOD RECORD*, pages 99 – 108. ACM Press, June 1993.

[AWH92] A. Aiken, J. Widom, and J.M. Hellerstein. Behaviour of Database Production Rules: Termination, Confluence, and Observable Determinism. In M. Stonebraker, editor, *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, volume 21 of *SIGMOD Record*, pages 59–68, June 1992.

[BBKZ92] A.P. Buchmann, H. Branding, T. Kudrass, and J. Zimmermann. REACH: A REal-time, ACtive, and Heterogeneous Mediator System. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4), 1992.

[BBKZ93] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In *Proc. of the 1st Intl. Workshop on Rules in Database Systems, Edinburgh, Scotland*, 1993.

[BBL76] B.W. Boehm, J.R. Brown, and M. Lipow. Quantitative Evaluation of Software Quality. In *Proc. of 2nd Intl. Conf. on Software Engineering*, pages 286–299. IEEE, 1976.

[BC79] P. Buneman and E. Clemons. Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems*, 4(3):368–382, September 1979.

[BCN92] C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design.* Benjamin/Cummings, 1992.

[BCP96] E. Baralis, S. Ceri, and S. Paraboschi. Modularization Techniques for Active Rules Design. *ACM Transactions on Database Systems*, 21(1):1–29, March 1996.

[Ber94] Mikael Berndtsson. Reactive Object-Oriented Databases and CIM. In *Proceedings of the 5th International Conference on Database and Expert Systems Applications, Athens, Greece*, September 1994.

[Bic98] P. Bichler. *Conceptual Design of Secure Workflow Systems: An Object-Oriented Approach to the Uniform Modeling of Workflows, Organizations, and Security.* Dissertation, University of Linz, Austria, 1998.

[BL92] M. Berndtsson and B. Lings. On developing reactive object-oriented databases. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4), 1992.

[BL95] Mikael Berndtsson and Brian Lings. Logical Events and ECA Rules. Technical Report HS-IDA-TR-95-004, University of Sövde, 1995.

[BLT86] J. Blakely, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 61–71, 1986.

[BM91a] E. Bertino and L. Martino. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, 24(4), April 1991.

[BM91b] M. Bouzeghoub and E. Metais. Semantic Modeling of Object-Oriented Databases. In Lohman et al. [LSC91].

[BS94] P. Bichler and M. Schrefl. Active Object-Oriented Database Design Using Active Object/Behavior Diagrams. In J. Widom and S. Chakravarthy, editors, *Proc. of the 4th Intl. Workshop on Research Issues in Data Engineering (RIDE)*, Active Databse Systems. IEEE Computer Society, February 1994.

[BZBW95] A.P. Buchmann, J. Zimmermann, J.A. Blakeley, and D.L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proc. of the 11th Intl. Conf. on Data Engineering (ICDE)*, 1995.

[CAM93] S. Chakravarthy, E. Anwar, and L. Maugis. Design and Implementation of Active Capability for an Object-Oriented Database. Technical Report UF-CIS-TR-93-001, University of Florida, Computer and Information Sciences, January 1993.

[CB97] R.G.G. Cattell and D. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[CBFP95] S. Ceri, E. Baralis, P. Fraternali, and S. Paraboschi. Design of Active Rule Applications: Issues and Approaches. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Proc. of the 4th Intl. Conf. On Deductive and Object-Oriented Databases (DOOD)*, LNCS 1013, pages 1–18. Springer, December 1995.

[CC96] C. Collet and T. Coupaye. Composite Events in NAOS. LNCS 1134. Springer, September 1996.

[CCRL+90] S. Ceri, S. Crespi-Reghizzi, G. Lamperti, L. Lavazza, and R. Zicari. Algres: An advanced database for complex applications. *IEEE Software*, 7(4), July 1990.

[CCS94] C. Collet, T. Coupaye, and T. Svensen. NAOS: Efficient and modular reactive capabilities in an Object-Oriented Database System. In *Proc. of the 20th Intl. Conf. on Very Large Databases, Santiago, Chile*, 1994.

[CF97] S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules: the IDEA Methodology*. Addison-Wesley, 1997.

[CFPT95] S. Comai, P. Fraternali, G. Psaila, and L. Tanca. A Customizable Model for the Semantics in Active Databases. In *6th IFIP TC-2 Working Conference on Data Semantics, Atlanta*, May 1995.

[CFPT96] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. *Active Rule Management in Chimera*, chapter 6, pages 151–176. In Widom and Ceri [WC96a], 1996.

[CGMR95] S. Castangia, G. Guerrini, D. Montesi, and G. Rodriguez. Design and Implementiation for the Active Rule Language of Chimera. In N. Revell and A M. Tjoa, editors, *Proc. of the Workshop on Database and Expert Systems Applications (DEXA)*. OMNIPRESS, 1995.

[Cha89] S. Chakravarthy. Rule Management and Evaluation: An Active DBMS Perspective. *ACM SIGMOD RECORD*, 18(3):20–28, September 1989.

[Cha95] S. Chakravarthy. Architectures and monitoring techniques for active databases: An evaluation. *Data & Knowledge Engineering*, 16(1):1–26, July 1995.

[Cha97] S. Chakravarthy. SENTINEL: An Object-Oriented DBMS with Event-Based Rules. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Tucson, Arizona*, 1997.

[Che76] P.P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–37, March 1976.

[Che77] P.P. Chen. The Entity-Relationship Model: A Basis for the Enterprise View of Data. In *Proceedings of the 10th Int. Conference of Data Engineering (ICDE)*, number 46, pages 76–84, 1977.

[CHS92] S. Chakravarthy, E. Hanson, and S.Y.W. Su. Active Data/Knowledge Base Research at the University of Florida. *IEEE Date Engineering Bulletin, Special Issue on Active Databases*, 15(4), 1992.

[CKAK93] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Anatomy of a Composite Event Detector. Technical Report UF-CIS-TR-93-039, University of Florida, Computer and Information Sciences, December 1993.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proc. of the 20th Intl. Conf. On Very Large Databases (VLDB)*, 1994.

[CKTB95] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R.H. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. In *Proc. of the 11th Intl. Conf. on Data Engineering (ICDE), Taipei, Taiwan.* IEEE Computer Society Press, 1995.

[CLM89] J. Clifford, B. Lindsay, and D. Maier, editors. *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon*, volume 18 of *SIGMOD Record*. ACM Press, May 1989.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data & Knowledge Engineering*, 14(1):1–26, November 1994.

[Com73] CODASYL Data Description Language Committee. CODASYL Data Description Language Journal of Development. *NBS Handbook 113*, June 1973.

[Con96] ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *SIGMOD RECORD*, 25(3):40–49, September 1996.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proc. of the 17th Intl. Conf. on Very Large Data Bases (VLDB)*, September 1991.

[CW96a] S. Ceri and J. Widom. *Applications of Active Databases*, chapter 10, pages 259–291. In Widom and Ceri [WC96a], 1996.

[CW96b] S. Ceri and J. Widom. *Standards and Commercial Systems*, chapter 9, pages 232–258. In Widom and Ceri [WC96a], 1996.

[Dat83] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, Massachusetts, 1983.

[Day88] U. Dayal. Active Database Management Systems. In *Proc. of the 3rd Intl. Conf. on Data and Knowledge Bases*, pages 150–167, June 1988.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, and S. Sarin. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD RECORD*, 17(1):51–70, March 1988.

[DBC96] U. Dayal, A. P. Buchmann, and S. Chakravarthy. *The HiPAC Project*, chapter 7, pages 177–206. In Widom and Ceri [WC96a], 1996.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are Objects too: A Knowledge Model for an Active, Object-Oriented Database System. In *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, Lecture Notes in Computer Science 334, pages 129–143. Springer, 1988.

[DD91] U. Dayal and K. R. Dittrich. Active Database Systems. Tutorial, 17th International Conference on Very Large Data Bases, Barcelona, Spain, September 1991.

[Deu91] O. Deux. The O_2 System. *Communication of the ACM*, 34(10), October 1991.

[DG90] O. Diaz and P. Gray. Semantic-rich User-defined Relationships as a Main Constructor in Object-Oriented Databases. In R. Meersman and W. Kent, editors, *Proceedings of the IFIP TC2 Working Conference on Object-Oriented Databases (DS-4)*, North-Holland, 1990.

[DG93] K. R. Dittrich and S. Gatziu. Time Issues in Active Database Systems. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases, Arlington, Texas*, June 1993.

[DJ94] O. Diaz and A. Jaime. EXACT: An EXtensible approach to ACTive object-oriented databases. Technical report, Universidad del Pais Vasco / Euskal Herriko Unibertsitatea, Spain, 1994.

[DJPaQ94] O. Diaz, A. Jaime, N.W. Paton, and G. al Qaimari. Supporting Dynamic Displays Using Active Rules. *ACM SIGMOD Record*, 23(1), 1994.

[DP91] O. Diaz and N. W. Paton. Sharing Behaviour in an Object-Oriented Database Using a Rule-Based Mechanism. In M. Jackson and A. Robinson, editors, *Proceedings of the 9th British National Conference on Object-Oriented Databases*, pages 17–37, 1991.

[DPG91] O. Diaz, N. W. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Uniform Approach. In Lohman et al. [LSC91], pages 317–326.

[EC75] K. P. Eswaran and D. D. Chamberlain. Functional Specification of a Subsystem for Data Base Integrity. In *Proceedings of the 1st International Conference on Very Large Data Bases*, September 1975.

[EJK93] R. Elmasri, S. James, and V. Kouramajian. Automatic Class and Method Generation for Object-Oriented Databases. In *Proc. of 3rd Intl. Conf. on Deductive and Object-Oriented Databases*, 1993.

[EKW92] D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-Oriented Systems Analysis—A Model-Driven Approach.* Yourdon Press, 1992.

[Esw76] K. Eswaran. Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated Data Base System. Research Report RJ1820, IBM, August 1976.

[FV95] Ch. Fahrner and G. Vossen. A survey of database design transformations based on the Entity-Relationship model. *Data & Knowledge Engineering*, (15):213–250, 1995.

[GD92] S. Gatziu and K. R. Dittrich. SAMOS: An Active Object-Oriented Database System. In *Bulletin of the Technical Committee on Data Engineering*, volume 15, pages 23–26. IEEE Computer Society, December 1992.

[GD93a] S. Gatziu and K. R. Dittrich. Eine Ereignissprache für das aktive, objektorientierte Datenbanksystem SAMOS. In W. Stucky and A. Oberweis, editors, *Datenbanksysteme in Büro, Technik und Wissenschaft. GI-Fachtagung, Braunschweig.* Gesellschaft für Informatik, Springer Verlag, March 1993. Veröffentlichung von Gatz92a.

[GD93b] S. Gatziu and K. R. Dittrich. Events in an Active Object-Oriented Database System. In N. W. Paton and M. H. Williams, editors, *Proceedings of the 1st International Workshop on Rules in Database Systems, Edinburgh, Scotland.* British Computer Society, Springer Verlag, September 1993.

[GD94] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In J. Widom and S. Chakravarthy, editors, *Proc. of the 4th Intl. Workshop on Research Issues in Data Engineering (RIDE)*, Active Databse Systems, pages 2–9. IEEE Computer Society, February 1994.

[Gem96a] GemStone Systems, Inc. *GemBuilder for C Version 5.0*, 1996.

[Gem96b] GemStone Systems, Inc. *GemBuilder for VisualWorks Version 5.0*, 1996.

[Gem96c] GemStone Systems, Inc. *GemStone Programming Guide Version 5.0*, 1996.

[GGD91] S. Gatziu, A. Geppert, and K. R. Dittrich. Integrating Active Objects into an Object-Oriented Database System. In *Proceedings of the 3rd International Workshop on Database Programming Languages (DBPL), Nafplion (Greece)*, August 1991.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In Lohman et al. [LSC91], pages 327–336.

[GJ96] N. Gehani and H. V. Jagadish. *Active Database Facilities in Ode*, chapter 8, pages 207 – 232. In Widom and Ceri [WC96a], 1996.

[GJMS93] N.H. Gehani, H.V. Jagadish, Inderpal Singh Mumick, and Oded Shmueli. Temporal Queries for Active Database Support. Technical report, AT&T Bell Laboratories, 1993.

[GJS92a] N.H. Gehadi, H.V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In L.-Y. Yuan, editor, *Proc. of the 18th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 327 – 338, August 1992.

[GJS92b] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Object-Oriented Database. volume 21 of *SIGMOD Record*, pages 81–90. ACM Press, June 1992.

[GJS92c] N.H. Gehani, H.V. Jagadish, and Oded Shmueli. COMPOSE. A System for Composite Event Specification and Detection. Technical report, AT&T Bell Laboratories, 1992.

[GMS92] N.H. Gehani, Inderpal Singh Mumick, and Oded Shmueli. Events with Attributes in an Active Database. Technical report, AT&T Bell Laboratories, 1992.

[GN93] S.K. Gadia and S.S. Nair. Temporal Databases: A Prelude to Parametric Data. In *Temporal Databases: Theory, Design, and Implementation*, chapter 2, pages 28–66. Benjamin Cummings, 1993.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Han87] E. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 440–453, May 1987.

[Han89] E. N. Hanson. An Initial Report on The Design of Ariel: A DBMS With an Integrated Production Rule System. *ACM SIGMOD RECORD*, 18(3):12–19, September 1989.

[Han96] E.N. Hanson. *The Ariel Project*, chapter 3. In Widom and Ceri [WC96a], 1996.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Her95] H. Herbst. A Meta-Model for Business Rules in Systems Analysis. In J. Iivari, K. Lyytinen, and M. Rossi, editors, *Proc. of the 7th Intl. Conf. on Advanced Information Systems Engineering (CAiSE'95)*, pages 186–199, 1995.

[Her97] H. Herbst. *Business Rule-Oriented Conceptual Modeling*. Physica, 1997.

[Hie96] R. Hiebl. Komplexe Ereignisse in TriGS–Konzeption und Implementierung der Ereigniserkennung und Regelausfürung. Master's thesis, University of Linz, Austria, 1996. in German.

[HK86] S. Hudson and R. King. A Database System for Specifying Functionally-Defined Data. In *Proceedings of the 1st International Workshop on Object-Oriented Database Systems*, pages 26–37. IEEE Computer Science Press, September 1986.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 10(2), 1987.

[HKMS94] H. Herbst, G. Knolmayer, T. Myrach, and M. Schlesinger. The specification of business rules: a comparison of selected methodologies. In A.A. Verrijn-Stuart and T.W. Olle, editors, *Methods and Associated Tools for the Information System Life Cycle*, pages 29–46. IFIP, North-Holland, 1994.

[HLM88] M. Hsu, R. Ladin, and D. R. McCarthy. An Execution Model for Active Data Base Management Systems. pages 171–179, June 1988.

[HM96] H. Herbst and T. Myrach. A Repository System for Business Rules. In R. Meersman and L. Mark, editors, *Proc. of the 6th IFIP TC-2 Working Conference on Data Semantics*, London, 1996. Chapman & Hall.

[JCJO92] I. Jacobson, M. Christerson, M. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesely, 1992.

[Kap91] G. Kappel. Reorganizing Object Behavior by Behavior Composition – Coping with Evolving Requirements in OfficeSystems. In H.J. Appelrath, editor, *Proc. BTW'91 (Datenbanken für Büro, Technik und Wissenschaft).* Springer IFB 270, March 1991.

[KNS89] W. Klas, E.J. Neuhold, and M. Schrefl. Tailoring Object-Oriented Data Models through Metaclasses. In *Proceedings of Advanced Database System Symposium, Kyoto, Japan*, pages 169–178. Special Interest Group on Database Systems, Information Processing Society of JAPAN, December 1989.

[KP81] S. Koenig and R. Paige. A Transformational Framework for the Automatic Control of Derived Data. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 306–318, September 1981.

[KRSR98] G. Kappel, S. Rausch-Schott, and W. Retschitzegger. A Tour on the TriGS Active Database System—Architecture and Implementation. In *Proc. of the ACM Symposium on Applied Computing, Atlanta, Georgia*, 1998.

[KRSRS96] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen. From Rules to Rule Patterns. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Proc. of the Conf. on Advanced Information Systems Engineering (CAiSE)*, LNCS 1080, pages 99–115. Springer, May 1996.

[KRSRV94] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and S. Vieweg. TriGS: Making a Passive Object-Oriented Database System Active. *Journal of Object-Oriented Programming*, 7(4):40–51, July/August 1994.

[KS89] G. Kappel and M. Schrefl. A Behavior Integrated Entity-Relationship Approach for the Design of Object-Oriented Databases. In C. Batini, editor, *Proc. of the 7th Intl. Conf. on the Entity-Relationship Approach*, pages 311–328. North-Holland, 1989.

[KS91a] G. Kappel and M. Schrefl. Object-Behavior Diagrams. In *Proceedings of the 7th International Conference on Data Engineering, Kobe, Japan*, pages 530–539, 1991.

[KS91b] G. Kappel and M. Schrefl. Using an Object-Oriented Diagram Technique for the Design of Information Systems. In H.G. Sol and K.M.Ṽan Hee, editors, *Proc. of the Intl. Working Conf. on Dynamic Modelling of Information Systems.* North-Holland, 1991.

[KS92] G. Kappel and M. Schrefl. Local Referential Integrity. In G. Pernul and A M. Tjoa, editors, *Proc. of the Intl. Conf. on Entity-Relationship Approach.* Springer LNCS 645, 1992.

[KS93] G. Kappel and M. Schrefl. Panel Discussion on Active Database Systems. In C. Beeri, A. Heuer, and G. Saake, editors, *Formal Aspects of Object Base Dynamics*, number 62 (9317) in Dagstuhl-Seminar-Report, April 1993.

[KS94] G. Kappel and M. Schrefl. Inheritance of Object Behavior – Consistent Extension of Object Life Cycles. In J. Eder and L. Kalinichenko, editors, *Proc. of the Intl. East/West Database Workshop*. Springer WSCS, 1994.

[KS96a] G. Kappel and M. Schrefl. Modeling Object Behavior: To use rules or methods or both. In R.R. Wagner and H. Thoma, editors, *Proc. of the 7th Intl. Conf. on Database and Expert Systems Applications (DEXA'96)*, LNCS 1134. Springer, 1996.

[KS96b] G. Kappel and M. Schrefl. *Objektorientierte Informationssysteme—Konzepte, Darstellungsmittel, Methoden*. Springer, 1996. (in German).

[KT97] W. Kraus and T. Thalhammer. Design and Implementation of a Graphical Editor for Active Object/Behavior Diagrams. Master's thesis, Universität Linz, Austria, September 1997.

[Lan97] P. Lang. *Conceptual Active Object-Oriented Database Design*. Dissertation, Universität Linz, September 1997.

[LGA96] D.F. Lieuwen, N.H. Gehani, and R.M. Arlein. The Ode Active Database: Trigger Semantics and Implementation. In *Proc. of the 12th Intl. Conf. on Data Engineering (ICDE)*, pages 412–420. IEEE Computer Society, March 1996.

[LH89] K.J. Lieberherr and I. Holland. Assuring good style for object-oriented programming. *IEEE Software*, pages 38–48, September 1989.

[LHM86] B. Lindsay, L. Haas, and C. Mohan. A Snapshot Differential Refresh Algorithm. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 53–60, 1986.

[LOKT98] P. Lang, W. Obermair, W. Kraus, and T. Thalhammer. A Graphical Editor for the Conceptual Design of Business Rules. In *Proc. of the 14th Intl. Conf. on Data Engineering, Orlando, Florida*. Computer Society Press, 1998.

[Lor95] M. Lorenz. *Rapid Software Development with Smalltalk*. SIGS Books, 1995.

[LOS96] P. Lang, W. Obermair, and M. Schrefl. Situation Diagrams. In R.R. Wagner and H. Thoma, editors, *Proc. of the 7th Intl. Conf. on Database and Expert Systems Applications*, LNCS 1134, pages 400–421. Springer, September 1996.

[LOS97] P. Lang, W. Obermair, and M. Schrefl. Modeling Business Rules with Situation/Activation Diagrams. In A. Gray and P. Larson, editors, *Proc. of the 13th Intl. Conf. on Data Engineering (ICDE)*, pages 455–464. IEEE Computer Society Press, April 1997.

[LSC91] Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors. *Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, Spain*. Morgan Kaufmann Publishers, September 1991.

[LSR87] M. Loomis, A. Shah, and J. Rumbaugh. An Object Modelling Technique for Conceptual Design. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 325 – 335, 1987.

[Man87] F. A. Manola. PDM: An Object-Oriented Data Model for PROBE. Technical report, CCA, September 1987.

[MD86] F. A. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In *Proceedings of the 1st International Workshop on Object-Oriented Database Systems*. IEEE Computer Science Press, September 1986.

[MD89] D. R. McCarthy and U. Dayal. The Architecture of an Active Data Base Management System. In Clifford et al. [CLM89], pages 215–224.

[ME96] W.Y. Mok and D.W. Embley. Transforming Conceptual Models to Object-Oriented Database Design: Practicalities, Properties, and Peculiarities. In *Proc. of 15th Intl. Conf. on Conceptual Modelling, Cottbus, Germany*, 1996.

[Mor83] M. Morgenstern. Active Databases as a Paradigm for Enhanced Computing Environments. In *Proceedings of the 9th International Conference on Very Large Data Bases*, pages 34–42, 1983.

[Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT, Cambridge (MA), 1981.

[MPC96] R. Meo, G. Psaila, and S. Ceri. Composite Events in Chimera. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, LNCS 1057. Springer, March 1996.

[MS93] J. Melton and A.R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.

[Müc94] T.A. Mück. Design Support for Initiatives and Policies in Conceptual Models of Information Systems—A Statechart Approach. In *Proc. of the 27th Hawaiian Intl. Conf. om System Sciences (HICSS)*, volume IV. IEEE Computer Society Press, 1994.

[Müc95] T.A. Mück. *Active Databases—Concepts and Design Support.* Advances in Computers. Academic Press, 1995.

[OHM+91] T.W. Olle, J. Hagelstein, I.G. Macdonald, C. Rolland, H.G. Sol, F.J.M.Ṽan Assche, and A.A. Verrijn-Stuart. *Information Systems Methodologies: A Framework for Understanding.* Addison-Wesley, 1991.

[ORS95] W. Obermair, B. Röck, and M. Schrefl. Active Object-Oriented Database Systems. Manuscript, Institut für Wirtschaftsinformatik, Universität Linz, 1995.

[OY87] Z. Oszoyoglu and L. Yuan. A New Normal Form for Nested Relations. *ACM Transactions on Database Systems*, 12(1), March 1987.

[PCFW95] N. W. Paton, J. Campin, A. Fernandes, and M. Howard Williams. Formal Specification Of Active Database Functionality: A Survey. Lecture Notes in Computer Science. Springer, September 1995.

[PDB93] N. W. Paton, O. Diaz, and M. L. Barja. Combining Active Rules and Metaclasses for Enhanced Extensibility in Object-Oriented Systems. *Data & Knowledge Engineering*, 10:45–63, 1993.

[PDW+93] N. W. Paton, O. Diaz, M. H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of Active Behavior. In N. W. Paton and M. H. Williams, editors, *Proceedings of the 1st International Workshop on Rules in Database Systems, Edinburgh, Scotland*, pages 40–57. British Computer Society, Springer Verlag, September 1993.

[Pet86] J.L. Peterson. *Petri nets and the theory of modelling systems.* Prentice Hall, 1986.

[PS96] S. Potamianos and M. Stonebraker. *The POSTGRES Rule System*, chapter 2. In Widom and Ceri [WC96a], 1996.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[Ret96] W. Retschitzegger. *A Tour on TriGS: Development of an Active System and Application of Rule Patterns for Active Database Design.* PhD thesis, University of Linz, Austria, 1996.

[Rie96] A.J. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

[RKB87] M. Roth, H. Korth, and D. Batory. SQL/NF: A Query Language for ¬1NF Relational Databases. *Information Systems*, 12(1), 1987.

[Rou82] N. Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems*, 7(4):258–290, June 1982.

[Rum87] J. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, volume 22 of *SIGPLAN Notices*, pages 466 – 481, December 1987.

[SC96] M.J.V. Silva and C.R. Carlson. Conceptual Design of Active Object-Oriented Database Applications Using Multi-Level Diagrams. In P. Cointe, editor, *Proc. of the 10th European Conf. on Object-Oriented Programming (ECOOP)*, LNCS 1098. Springer, July 1996.

[Sch90] M. Schrefl. Behavior Modeling by Stepwise Refining Behavior Diagrams. In *Proceedings of the 9th International Conference on Entity Relationship Approach, Lausanne*, pages 113–128, 1990.

[SHP89] M. Stonekraker, M. Hearst, and S. Potamianos. A Commentary on the POSTGRES Rule System. *ACM SIGMOD RECORD*, 18(3), September 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Atlantic City*, May 1990.

[SK91] M. Schrefl and G. Kappel. Cooperation Contracts. In T.J. Teorey, editor, *Proceedings of the 10th International Conference on Entity Relationship Approach, San Mateo, California*, pages 285–307, 1991.

[SK96] E. Simon and J. Kiernan. *The A-RDL System*, chapter 5. In Widom and Ceri [WC96a], 1996.

[SKD95] E. Simon and A. Kotz-Dittrich. Promises and Realities of Active Database Systems. In *Proc. of the 21st Intl. Conf. on Very Large Databases (VLDB)*, pages 642–653, 1995.

[SNF80] C.S. Dos Santos, E.J. Neuhold, and A.L. Furtado. *A Data Type Approach to the Entity-Relationship Model*, pages 103–120. North-Holland, 1980.

[SR86] M. Stonebraker and L. Rowe. The Design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 340–355, 1986.

[SRL+90] M. Stonebraker, L. A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, and D. Beech. Third Generation Database System Manifesto. *ACM SIGMOD Record*, 19(3):31–44, September 1990.

[SS77] J. M. Smith and D. C. P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2), 1977.

[SS86] H.J. Schek and M.H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2):137–147, 1986.

[SS95] M. Schrefl and M. Stumptner. Behavior Consistent Extension of Object Life Cycles. In *Proc. of the 14th Intl. Conf. on the Entity-Relationship Approach*. Springer LNCS, 1995.

[SS97] M. Schrefl and M. Stumptner. Behavior Consistent Refinement of Object Life Cycles. In *Proc. of the 16th Intl. Conf. on Conceptual Modeling (ER)*, 1997.

[SSW80] P. Scheuermann, G. Schiffner, and H. Weber. Abstraction Capabilities and Invariant Properties Modeling within the Entity-Relationship Approach. In P. Chen, editor, *Proc. of the 1st Intl. Conf. on Entity-Relationship Approach*, pages 121–140. North-Holland, 1980.

[Sto82] M. Stonebraker. A Rules System for a Relational Data Base Management System. In *Proceedings of the 2nd International Conference on Databases, Jerusalem*, June 1982.

[Sto85] M. Stonebraker. *Triggers and Inference in Database Systems*. Springer Verlag, June 1985.

[Sto87] M. Stonebraker. The Design of the POSTGRES Rules Systems. In *Proceedings of the 13th International Conference on Very Large Data Bases, Brighton*, September 1987.

[SW95] A.P. Sistla and O. Wolfson. Temporal Triggers in Active Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):471 – 486, June 1995.

[Tan92] A.K. Tanaka. *On Conceptual Design of Active Databases*. PhD thesis, Georgia Institute of Technology, 1992.

[Tar92] Z. Tari. On the Design of Object-Oriented Databases. In *Proc. of 11th Intl. Conf. on the Entity-Relationship Approach, Karlsruhe, Germany*, 1992.

[TNCK91] A.K. Tanaka, S.B. Navathe, S. Chakravarthy, and K. Karpalem. ER-R: An Enhanced ER Model with Situation-Action Rules to Capture Application Semantics. In T.J. Teorey, editor, *Proc. of the 10th Intl. Conf. on the Entity Relationship Approach*, pages 59–75, 1991.

[VGH93] G. Vossen and M. Groß-Hardt. *Grundlagen der Transaktionsverarbeitung*. Addison-Wesley, 1993.

[VS89] A. Verrijn-Stuart. Some Reflections on the Namur Conference on Information System Concepts. In E.D. Falkenberg and P. Lindgreen, editors, *Proc. of the IFIP TC 8/WG 8.1 Working Conf. on Information System Concepts: An In-depth Analysis*, pages ix–x. North-Holland, October 1989.

[WBT92] D.L. Wells, J.A. Blakeley, and C.W. Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10), 1992.

[WC96a] J. Widom and S. Ceri, editors. *Active Database Systems—Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann Publishers, 1996.

[WC96b] J. Widom and S. Ceri. *Introduction to Active Database Systems*, chapter 1, pages 1–41. In Widom and Ceri [WC96a], 1996.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In Lohman et al. [LSC91].

[WF90] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In H. Garcia-Molina and H.V. Jagadish, editors, *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, volume 19, June 1990.

[Wid92] Jennifer Widom. Active Database Rule Systems. Tutorial, 3rd International Conference on Extending Data Base Technologie, Vienna, Austria, March 1992.

[Wid96] J. Widom. *The Starburst Rule System*, chapter 4. In Widom and Ceri [WC96a], 1996.

Curriculum Vitae

Personal Record

Name:	Mag. Werner Obermair
Date of Birth:	24 June 1971
Place of Birth:	Schwaz
Citizenship:	Austria
Home Address:	A-4020 Linz, Weingartshofstraße 34/3

University Education

1990–95: Study of Business Information Systems at the Johannes Kepler University of Linz, Austria

1995–98: Ph.D. study at the Johannes Kepler University of Linz, Austria

Professional Experience

1991–94: Tutor for courses in programming languages at the Johannes Kepler University of Linz

1995–97: Teaching in object-oriented data modeling and object-oriented database systems at the Johannes Kepler University of Linz

1995–97: Research at the Department of Business Information Systems in project "Active Object-Oriented Database Design," funded by the Austrian Science Foundation (FWF) under contract P10320-TEC, headed by Prof. Michael Schrefl

1997 onwards: Teaching and research associate ("Universitätsassistent") at the Department of Business Information Systems, Data & Knowledge Engineering, at the Johannes Kepler University of Linz, headed by Prof. Michael Schrefl