Conceptual Design of Active Object-Oriented Databases

DISSERTATION
zur Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

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: **Dipl.-Ing. Peter Lang**

Linz, im September 1997

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 5. September 1997

Danksagung

An erster Stelle möchte ich mich bei Prof. Michael Schrefl bedanken, daß er mir im Rahmen eines Projektes die Möglichkeit zu dieser Arbeit gegeben hat und in der Folge in zahlreichen Diskussionen mit vielen Anregungen zum Gelingen wesentlich beigetragen hat. Ganz besonders möchte ich mich bei meinem Kollegen Werner Obermair bedanken, daß er mit seinem Ideenreichtum und in einer konstruktiv kritischen Art immer versucht hat, die Qualität des Ergebnisses zu verbessern und mich in seiner heiteren Art durch die letzten zweieinhalb Jahre begleitet hat. Bei Peter Bichler möchte ich mich für sein kritisches Hinterfragen so mancher aufgestellter Thesen und für den darauffolgenden Gedankenaustausch bedanken. Bei Prof. Gerti Kappel bedanke ich mich für die Übernahme der Zweitbegutachtung.

Allen weiteren Kolleginnen und Kollegen am Institut für Wirtschaftsinformatik der Universität Linz (Forschungsschwerpunkt Data & Knowledge Engineering), Ingeborg, Margit, Günter, Roman und Thomas, danke ich herzlich für die gute Zusammenarbeit.

Bei Werner Kraus und Thomas Thalhammer bedanke ich mich für die intensive Arbeit an der Entwicklung des grafischen Editors, der nicht nur das Zeichnen der hierin abgebildeten Diagramme wesentlich erleichtert, sondern auch immer neues Verständnis geschaffen hat.

Besonderer Dank gebührt meinen Eltern, die mich über all die Jahre hinweg unterstützt haben. Ein sehr großes Dankeschön gilt natürlich Claudia, die für mich und meine Arbeit soviel Verständnis gezeigt hat.

Danke!

Peter Lang

Kurzfassung

Die Geschäftspolitik eines Unternehmens kann durch Geschäftsregeln mit der Struktur Ereignis-Bedingung-Aktion beschrieben werden. Derartige Regeln werden von aktiven Datenbanksystemen unterstützt und erlauben, auf bestimmte vordefinierte Situationen mit der Ausführung einer Aktion zu reagieren, falls die entsprechende Bedingung beim Eintritt des Ereignisses erfüllt ist.

Der konzeptuelle Entwurf von aktiven objektorientierten Datenbanken umfaßt die Modellierung der Objektstruktur, des passiven Objektverhaltens (Operationen, die auf Objekten ausgeführt werden können) und des aktiven Objektverhaltens (Regeln, die Objekten erlauben, selbständig auf vordefinierte Situationen mit der Ausführung einer Operation zu reagieren). Die Analyse von Fallbeispielen zeigt, daß die Modellierung von Geschäftsregeln auf konzeptueller Ebene mittels aktivem Objektverhalten 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.

Die vorliegende Arbeit stellt *Aktive Objekt/Verhaltensdiagramme* vor, eine grafische Sprache, die die gestellten Anforderungen an die Modellierung von Geschäftsregeln auf konzeptueller Ebene erfüllt. Diese Sprache erweitert Objekt/Verhaltensdiagramme um *Situations/Aktivierungsdiagramme*, eine grafische Sprache zur Spezifikation von Ereignissen, Bedingungen und Regeln.

Um die Komplexität von Geschäftsanwendungen bewältigen zu können, wurden bereits Konzepte zur Spezialisierung von Objektstruktur und von passivem Objektverhalten erarbeitet und in verschiedener Literatur vorgestellt. Die Spezialisierung von aktivem Objektverhalten (Geschäftsregeln) wurde bisher kaum berücksichtigt. Die vorliegende Arbeit erweitert die bereits für Objekt/-Verhaltensdiagramme entwickelten Spezialisierungskonzepte um Konzepte zur Spezialisierung von Situations/Aktivierungsdiagrammen.

Konsistenzregeln gewährleisten eine korrekte Anwendung von Modellierungskonzepten und unterstützen so den Modellierer bei der Entwicklung eines konsistenten Datenbankschemas. Trotz Einhaltens aller Konsistenzregeln können verschiedene Modellierer verschiedene konsistente Datenbankschemata erzeugen, die sich in ihrer Qualität unterscheiden. Diese Arbeit führt Richtlinien für den Entwurf von aktivem Verhalten in aktiven objektorientierten Datenbanken an, welche dem Modellierer helfen, sowohl ein konsistentes als auch ein qualitativ hochwertiges, d.h. ein verständliches und wartbares Datenbankschema zu erzeugen.

Abstract

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.

The conceptual design of active object-oriented databases includes modeling the structure of objects, their passive behavior (the operations that can be peformed on objects), and their active behavior (rules that allow objects to react autonomously to predefined situations by performing an operation). Modeling business rules at the conceptual level by means of the active behavior of objects requires different concepts than currently provided by active object-oriented database systems and by recent approaches to active object-oriented database design.

This thesis presents a high-level graphical language called *Active Object/Behavior Diagrams*, which meets the requirements identified for modeling business rules at the conceptual level. Active Object/Behavior Diagrams extend the already introduced Object/Behavior Diagrams with *Situation/Activation Diagrams*, a graphical rule and event language.

In order to cope with the complex domain of business applications, concepts for specializing the object structure and the passive behavior have been elaborated and presented in the diverse literature. Specializing active behavior (business rules) has received little attention so far. This thesis extends the specialization concepts already defined for Object/Behavior Diagrams with the concepts for specializing Situation/Activation Diagrams.

The usage of modeling concepts is constrained by a set of consistency rules. They have the designer stick to a correct usage of the modeling concepts in order to achieve a consistent database schema. Even if all consistency rules are obeyed, different designers may yield different consistent database schemata, which may differ in their quality. This thesis presents guidelines for the conceptual design of active behavior in active object-oriented databases. These design guidelines should help the designer to achieve a consistent as well as a high-quality database schema, i.e., a comprehensible and maintainable database schema.

Contents

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, CKAK94, GD94, GJS92a] 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 [KS93]. (b) In many cases rules and methods are alternative ways to express the same effect [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 [KD93], further by providing simulation, testing, and debugging tools [KD93] and by using abstraction techniques like rule stratification [BCP96], which enables the partitioning of complex rule sets into modules of rules with comprehensible and predictable behavior, or rule patterns [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 the high-level, graphical active object-oriented design method *Active Object/Behavior Diagrams*. Our approach extends *Object/Behavior Diagrams* [KS91a], a graphical object-oriented design method 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].

A conceptual database schema developed by means of Active Object/Behavior Diagrams may be translated automatically into a logical database schema of some existing active object-oriented database system. Obermair [Obe97] gives a detailed description of such a mapping to the object-oriented database system GemStone (GemStone Systems, Inc.) combined with the active component TriGS ("Trigger system for GemStone") [KRSRV94]. The resulting logical database schema may contain a large set of rules. But these rules have to be maintained only by the active database system and not by the designer because they are generated automatically.

Specialization. Object-oriented systems define inheritance as the basic concept indicating specialization. Kappel, Schrefl, and Stumptner [Sch90, KS94, SS95, KS96b, SS97] have already discussed the specialization of object structure and passive behavior for conceptual database schemata modeled by means of Object/Behavior Diagrams. The specialization of active behavior has received little attention so far. Most object-oriented design methods like OMT [RBP+91] define the inheritance of object behavior by saying how the inherited methods may be overridden at subtypes. The inheritance of active behavior is usually defined by saying how the inherited active behavior may be overridden at subtypes [SC96]. Specializing object structure, passive behavior, and active behavior does not only mean to override (redefine) these features at subtypes but also means to extend these features at subtypes by adding new ones and to refine already existing features by newly introduced features. Inspired by the approach of consistent specialization of passive behavior [SS95, SS97], we present rules that ensure the consistent specialization of active behavior.

Design Guidelines. Insufficient methodological support in analysis and design is encountered by Simon and Kotz-Dittrich [SKD95] as one of the main problems of using active database technology. Active Object/Behavior Diagrams try to fill this need by providing a homogenous set of graphical modeling concepts for conceptual active object-oriented database design and associated design guidelines.

Using the modeling concepts such as provided by Active Object/Behavior Diagrams is restricted by a set of consistency rules. Consistency rules are either guaranteed by the modeling concepts or have to be obeyed by the designer. They have the designer stick to a correct usage in order to achieve a consistent database schema. In addition to consistency rules, design guidelines help to achieve a "high-quality" conceptual database schema, i.e., a comprehensible and maintainable conceptual database schema.

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 the basic concepts of Active Object/Behavior Diagrams. Chapters 7, 8, and 9 deal with extended aspects of active object-oriented database design, they treat the specialization of Active Object/Behavior Diagrams and establish design guidelines. 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 can serve as a 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 revises some of the concepts of Active Object/Behavior Diagrams as described in Chapters 4, 5, and 6 in order to enable a consistent specialization of passive and active behavior as presented in the successive Chapter 8.
- Chapter 8 presents a complete overview of specializing object structure, passive behavior, and active behavior in Active Object/Behavior Diagrams based on the revisions introduced in Chapter 7.
- Chapter 9 establishes guidelines for designing the active behavior of active object-oriented databases. The design guidelines should help to improve the quality of the conceptual database schema.
- Chapter 10 summarizes the main contributions of Active Object/Behavior Diagrams, shortly describes a tool prototype that supports developing a conceptual schema by means of 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 conceptual design, an accompanying thesis, written by Werner Obermair [Obe97], treats the logical design of active object-oriented databases. The core concepts of a conceptual model for active object-oriented database design 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

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, BM91]) 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.

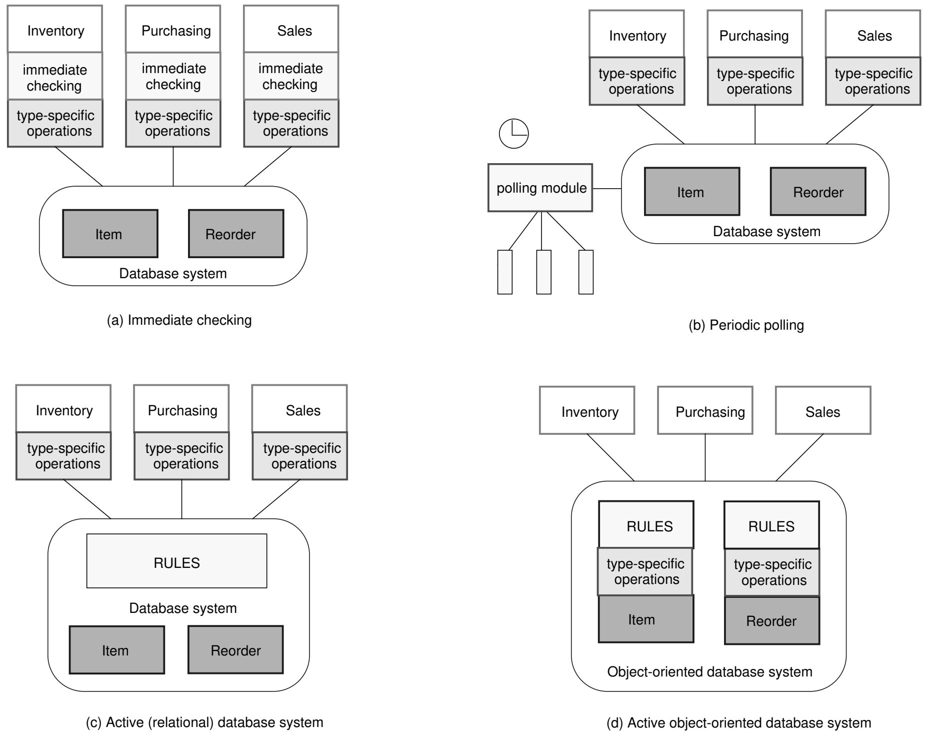
The schema of an object-oriented database consists of a set of object classes. An object class contains the description of the data structure used to model real-world entities of that class together with the description of the associated operations, also called methods. The data of one real-world entity 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]):



(a) Immediate checking
(b) Periodic polling
(c) Active (relational) database system
(d) Active object-oriented database system

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 ([HKRSR97]), 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, HKRSR97] 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 explained. 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 $E = (F \bigtriangleup G)$, occurs when one of both events occurs and the other occurs, too, or has already occurred in the past; i.e., $F \bigtriangleup G = (F \wedge G) \vee (F,_{prior}\, G) \vee (G,_{prior}\, F)$.
  - A *choosing n sequence* E of an event F, denoted $E = (\mathit{choose}\ n\ F)$, 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. [HKRSR97]).

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, CM93, CKAK93, CKTB94, CKAK94].

- *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. Details on TriGS can be found in [KRSRV94, Hie96, Ret96, HKRSR97].
- *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

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-

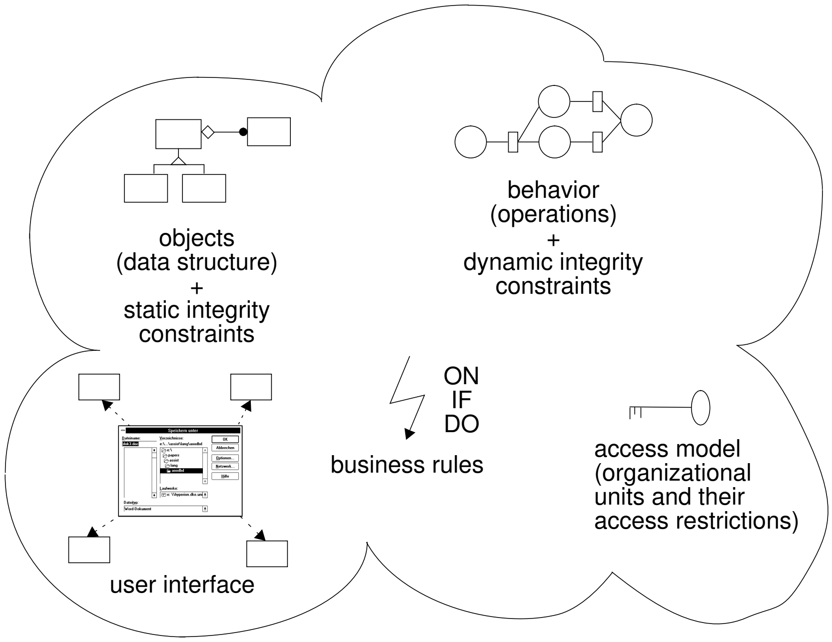


Figure 3.1: Elements of the business area

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, [Bic97]).

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.

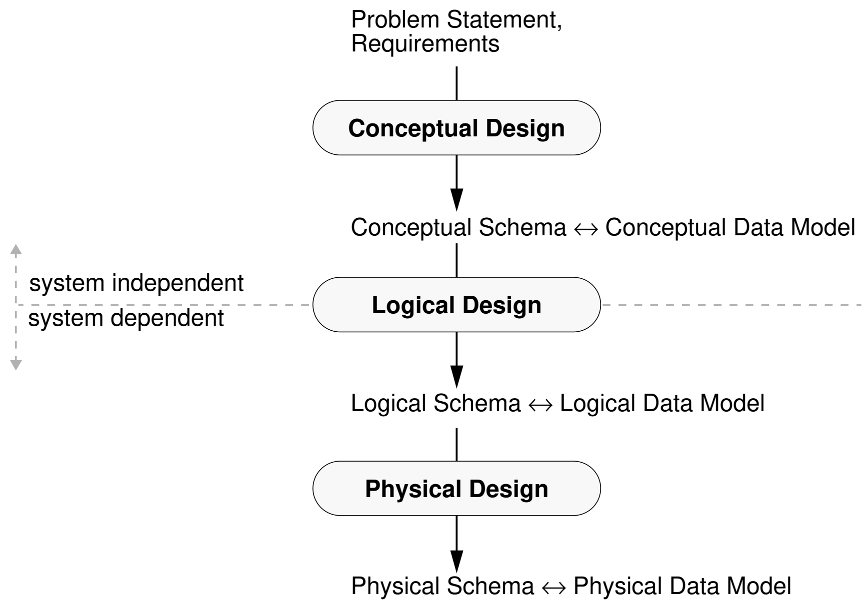


Figure 3.2: Database design process

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

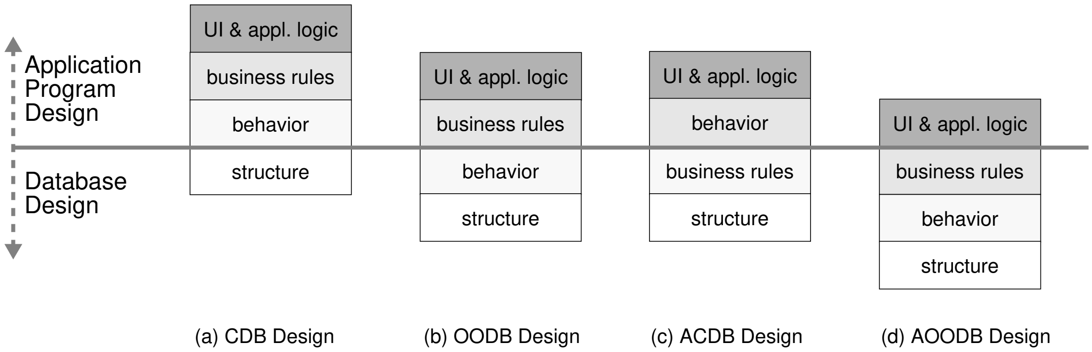


Figure 3.3: Where the elements of the business area are reflected in information-system design

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 bi-temporal 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 especially in Chapter 9.

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 $(\mathrm{ER})^2$, 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 Chapter 8.

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]. Chapter 9 will discuss guidelines for active object-oriented database design in detail.

- 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], a trigger system for the object-oriented database management system GemStone (GemStone Systems, Inc.). 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, rules cannot be attached to specific object types even if they model 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

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.

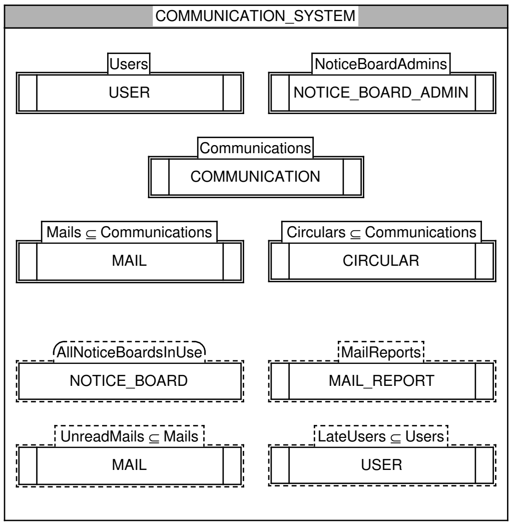


Figure 4.1: The global object type COMMUNICATION_SYSTEM

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

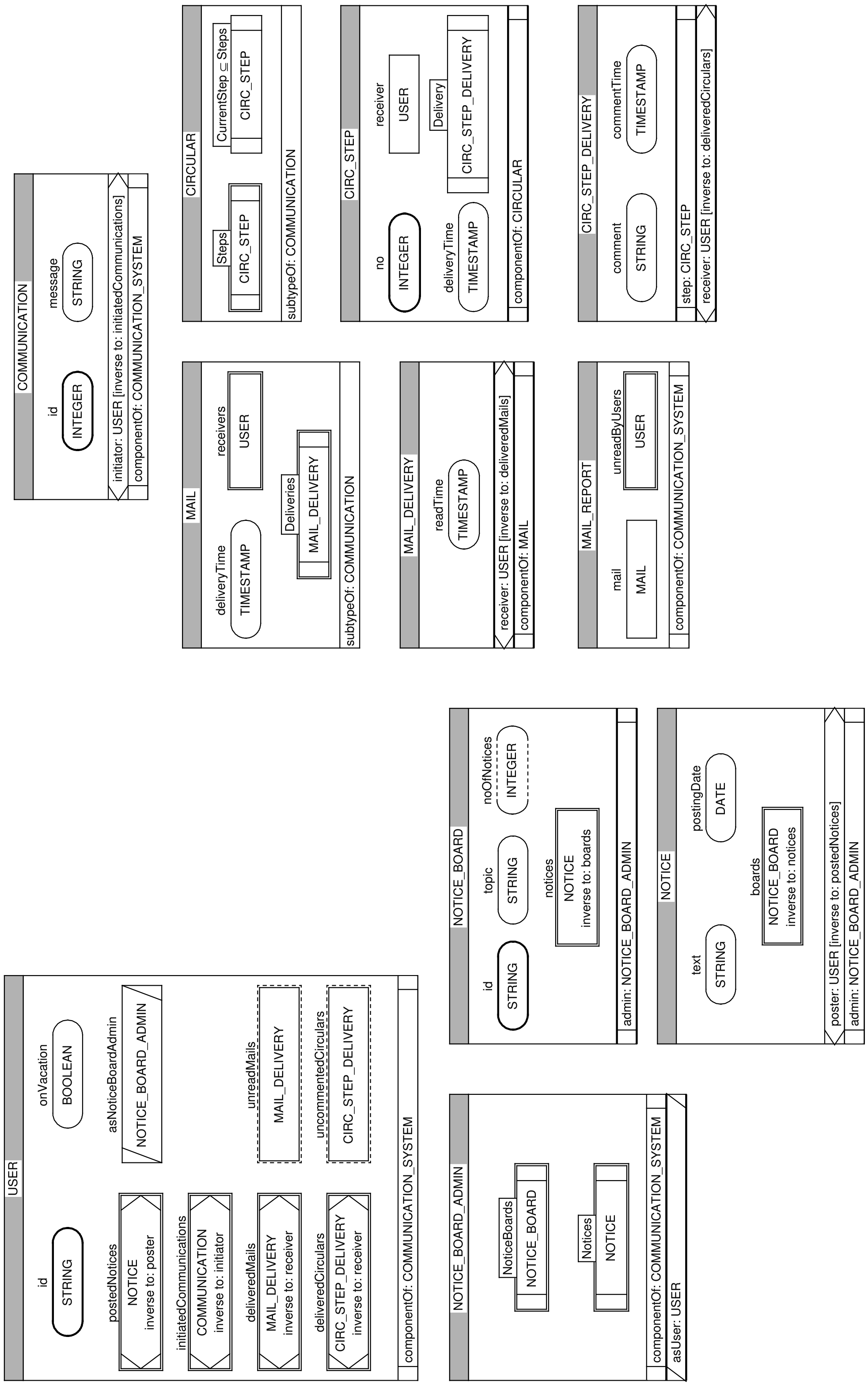


Figure 4.2: Object types representing users and communications

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

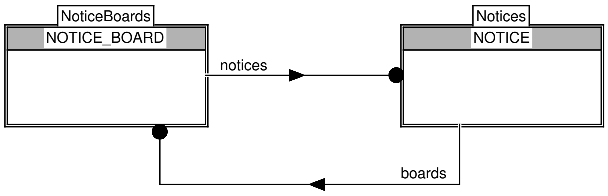


Figure 4.3: Object-class diagram of object type NOTICE_BOARD_ADMIN

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 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

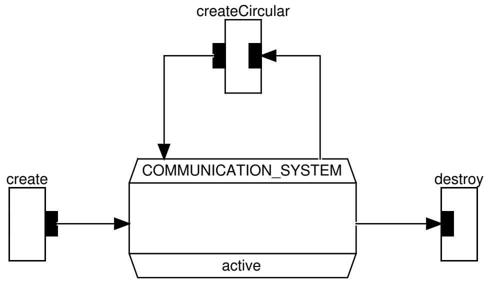


Figure 4.4: Life-cycle diagram of object type COMMUNICATION_SYSTEM (selection)

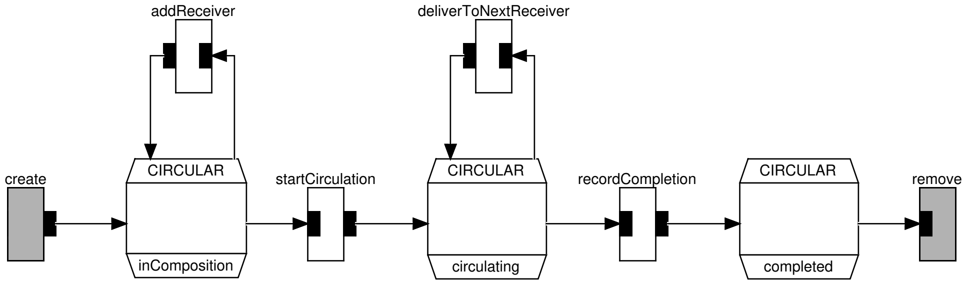


Figure 4.5: Life-cycle diagram of object type CIRCULAR

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 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

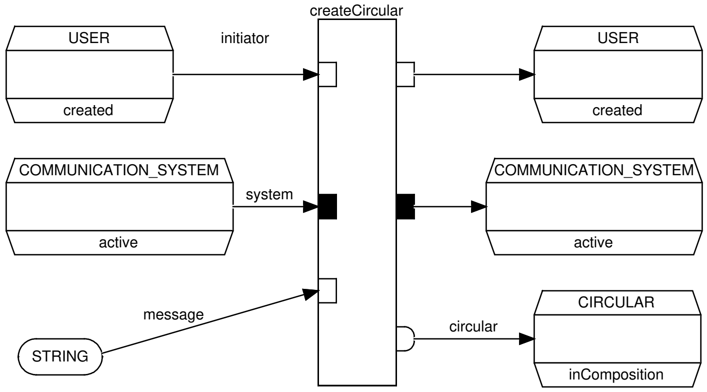


Figure 4.6: Specification of activity createCircular

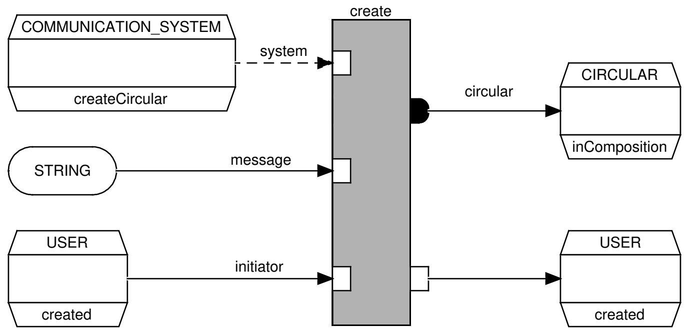


Figure 4.7: Specification of activity create

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,

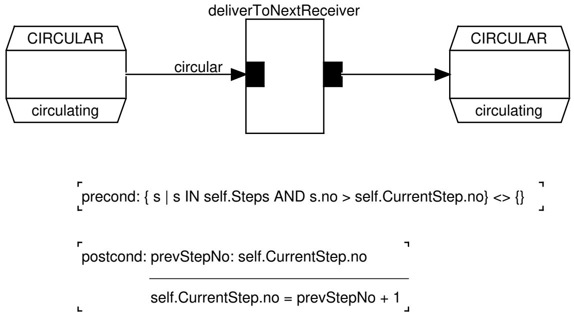


Figure 4.8: Specification of activity deliverToNextReceiver

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.

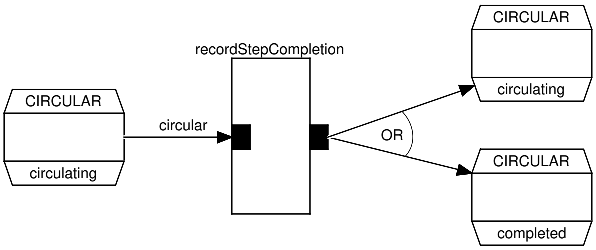


Figure 4.9: Specification of compound activity recordCompletionOfStep

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.

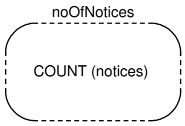


Figure 4.10: Query diagram for property noOfNotices

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

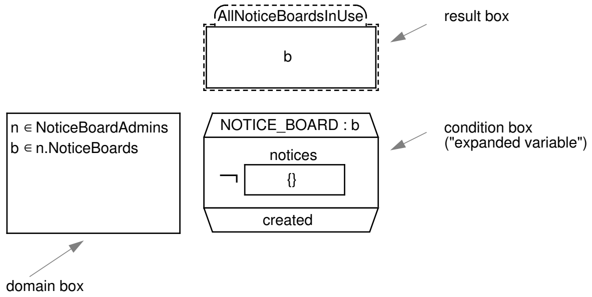


Figure 4.11: Query diagram for property AllNoticeBoardsInUse

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

$x_1 \in c_1$ $x_2 \in c_2(x_1)$ ... $x_n \in c_n(x_1,x_2,...,x_{n-1})$

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 multivalued.

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-

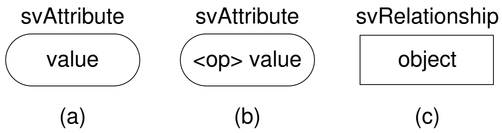


Figure 4.14: Scalar-comparison predicates

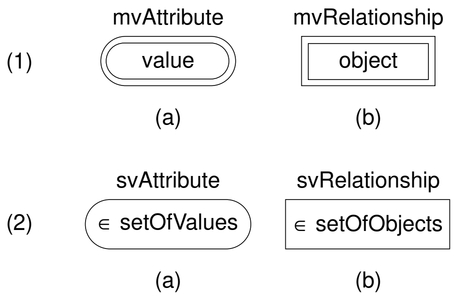


Figure 4.15: Scalar/set-comparison predicates

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 $\neg$ to the left of the property.

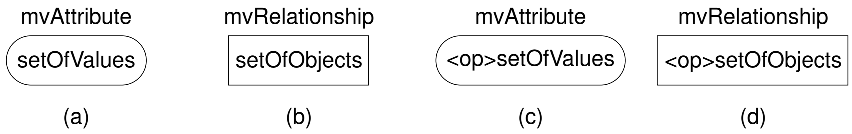


Figure 4.16: Set-comparison predicates

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

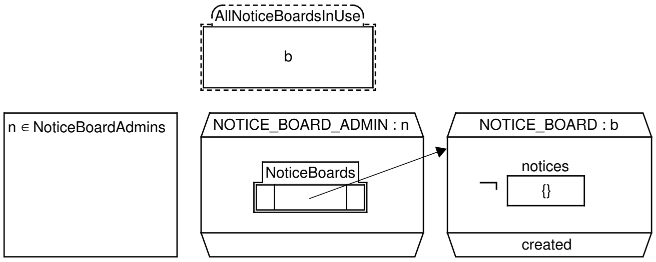


Figure 4.17: Query diagram for property AllNoticeBoardsInUse

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

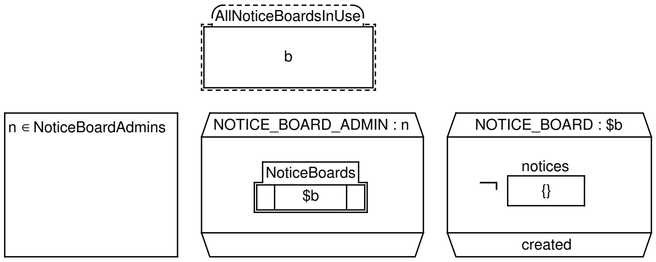


Figure 4.18: Query diagram for property AllNoticeBoardsInUse

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

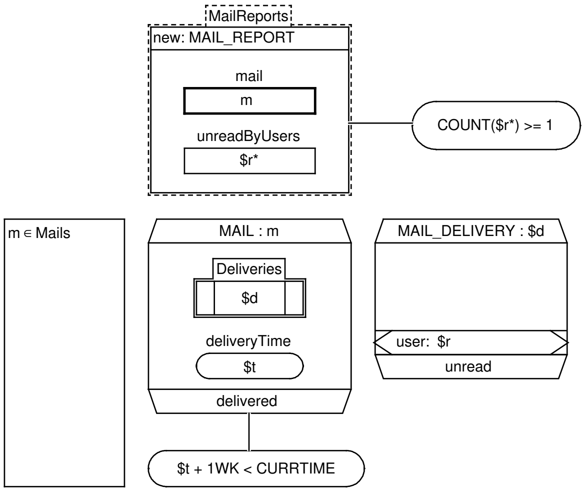


Figure 4.19: Query diagram for property MailReports

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.

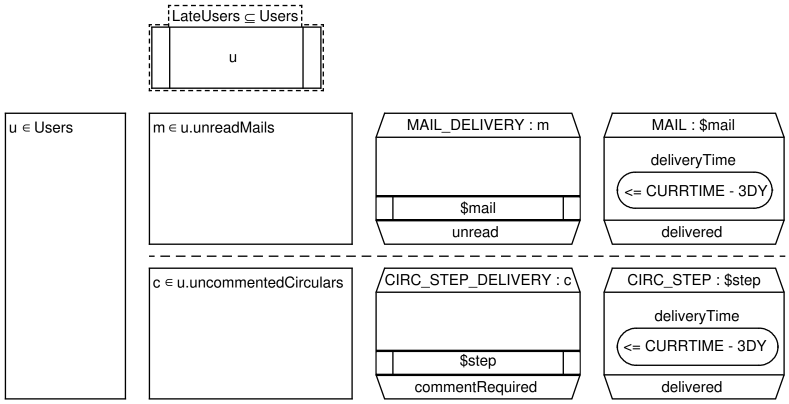


Figure 4.20: Query diagram for property LateUsers

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.

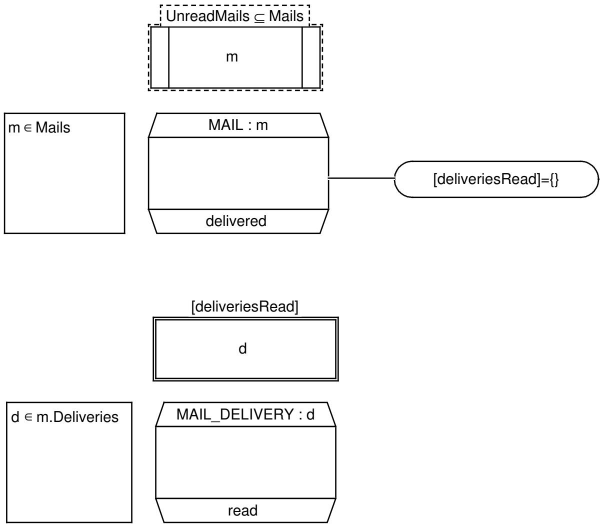


Figure 4.21: Query diagram for property UnreadMails

Chapter 5

Situation Diagrams

Contents

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.

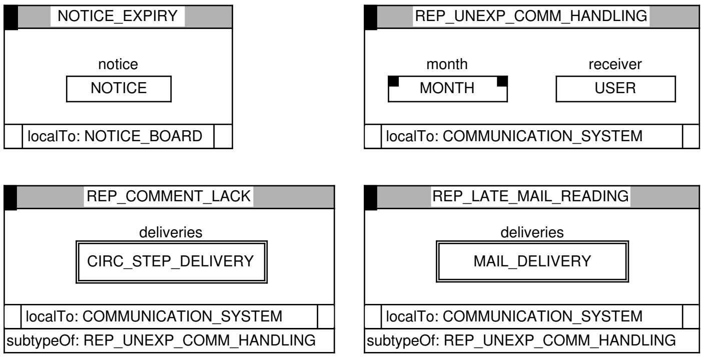


Figure 5.1: Situation-type diagrams

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 Chapter 8.

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

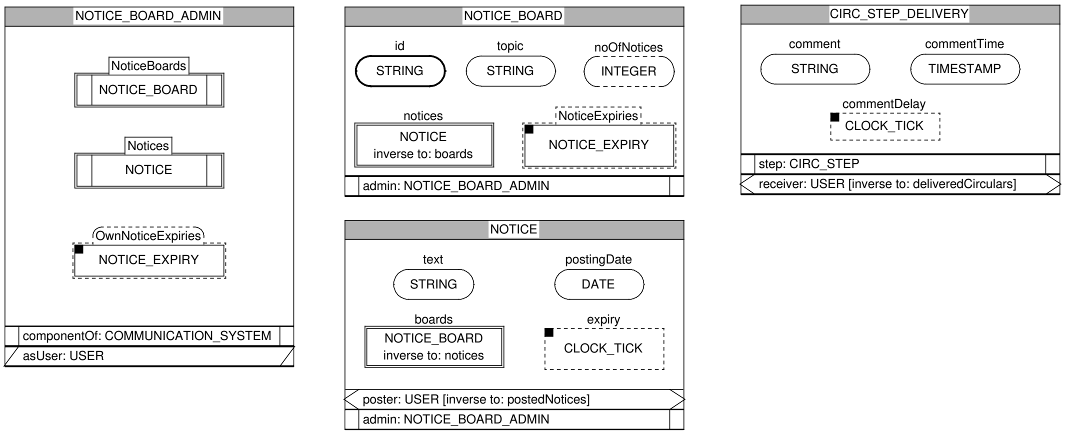


Figure 5.2: Object-type diagrams including situation classes

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-

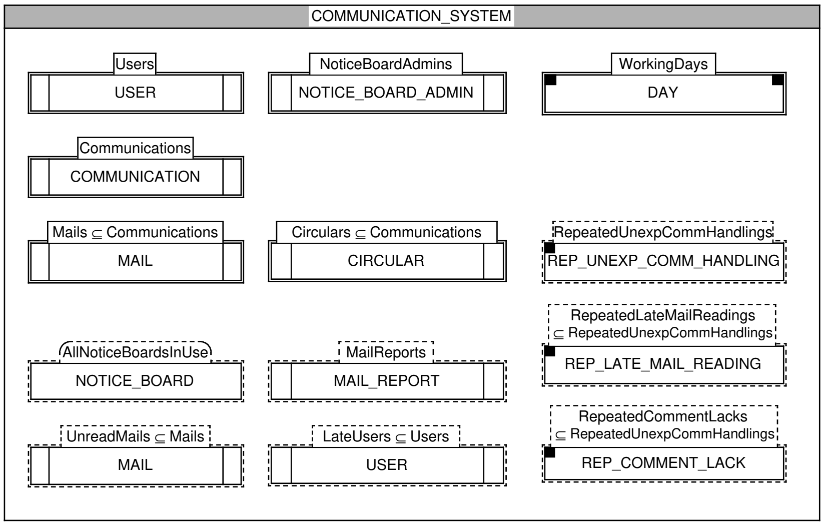


Figure 5.3: The global object type including situation classes

> 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 of 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

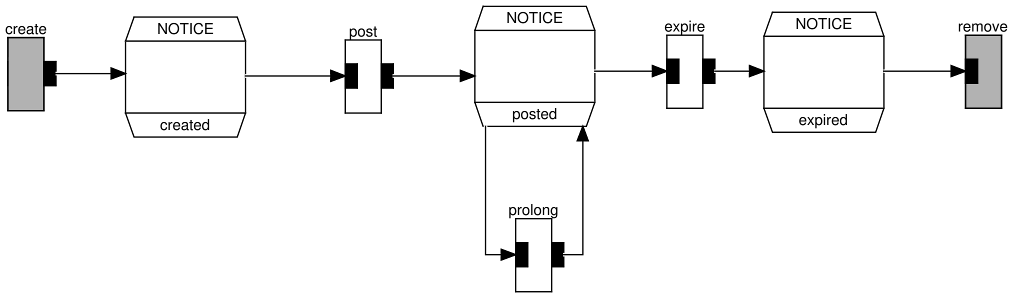


Figure 5.4: Life-cycle diagram of object type NOTICE

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 $(\langle ObjectTypeName \rangle$_STATE_$\langle StateName \rangle)$. 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).

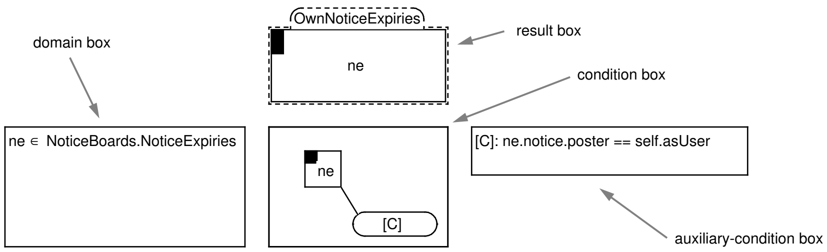


Figure 5.5: Situation diagram for situation class OwnNoticeExpiries

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)

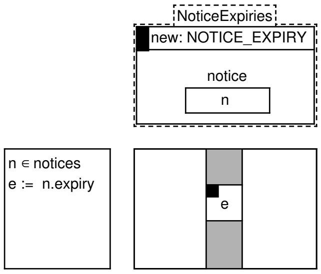


Figure 5.6: Situation diagram for situation class NoticeExpiries

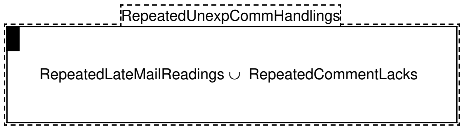


Figure 5.7: Situation diagram of situation class RepeatedUnexpCommHandlings

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

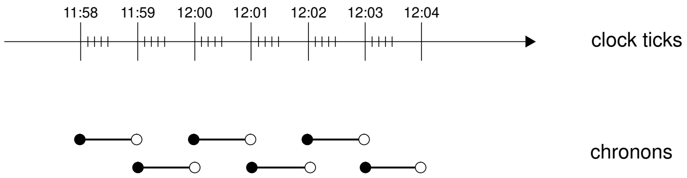


Figure 5.8: Clock ticks and chronons

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,

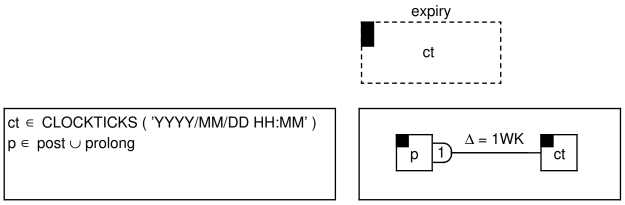


Figure 5.9: Situation diagram for situation class expiry

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.

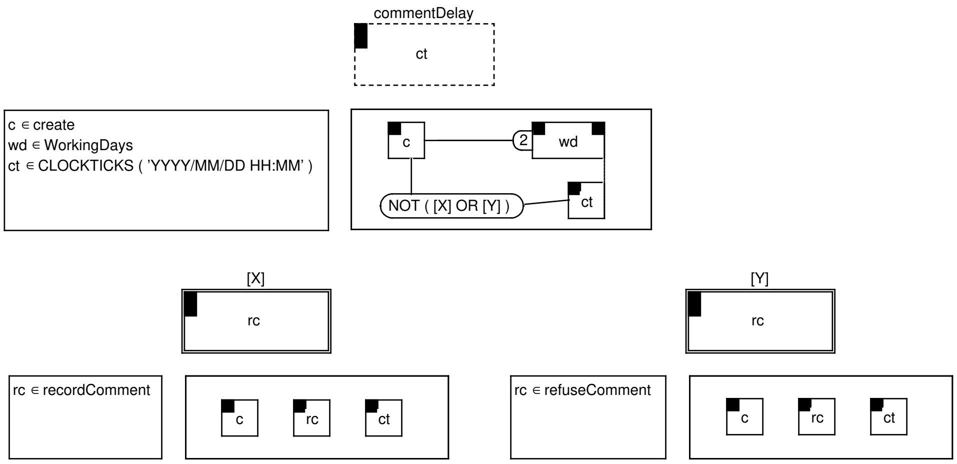


Figure 5.10: Situation diagram for situation class commentDelay

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

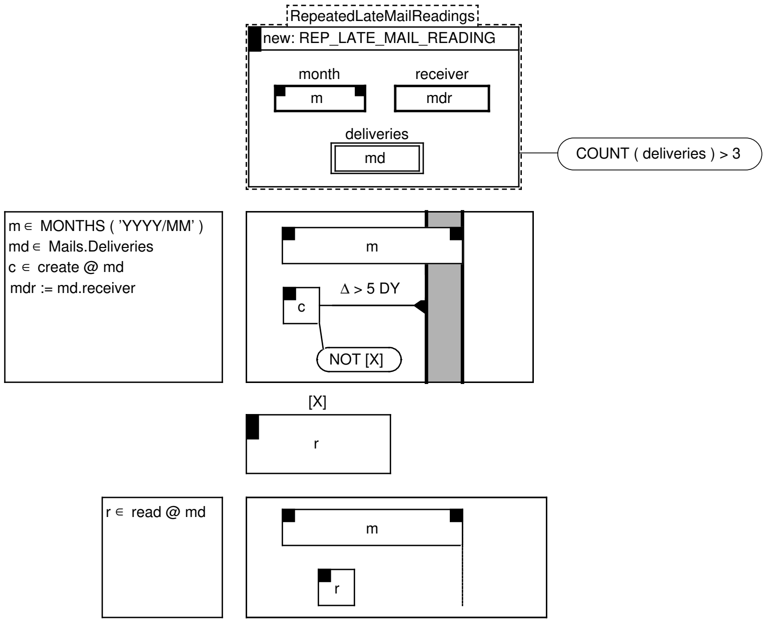


Figure 5.11: Situation diagram for situation class RepeatedLateMailReadings

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

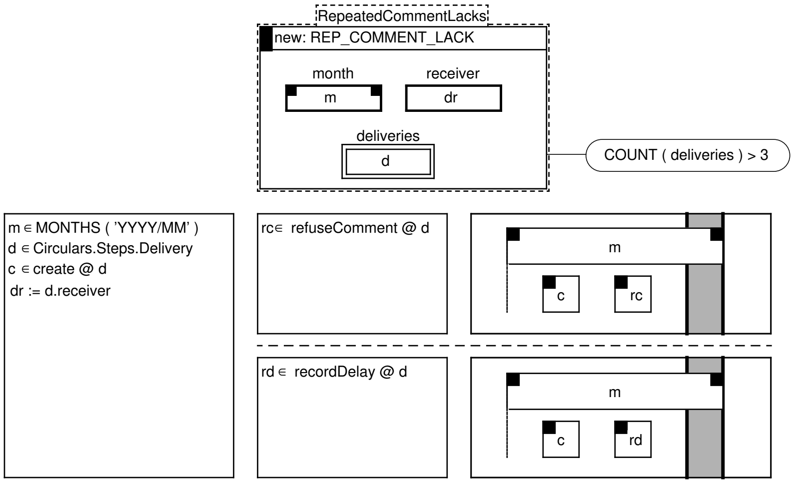


Figure 5.12: Situation diagram for situation class RepeatedCommentLacks

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 RepeatedComment-Lacks 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).

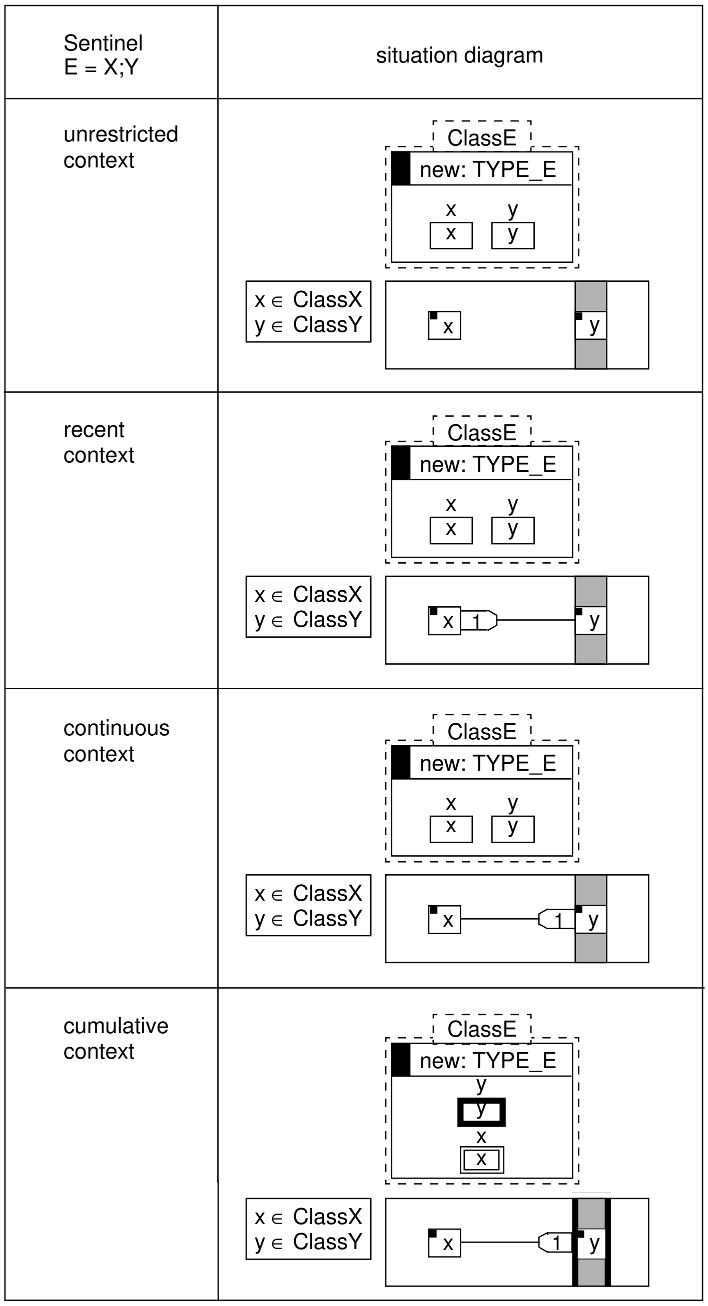


Figure 5.13: The declarative semantics of Sentinel's sequence operator

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 (see [Obe97]).

Chapter 6

Activation Diagrams

Contents

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. [Obe97]).

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.

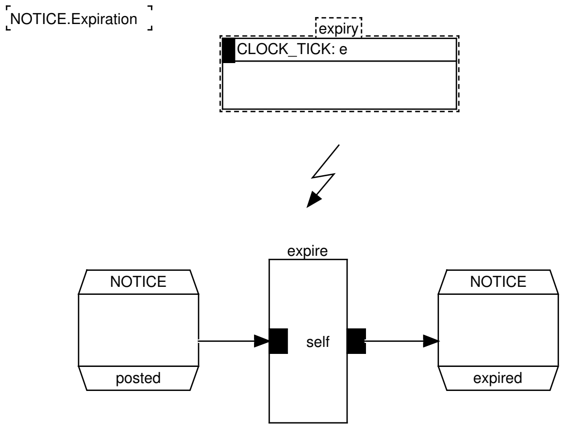


Figure 6.1: Rule Expiration of object type NOTICE

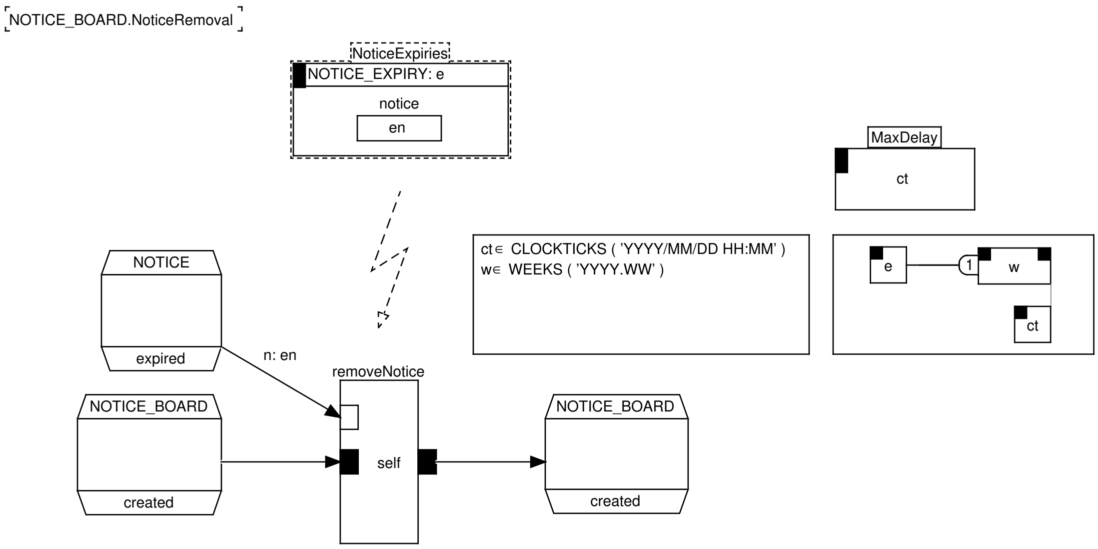


Figure 6.2: Rule NoticeRemoval of object type NOTICE_BOARD

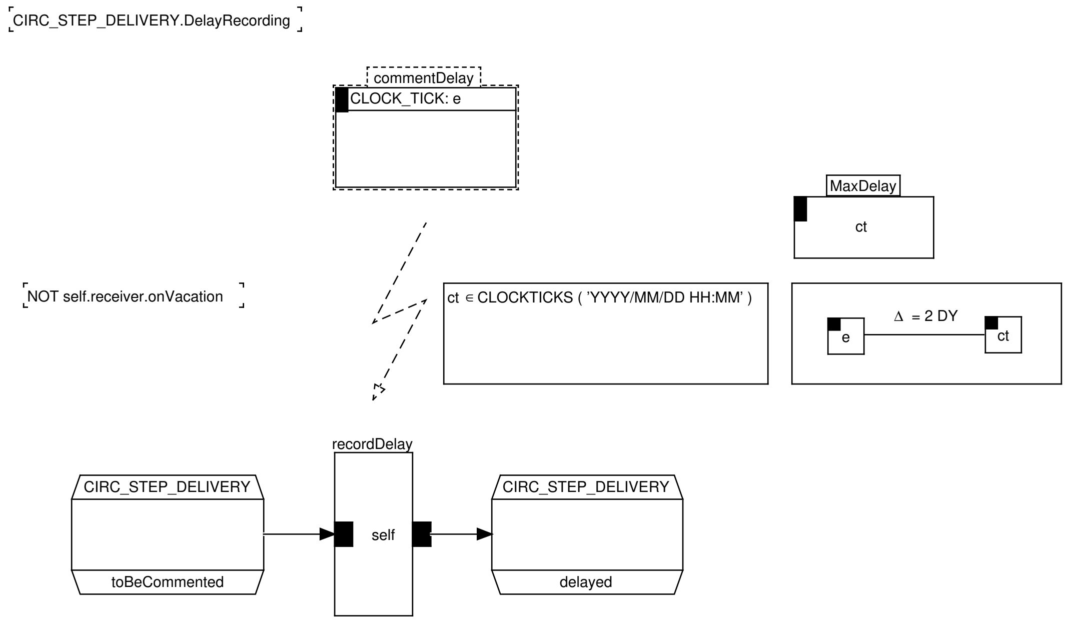


Figure 6.3: Rule DelayRecording of object type CIRC_STEP_DELIVERY

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 $S_{(r,e)}$-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.

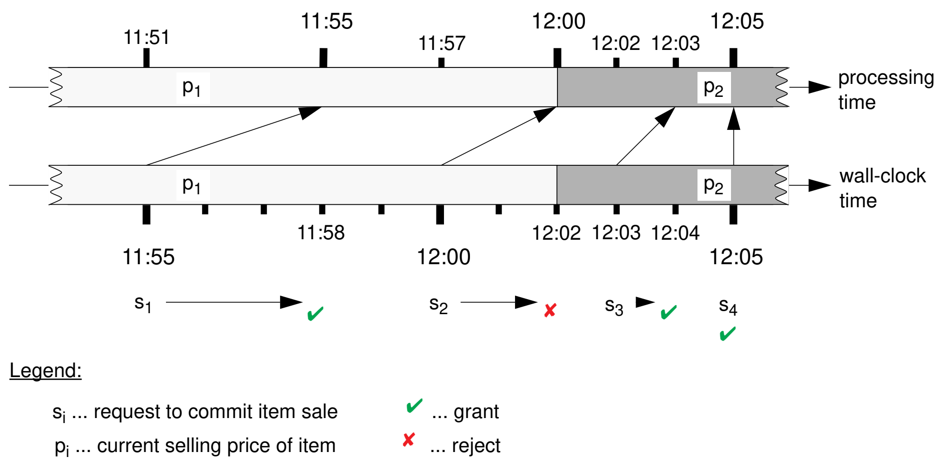


Figure 6.4: Schematic representation of temporally faithful execution

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 [Obe97] 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

Active Object/Behavior Diagrams Revisited

Contents

Active Object/Behavior Diagrams introduced so far provide excellent technical concepts for modeling business rules. However, business experts are usually more familiar with the notions permission, prohibition, and obligation than with the concepts precondition, situation, and rule. Permission, prohibition, and obligation represent the core concepts in a branch of philosophical logic, called deontic logic. Deontic logic is a modal logic that deals with the actual as well as the ideal behavior of systems. As active object-oriented database design allows to specify the behavior of objects, which describe a system at the conceptual level or are part of such a system, the notions permission, prohibition, and obligation may also be used in Active Object/Behavior Diagrams.

The objective of this chapter is to incorporate permissions, prohibitions, and obligations into Active Object/Behavior Diagrams in order to make the specification of object dynamics more intelligible and also to prepare the specialization of object behavior as discussed in detail in Chapter 8.

The remainder of this chapter is structured as follows. Section 7.1 introduces the field of deontic logic and restricts the interest on deontic logic to permissions, prohibitions, and obligations used for specifying integrity constraints. Section 7.2 picks up the ideas about specifying integrity constraints with deontic logic and shows how permissions, prohibitions, and obligations are adopted for Active Object/Behavior Diagrams. It is important to mention in advance that the deontic concepts that are going to be used in Active Object/Behavior Diagrams do *not* match exactly the concepts provided by deontic logic.

7.1 Permission, Prohibition, and Obligation in Deontic Logic

Deontic logic is a branch of philosophical logic concerning reasoning about norms and represents the logical foundations for ethics and law. The real world defines a lot of norms, and so does the business area, which is implemented in an information system. In the following, we present the general ideas of deontic logic and discuss those parts of deontic logic which are applicable to active object-oriented database design, particularly to Active Object/Behavior Diagrams.

The remainder of this section is organized as follows. Section 7.1.1 gives a short overview of the applications where deontic logic is used and restricts further interest on deontic logic to its impact on information systems design, particularly on database design. Section 7.1.2 focuses on the specification of integrity constraints, the area for which deontic concepts will be used in database design. Section 7.1.3 presents the basic ideas of deontic logic in an informal way.

7.1.1 Applications of Deontic Logic

Before we discuss the use of deontic logic in database design, we should have a short look on the many applications of deontic logic in computer science, which are categorized in [WM93] as follows.

1. *Fault-tolerant systems*: The application of deontic logic may be considered as the specification of exception-handling, where the exceptions are generated by the computer system.

2. *Normative user behavior*: Users behave in all sorts of ways, which are not ideal. The behavior that is desired from the user and the behavior he or she may actually engage in has to be distinguished. This application of deontic logic may be considered as the specification of error-handling.

3. *Normative behavior in or of an organization*: The behavior of emloyees in an organization may be specified by policies whose specification is based on deontic logic. Furthermore, deontic logic can be used for formulating security policies, a special case of organizational policies. Deontic logic can be used for specifying also the behavior of an organization as a whole.

4. *Normative behavior of an object system*: An object system itself is a part of the real world and its objects model a part of the real world. The behavior of the objects contained in the object system can be described using deontic logic. There are various applications of deontic logic in this area, like the specification of law, the simulation of legal thinking, the specification of deontic integrity constraints, and other applications, for example, the specification of process scheduling.

All applications of deontic logic involve some sort of specification of norms and actual behavior in deontic logic. Wieringa and Meyer [WM93] point out that specifying the behavior of a system in deontic logic has to be distinguished from programming a computer in deontic logic. In most cases, the deontic specification of a system's behavior can be useful without being implemented in a computer. And if it is implemented, it need not be implemented using a deontic programming language.

Active object-oriented database design models a part of the real world in an active object-oriented database schema in terms of object structure and (active and passive) object behavior. Deontic logic can be used for describing the normative behavior of objects, i.e., the policies defined in the business area that represents the part of the real world to be modeled in the database.

In general, the behavior of objects is specified by the operations (activities) that may be executed on top of them. Dynamic integrity constraints specify under which circumstances operations may be or have to be executed. This is the place where deontic ideas are considered later on: Deontic concepts like permission, prohibition, and obligation can be used for specifying deontic integrity constraints in addition to conventional integrity constraints in database schemas.

7.1.2 Specifying Integrity Constraints

Specifying integrity constraints is a core task in database design. In Chapter 3, we distinguished between static integrity constraints and dynamic integrity constraints . Wieringa et al. [WMW89] distinguish the two types of integrity constraints according to a different aspect: necessary constraints and deontic constraints. *Necessary constraints* are formulas that are always true in the real world because they cannot be violated there. For example, the age of a person cannot be lower than zero. It cannot be violated in the real world because this is an analytical truth. In addition, a person's age may be constrained in the database to be at most 150. Such a constraint is based on empirical truth and not on analytical truth. It is empirically true that persons cannot get older than 150 years. As necessary contraints do not constrain the real world but only the database, they should be called simply "database constraints."

Furthermore, Wieringa et al. argue that most of the constraints published in the database literature—they are called "database constraints" there, too—are not necessary truths in the sense explained above, but are instead normative statements that apply to the real world and that can be violated in the real world. An example given by Wieringa et al. is that a salary should not decrease. Violating such a constraint in the world is not a falsification of empirical truth but represents a violation of a real-world norm. Thus, this is a truly constraint on the real world and not a constraint on a database of facts about the real world. Such constraints are called *deontic constraints*. As deontic constraints constrain the real world, they should be called simply "real-world constraints."

Active object-oriented database design is concerned with both necessary constraints and deontic constraints. Before we discuss the impact of these constraints on database design, we present some basic aspects of deontic logic.

7.1.3 The Idea of Deontic Logic

As already mentioned, deontic logic is the logic that deals with the ideal and the actual behavior of systems. For this, the operators permission, prohibition, and obligation are introduced. Usually, a system's actual behavior differs from its ideal behavior because norms specified with the ideal behavior are violated while the system is in use. Specifying the actual and the ideal behavior requires different description techniques.

The following definitions will be presented in an informal way. The objective of this section is to show only the ideas behind deontic logic and is not to define a formal framework for further work based on deontic logic. Meyer and Wieringa [MW93a] give a concise overview of deontic logic and certain actual research directions. Åqvist [Aqv86], for instance, presents deontic logic in more detail.

Wieringa et al. [WWMD91] distinguish two types of dynamic modalities: objective modalities and deontic modalities. *Objective modalities* state concern what objectively can happen. Human beings, their access rights or execution obligations are not considered at all. Following objective modalities are defined for an acitivity α:

1. α can possibly happen.

2. α necessarily happens.

3. α is discretionary, α may not happen.

4. α can impossibly happen.

Objective modalities can help to represent necessary constraints, which are never—at least in all probability in case of contraints based on empirical truth—violated in the real world and therefore must not be violated in the database. Several equivalances can be given for objective modalities:

1. Iff α necessarily happens, not executing α can impossibly happen.

2. Iff α is discretionary (may not happen), not executing α can possibly happen.

3. Iff α can impossibly happen, executing α would lead to an inconsistent database state.

Deontic modalities state concern what is admissible. They specify norms of the real world, which in turn may be violated. Thus, the database has to define states, so-called violation states, and possible or admissible reactions to the entrance of such a state, i.e., reactions to the violation of a norm. Following deontic modalities are defined for an activity α:

1. α is permitted.

2. α is obligatory.

3. α is discretionary, not obligatory.

4. α is forbidden (α is not permitted).

Deontic modalities can help to represent deontic constraints, which may be violated in the real world and thus may be violated also in the database. Since reactions to violations of norms are specified in the real world, such reactions have to be specified also in the database. Similar to objective modalities, following equivalances can be given for deontic modalities:

1. Iff α is obligatory, not executing α is not permitted.

2. Iff α is discretionary, not executing α is permitted.

3. Iff α is forbidden, executing α leaves the database in a violation state, but not in an inconsistent state!

Objective modalities and deontic modalities cannot be specified completely independently from each other. They are tightly interrelated. Wieringa et al. present some very interesting implications between them:

1. If α is permitted, α can possibly happen.
2. If α necessarily happens, α is obligatory.
3. If α impossibly happens, α is forbidden.
4. If α is discretionary (not obligatory), α may not happen.

These ideas constitute the base for extending active object-oriented data models with permissions, prohibitions, and obligations as shown below. It is important to mention here that the deontic modalities like permission and prohibition as well as obligation and discretion are usually defined to be complementary in deontic logic. This means that performing an activity is either permitted or forbidden. Deontic logic usually disallows to specify an activity that is neither permitted nor forbidden. The property of deontic logic requiring permission and prohibition to be complementary will be relaxed when deontic ideas are adopted for database design.

7.2 Adopting Deontic Ideas for Database Design

Inspired by the ideas of deontic logic, we introduce the notions permission, prohibition, and obligation for active object-oriented database design in order to allow the database designer to model the real world in a more natural way. Specifying the dynamics of objects in terms of permissions, prohibitions, and obligations comes much closer to the real world than specifying the dynamics in terms of conventional conditions (e.g., necessary preconditions). *But notice that the deontic concepts that are going to be used in database design do not correspond exactly to the concepts used in deontic logic and to what has been introduced in Section 7.1.3.* The differences will be explained at the end of this section.

The remainder of this section is structured as follows. The state-of-the-art in modeling the passive and active behavior of objects is revisited in Section 7.2.1. The changes that are applied to Active Object/Behavior Diagrams to incorporate permissions, prohibitions, and obligations are presented in Section 7.2.2. Consistency rules that restrict the use of the introduced concepts are defined in Section 7.2.3. As the introduced concepts do not match exactly with the deontic concepts described above, last but not least, the differences between the notions permission, prohibition, and obligation used in database design and the same-named concepts in deontic logic are discussed in Section 7.2.5.

7.2.1 The State-of-the-Art in Modeling Dynamics

Currently, an object's dynamics is specified by its passive behavior and by its active behavior. An object's passive behavior is specified in terms of an object life cycle. In Object/Behavior Diagrams the object life cycle is based upon a Petri net. Its transitions are represented by activities, its places are represented by states. As opposed to Petri nets, where transitions are fired automatically if all places that are connected to a transition contain a token, an activity has to be invoked explicitly for an object that resides in all prestates. Additional *preconditions* can be given to avoid an activity's invocation in a context under which the activity may never be completed successfully. Both prestates and additional preconditions represent a *necessary precondition*. This means that satisfying the precondition is necessary to be able to invoke the activity at all but does not guarantee that it is completed successfully. A *sufficient precondition* guarantees that an activity is executed successfully if the precondition is met when the activity is invoked and therefore is much stronger than a necessary condition [WWMD91]. Currently, Active Object/Behavior Diagrams allow to specify necessary preconditions but not to specify sufficient preconditions. A *postcondition* ensures that an activity produces the desired result. Further, necessary and sufficient postconditions could be distinguished, but only necessary postconditions are of interest [WWMD91]. Preconditions and postconditions of activities are referred to as *dynamic integrity constraints*.

An object's active behavior is currently specified by means of Situation/Activation Diagrams, a graphical representation of event-condition-action rules (ECA rules). They define that an activity has to be performed when an event occurs and if the associated condition is satisfied.

The constraints on the dynamics of objects, i.e., dynamic integrity constraints, are usually not specified in terms of necessary or sufficient preconditions. Business experts and process modelers are used to specify such constraints in form of policies. They specify the circumstances under which it is forbidden, permitted, or even obligatory to do something. Providing only concepts like necessary preconditions breaks with the traditional concepts in the business area and complicates the communication between business experts, process modelers, and database designers. Since different concepts require different thinking, they cause misunderstandings if their semantics are misinterpreted. The aim of this section is to make use of the deontic ideas to facilitate the communication between the experts in the business area and in conceptual database design and to prevent misunderstandings by providing concepts appropriate for all kinds of experts.

The specification of dynamic integrity constraints by means of preconditions and rules will be revised below to allow to specify constraints similar to deontic integrity constraints.

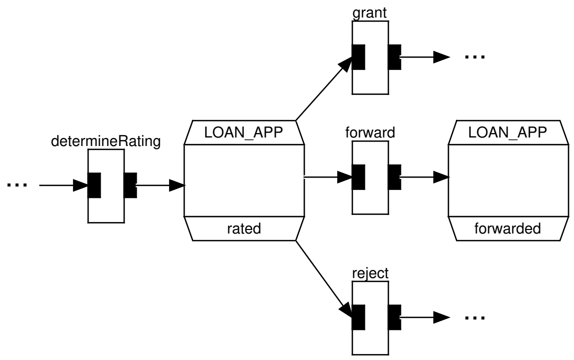


Figure 7.1: Life-cycle diagram of object type LOAN (partially shown)

7.2.2 Active Object/Behavior Diagrams with Permissions, Prohibitions, and Obligations

Incorporating the notions permission, prohibition, and obligation into Object/Behavior Diagrams has been first proposed by Kueng and Schrefl in [KS95] and by Schrefl and Bichler in [BS95]. Incorporating the notions permission, prohibition, and obligation into Active Object/Behavior Diagrams does not cause major changes. The concept activity precondition is modified only slightly.

First, we show how Active Object/Behavior Diagrams are modified and extended in order to express permissions, prohibitions, and obligations. Then, we describe how our approach of modeling permissions and prohibitions that are not complementary resembles three-valued logic.

All the modifications are tailored towards an easy application for consistent specialization of object behavior and business rules. Specialization is discussed in Chapter 8 and is omitted in this chapter as far as possible. Nevertheless, the modifications presented here lay the foundations for specializing passive and active behavior. But first of all, we introduce a new running example before we start describing the details of modifications.

Example: Consider a customer who applies for a loan at a loan bank. A loan application is created and processed by the clerks of the loan bank. Figure 7.1 shows a part of the life-cycle diagram of object type LOAN_APP. Type LOAN_APP represents the loan application. It defines (among others) the attributes amount (the loan's amount), installment (the borrower's monthly payment), and dispIncome (the borrower's disposable income). The life-cycle diagram in Figure 7.1 describes that after the borrower's credit rating has been determined, the loan application may be granted, rejected, or forwarded to the board.

Modifications and Extensions. Activity preconditions, situation diagrams, and rule conditions are the basic concepts to express dynamic integrity constraints. Basicly, only

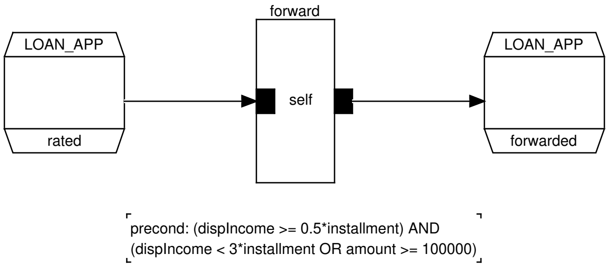


Figure 7.2: Activity-specification diagram of activity forward before revision

activity preconditions need to be extended such that they can represent permissions and prohibitions. Situations and rules need not be modified to represent obligations.

First, we present how activity preconditions are extended. Then, we illustrate how permissions, prohibitions, and obligations are represented in Active Object/Behavior Diagrams.

Permission and prohibition represent two contrary conditions. In order to model permissions and prohibitions, the activity precondition is replaced by two counterpart conditions, where one of them corresponds to the precondition so far specified with activities. Counterpart conditions need not be complementary as it is required in deontic logic. Permission P and prohibition F of an activity are said to be complementary if for each parameter binding either $P \wedge \neg F$ or $\neg P \wedge F$ holds. Since we allow to specialize the active and passive behavior during design (by using the concept of inheritance), we require the counterpart conditions to be complementary only at object types that can be instantiated. Chapter 8 shows how permissions, prohibitions, and obligations are specialized.

Up to now, the *activity precondition* is composed of the textually specified precondition, which is labeled precond, and of the graphically specified prestates of the activity's input ports.

As already mentioned in Section 7.2.1, the activity precondition has always represented a necessary precondition. Having satisfied the precondition did not require the activity to complete successfully. If the precondition failed when trying to invoke the activity, it could not be executed at all. Therefore, the activity precondition has represented a necessary precondition.

Example: Figure 7.2 shows the original activity-specification diagram of activity forward with the graphically specified prestate rated and the textually specified precondition labeled precond. Activity forward can be performed only if the precondition is satisfied (i.e., if the loan application is in state rated and if the textual condition at label precond is satisfied).

Now, instead of associating a necessary precondition with an activity, a permission and a prohibition are associated with an activity, where the former necessary precondition corresponds to the negated prohibtion. The *permission* represents a sufficient precondition,

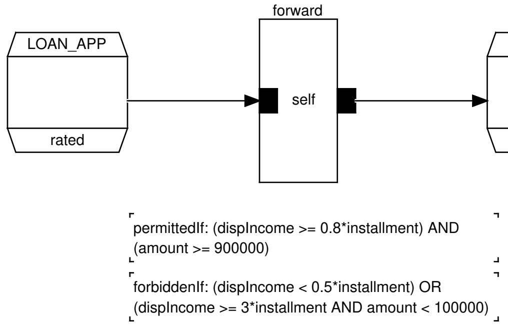


Figure 7.3: Activity-specification diagram of activity forward after revision

which guarantees that the activity is executed successfully if it is satisfied when the activity is invoked. The *prohibition* represents the negated necessary precondition. If the prohibition is satisfied when an activity is invoked, executing the activity is forbidden. Permission and prohibition need not be complementary (except at instantiable object types). The permission and the prohibition, respectively, are given the labels permittedIf and forbiddenIf, respectively.

Again, the permission and the prohibition have to be considered in combination with the prestates, which still remain. The prestates shown for the input ports of an activity express a *prestate condition*, which is satisfied if each actual parameter resides in each prestate connected to the input port that represents the corresponding formal parameter. It is permitted to perform an activity if the permission as well as the prestate condition are satisfied. It is forbidden to perform an activity if the prohibition is satisfied or if the prestate condition is not satisfied.

Example: Figure 7.3 shows the revised activity-specification diagram of activity forward. The precondition is replaced by a permission and a prohibition. The prohibition at label forbiddenIf (Figure 7.3) represents the negated necessary precondition at label precond (Figure 7.2).

Specifying no permission means that performing the activity is not allowed explicitly. Specifying no prohibition means that performing the activity is not forbidden explicitly. The logical value of the textual conditions that are not given is “false” by default. The prestate condition still represents a necessary precondition, which if violated forbids to execute the activity.

Now, let us summarize how the concepts permission, prohibition and obligation are represented using the above modifications.

1. A *permission* is defined by the activity precondition labeled permittedIf and by the prestate condition. If *both* conditions are satisfied, executing the activity is permitted.

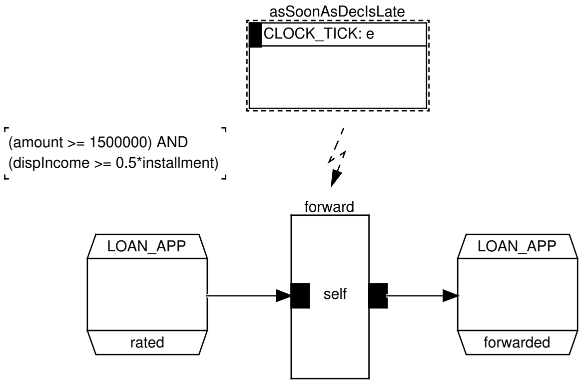


Figure 7.4: Activation diagram for rule Forward before revision

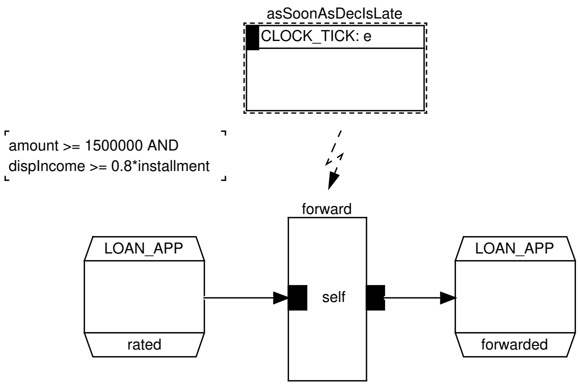


Figure 7.5: Activation diagram for rule Forward after revision

2. A *prohibition* is defined by the activity precondition labeled forbiddenIf and by the prestate condition. If the condition labeled forbiddenIf is satisfied *or* the prestate condition is not satisfied, executing the activity is forbidden.

3. An *obligation* is represented by a rule. The obligation to execute an activity is defined by the rule's situation class and by the rule condition. It is meaningful in practice that the obligation to execute an activity requires also the permission to do it. Wieringa et al. [WWMD91] remarked that this is not provable in their deontic language. Nevertheless, we require that the permission to execute the rule action is included into the rule condition.

   Activation diagrams help the designer to include at least the prestate condition concerning the object on which the activity is performed into the rule condition by defining a rule to be "experimental" (cf. Chapter 6). Experimental rules require the object on which the activity is performed to be in all prestates specified at the activity's primary input port.[1]

   Example: The loan bank has established a business rule saying that forwarding a loan application to the board as soon as the decision (whether to grant, to reject, or to forward the application) is late is obligatory if the amount is 1,500.000 ATS or higher. Assume the decision about a loan application to be late when either two or five weeks (depending on whether the borrower has applied for a personal loan or for a mortgage loan) have passed since the credit rating has been determined.

   This obligation is implemented by rule Forward, which invokes activity forward. The activation diagram depicted in Figure 7.4 illustrates rule Forward before the division of the activity precondition into permission and prohibition. The rule-triggering event e of class asSoonAsDecIsLate occurs on every Monday at eight a.m. if the decision about the loan application is already late. The action is executed actually at eight o'clock on the first Monday after the decision is late. This rule obeys the necessary precondition of the activity forward by including it explicitly into the rule condition. Of course, by requiring the amount to be at least 1,500.000 ATS, parts of the necessary precondition are dispensable.

   Figure 7.5 depicts the activation diagram of the same rule after the revision. Now, the rule condition obeys the permission of the revised activity forward. By requiring the amount to be at least 1,500.000 ATS, parts of the permission are again dispensable.

The reader might ask why Active Object/Behavior Diagrams provide two concepts (the condition part of situation diagrams and the rule condition) to specify the condition part of an obligation. Situation diagrams seem to be powerful enough to specify

[1] Introducing an additional kind of rules (besides experimental and definite rules) that includes automatically the whole permission into the rule condition would facilitate the specification of obligations, which have to obey the permission of the activity to be invoked.

an obligation's conditions. The rule condition could be incorporated into the situation diagram and specifying rule conditions could be left out at all. But separating rules from situations on which they react makes situation classes much more reusable. Otherwise, including the rule condition into the situation diagram ties the situation class to a specific rule and prevents it from being reused. This represents a design guideline for active object-oriented database design (cf. Chapter 9).

We do not provide a concept (like rules for modeling obligations) to specify that executing an activity is *discretionary* because we believe that discretionary tasks play only a secondary role in conceptual database design. The database schema has to represent the facts about the real world, e.g., when something has to be done (obligation), whether something is allowed to be done (permission), or whether something is forbidden to be done (prohibition). The end user can execute an activity everytime he is allowed to do it. Specifying discretionary tasks is therefore not necessary. Providing an additional concept to model discretionary tasks may confuse the designer and thus may encourage misuse, especially with respect to specialization of active behavior.

Notice once more that the terms "permission" and "prohibition" used in the context of Active Object/Behavior Diagrams include the condition labeled permittedIf or forbiddenIf, respectively, as well as the prestate condition as mentioned above. In the following, the term "activity precondition" refers to both counterparts, i.e., to the permission and the prohibition of an activity.

The division of preconditions into counterparts is useful especially for specialization. The designer may leave an undefined gap between permission and prohibition and may close this gap step by step downwards the specialization hierarchy (cf. Chapter 8).

Parallels to Three-Valued Logic. Remember that preconditions are replaced by two contrary conditions (i.e., into permission and prohibition), which need not be complementary. These contrary conditions resemble three-valued logic. The basic difference to "conventional" two-valued logic is that with three-valued logic, a logical expression can result in one of three truth values: "true," "undefined," or "false." There may be some bindings for a logical expression that lead to an undefined result. For more details about three-valued logic, see, for example, [Urq86].

The concept of incomplete specification is similar to the specification of contrary conditions in Active Object/Behavior Diagrams now. Performing an activity is permitted if its associated permission is satisfied, and it is forbidden if its associated prohibition is satisfied. For all parameter bindings that let both conditions fail, executing the activity is neither permitted explicitly nor forbidden explicitly; it is undefined. Specifying two conditions that need not be complementary reduces the underlying logic to two-valued logic with truth values "true" and "false." Several rules are defined concerning the consistent specification of such conditions. They will be introduced below.

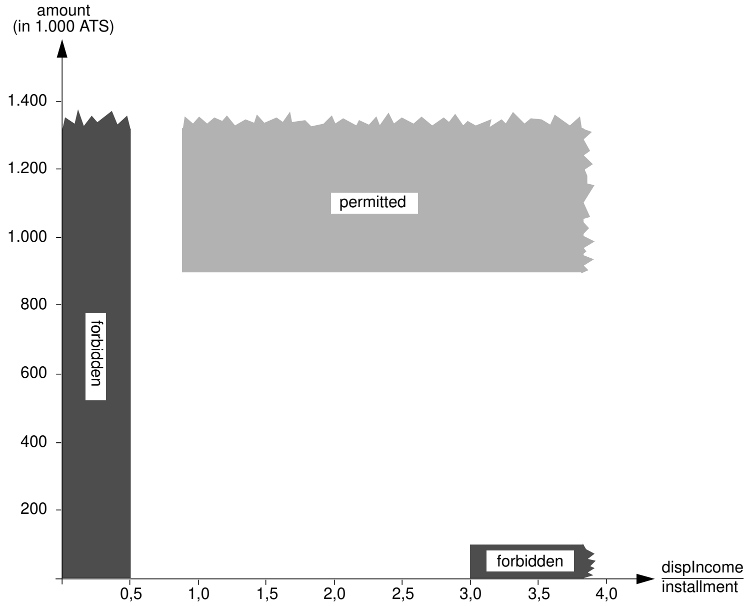


Figure 7.6: Graphical representation of permission and prohibition of activity forward

7.2.3 Consistency Rules for Permissions, Prohibitions, and Obligations

Applying modeling concepts is usually restricted by additional rules in order to achieve a consistent database schema. Introducing the concepts permission, prohibition, and obligation requires also additional consistency rules.

The following rules do not consider specialization. If object structure, passive behavior, or active behavior is specialized, additional consistency rules apply to specialization. They will be discussed in Chapter 8.

Before we start describing rules that restrict the use of the introduced concepts, we give a "liberty," which will be restricted when the involved concepts are specialized. In general, permission and prohibition need not be complementary. Permission P and prohibition F of an activity are said to be complementary if for each possible parameter binding of the activity either $P \wedge \neg F$ or $\neg P \wedge F$ holds. There may be some parameter bindings such that neither P nor F are satisfied and, therefore, executing the activity is neither permitted explicitly nor forbidden explicitly (*free activity execution*). As we will see in Chapter 8, the consistent specialization of preconditions requires that this rule is strengthened for concrete object types. Concrete object types are those objects types that can be instantiated. Permission and prohibition of activities at concrete object types have to be complementary.

Example: Consider the permission and the prohibition of activity forward in Figure 7.3. Both conditions are not complementary. Figure 7.6 illustrates the gap between permission and prohibition in a two-dimensional graphical representation. The horizontal axis depicts the ratio of the disposable income to the monthly installment, the vertical axis depicts the loan amount (in 1.000 ATS). The areas for which performing the activity forward is either permitted or forbidden are shaded and labeled. Whether the execution of the activity with a parameter binding covered by the blank area is permitted or forbidden, has to be defined at subtypes.

The following consistency rules have to be obeyed when permissions, prohibitions, or obligations are specified.

1. Permissions and prohibitions must not contradict each other (*rule of freedom of contradiction for activity preconditions*). Thus, both conditions must not be satisfied for the same parameter binding. Otherwise, if a parameter binding satisfies both conditions, this would mean that executing the activity is permitted as well as forbidden for this parameter binding.

   Example: Figure 7.6 illustrates all possible parameter bindings for activity forward. The areas for permission and prohibition do not overlap. Thus, the precondition of activity forward follows the rule of freedom of contradiction for activity preconditions.

2. If executing an activity is obligatory, it has to be permitted, too (*empowerment rule for obligations*). Thus, either the situation diagram of the rule's situation class—applicable for logical situations only—or the rule condition has to obey the permission to execute the action. In other words, the permission to execute the action has to be included either into the situation diagram or into the rule condition.

   Example: Rule Forward, which is represented by the activation diagram depicted in Figure 7.5, follows the empowerment rule for obligations because the rule condition implies the permission of activity forward.

3. Similar to the rule of freedom of contradiction for activity preconditions, performing conflicting rule actions must not be obligatory at the same time (*rule of freedom of contradiction for obligations*).

   There are two alternative definitions for conflicting rules: Rules can be said to conflict if they are commutative or if they are confluent. A set of rules that are considered for execution at the same point in time is *commutative* [FG90] if (1) each rule of the set can be executed (rule condition and activity permission still hold) after another one of this set has been executed and if (2) the final database state does not depend on the order in which the rules are executed. A set of rules is *confluent* [FG90] if for all database states s, s_1, and s_2 where s_1 and s_2 can be reached from s by executing some rules of the rule set, there is a database state s' such that s' can be reached from s_1 and s_2 by executing the remaining rules of the rule set.

Commutativity and confluence are very similar properties of rule sets, but confluence is less stringent because it does not require that the rules are applicable in any order, but only that the result does not depend on the order in which the rules are executed. The terms commutativity and confluence of rule sets have been defined already for production rule systems in Artificial Intelligence (cf. [FG90]) and are rediscovered and adapted for active rule systems in [AWH92]; there, they are used for static rule analysis.

Example: Consider the life cycle of a loan application depicted in Figure 7.1. Activities grant, forward, and reject have the same prestate rated. E.g., executing grant before reject prevents reject from being executed and vice versa. A rule that performs grant does not commute with a rule that performs reject if they are triggered at the same point in time. The resulting database state depends on the order in which they are executed.

If executing two activities is obligatory at the same time and if these two activities should be executed in a certain order, a compound activity can be specified. The activities' execution order can be specified within the activity-realization diagram of the compound activity (cf. Section 4.4.4). In other active languages (e.g., Chimera [CGMR95]), two rules may be given different priorities to specify their action's execution order.

Aiken et al. introduce in [AWH92] another criterion called *termination* requiring that rule processing is guaranteed to terminate. In Active Object/Behavior Diagrams, rule processing always terminates within a chronon because the effects of executing a rule action are not visible before the next chronon. However, rule processing may not terminate across chronons although no additional basic situations are raised. We have not considered termination so far. Further information regarding termination can be found, for example, in [VGD97].

The above consistency rules represent general rules restricting the specification of permissions, prohibitions, and obligations. Additional consistency rules constrain the rules above if passive and active behavior are specialized (cf. Chapter 8).

7.2.4 Violating Consistency Rules

The consistency rules introduced in Section 7.2.3 may be violated by the designer. In general, there are two ways to deal with their violation: First, if possible, inform the designer about the misuse at "compile-time" and have him to correct the error. Second, if the violation are not detected at compile-time, then try to resolve the arising conflict at "run-time" and generate an exception. The advantages, the disadvantages, and the applicability of both approaches will be discussed in the following.

Conflict Detection at Compile-Time. The permissions, prohibitions, and obligations specified by the designer during the modeling process may violate one or more of the consistency rules defined above. For example, executing an activity may be permitted as well as forbidden for the same parameter binding.

The correctness of sentences—consistency rules represent sentences about the database schema—is usually checked by establishing a truth table (cf. [RN95]). Building a truth table is often impractical because of the computation time is exponential in n. The process of building truth tables can be extended also to classes of inferences. There are certain patterns of inferences that occur over and over again, and their soundness can be shown once and for all. Such a pattern of inference is called an *inference rule*. One well-known inference rule is the *resolution rule*. The application of the resolution inference rule will find a proof if one exists, but its efficiency is not guaranteed. *Subsumption* is one strategy to implement resolution. Checking a set of sentences about a general database schema for satisfiability is NP-complete, and, therefore, unlikely to yield to a polynomial-time algorithm. Of course, there is a useful class of sentences, called Horn sentences, for which a polynomial-time inference procedure exists. But not every sentence can be represented as Horn sentence.

However, building a truth table for sentences about a active object-oriented database schema as it may be specified by Active Object/Behavior Diagrams, is very hard. E.g., all possible actual parameters of an activity would have to be tested to check whether the permission and the prohibition may ever contradict. Consequently, the consistency rules are not assumed to be checked at compile-time.

Conflict Resolution at Run-Time. Whether consistency rules are violated can be detected for a concrete binding at run-time. For example, if the permission and the prohibition are satisfied by the actual parameters, the rule of freedom of contradiction for activity preconditions is violated and, thus, the conflict has to be resolved. Both conditions have to be evaluated every time an activity is invoked to detect such specification errors. But this causes a significant performance decrease at run-time. For this reason, a kind of "consistency-checking switch" should allow to turn on checking of consistency rules only during the testing phase and turn it off when the implemented database schema is delivered.

Assume, the consistency-checking switch to be turned on. So, how to react if consistency rules are violated? The general conflict resolution might be not to perform an activity if a consistency rule is violated and to raise an exception that protocols the context in which the violation has occurred. But depending on the business area not performing an action upon the occurrence of a conflict might have more disastrous consequences than peforming it. For example, a loan bank usually freezes a loan if the borrower is late with repaying the installments. If a conflict is detected while trying to suspend granting further loans, the loan bank might want to perform the suspension despite of the conflict and not to reject the suspension.

Thus, what to do in the case of a conflict depends on the business area. The most flexible solution is to define a default conflict resolution for the violation of a consistency rule, to raise an exception event that signals the violation, and to allow the database designer to specify a reaction to such an exception event in advance by means of another business rule.

Another approach is, for example, to specify priorities between activities. If several activities are invoked at the same time by different rules, they are executed according to the priorities. Further, each activity may have an alternative activity, which is invoked if executing the activity fails (despite it is invoked by a rule or by a user).

In the remainder of this section, we describe each possible violation of the consistency rules set up in Section 7.2.3, define the default conflict resolution action, and show which exception event is raised in the case of a conflict.

The rule of contradiction for activity preconditions is violated by parameter bindings for which both counterparts are satisfied. Both counterparts have to be evaluated to detect the contradiction. Evaluating always two conditions instead of a single condition (e.g., the permission) decreases performance. Nevertheless, lower performance during the testing phase is less critical than specification errors are.

If executing an activity is permitted as well as forbidden for the same binding, permission and prohibition contradict. The default conflict resolution is to give the prohibition higher priority and thus to prevent the activity from being executed. An exception event is raised for an activity a and added to event class a_contradictingPrecond, which is local to the object type to which the activity belongs. Rules may be specified upon the occurrence of this event in order to react to the exception.

Whether the rule of freedom of contradiction for obligations is violated, cannot always be detected at run-time when rules are considered for execution. Contradicting obligations are rules that have to be executed at the same time but perform conflicting activities. Activities conflict if they do not commute or are not confluent. Assume two obligations, which are implemented by rules r_1 and r_2. Both rules are considered at the same time, i.e., their rule events have occurred and their rule condition and their actions' permissions are satisfied. The order in which considered rules are triggered for execution is not predefined. Before a rule execution is triggered, the rule condition and the action's permission are checked again. If both conditions still hold, the action is performed.

If the execution of r_1's action changes the database in a way that the permission of r_2's action is not satisfied any more, the exception event r_2_precondViolatedDuringExec is raised. If the rule condition of r_2 is not satisfied, r_2 is not executed at all. Contradicting obligations can be detected in this way and reactions to the violation can be specified upon the exception event. Executing one of the actions may not violate the other rule action's permission or rule condition, but may "only" influence the behavior of the other rule's action. Whether this is the desired effect, depends on the business area. However, this effect remains undiscovered.

If executing two conflicting activities at the same time is obligatory, a compound activity should be specified that defines the correct execution order of the activities. For example, consider a loan bank that prints bank statements about the deposits and the withdrawals that have occurred within a month on a customer's account at the end of every month and sends them to the customers. Further, the customers' bank accounts are debited with service charges and credited with the interest at the end of every quarter of a year (at the end of every third month). If both tasks are implemented by business rules that fire either at the end of every month or at the end of every third month, both rules may be executed in any order at the end of every third month. Since it seems to be meaningful to print the bank statements always after debiting and crediting the bank accounts, a compound activity that performs the printing last has to be defined.

As already mentioned above, another approach is to specify priorities between activities. Then, if activities have to be executed at the same time, they are executed according to the priorities. In the example above, debiting and crediting should have a higher priority than printing bank statements.

The empowerment rule for obligations is violated if a rule tries to perform its action although it is not permitted to do it. Remember that defining rules to be experimental facilitates including the prestate condition of the object on which the activity has to be executed into the rule condition. Nevertheless, the designer might want to implement the activity precondition within the situation diagram or within the rule condition explicitly. If the permission is not satisfied when the rule execution of rule r is triggered, an exception event of class $r\_\mathsf{precondViolated}$ is raised. The default conflict resolution is not to perform the action but to raise the appropriate exception event. Reactions to such a violation may be specified by a rule triggered when an exception event occurs.

The conflict resolution schemes as defined above do not consider specialization. Further consistency rules are required to ensure a consistent specialization of object structure, passive behavior, and active behavior. The new consistency rules need new conflict resolution schemes, which are discussed also in Chapter 8.

Related Work. The following approaches are selected from a wide range of papers that deal with the handling of violations of dynamic or deontic integrity constraints.

Dignum et al. [DMW94] refer to similar conflicts between permissions and obligations. They introduce a "safe context" for an activity. The safe context of an activity a consists of all activities that do not conflict with a. They advise to compute safe contexts for an activity to check whether two activities conflict or not.

Jonscher [Jon93] specifies the access rights of users in a database system by means of deontic "rights." The terms permission, prohibition, duty, and liberty always refer to the database users and state concern about their permission, prohibition, or obligation to do something, which is quite different to the understanding of deontic concepts in this thesis (see also Section 7.2.5). Jonscher resolves conflicts that arise during run-time according to priorities defined for deontic concepts in advance. E.g., the conflict raised by contradicting permission and prohibition is solved by giving the prohibition priority over the permission.

Feenstra and Wieringa [FW95] propose an approach to the validation of formal specification of integrity constraints. The validation problem of formal specifications consists of assuring whether the formal specification corresponds with what the domain specialist intends. Feenstra and Wieringa propose a system that can answer reachability queries, in which it is asked whether the system can evolve from one state into another without violating the integrity constraints. Techniques from planning and model generation become applicable by making reasonable assumptions. The given reasoning techniques may be applied also to Active Object/Behavior Diagrams, but this has not been considered up to now.

7.2.5 Differences to Deontic Logic

Objective and deontic modalities cannot be applied directly to conceptual database design for a number of reasons. The semantics of the basic deontic concepts like permission, prohibition, and obligation differ in conceptual database design (especially in Active Object/Behavior Diagrams) and in deontic logic. These differences will be discussed in the following.

First, in [WWMD91] Wieringa et al. have recognized that the logic of preconditions specified in conceptual specification languages based on Petri nets is usually such that if they are not satisfied, the action is not performed. This carries a connotation of actuality, which cannot be expressed exactly in deontic logic. This means that the execution semantics as defined for Active Object/Behavior Diagrams cannot be expressed exactly by means of objective and deontic modalities.

Second, in most conceptual database design languages (and thus also in Active Object/Behavior Diagrams), reactions to violations of deontic constraints are specified at the same level as actions that do not represent reactions to violations of deontic constraints. Reactions as well as "conventional" actions are specified in the same object life cycle because the currently available modeling concepts do not allow to distinguish explicitly between "normal" dynamics (representing real-world norms) and "exceptional" dynamics (representing a violation of norms).[2] As a consequence, necessary and deontic constraints are combined in a single life cycle. In addition, also the business area often does not differ between an action that represents a reaction to a violation of a real-world norm and an action that represents a usual business policy. E.g., fining the borrower of a book if he or she does not return the book in time may be interpreted either as an exception or as a usual task in the real world. Permissions are considered to be equivalent to sufficient preconditions and

[2]Providing explicit concepts to model exceptional dynamics could be a topic of further research in this area.

prohibitions are considered to be equivalent to negated necessary preconditions because (1) preconditions are usually used for constraining both normal and exceptional dynamics and (2) organizational units, which may be forbidden, permitted, or even obliged to execute operations, are not considered here as discussed below.

Third, we have omitted modeling the access model of information systems. Deontic constraints are often used for specifying the permission, the prohibition, or the obligation of some organizational subject (e.g., of a specific person or of a department) to perform a certain action. For example, Jonscher [Jon93] specifies access authorization by so-called "duties" and "liberties" of database users using deontic logic. But we do not consider the access model in our dynamic constraints at all. Weigand introduces in [Wei93] the "environment of discourse" as the organizational environment, where the information system supports communication and coordination. In addition, he argues that deontic integrity constraints in conceptual models deal typically with the environment of discourse rather than with the universe of discourse. We believe that there are also norms in the real world that need not be enforced necessarily by human beings or organizations. For example, consider a gift voucher that may be exchanged only until a certain date. An operation may be performed (without any intervention by the user) on a voucher in the database that marks the voucher to be invalid if the validity period is over. We believe also that specifying a "doer" for actions [Mai93] is not subject to database design but to a higher-level design, commonly referred to as workflow modeling (see, for example, [Bic97]). There, organizational units are given the ability and the responsibility to perform actions. Database design should abstract from organizational aspects and should concentrate on the exact representation of the business area in the database. Maibaum [Mai93] warns to use a precondition for both for identifying the context within which an action is performed and/or for checking whether performing this action is allowed. As we abstract from any access model, access rights are not mixed with permissions and prohibitions in our approach. But in general, combining the specification of necessary and deontic constraints in a single life cycle with a single modeling concept (formerly precondition, now permission and prohibition) prohibits a clear distinction between objective and deontic modalities. Krogstie and Sindre [KS96c] use deontic operators to specify the requirements of information systems. Krogstie and Sindre provide a rule concept that allows to specify permissions, prohibitions, and obligations also for actors and roles of an organizational model. We class rules that are addressed to actors or roles with "workflow modeling." Rules with the "system" as actor can be compared with the notion of rules provided by our approach. Krogstie and Sindre model also permissions and prohibitions by means of rules. They distinguish permissions, prohibitions, and obligations only by a single deontic operator.

Forth, as opposed to obligations, necessities cannot be modeled explicitly by means of Situation/Activation Diagrams. A necessity describes the context in which something has to be done; violating a necessity is not possible in the real world. An obligation "implements" a necessity such that it defines when to perform an action that executes what's necessary. If business processes are modeled by means of Petri nets or arbitrary state-transition diagrams, a business case is assumed to be processed until it resides in one or several of the possible target states (a sink in the net). If business processes are modeled by means of Active Object/Behavior Diagrams, the life cycle of the business object is ended (1) by reaching a state that is intended to be a target state of the business case or (2) by executing a removal activity. The necessity to execute an operation is specified by the prerequisite to move towards a target state or towards a removal activity, respectively, and by the preconditions or permissions and prohibitions, which reduce the possible ways of the database to evolve. Consider the example about borrowing books as used in [WWMD91]. Abstract from any person that borrows a book and consider the book only. If a book is borrowed (it resides in a state called "borrowed"), then it is necessary to return the book. If the book is already fined, the book has to be returned accompanioned by a payment. The necessity to return a book, regardless the book is fined or not, is expressed implicitly in the book's life cycle because there is no way back to state "available" except by returning the book. The state "available" represents the target and the source of the book's life cycle. Commonly, most of the necessities are specified already in the object life cycle and in additional preconditions of the activities. Active Object/Behavior Diagrams derive the necessity to do something from obligations, permissions, and prohibitions associated with activities, and from the life-cycle diagram of object types.

Chapter 8

Specialization

Contents

Specializing means to modify or to make something suitable for a special use or purpose. Consider the three phases which the database design process comprises: conceptual design, logical design, and physical design. Each design phase is tailored to a specific purpose. E.g., within logical database design, the designer tries to transform the conceptual schema into a logical schema in order to implement it on top of a particular database system. Hence, the conceptual schema is specialized into a logical schema. In general, the database schema resulting from a design phase is specialized in the successive design phase. Using the concept of specialization is not restricted to the data models of design phases as a whole. Within each design phase, the objects and its features represented by the schema, may be specialized. The counterpart of specialization, generalization, is used for representing the essential aspects of the business area by abstracting from details.

The aim of this chapter is to provide a comprehensive overview and the interdependencies of specializing the three main components of Active Object/Behavior Diagrams: specializing object structure, specializing object behavior, and specializing business rules.

Modeling object structure in a top-down manner is widely accepted within conceptual object-oriented database design. Inheritance is the basic concept used in all object-oriented design methods for specializing object structure and object behavior. Inheritance defines a relationship between two object types, where one object type, called the subtype, inherits the structure and the behavior from the other one, called the supertype.

Many object-oriented design methods like OMT [RBP+91], OOSA [EKW92], OOAD [Boo94] define the inheritance of object behavior only at object-type level by saying how an object's operations are inherited from the supertype and how their interface can be overridden at the subtype. These methods do not say how to inherit the object life cycle. Also the original version of Object/Behavior Diagrams [KS91a] did not address the inheritance of object life cycles. Follow-on papers on Object/Behavior Diagrams treat already the specialization of object life cycles [KS94, Sch90, SS95, SS97].

Although specializing object structure and object behavior has already been discussed it is presented here again. Incorporating the concepts permission, prohibition, and obligation in combination with the consistency rules in Chapter 7 requires a revision of the specialization concepts for structure and behavior. Specializing business rules has been discussed together with Object/Behavior Diagrams first in [BS95, KS95] and is elaborated in further detail here, where business rules are modeled by means of Situation/Activation Diagrams.

A major contribution of this chapter is that we distinguish three kinds of specialization of an object type's features (properties, activities, states, situation classes, situation diagrams, and rules) that are not always clearly distinguished in the literature: redefining inherited features at a subtype, adding new features at a subtype, and refining features.

The remainder of this chapter is structured as follows. Section 8.1 introduces the specialization of conceptual models and outlines various aspects of specialization. Section 8.2

presents the specialization of object structure. Section 8.3 presents the specialization of object behavior, i.e., of object life cycles, of activity specifications, and of activity realizations. Section 8.4 presents the specialization of business rules, i.e., of situations and of rules. Finally, Section 8.5 gives a small overview of related work in the area of specializing business rules.

8.1 Introduction

Specializing object types and their features helps the designer to develop a comprehensible conceptual schema of the business area. Numerous concepts are provided in order to specialize object types. This section introduces the main concepts of specialization on which the successive sections are based. The specialization of object structure, of object behavior, and of business rules rely on the concepts introduced here.

Section 8.1.1 presents inheritance as the basic concept for specialization, which is used in every object-oriented design and programming language. Section 8.1.2 presents some prerequisites necessary for specializing object structure, object behavior, and business rules in Active Object/Behavior Diagrams. Section 8.1.3 classifies specialization into three kinds of specialization, which are also distinguished with the specialization of object structure, of object behavior, and of business rules. Section 8.1.4 describes behavior composition as a means for generalizing object behavior. Section 8.1.5 shows the impact of the concepts permission, prohibition, and obligation on specialization. Section 8.1.6 points out that some of the consistency rules that are introduced for specialization are hard to check at compile-time and therefore are checked at run-time. Section 8.1.7 presents a concept called "dynamic evaluation," which allows to view an object as if it were an instance of one of its supertypes, as a helpful concept for specialization. Finally, Section 8.1.8 presents the background of this research on specialization.

8.1.1 Inheritance—The Basic Concept of Specialization

Inheritance represents the core concept in object-oriented design methods, which allows to specialize the features of an object-oriented schema. Two kinds of inheritance are distinguished with respect to the possible number of supertypes: single inheritance and multiple inheritance. *Single inheritance* differs from *multiple inheritance* as with the former, a subtype can define an inheritance relationship only to one supertype, whereas with the latter, an object type can define an inheritance relationship to several supertypes. In the context of Active Object/Behavior Diagrams, only single inheritance is considered. Using multiple inheritance in Active Object/Behavior Diagrams is possible, but not necessary. Taivalsaari [Tai96] mentions that multiple inheritance is often used for combining abstractions of equal importance. The classical example he gives is to inherit two object types, TEACHER and STUDENT, to form a type TEACHING_ASSISTANT. In practice, so he argues, the combination of conceptually equal abstractions using multiple inheritance can be quite laborious, however, because such abstractions tend to contain a large number of overlapping properties that have to be dealt with at the subtypes. Such situations can usually be expressed more naturally using the concept of roles. Using Active Object/Behavior Diagrams, we would specify an object type PERSON, whose instances can play two roles (also at the same time), TEACHER and STUDENT.

Since inheritance allows to base new object definitons upon existing ones, inheritance may be used for many different purposes [Tai96]. Inheritance can be considered a mechanism to implement code sharing and, thus, a favorite concept to reuse components of object-oriented programming languages. Contrary to programming languages as a tool for logical design, Active Object/Behavior Diagrams as a tool for conceptual design should use inheritance for a different purpose, i.e., to express conceptual specialization only. *Conceptual specialization* reflects the specialization of objects and its features of the business area in the conceptual schema.

In Active Object/Behavior Diagrams, the inheritance relationship between object types is defined by the subtype-of relationship specified at the object-type diagram of the subtype (cf. Section 4.2.3). Since we treat situations as first-class objects, also a situation type may inherit the properties of its supertype by specifying a subtype-of relationship (cf. Section 5.1). Both object types and situation types define type hierarchies based on this relationship. Subtype-of relationships must not define circles in the type hierarchies. According to the classification of inheritance concepts established in [KS96b], the inheritance concept of Object/Behavior Diagrams and of Active Object/Behavior Diagrams as presented in Chapter 4 provides a "covariant subtypeOf" relationship between a subtype and its supertype.

We distinguish between concrete and non-concrete[1] object types. A *concrete object type* can be instantiated, whereas a *non-concrete object type* cannot be instantiated. Subtypes and supertypes define an inheritance graph with the object types as nodes and the subtype-supertype relationships as arcs. The inheritance graph consists of a set of unconnected inheritance trees. The design of object types becomes more comprehensible if *only objects types representing leaves within an inheritance tree can be defined concrete* and thus can be instantiated [KS96b]. The power of the modeling concepts is not limited if we follow this approach. Concrete object types may become non-concrete by introducing new subtypes, and their instances become instances of the added concrete subtypes. But note that a leave of an inheritance tree is not concrete by default. Non-concrete leaves indicate that designing the inheritance tree is not finished yet. Since we treat situations as first-class objects, also situation types are either non-concrete or concrete.

In the remainder of this chapter, the phrase "some supertype" and "a supertype" or "some subtype" and "a subtype" of an object type denote any direct or indirect supertype of

[1]The term "non-concrete" is used intentionally instead of "abstract" because the latter will be used with refinement.

the involved object type or any direct or indirect subtype of the involved object type, respectively.

8.1.2 Prerequisites for Specialization

As already mentioned, we distinguish between the specialization of object structure, object behavior, and business rules. Object structure is represented by properties, object behavior is represented by activities and states, and business rules are represented by rules, situation classes, and situation diagrams. The three components have in common that all features (properties, activities, states, situation classes, situation diagrams, and rules) of the supertype are inherited by the subtype.

Each feature that is introduced at some object type O is inherited by each direct and indirect subtype of O (cf. "rule of full inheritance" in [SS95]). A feature belongs to the object type at which it is introduced and to the direct and indirect subtypes of this object type. The following rules apply to all features of an object type, i.e., to properties, activities, states, rules, situation classes, and situation diagrams. They represent a prerequisite for specializing object structure, object behavior, and business rules.

- The *rule of unique names of features* requires that the name of each feature of an object type O is unique at O, regardless of the feature being inherited from some supertype of O or being introduced at O.
- The *rule of origin for features* requires that if a feature belongs to several object types, it belongs also to their common supertype. This rule ensures that a feature is always introduced at a single object type.

How the inherited features may be modified at the subtypes is discussed separately for object structure, for object behavior, and for business rules in Sections 8.2, 8.3, and 8.4.

8.1.3 Classification of Specialization

Object structure, object behavior, and business rules may be specialized in three ways: by redefinition, by extension, or by refinement. *Redefinition* means overriding conditions over features at a subtype. *Extension* means adding new features at a subtype. *Refinement* means detailing features (not necessarily inherited features at a subtype) by introducing new features that refine other features.

Redefinition. A condition over a feature is, for example, the domain of a relationship, which constrains the objects that may be referenced by the relationship, to direct or indirect instances of the domain type. Pre- and postconditions are conditions over activities and constrain their invocation.

Basically, we distinguish between necessary and sufficient conditions. Their difference will be explained by means of necessary and sufficient preconditions of operations [WWMD91]. If a *necessary precondition* is not satisfied when an operation is invoked, the operation cannot complete successfully. Further, if a necessary precondition is satisfied when an operation is invoked, it is not guaranteed that the operation completes successfully (cf. Section 7.2.1).

If a *sufficient precondition* is satisfied when an operation is invoked, the operation is guaranteed to complete successfully. If a sufficient precondition is not satisfied when an operation is invoked, the operation may be performed, but is not guaranteed to complete successfully (cf. Section 7.2.1).

Necessary and sufficient preconditions of operations may be redefined at subtypes as follows. A necessary precondition may be only *strengthened* at subtypes. Thus, a necessary precondition of an operation at an object type O represents the weakest necessary precondition of the operation's necessary preconditions at subtypes of O, i.e., it is implied by the operation's necessary precondition of each subtype of O. A sufficient precondition may be only *weakened* at subtypes. Thus, a sufficient precondition of an operation at an object type O represents the strongest sufficient precondition of the operation's sufficient preconditions at subtypes of O, i.e., it implies the operation's sufficient precondition of each subtype of O.

In Active Object/Behavior Diagrams, the negated prohibition of an activity is a necessary precondition, the permission of an activity represents a sufficient precondition (cf. Section 7.2.2).

There are two ways to redefine ("override") conditions at a subtype: replacement or syntactically restricted modification. If conditions are replaced, the replaced conditions may be concise but it is more difficult to determine whether the condition is weakened or strengthened. A condition C' is weaker than or equal to a condition C if $C \rightarrow C'$ holds. If $C' \rightarrow C$ holds, then, condition C' is stronger than or equal to condition C.

If a condition is enlarged incrementally only in a restricted way (e.g., by adding the new terms with a logical *or* or with a logical *and* depending whether the condition should be weakened or strengthened), the terms of conditions are spread all around over the inheritance hierarchy and, therefore, the complete condition must be constructed from terms indicated at different object types. Rules that require a condition to become weaker or stronger are enforced implicitly.

Meyer [Mey92a, Mey92b] has choosen the latter strategy, syntactically restricted modification, for Eiffel. If a precondition is specified once with the keyword "require," it may be enlarged (weakened) at a subtype only with the keyword "require else." If a postcondition is specified once with the keyword "ensure," it may be enlarged (strengthened) at a subtype only with the keyword "ensure then."

Active Object/Behavior Diagrams follow the same strategy. Conditions that are specified at subtypes do not replace conditions at supertypes. They are interpreted as disjunctive extension if conditions have to be weakened or as conjunctive extension if conditions have to be strengthened. This stepwise specification of conditions downwards the inheritance tree resembles a modular construction and an incremental enlargement of conditions.

Extension. Each feature once introduced at an object type O is inherited by all direct or indirect subtypes of O. Besides the inherited features, each subtype may define (introduce) additional features. Object structure may be extended by properties, passive behavior by activities and states, and active behavior by situation classes, situation diagrams, and rules. Additional consistency rules are necessary to guarantee that, for example, object life cycles are extended in a consistent way when activities and states are added.

Refinement. A feature of an object O (either introduced at O or inherited from a supertype) may be refined by introducing *new* features and by establishing a refinement relationship between the refined feature and the refining features. A feature may never be refined at an object type O by a feature that is inherited from some supertype of O.

Not every feature may be refined. A feature that the designer intends to refine has to be introduced as *abstract* feature. An abstract feature need not be refined at that object type O at which it is introduced, but may be refined at some subtype of O. *Basic* features cannot be refined. Abstract features and basic features as well as the refinement relationships are inherited downwards.

The following consistency rules concern the refinement of each kind of abstract features (properties, activities, states, situation classes, situation diagrams, and rules).

- The *refinement rule for abstract features* says that once an abstract feature is refined at an object type O, the refinement of the feature may not be modified at some subtype of O. Yet the features in the refinement may be further refined.
- The *rule of limited usability of abstract features* says that abstract features (except abstract pre- and poststates for activities and the basic interval classes that exist for each abstract activity and state) cannot be used in any expression. E.g., an abstract property cannot be used in a condition specified in a situation diagram.
- The *up-to-basic refinement rule for abstract features of concrete object types* requires that each abstract feature of a concrete object type O is either refined at O or inherits a refinement from some supertype of O.

Since only abstract features may be refined, and since abstract features have to be refined (or have to inherit a refinement) at concrete object types, we need not define a "rule of unique distinction of basic and abstract features at concrete object types," which requires that a feature is either basic or abstract at each concrete object type, as it is defined in [KS96b].

An abstract feature is *locally abstract* at an object type O if it is refined at O or if it inherits a refinement from some supertype of O. An abstract feature is *locally basic* at an object type O if it is neither refined at O nor inherits a refinement from some supertype of O. An abstract feature that is locally basic at an object type O, may be refined at some subtype O' and so become locally abstract at O' and at the subtypes of O'. An abstract feature is *globally abstract* if it is locally abstract at the object type at which it is introduced and at each direct or indirect subtype. Analogously, a feature is *globally basic* if it is locally basic at the object type at which it is introduced and at each direct or indirect subtype. Of course, basic features are locally basic at each object type to which they belong and are always globally basic. Because of the up-to-basic refinement rule for abstract features of concrete object types, abstract features may be globally basic only during the design process if they are introduced at a non-concrete leave of an inheritance tree, or if they are introduced at an object type that has so far only non-concrete (direct and indirect) subtypes.

8.1.4 Behavior Composition

Remember the concept of a *compound activity* described earlier in Section 4.4.4 and introduced by Kappel in [Kap91]. A compound activity represents a means for behavior composition, where globally 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 globally basic activities and other compound activities of one or several object types by specifying a particular execution order on them. Compound activities do not augment the basic behavior of an object type and are not shown in the life-cycle diagram. A compound activity may implement an abstract activity such that it changes an object from the abstract activity's prestates into the abstract activity's poststates.

8.1.5 Specialization of Permissions, Prohibitions, and Obligations

Active Object/Behavior Diagrams have been revised in Chapter 7 not only to provide modeling concepts like permission, prohibition, and obligation, with which the business expert is more familiar. The revision has laid the foundations for the appropriate specialization of object behavior and business rules in order to achieve a more comprehensible design.

The activity precondition as introduced in Chapter 4 represented a necessary precondition. During the revision in Chapter 7, this necessary precondition has been negated in order to represent a prohibition. In addition, a sufficient precondition has been introduced, which represents a permission. Now, the activity precondition is represented by a permission as well as by a prohibition.

As already introduced in Chapter 7, in the specification diagram of an activity, the permission of the activity is given at the label permittedIf, the prohibition of the activity is given at the label forbiddenIf. To indicate that permissions and prohibitions are always enlarged disjunctively (cf. Section 8.1.3), the labels of permissions and prohibitions in activity specifications that are inherited from a supertype and are redefined at the subtype are extended. If an activity specification is redefined at a subtype, the additional disjunctive term of a permission is given at label alsoPermittedIf and the disjunctive term of a prohibition is given at label alsoForbiddenIf. Correspondingly, the postcondition of an activity is given at the label postcond. To indicate that postconditions are always enlarged conjunctively when redefined, the label of the postcondition at the redefined activity specification is extended to addPostcond.

If we speak of the permission, the prohibition, or the postcondition of an activity at an object type O, we mean the complete permission, prohibition, or postcondition including all disjunctive or conjunctive terms specified for the activity at all supertypes of O.

A permission and a prohibition need not be complementary only at non-concrete object types. A so-called "decision gap" may left open between them. However, at concrete object types, the decision gap must be closed. Then, the preconditions must not be specialized any more at a subtype and therefore have to be defined *definite*. Activity preconditions that may be redefined are called *indefinite*. Activity preconditions at concrete object types are always definite.

To indicate that rule conditions are always enlarged disjunctively when redefined, the rule condition of a redefined rule is given the label or at the subtype.

The consistency rules given in Section 7.2.3 do not consider specialization. New consistency rules will be introduced with the specialization of object behavior in Section 8.3 and with the specialization of business rules in Section 8.4.

8.1.6 Conflict Resolution

As the consistency rules for permissions, prohibitions, and obligations are adapted and extended in order to cope with the specialization of object behavior and of business rules, also the conflict resolution schemes have to be adapted and extended for the new consistency rules. Conflicts may arise when object structure, object behavior, or business rules are specialized. When the conflicts are detected (at compile-time or at run-time) and how run-time conflicts are resolved is discussed separately for object structure, for object behavior, and for business rules. See Section 7.2.4 for a general discussion about conflict detection and conflict resolution in Active Object/Behavior Diagrams.

The best way to prevent conflicts is to avoid misuse by syntactically constraining the way in which features may be specialized. Conflicts that are due to the specialization of conditions—permissions, prohibitions, etc., represent conditions—are hard to detect at compile-time. Conflicts between permissions, prohibitions, and obligations are detected at run-time and are resolved by giving one of the conflicting parties (i.e., one of the counterparts) higher priority.

8.1.7 Dynamic Evaluation

Polymorphic programming languages allow variables and values to have more than one type [CW85]. Polymorphism in Active Object/Behavior Diagrams allows a variable and an object to reference[2] objects of different types. The domain type specified for variables and relationships represents their *static type*. Object-oriented systems provide the concept of *dynamic binding* of objects, which allows variables and relationships to reference objects at run-time that are instances of a different type than the static type. The type of an object that is referenced by a variable or relationship at run-time is called the *dynamic type*. The dynamic type of a variable or of a relationship has to be the static type or a direct or indirect subtype of the static type. Further, the implementation of an operation that is bound (chosen for execution) at run-time can depend on the type of the receiver (*simple-polymorphic operation*) and also on both the type of the receiver and the type of its actual parameters (*multi-polymorphic operation*) [KS96b]. Activities in Active Object/Behavior Diagrams represent simple-polymorphic operations.

The redefinition of operations and properties takes its power from polymorphism and dynamic binding [Mey92a]. Dynamic binding is applied not only to activities and object relationships of Active Object/Behavior Diagrams, but also to rules and situations. Situation diagrams may be overridden at subtypes (with respect to some consistency rules). A situation variable that refers to an object's situation class is bound dynamically to a situation diagram the same way as a method name in a message to an object is bound to a method implementation. Consequently, the redefinition of active behavior takes its power from polymorphism and dynamic binding, too. E.g., at run-time, situation variables that are constrained to range over situations of type S (static type) may range over situations of type S' supposed that S' (dynamic type) is a direct or indirect subtype of S.

The dynamic type of a variable or relationship can differ from the static type, i.e., it can be a direct or indirect subtype of the static type. Remember that only concrete object types may be instantiated and that only leaves of an inheritance tree can be concrete. Opposed to the concept of static and dynamic types, a concrete instance might be viewed as if it were an instance of one of its supertypes and, for example, all the associated conditions or situations are *evaluated dynamically* according to that type. Such a feature may be used (1) by the designer for performing tests to check the constraints that are common to instances of all subtypes, or (2) by the system for checking conditions and situations

[2] Objects reference other objects by means of a relationship.

incrementally from the top of the inheritance hierarchy down to the concrete object type in order to save time or (3) by the system for checking the correct specialization of conditions.

> The phrase "instance of a supertype," which is used in the remainder of this chapter, always denotes an instance of a concrete object type (a leave of an inheritance tree), which is viewed as if it were an instance of a supertype.

Cardelli and Wegner [CW85] have presented a concept very similar to dynamic evaluation, called coercion. A *coercion* is a semantic operation, which is needed to convert an argument to the type expected by a function, in a situation that would otherwise result in a type error. Coercions can be provided statically, by automatically inserting them between parameters and arguments at compile-time, or may be determined dynamically by run-time test on arguments. For example, when a float is multiplied by an integer, the integer will be converted to a float, the multiplication performed, and a float calculated for the expression value.

8.1.8 Research Background

The specialization of object structure and object behavior is based mainly on the work of Kappel, Schrefl, and Stumptner in [KS94, Sch90, SS95, SS97]. The specialization of business rules is inspired by the work of Bichler, Kueng, and Schrefl in [BS95, KS95].

The work in the literature mentioned above is revised, reorganized, and extended for Active Object/Behavior Diagrams (1) to deal also with the specialization of the new concepts like permission, prohibition, and obligation, (2) to present the specialization of business rules modeled by means of Situation/Activation Diagrams, which have not been available in the above literature, and (3) to present a complete overview of the redefinition, the extension, and the refinement of object structure, object behavior, and business rules.

The major contribution of structure specialization (besides the adaption of rules dealing with the special kinds of relationships in Active Object/Behavior Diagrams) is that also the refinement of object structure is considered here (Section 8.2). The specialization of object structure is discussed in detail actually because the specialization of situation types is based on the specialization of object types, and the concepts established for object types will be reused for situation types.

The revision of Active Object/Behavior Diagrams in Chapter 7 requires also the revision of the specialization concepts for object behavior that have been presented in [KS94, Sch90, SS95, SS97]. In addition to the literature, we model also sufficient conditions besides necessary conditions by introducing the specification of permissions and obligations. Specializing necessary and sufficient conditions helps to develop a more comprehensible conceptual schema. Further, we consider also the specialization of textual conditions, whereas the above literature deals only with the specialization of preconditions and postconditions that are specified graphically by means of prestates and poststates (Section 8.3). The specialization of object behavior has to be treated because active and passive behavior are highly interrelated. Business rules depend on the passive behavior because they are often triggered upon events raised because of the execution of activties and because they invoke again activities if they are triggered.

The basic ideas about specializing business rules have already been mentioned in [BS95, KS95]. Section 8.4 presents a comprehensive set of rules dealing with the redefinition, the extension, and the refinement of business rules in Active Object/Behavior Diagrams.

8.2 Specializing Object Structure

Object structure is specialized by redefining conditions over properties at subtypes (redefinition), by adding properties at a subtype (extension), or by refining abstract properties with new properties.

Most object-oriented modeling languages allow to specialize object structure only by adding new properties at a subtype or by redefining the domain of inherited properties at a subtype. Object/Behavior Diagrams in their original version and also Active Object/Behavior Diagrams as described in Chapter 4 have provided the same concepts. This section presents also how an object's properties may be refined at an object type without necessarily considering inheritance.

In order to demonstrate the essential aspects of specialization, we continue the running example of loan applications, which has been introduced in Chapter 7. We distinguish two types of loans: short-term personal loans and long-term mortgage loans. A customer who applies for a loan represents a potential borrower. Borrowers of personal loans have to provide different information for the application than borrowers of mortgage loans. The following figures should be self-explanatory. The successive examples will point out only those parts of the figures that are relevant for understanding the details of specialization.

Example: Figure 8.1 shows the object-type diagrams of all objects involved in the running example about loan appliations.

The global object type is omitted in the subsequent examples because it is not necessary for demonstrating the essential aspects of specialization. To hold the running example as simple as possible, the local-to relationships to the global object type are omitted, too. If an object type is local to a different object type than the global one, the local-to relationship is depicted.

The basic inheritance relationship, the subtype-of relationship (cf. Section 8.1.1), establishes an inheritance graph, which consists of a set of unconnected inheritance trees. Object types that have no subtypes represent a degenerated tree, which has only a single node.

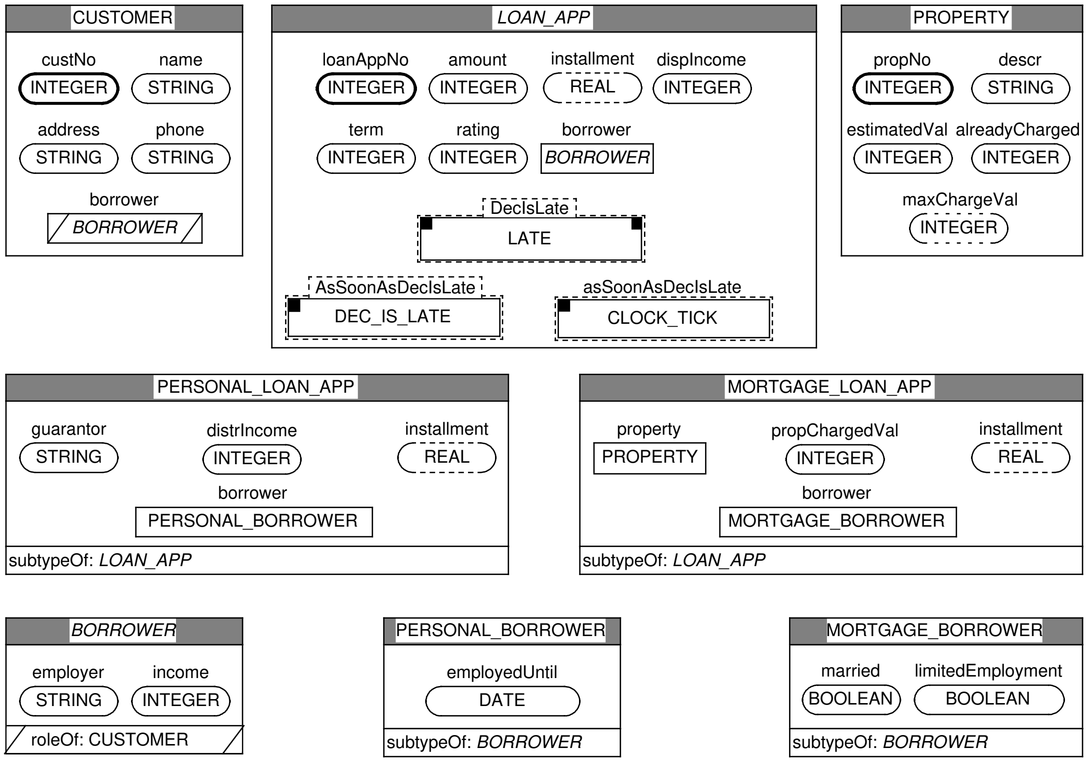


Figure 8.1: Object types

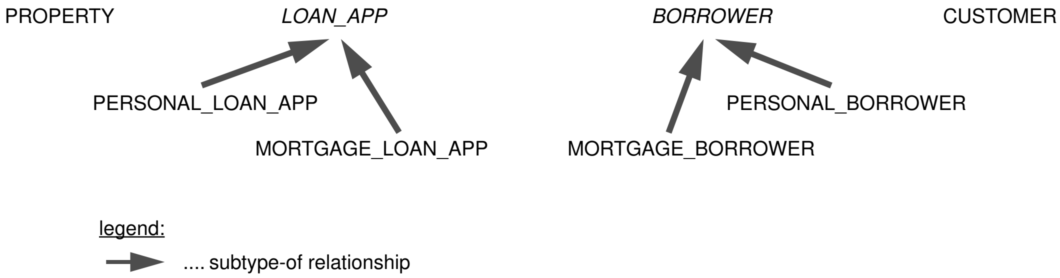


Figure 8.2: Inheritance graph: a set of unconnected trees

Example: Figure 8.2 depicts the inheritance graph with unconnected inheritance trees of the object types shown in Figure 8.1.

Remember the distinction between concrete and non-concrete object types in Section 8.1.1. Only object types that are defined explicitly to be concrete can be instantiated; only object types that represent "leaves" of an inheritance tree can be concrete. The names of non-concrete object types are depicted in a *slanted sans serif* type style. The names of concrete object types are depicted in a normal sans serif type style.

Example: Consider Figures 8.1 and 8.2. Object types *LOAN_APP* and *BORROWER* are non-concrete, all others are concrete.

The structure of an object is described by the properties that belong to its object type. Derived properties have an associated derivation expression, which is modeled by means of a query diagram (cf. Section 4.5). The following general rule is not specific to any kind of specialization, but applies to object types with derived properties in general.

- The *rule of existing query diagram for derived properties of concrete object types* requires that a concrete object type either defines a query diagram for each derived property or inherits a query diagram for each derived property from some supertype.

  Example: Concrete object types PERSONAL_LOAN_APP and MORTGAGE_LOAN_APP (Figure 8.1) have to provide a query diagram for their derived property installment. The query diagrams are not depicted here. The installment represents the monthly payment computed from the loan's amount, its term, and an interest rate, which differs for personal loans and mortgage loans.

The remainder of this section is structured as follows. The successive subsections present the redefinition, the extension, and the refinement of object structure. Finally, conflicts that may arise during the specialization of object structure are presented in the last subsection. Remember also the general rules defined in Sections 8.1.2 and 8.1.3 for each subsection.

Redefining Object Structure

A property (either relationship or attribute) is introduced at an object type as single-valued or multi-valued property, as key or "non-key" property, or as derived or "non-derived" property. These features of a property may never be changed at subtypes. The following rules say how constraints on properties may be redefined at a subtype and how query diagrams must be specified for derived properties.

- The *rule for redefinition of relationship domains* says that the type domain of an inherited relationship property may be restricted at the subtype only to a direct or indirect subtype of that domain type defined for it at the supertype.

  Example: Domain type *BORROWER* of general relationship borrower of object type *LOAN_APP* (Figure 8.1) is redefined at subtypes PERSONAL_LOAN_APP and MORTGAGE_LOAN_APP to domain types PERSONAL_BORROWER and MORTGAGE_BORROWER, respectively.

  Similar to necessary and sufficient preconditions, we could also distinguish between a necessary and a sufficient domain. At concrete object types, the necessary domain would have to match the sufficient domain. But currently, the property domain represents only a necessary condition in Active Object/Behavior Diagrams and therefore may be only strengthened at subtypes.

- The *rule of forbidden redefinition of attribute domains* says that the domain of attributes must not be redefined at subtypes because data types do not define any type hierarchy.

- The *rule of forbidden redefinition of query diagrams of derived properties* says that an inherited query diagram of a derived property must not be redefined at a subtype.

  To allow redefining inherited query diagrams at a subtype, the range of valid results (in addition to the type domain) should be given for the involved derived property because otherwise it cannot be determined whether a query diagram is redefined consistently. The range may be restricted at the subtypes.

  Example: Derived attribute installment should define a range of valid results besides the query diagram and the usual type domain at type *LOAN_APP*. Then, the query diagram at subtypes PERSONAL_LOAN_APP and MORTGAGE_LOAN_APP would have to produce a result being in the range defined at type *LOAN_APP*.

  Currently, Active Object/Behavior Diagrams do not provide the ability to specify a range for derived properties. Redefining query diagrams is forbidden to prevent their inconsistent specialization.

Extending Object Structure

The following rules have to be obeyed if object types are extended by introducing new properties at subtypes.

Example: Object type PERSONAL_BORROWER (depicted in Figure 8.1) inherits all properties from its supertype *BORROWER* and adds attribute employedUntil.

- The *rule of immediate definition for object-defining properties* requires that each object-defining property (has-constituent relationship, component-of relationship, and role-of relationship) of a subtype is already defined at a supertype. Thus, object-defining properties must not be added at a subtype but have to be introduced at root types.

  Example: The role-of relationship of object type PERSONAL_BORROWER is introduced already at supertype *BORROWER* and is inherited by subtype PERSONAL_BORROWER.

- The *rule of immediate definition for key properties* requires that each key property (attribute, role-of relationship, component-of relationship, and has-constituent relationship may be key if they are single-valued) is already defined at the root type. Like object-defining properties, key-properties may only be introduced at root types.

  Example: Key attribute loanAppNo, which is inherited by PERSONAL_LOAN_APP and MORTGAGE_LOAN_APP, is defined already at root type *LOAN_APP*.

Refining Object Structure

An object type's properties may be further refined during conceptual modeling to present the structure in more detail. Traditional Object/Behavior Diagrams have allowed properties to be "refined" only along the inheritance hierarchy by redefining the domain type of a relationship property at a subtype. For this, the relationship property is given a "dummy" domain type for which no properties are defined at first. Afterwards, the relationship property could be refined either by modifying this dummy domain type (by introducing its properties). If the property is refined at a subtype actually, it is sufficient to subtype also the dummy domain type (and add the new properties) and to redefine the domain type of the property to the newly created subtype of the previous domain type.

Example: Consider property address of object type CUSTOMER depicted in Figure 8.1. Assume address to be a general relationship with domain type ADDRESS instead of an attribute with data type STRING.

Object type ADDRESS may be subtyped by object type REFINED_ADDRESS, which defines the desired properties street, city, state, zipcode, and country. Then, property address is redefined at subtype CUSTOMER_WITH_REFINED_ADDRESS with domain type REFINED_ADDRESS.

The approach above does not represent pure refinement but only a workaround by extending the domain type. This solution always combines refinement with the redefinition of the property and the extension of a property's domain type.

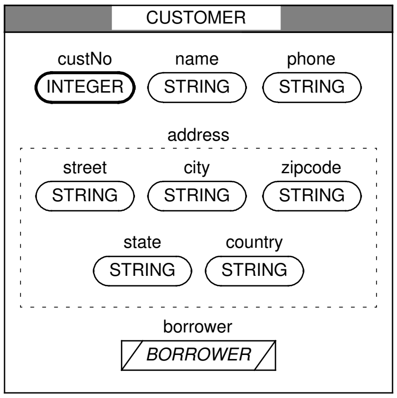


Figure 8.3: Refined object type CUSTOMER

To make refinement of object structure also available for Active Object/Behavior Diagrams, a new concept similar to "composite attributes" of ER diagrams [BCN92] is introduced. A property that the designer intends to refine is called an *abstract property*. An abstract property does not define a domain type and a multiplicity and may never be a key or a part of it.

An abstract property is refined by depicting the refining properties within the abstract property. The refining properties have their own name and may also be accessed directly via their name. Like all other abstract features, the names of properties (whether basic or abstract) have to be unique. An abstract property is depicted by a rectangle with a dashed border and with its name given above the rectangle. This representation differs from a derived single-valued general relationship only in the missing domain type.

Example: Consider the revision of object type CUSTOMER depicted in Figure 8.3. Property address represents an abstract property. It is refined by properties street, city, zipcode, state, and country depicted inside the rectangle with the dashed border.

Following rules restrict the refinement of object structure.

- The *refinement rule for object-defining properties* says that object-defining properties must not refine an abstract property. Abstract properties are always descriptive and not defining.

  Example: Abstract property address is not refined by an object-defining property.

- The *refinement rule for key properties* says that key properties must not refine an abstract property.

  Example: Abstract property address is not refined by a key property.

Resolving Conflicts

Conflicts that arise because of specializing object structure can be detected already at compile-time. Conflict resolution strategies at run-time need not be established. Nevertheless, naming conflicts, which arise during compile-time, are shortly touched on in the following.

Naming conflicts are usually due to multiple inheritance. Although Active Object/Behavior Diagrams support only single inheritance, they could cope with naming conflicts that are due to multiple inheritance. Features of an object type are assumed to have a unique identifier in Active Object/Behavior Diagrams. Each feature originates at a single object type, i.e., the object type at which it is introduced. Further, the rule of unique names of features (cf. Section 8.1.2) requires that all (inherited and introduced) features of an object type have a unique name. Since naming conflicts can be detected at compile-time, subtyping two object types by a single object type would not be allowed if naming conflicts arise.

8.3 Specializing Object Behavior

Analogously to object structure, object behavior can be specialized by redefinition (e.g., redefining pre- and postconditions), by extension (e.g., by adding new activities and states), or by refinement (e.g., by refining abstract activities and abstract states into subnets).

The concept of inheritance, implemented by means of a subtype-of relationship, represents the core concept also for specializing object behavior. Many object-oriented design methods like OMT [RBP+91], OOSA [EKW92], OOAD [Boo94] define the inheritance of object behavior only at object-type level by saying how an object's operations are inherited from the supertype and how their interface can be overridden at the subtype. They do not define details about how to inherit the object life cycle from the supertype. The original version of Object/Behavior Diagrams [KS91a] did not address the inheritance of object life cycles, too. Follow-on papers on Object/Behavior Diagrams deal already with the specialization of object life cycles [KS94, Sch90, SS95, SS97]. If the specialization rules defined there are obeyed by the designer, a consistent specialization of an object type's behavior is guaranteed. But the specialization rules consider only the specialization of the graphical elements (e.g., pre- and poststates of activities). None of the specialization rules addresses the specialization of the textually specified conditions (pre- and postcondition of activities). Since we have extended the definition of object behavior with permissions, prohibitions, and obligations in Chapter 7, the specialization of these concepts requires also a revision of the specialization rules for object behavior.

We distinguish between the specialization of object life cycles, activity specifications, and activity realizations. Although activity specifications represent only a detailed view of activities that are already present in the object life cycle, the specialization of life cycles and activity specifications is discussed separately because they are always depicted in different diagrams, in life-cycle diagrams and in activity-specification diagrams, respectively. Nevertheless, the diagrams and the specialization of both life cycles and activity specifications are highly interrelated. An activity realization implements an activity and has to stick to the pre- and postconditions provided by the activity specification.

As already mentioned, most object-oriented design methods define the inheritance of object behavior only by saying how an object's operations are inherited from the supertype and how their interface may be overidden at the subtype. For this, two important notions are used in the diverse literature: covariance and contra-variance. *Covariance* requires that input and output parameters of operations are restricted to subtypes when operations are redefined at a subtype. *Contra-variance* requires that input parameters are generalized to supertypes and output parameters are restricted to subtypes [ES94, SS95]. Covariance is favored by object-oriented design methods as it supports the concept of specialization in the tradition of semantic networks in Artificial Intelligence. Contra-variance is favored by type theoreticians as it supports static type checking in the presence of type substitutability [Car88, WZ88], which allows using an instance of a subtype whenever an instance of a supertype is expected. Meyer [Mey92a] follows the contra-variance rule for the specification of operations and, thereby, introduces the paradigm "design by contract" for the programming language Eiffel [Mey92b]: Subcontractors honor the prime contractor's promises in the original contract. In other words, what is promised to work for instances of the supertype should work also for instances of its subtypes. However, Eiffel does not ensure type substitutability since object types are specialized covariantly. Also Harris [Har91] suggests type substitutability to be the *only* specialization principle for programming languages. Obviously, both covariance and contra-variance have their benefits. Unfortunately, covariance and contra-variance are conflicting views of the world as remarked by Even and Sakkinen in [ES94].

So, which approach follow Active Object/Behavior Diagrams actually? Remember the distinction between necessary and sufficient preconditions in Section 7.2.1 and 8.1.3. Necessary preconditions are always specialized according to the covariance rule because they have to be strengthened downwards the inheritance hierarchy. Sufficient preconditions are always specialized according to the contra-variance rule because they have to be weakened downwards the inheritance hierarchy. Due to the division of the activity precondition into permission (represents a sufficient precondition) and prohibition (represents a negated necessary precondition), Active Object/Behavior Diagrams allow to follow both the covariance rule and the contra-variance rule. As a consequence, Active Object/Behavior Diagrams allow to model the business area at conceptual level similar to the tradition of semantic networks, but allow also to benefit from paradigms like type substitutability, which have usually been used for programming languages. Following the contra-variance rule means for Active Object/Behavior Diagrams that permissions may be only weakened at subtypes. Following the covariance rule means that also prohibitions may be only weakened at subtypes because prohibitions represent negated necessary preconditions, which may be only strengthened at subtypes. Postconditions may be only strengthened at subtypes because they represent necessary conditions.

Covariance and contra-variance are appropriate only for specializing activity specifications. Specializing life cycles requires different concepts, which obey the specialization of the connections between activities and states. Schrefl and Stumptner [SS95, SS97] define three different kinds of *behavior consistency* for specializing object life cycles: oberservation consistency, weak invocation consistency, and strong invocation consistency. *Observation consistency* requires that the behavior of each instance of a subtype can be observed as an instance of its supertype if the activities and states that are added at the subtype are neglected. *Weak invocation consistency* requires that any sequence of activities that can be performed on instances of a supertype can also be performed on instances of its subtypes, thereby, producing the effect expected at the supertype. *Strong invocation consistency* requires that instances of a subtype can be used in the same way as instances of the supertype, despite using or having used new activities of the subtype. Thus, it requires additionally to weak invocation consistency that activities added at a subtype do not infere with activities inherited from the supertype.

Observation-consistent specialization of object life cycles corresponds to covariant specialization of activity specifications. Weak invocation-consistent specialization of object life cycles corresponds to contra-variant specialization of activity specifications. Since we require both covariance and contra-variance for activity specialization, we require strong invocation-consistent specialization for object life cycles in Active Object/Behavior Diagrams. Strong invocation consistency corresonds to both covariant and contra-variant specialization of activity specifications.

In Section 7.2.3, we have tried to abstract from specialization when we defined consistency rules about the specification of permissions and prohibitions. Now, we discuss the general principles of specializing permissions and prohibitions. Only permission and prohibition are considered in the following discussion. Obligations represent the active behavior of object types and will be treated in Section 8.4. The following consistency rules are defined for all instances of an object type. We know that only concrete object types can have instances (that are created by instantiating the object type) and that only those object types can be concrete that represent leaves of an inheritance tree. If we speak of instances of a non-concrete object type, we mean the union of the instances of the concrete object types beneath the non-concrete object type in the type hierarchy.

- What is permitted for instances of a supertype, has to be permitted for instances of a subtype (*specialization rule for permissions*). This means that permissions may become only more general downwards the inheritance hierarchy. This rule follows from the contra-variance rule, which says that sufficient conditions may be only weakened at subtypes.

- What is forbidden for instances of a supertype, has to be forbidden for instances of a subtype (*specialization rule for prohibitions*). This means that prohibitions may become only more general downwards the inheritance hierarchy, too. This rule follows from the covariance rule, which says that necessary conditions may be only strengthened at subtypes. Since a prohibition represents a negated necessary precondition, it may be only weakened at subtypes.

- Performing an activity on instances of concrete object types is either permitted or forbidden (*definiteness rule for permissions and prohibitions at concrete object types*).

These consistency rules represent only the general idea about inheriting permissions and prohibitions. The details about how these rules are enforced by specific rules for extending object life cycles and for redefining activity specifications are presented later on.

Both the covariance rule and the contra-variance rule say that an activity's postcondition may be only strengthened at subtypes. Since the postcondition is not part of the introduced concepts permission and prohibition, it has not been mentioned above. Remember that two kinds of postconditions could be distinguished as well: necessary and sufficient postcondition. Since only necessary postconditions are of interest [WWMD91], the sufficient ones are omitted. Necessary conditions are always strengthened downwards the inheritance hierarchy.

The remainder of this section is organized as follows. Section 8.3.1 presents the specialization of object life cycles. Section 8.3.2 presents the specialization of activity specifications. Section 8.3.3 concludes the specialization of object behavior with some remarks on the specialization of activity realizations. Remember also the general consistency rules defined in Sections 8.1.2 and 8.1.3 for each subsection.

8.3.1 Specializing Object Life Cycles

An object life cycle specifies all states in which an object type's instances may reside and the available transitions between these states. A life cycle is depicted in a life-cycle diagram (cf. Section 4.4). Object life cycles are specialized either by extension or by refinement.

The following general rule is tailored neither to extension nor to refinement but deals with the specification of pre- and poststates of activities within the inheritance hierarchy only.

- The *rule of immediate definition for pre- and poststates* requires that an activity does not define a pre- and poststate at a subtype if both the activity and the state are defined already at a supertype. The pre- or poststate of the activity must be defined already at the supertype to which both the activity and the state belong.

  Example: Figure 8.4 shows an imaginary life cycle and illustrates that it is forbidden to connect an activity with a state if both belong already to the supertype.

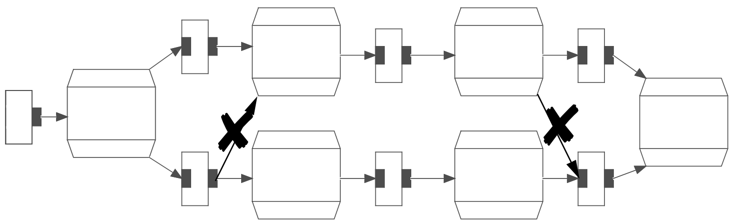


Figure 8.4: Immediate definition of pre- and poststates

The remainder of this section is structured as follows. The successive subsections show how object life cycles are extended or refined. Since the consistency rules given to ensure that an object life cycle is extended or refined consistently can be checked at compile-time, conflicts need not be resolved at run-time and therefore will not be discussed for the specialization of object life cycles.

Extending Object Life Cycles

An object life cycle is extended by adding new activities or states to the life cycle inherited from the supertype.

Example: In order to decide (i.e., to grant, to reject, or to forward an application) about a mortgage-loan application, the mortgage value must be determined. For this, the property's value must be estimated and an extract of the land register must be obtained. If a mortgage loan is granted, the mortagage has to be registered in the land register.

Consider the life cycle of object type *LOAN_APP* depicted in Figure 8.5 and the life cycle of the subtype MORTGAGE_LOAN_APP depicted in Figure 8.6. The inherited life cycle is extended by activities estimateProperty, obtainExtract, determineMortgageValue, and registerMortgage and by states propToEstimate, propEstimated, extractToObtain, extractObtained, valDetermined, toRegister, and registered.

Besides the rule of full inheritance [SS95] (cf. Section 8.1.2) and the rule of immediate definition for pre- and poststates, which is introduced above, the following rules, which have been introduced by Schrefl and Stumptner [SS95], must be obeyed to ensure strong invocation consistency if activities or states are added at subtypes.

- The *rule of forbidden parallel extension* requires that an inherited activity at the subtype consumes at most from those states from which the activity at the supertype consumes unless its permission is "false" at the supertype. This rule ensures a weak invocation-consistent specialization of object life cycles. Figure 8.8 illustrates what is forbidden by the rule of forbidden parallel extension.

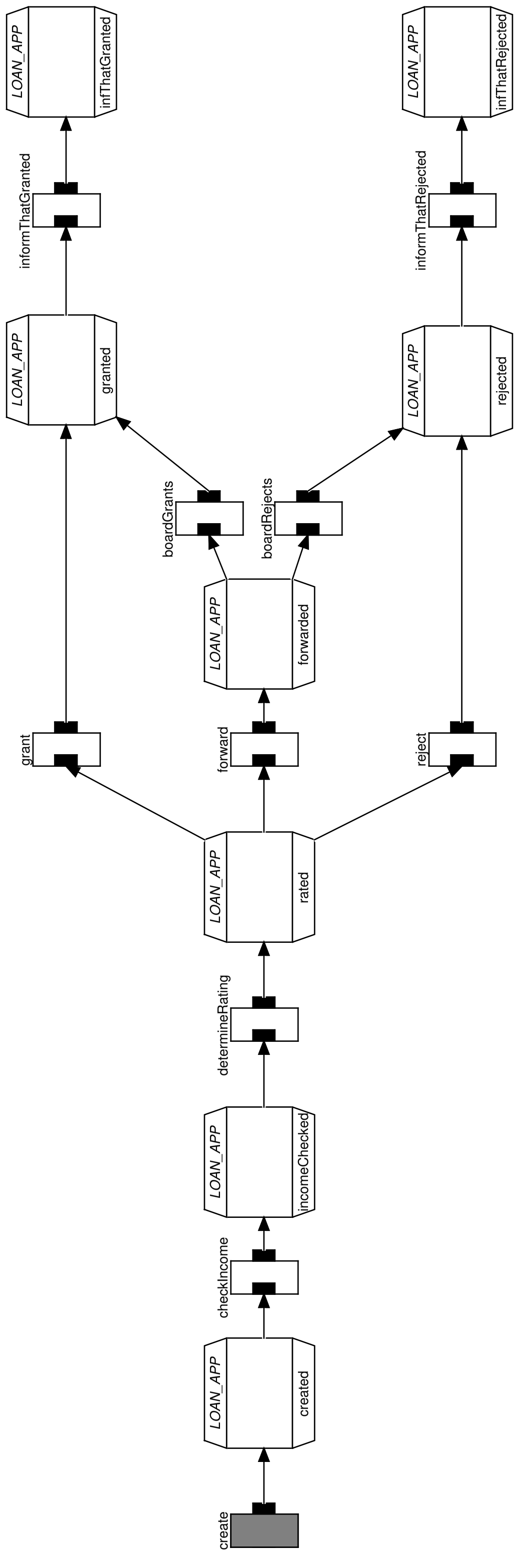


Figure 8.5: Life cycle of object type *LOAN_APP*

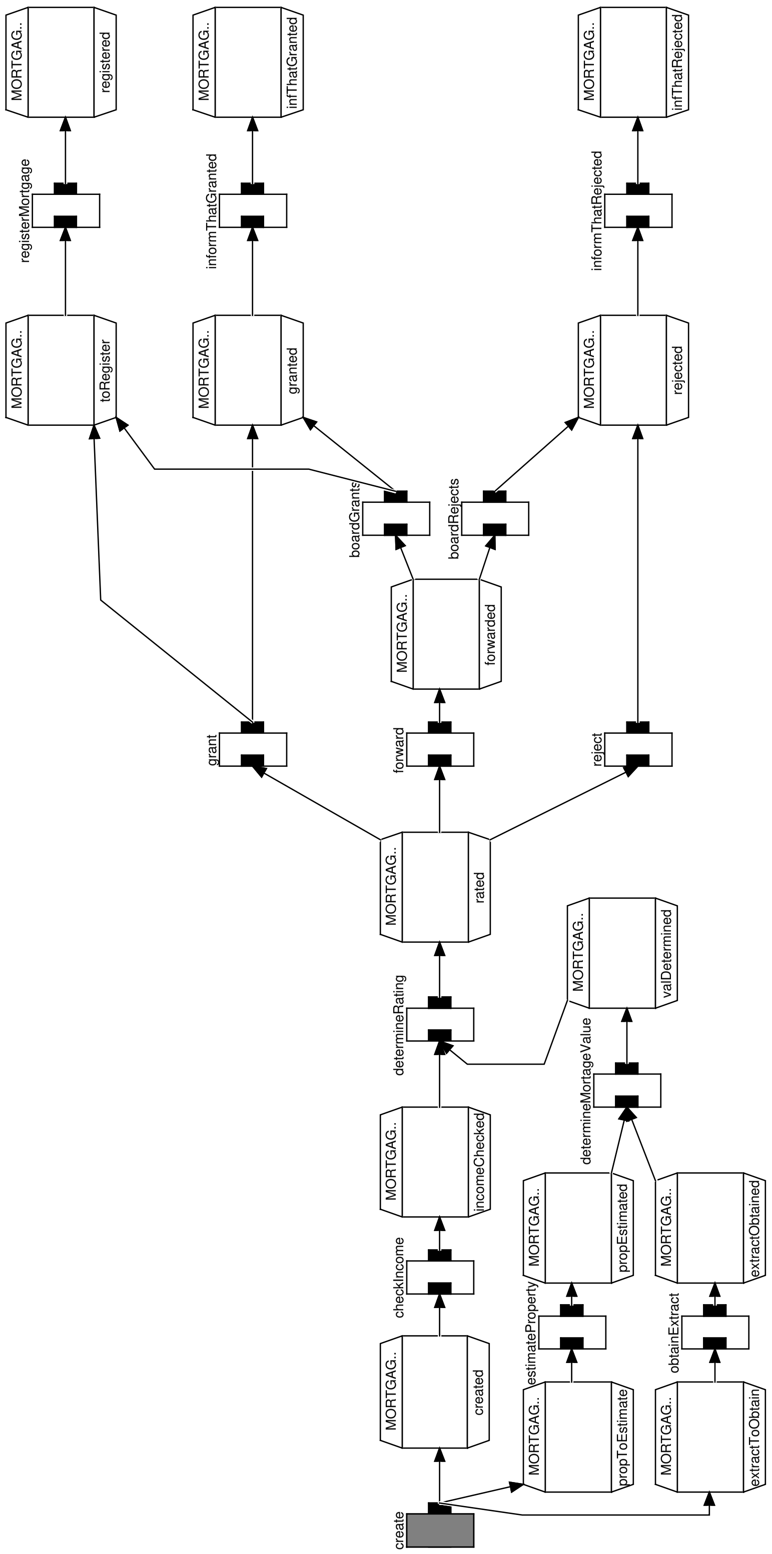


Figure 8.6: Life cycle of object type MORTGAGE_LOAN_APP

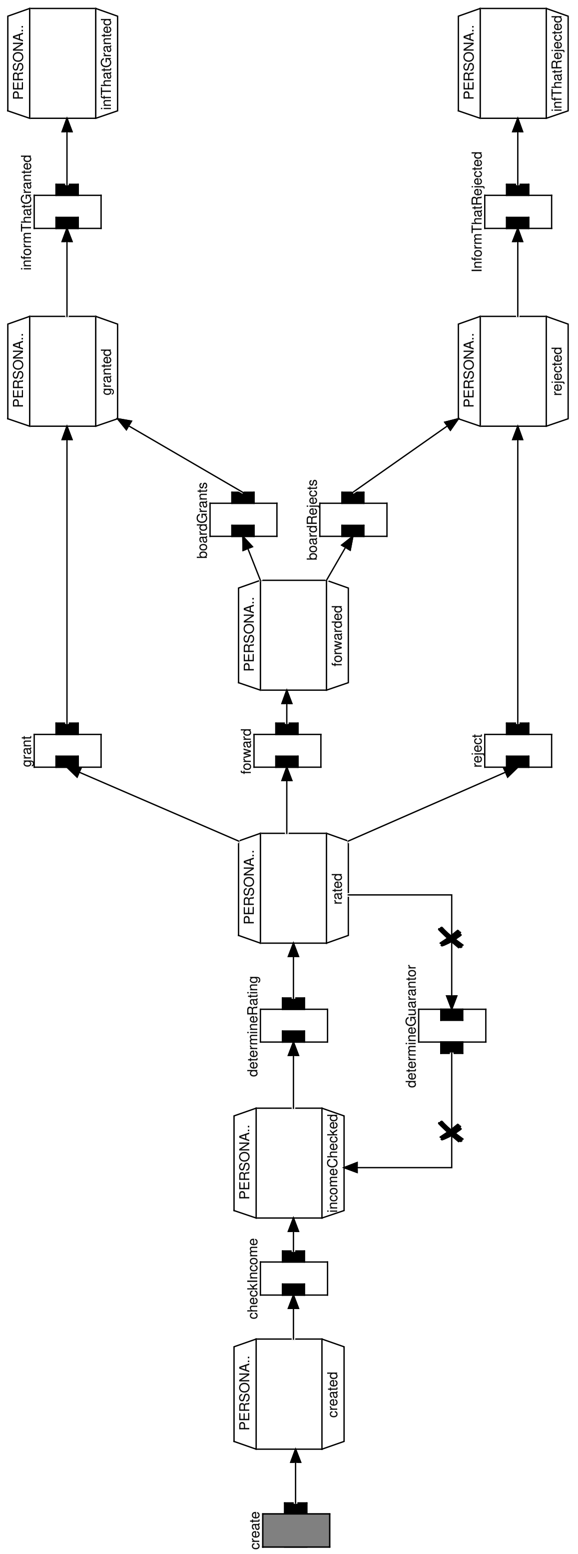


Figure 8.7: Life cycle of object type PERSONAL_LOAN_APP

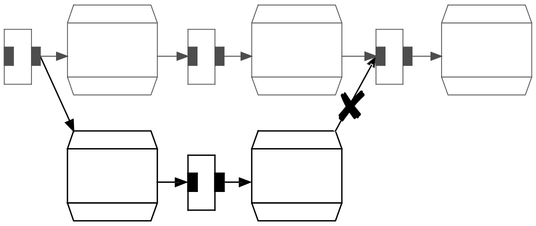


Figure 8.8: Parallel extension

In order to add a prestate to an inherited activity of some object type, the permission of the inherited activity has to be false at the supertype for the following reason. The permission to perform an activity is determined by the prestate condition (C^{pre}) and the textual permission (P). The prohibition is determined also by the prestate condition (C^{pre}) and by the textual prohibition (F). Hence, the prestate condition is used for both permission and prohibition. The permission is defined as $P \wedge C^{pre}$, the prohibition as $F \vee \neg C^{pre}$. If a prestate is added to an inherited activity at the subtype, C^{pre} is strengthened. If C^{pre} is strengthened, the permission is strengthened, too. And this contradicts the contra-variance rule and thus contradicts also weak invocation consistency. Nevertheless, if the permission is false, it cannot be strengthened any more, and therefore prestates may be added.

This dependence between the prestates specified in the life-cycle diagram and the textually specified permission in the activity specification diagram is evident since an activity-specification diagram is only a more detailed view on an activity of the life cycle.

Example: The life cycle of object type MORTGAGE_LOAN_APP (Figure 8.6) extends the object life cycle inherited from object type *LOAN_APP* (Figure 8.5) in a consistent way although the extension is parallel. The permission of activity determineRating must be false at supertype *LOAN_APP* (not shown).

Making a state that is introduced at a subtype a poststate of an inherited activity at a subtype is not forbidden by the rule of forbidden parallel extension. Both, the covariance rule and the contra-variance rule allow to strengthen the postcondition at subtypes. The poststate condition is a (conjunctive) part of the postcondition.

Example: The mortgage on the property has to be registered in the land register if the loan is granted (either by the clerk or by the board). Poststate toRegister is added to activities grant and boardGrants at object type MORTGAGE_LOAN_APP (Figure 8.6).

- The *rule of forbidden alternative extension* requires that an activity added at the subtype does not consume from a state already present at the supertype and does not produce into a state already present at the supertype. This rule ensures an

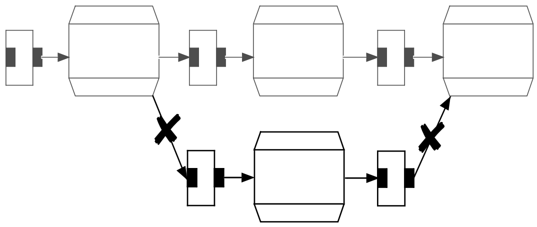


Figure 8.9: Alternative extension

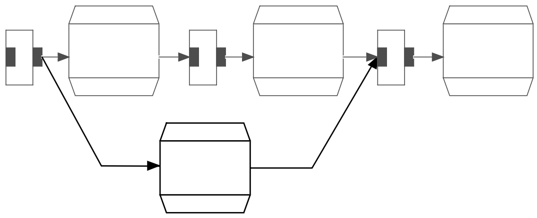


Figure 8.10: No substantial (real) extension

observation-consistent specialization of life cycles. Figure 8.9 illustrates what is forbidden by the rule of forbidden alternative extension.

Example: If the credit rating of a customer applying for a personal loan is rather low, a guarantor should be nominated in order to raise the rating. If a guarantor is determined, the application has to be rated once more.

The life cycle of object type *LOAN_APP* is extended at type PERSONAL_LOAN_APP (Figure 8.7) by activity determineGuarantor. Since this activity consumes from state rated and produces into state incomeChecked—both are inherited from supertype LOAN_APP—, the rule of forbidden alternative extension is violated.

- The *rule of partial inheritance* requires that an inherited activity at the subtype inherits all pre- and poststates from the activity at the supertype.

  Removing an inherited prestate from an inherited activity at a subtype would weaken the prestate condition and thus would strengthen the prohibition $F \vee \neg C^{pre}$, which is forbidden. Removing an inherited poststate from an inherited activity at a subtype would weaken the poststate condition, which is also forbidden.

Schrefl and Stumptner [SS95] call a life cycle L' of a subtype a *substantial extension* (or also *real extension*) of a life cycle L of a supertype if a state added in L' is not both prestate of an activity already present in L and poststate of an activity already present in L. Adding merely a state between two activities inherited from a supertype is no substantial extension because an intermediate activity is missing.

Figure 8.8 illustrates a substantial (parallel) extension of a life cycle. Figure 8.10 illustrates a non-substantial (parallel) extension. If a life cycle extension is not substantial, the rule

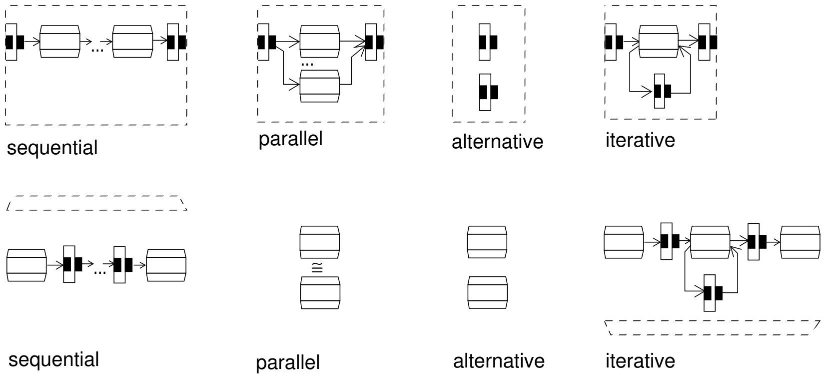


Figure 8.11: Refinement primitives [KS96b]

of forbidden parallel extension need not be obeyed in order to achieve a weak invocation-consistent and thus to achieve a strong invocation-consistent specialization.

Refining Object Life Cycles

Similar to complex systems, an object type's life cycle is defined usually in a top-down fashion. First, the designer tries to abstract from details and gives an overview of the possible life cycles of an object types' instances. Then, the given life cycle is considered in more detail by expanding activities and states into subnets. The following refinement primitives and consistency rules have been established already in [Sch90]. The modifications to Object/Behavior Diagrams during the development of Active Object/Behavior Diagrams and the revision of Active Object/Behavior Diagrams in Chapter 7 require a slight modification of the refinement concepts.

Basicly, object life cycles are refined such that an *abstract activity* is replaced by a subnet with activities as sources and as sinks, and that an *abstract state* is replaced by a subnet with states as sources and as sinks. The interface of an abstract activity cannot be specified by means of an activity-specification diagram. Further, an abstract activity cannot be implemented (it is replaced by subactivities and substates), and thus may not define an activity-realization diagram. Last but not least, the permission of an abstract activity is always false because it may never be executed.

Schrefl [Sch90] defines four refinement primitives based on structured diagrams and a set of rules based on semantic refinement criteria. The four refinement primitives are *sequential*, *parallel*, *alternative*, and *iterative* refinement. Figure 8.11 [KS96b] shows the four primitives for activities in the upper row and for states in the lower row. Most of them should be self-explanatory from their graphical representation. The difference between alternative and parallel refinement of states is that with the former, an object may reside only in one of the substates, and with the latter, an object must reside either in all or in none of the substates.

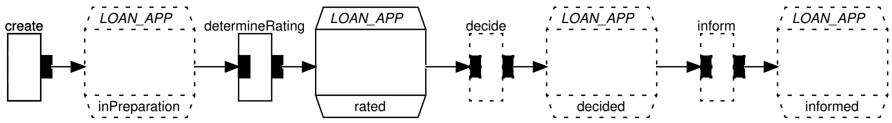


Figure 8.12: Initial life cycle of object type *LOAN_APP*

The sources of the refinement of a creation activity have to be creation activities, and its sinks have to be inner activities. The sources of the refinement of a removal activity have to be inner activities, and its sinks have to be removal activities. The sources and sinks of the refinement of an inner activity have to be inner activities.

Example: Consider the initial life cycle of type *LOAN_APP* in Figure 8.12. Also the initial object life cycle has to correspond to a refinement primitive. Assume the starting life cycle to consist only of a creation activity and of a poststate. Then, this poststate has to be refined according to a refinement primitive. Here, it is refined sequentially with state inPreparation as source and with state informed as sink.

Figure 8.13 shows the same life cycle after a few refinement steps. E.g., abstract state inPreparation is refined sequentially, activity inform is refined alternatively. The refinement primitives are applied repeatedly for the refinement of abstract activity decide.

If activities and states are refined according to the refinement primitives given above and if the following rules are obeyed, a consistent refinement is guaranteed (cf. [Sch90]).

- The *structured refinement rule* requires that no side-entries or side-exits exist. Only sinks and sources may be connected by arcs with foreign elements.

  Example: There are no side-entries in the life cycle of *LOAN_APP* (Figure 8.13). Only sources and sinks are connected to elements outside the refinement.

- The *pre- and poststate satisfaction rule* requires that each prestate of an abstract activity is prestate of each source in its refinement, and that each poststate of an abstract activity is poststate of each sink in its refinement. Of course, the poststate statisfaction rule may be applied only to refinements of inner activities and of removal activities, and the prestate satisfaction rule may be applied only to refinements of inner activities and of creation activities.

  Example: Consider abstract activity decide. Its basic prestate rated is also prestate of its sources grant, forwardToBoard, and reject.

- The *pre- and poststate refinement rule* requires that each prestate of a source activity of an abstract activity a either is also prestate of a or is a substate of some prestate of

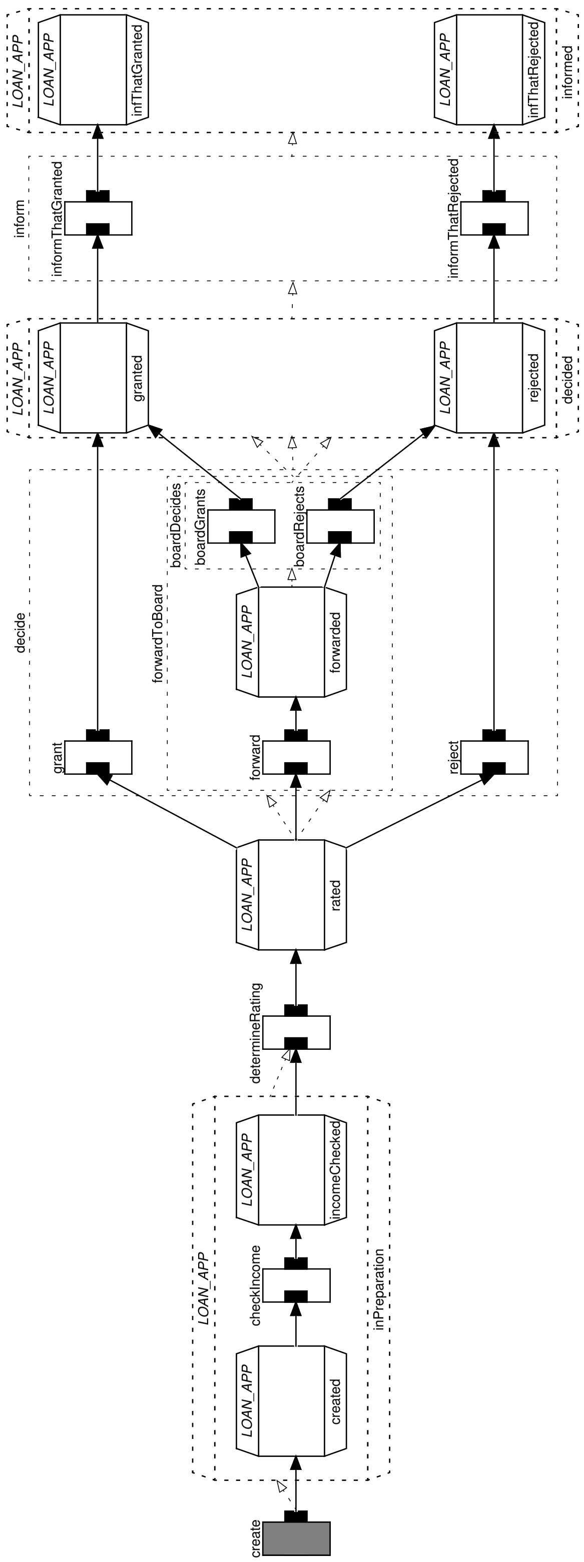


Figure 8.13: Refined life cycle of object type *LOAN_APP*

a. Analogously, each poststate of a sink activity of an abstract activity a must either be also a poststate of a or be a substate of some poststate of a. This rule ensures that the subnet does not define new pre- or poststates for its sources or sinks.

Example: Consider abstract activity decide again. Prestate rated of its source activities grant, forwardToBoard, and reject is also prestate of activity decide. Poststate granted of its sink activity grant is a substate of decided, which is the poststate of activity decide.

- The *insertion and removal specialization rule* says that an activity may insert an object into some source state of a state s only if it inserts the object also into state s and that an activity may remove an object from some sink state of a state s only if it removes the object also from state s. This rule ensures that the refinement obeys the object insertion and removal semantics of its superstate.

  If an abstract poststate s of an activity a is refined according to the parallel refinement primitive, a has to insert the object into each source state of s. Otherwise, if s is refined according to the alternative, sequential, or iterative refinement primitive, a has to insert the object into exactly one source state of s.

  Example: Activity create inserts a loan application into the only source state created of its abstract poststate inPreparation, which is refined sequentially.

  If an abstract prestate s of an activity a is refined according to the parallel refinement primitive, a has to remove the object from each sink state of s. Otherwise, if s is refined according to the alternative, sequential, or iterative refinement primitive, a has to remove the object from exactly one sink state of s.

  Example: Activity determineRating removes a loan application from the only sink state incomeChecked of its abstract prestate inPreparation, which is refined sequentially.

- The *up-to-basic refinement rule for arcs* requires that every basic activity with an abstract prestate s has at least one sink state of s as prestate. Analogously, every basic activity with an abstract poststate s must have at least one source state of s as poststate. This rule ensures that each basic activity has also defined basic pre- and poststates.

  Example: Basic activity create has the basic poststate created, which is a source state of abstract poststate inPreparation. Basic activity determineRating has the basic prestate incomeChecked, which is a sink state of abstract prestate inPreparation.

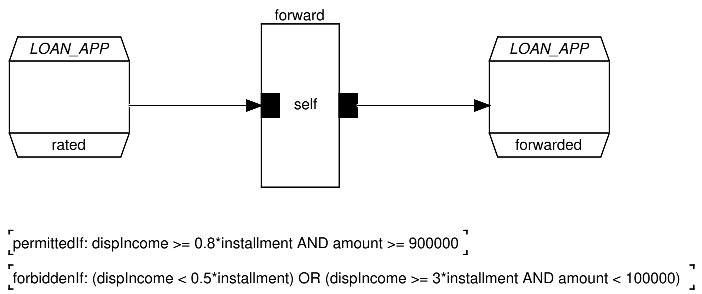


Figure 8.14: Specification of activity forward at object type *LOAN_APP*

8.3.2 Specializing Activity Specifications

The activitiy-specification diagram describes the formal input and output parameters of an activity as well as the conditions over these parameters and over the object on which the activity is invoked. These conditions are the permission, the prohibition, and the postcondition (cf. Sections 4.4.2 and Section 7.2.2).

The activity-specification diagram defines with the interface also the structure of the activity-events. These events denote the completion of an activity. Thus, specializing the activity's specification specializes also the associated event type's structure and is therefore similar to the specialization of object structure.

Properties, activities, and states are features of object types; similarly, ports are features of activity specifications. Specializing an activity specification means specializing the activity's ports. The general consistency rules defined for properties etc. in Section 8.1.2 can be applied also to ports:

- The *rule of unique port names* requires that the name of each port of an activity a at an object type O is unique for a, regardless of the port being inherited from a of some supertype of O or being introduced for a at O.
- The *rule of origin for ports* requires that if a port belongs to an activity a of several object types, it belongs also to a at their common supertype. This rule ensures that a port is always introduced for an activity at a single object-type.

Further, the following general rules must be obeyed if activity specifications are specialized.

- The *definiteness rule for permissions and prohibitions at concrete object types* requires that performing an activity is either permitted or forbidden for instances of a concrete object type. It is not allowed that neither the permission nor the prohibition is satisfied. Permission and prohibition have to be complementary.[3]

[3]See Section 7.2.2 for a definition of complementary conditions.

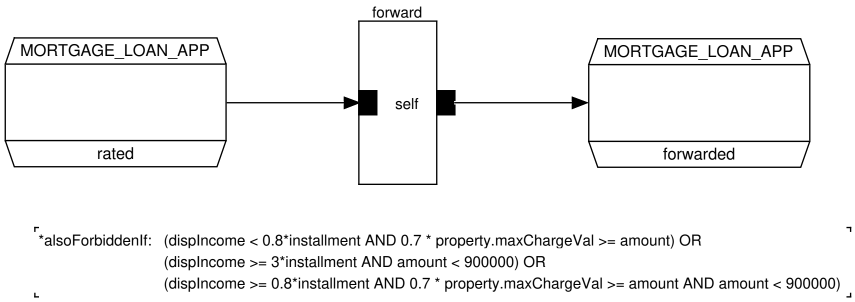


Figure 8.15: Specification of activity forward at object type MORTGAGE_LOAN_APP

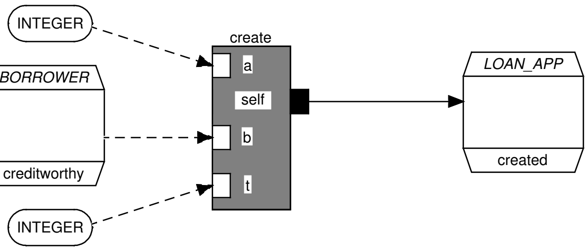


Figure 8.16: Specification of activity create at object type *LOAN_APP*

To simplify the specification of complementary conditions, only one kind of precondition may be given for concrete object types, either the permission or the prohibition. If only one of them is given, the other one is assumed to be complementary (to the complete condition and not only to the disjunctive term specified at the concrete object type). To distinguish an implicitly complementary condition from the default value "false" if a permission or prohibition is not specified, the label of the given one is prefixed with an asterisk, such that we have either the label *permittedIf or *forbiddenIf. If the activity is already introduced at some supertype and is redefined at a concrete subtype, the labels *alsoPermittedIf and *alsoForbiddenIf have to be given.

Example: Figure 8.15 illustrates the complementary definition of permission and prohibition of activity forward at concrete object type MORTGAGE_LOAN_APP, which is simplified and guaranteed by giving the label *alsoForbiddenIf. As a consequence, the permission is not depicted there, but it is assumed to be the complement of the complete prohibition instead.

- The *rule of immediate definition of port pre- and port poststates of an activity* requires that a port does not introduce a prestate or a poststate of an activity at an object type O if the port is specified already at the supertype of O and the state belongs to the domain given for the port at the supertype.

Example: Activity create of object types MORTGAGE_LOAN_APP (Figure 8.17) and PERSONAL_LOAN_APP (Figure 8.18) obey this rule. It receives the inputs a

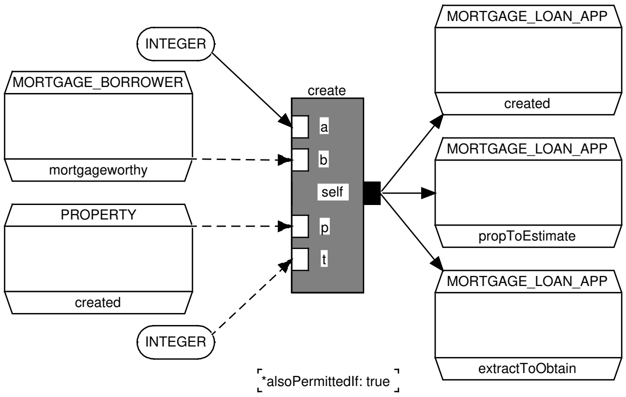


Figure 8.17: Specification of activity create at object type MORTGAGE_LOAN_APP

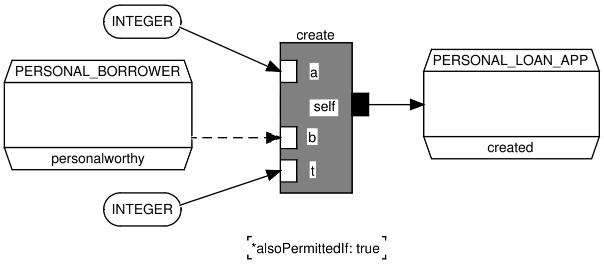


Figure 8.18: Specification of activity create at object type PERSONAL_LOAN_APP

(the loan's amount), b (the borrower), and t (the loan's term). In addition, a mortgage-loan application receives the input p, the mortgage property. The prestates mortgageWorthy and personalWorthy are available only for mortgage borrowers or personal borrowers, respectively.

We distinguish again between redefinition, extension, and refinement of activity specifications. The three following subsections treat these concepts.

Redefining Activity Specifications

The following rules have to be obeyed if the activity specification is redefined.

- The *rule for redefinition of object port domains* says that the domain type of an inherited object port may be restricted only to a direct or indirect subtype of that domain type defined for the inherited port at the supertype, provided that the inherited activity's permission is false at the supertype. This rule is in the line of covariance.

  The permission of the activity has to be false because the prestate condition C^{pre} is (together with the textually specified permission P) part of the permission, which is defined as $P \wedge C^{pre}$. Strengthening the prestate condition by restricting the port domain would also strengthen the permission (unless the permission is false at the supertype), which is forbidden.

Example: Port domain *BORROWER* of port b of activity create of object type *LOAN_APP* (Figure 8.16) is overridden with object types MORTGAGE_BORROWER and PERSONAL_BORROWER at the subtypes MORTGAGE_LOAN_APP (Figure 8.17) and PERSONAL_LOAN_APP (Figure 8.18).

Since the actual parameters assigned to an activity's port when the activity is invoked must be an instance of the port domain, indicating port domains represents a necessary condition, which may be only strengthened at subtypes.

- The *rule of forbidden redefinition of data port domains* says that the domain of data ports, which hold values of a data type, may not be redefined at a subtype because data types do not define any type hierarchy.

- The *rule of redefinition for port prestates* says that an additional prestate may be introduced for an inherited object input port at a subtype only if the inherited activity's permission is false at the supertype. This rule is in the line of covariance.

  The prestates of an activity's ports build the prestate condition (cf. Section 7.2.2). Adding a prestate, provided that the permission is false at the supertype, strengthens the prestate condition, which is allowed because the prestate condition represents a necessary condition. Weakening the prestate condition by removing a prestate is forbidden.

- The *rule of redefinition for port poststates* says that an additional poststate may be introduced for an inherited object output port at a subtype. This rule is in the line of both covariance and contra-variance.

  The poststates of an activity's ports build the poststate condition (cf. Section 7.2.2). Adding a poststate strengthens the poststate condition, which is allowed because the poststate condition represents a necessary condition. Weakening the postcondition by removing a poststate is forbidden.

  Note that introducing additional poststates of an inherited port at a subtype strengthens also the postcondition, which consists of the poststate condition (C^{post}) and of the textually specified postcondition (P) and is defined as $P \wedge C^{post}$.

- The *rule for redefinition of prohibitions* says that the prohibition of an inherited activity may be only weakened at a subtype. The prohibition represents a negated necessary precondition and, therefore, may be only weakened at subtypes. This rule is in the line of covariance.

Example: Figures 8.14 and 8.15 depict the specification of activity forward at object types *LOAN_APP* and MORTGAGE_LOAN_APP, respectively. The prohibition at object type MORTGAGE_LOAN_APP is weaker than the prohibition at object type *LOAN_APP* because term specified at object type MORTGAGE_LOAN_APP is enlarged disjunctively with the term specified at supertype *LOAN_APP*.

- The *rule for redefinition of postconditions* says that the postcondition of an inherited activity may be only strengthened at a subtype. The postcondition specified in Active Object/Behavior Diagrams represents a necessary postcondition, and thus may be only weakened at subtypes. This rule is in the line of both covariance and contra-variance.

- The *rule for redefinition of permissions* says that the permission of an inherited activity may be only weakened at a subtype. The permission represents a sufficient condition, and therefore it may be only weakened at subtypes. This rule is in the line of contra-variance.

  Example: Consider again the specification of activity forward at object types *LOAN_APP* and MORTGAGE_LOAN_APP, which are depicted in Figures 8.14 and 8.15, respectively. The permission at type MORTGAGE_LOAN_APP (the complete prohibition's complement) is weaker than the term specified at supertype *LOAN_APP*.

Remember from Section 8.1.3 that if a condition is redefined at a subtype, the condition introduced at the subtype is always enlarged implicitly by the inherited condition. If a condition may be only weakened at subtypes (like the permission and the prohibition), the condition at the subtype is enlarged disjunctively with the inherited condition. If a condition may be strengthened at subtypes (like the postcondition), the condition at the subtype is enlarged conjunctively with the inherited condition.

Extending Activity Specifications

An inherited activity specification may be extended by introducing new ports at a subtype, provided that the activity's permission is false at the supertype. If the specification of the activity interface is not completed, executing the activity must not be permitted.

Example: Consider the activity specification of activity create at object types *LOAN_APP* (Figure 8.16) and MORTGAGE_LOAN_APP (Figure 8.17). Ports a (the loans's amount), b (the borrower) and t (the term) are inherited from object type *LOAN_APP*. Port p (the mortgage property) is added at object type MORTGAGE_LOAN_APP. The permission of activity create is false at object type *LOAN_APP*. It is false by default because a permission is not given explicitly there.

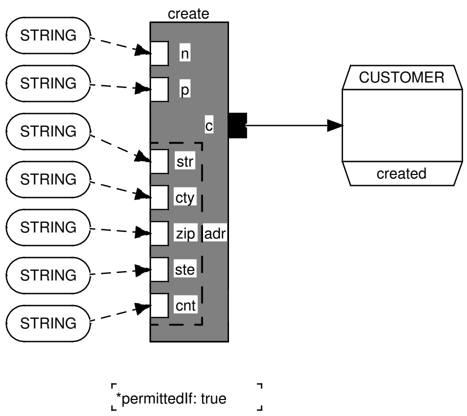


Figure 8.19: Refined specification of activity create at object type CUSTOMER

Refining Activity Specifications

Similar to an abstract property, an *abstract port* does not define any multiplicity, has no domain and therefore has no pre- or poststates. Since the primary port is determined by the object life cycle and since it cannot be modified by the activity specification, the primary port cannot be abstract.

Executing an activity a of an object type O with an abstract port that is locally basic at O (i.e., it is neither refined at O nor inherits a refinement from a supertype of O) is not permitted. Thus, its permission has to be false.

The general refinement rules defined for abstract features like property, activity, state, etc. may be applied also to abstract ports.

- The *refinement rule for abstract ports* says that once an abstract port is refined at an object type O, the refinement of the port may not be modified at some subtype of O. Yet the ports in the refinement may be further refined.
- The *rule of limited usability of abstract ports* says that abstract ports cannot be used in any expression.
- The *up-to-basic refinement rule for abstract ports at concrete object types* requires that each abstract port of an activity at a concrete object type O is either refined at O or inherits a refinement from some supertype of O.

Example: Figure 8.19 shows the activity-specification diagram of activity create at object type CUSTOMER. Remember how the customer's property address has been refined (cf. Figure 8.3). Abstract input port adr of activity create is refined in the same way. It is refined by ports str, cty, zip, ste, and cnt, which receive the values that are mapped to properties street, city, zipcode, state, and country, respectively. Since port adr is locally basic at object type CUSTOMER because it is refined there, executing activity create can be permitted.

Resolving Conflicts

It can be checked at compile-time whether the graphically specified pre- and poststate conditions of an activity-specification diagram are specialized correctly. As the redefined permissions and prohibitions are enlarged disjunctively and the redefined postconditions are enlarged conjunctively with the conditions inherited from the supertype, conflicts to the specialization rules cannot occur here. The rules for redefinition of permissions, prohibitions, and postconditions are satisfied automatically. Conflict resolution schemes are only necessary for the remaining consistency rules.

The permission and the prohibition of activities at concrete object types are usually reduced to one kind of precondition labeled either *permittedIf or *forbiddenIf (or *alsoPermittedIf and *alsoForbiddenIf if activities are redefined). Then, the definiteness rule for permissions and prohibitions at concrete object types is satisfied also automatically.

Nevertheless, the permission enlarged by *alsoPermittedIf may contradict the prohibition at the supertype because the complement of the complete permission at the concrete object type may be stronger than the prohibition at the supertype. Thus, the specialization rule for prohibitions is violated. This conflict is detected when the precondition of an activity a is evaluated and if the prohibition evaluated at the concrete object type (i.e., the negated permission) is not satisfied but the prohibition at the supertype is satisfied. As a consequence, the activity is not performed and an event is raised and added to the exception event class a_wrongProhibSpec. Analogously, the prohibition enlarged by *alsoForbiddenIf may contradict the permission at the supertype because the complement of the complete prohibition at the concrete object type may be stronger than the permission at the supertype. In this case, the raised exception event is added to the event class a_wrongPermSpec.

8.3.3 Specializing Activity Realizations

An activity is implemented by means of an activity-realization diagram (cf. Section 4.4.3). Since activity realization is not further considered in this thesis, its specialization is considered not in detail here.

The implementation of an activity can be overridden arbitrarily provided that the implementation sticks to the activity specification, in particular to the pre- and postcondition. Only a single rule has to be considered with the implementation of activities.

- The *rule of existing implementation for basic activities at concrete object types* requires that basic activities of a concrete object type O are implemented at O or inherit an implementation from a supertype of O.

This represents a general rule for implementing activities and does not constrain the specialization of activity implementations.

8.4 Specializing Business Rules

Business rules are represented as active behavior of objects. They are specified by means of activation diagrams. An activation diagram depicts a rule, which is composed of a triggering event class, of a rule condition, and of an activity, which serves as action. The event class and the activity must belong to the same object type to which the rule belongs. The rule's event class describes when the action has to be performed given that the rule condition is satisfied. The term "obligation," which has been introduced in Chapter 7, will be used as synonym for the term "business rule" in the following.

Rules are triggered when an event occurs. Situations (events and intervals) are collected in situation classes, which are not necessarily defined in combination with a rule because situation classes might be used only in situation diagrams and not as rule-triggering event classes. Currently, only event classes (and not interval classes) can serve as rule-triggering event classes. Rule conditions are assumed to be part of the rule because they cannot be defined separately and therefore cannot exist independently from a rule. Since situation classes and rules can be specified independently, they can be also specialized independently.

Since we treat situations as first-class objects, we distinguish between situation types and situation classes. Like object types, situation types are organized in inheritance hierarchies, which are defined by the subtype-of relationship. Each situation class has an associated member type; the situations collected by a situation class have to be direct or indirect instances of the situation class's member type.

Now, business rules are expressed by four features that may be specified and specialized individually: situation types, situation classes, situation diagrams, and rules. All these features may be specialized by redefinition, by extension, and by refinement. The basic relationship indicating specialization is the subtype-of relationship between object types. A subtype inherits besides properties, activities, and states also the situation classes, the situation diagrams, and the rules from its supertypes. Situation types are specialized like object types, i.e., by specializing their properties.

The specialization of obligations is constrained by a set of consistency rules, which ensure (if obeyed) that obligations are specialized consistently. Bichler, Kueng, and Schrefl have laid the foundations for obligation specialization in [BS95, KS95]. Business rules are called decision rules there, and the concept of consistent specialization is therefore called *decision consistency*. Since business rules are specified by means of activation diagrams in Active Object/Behavior Diagrams, the term *activation consistency* could be used instead of "decision consistency." However, a business expert is probably more familiar with decision consistency.

Obligations are assumed to be sufficient conditions with the meaning that an action (i.e., the rule action) has to be done when a certain condition over the history of time and the database is satisfied (i.e., when the rule-triggering event occurs and if the rule condition is satisfied). Since an obligation represents a sufficient condition, it may be only weakened at subtypes (cf. Section 8.1.3). Thus, decision consistency is in the line of "design by contract" [Mey92a] because it requires that if something to do is obligatory for an object viewed as if it were an instance of a supertype, it has to be obligatory also for the object viewed as if it were an instance of a subtype (cf. "dynamic evaluation" in Section 8.1.7).

The following general rules about the specialization of obligations and its components ensure decision consistency.

- If a situation occurs for an instance of a supertype, it has to occur also for an instance of a subtype (*specialization rule for situations*).
- If a rule is triggered by the occurrence of an event, the rule has to perform the action for instances of subtypes at least in those cases in which it is performed for instances of a supertype (*specialization rule for rule conditions*).

Permissions and prohibitions may be overridden only by enlarging them in a semantically restricted way to guarantee that they are weakened at subtypes according to the contravariance rule and to the covariance rule. Analogously, an inherited rule condition may not be simply replaced at a subtype but is enlarged disjunctively at the subtype. For more details see below.

The remainder of this section is structured according to the above distinction between four elements on which a business rule is based. Section 8.4.1 presents the specialization of situation types analogously to the specialization of object types. Section 8.4.2 presents the specialization of object types with situation classes, where situation classes represent another kind of properties. Section 8.4.3 presents the specialization of situation diagrams, which determine the extension of logical situation classes. Finally, Section 8.4.4 presents the specialization of rules, which implement obligations by using a situation class and a rule condition to fire an action.

8.4.1 Specializing Situation Types

Situations are treated as first-class objects. They are instances of a situation type and are collected in situation classes. The subtype-of relationship defines again the inheritance relationship between subtypes and supertypes. Situation type hierarchies are defined in same way as object-type hierarchies: situation types represent the nodes and the subtype-of

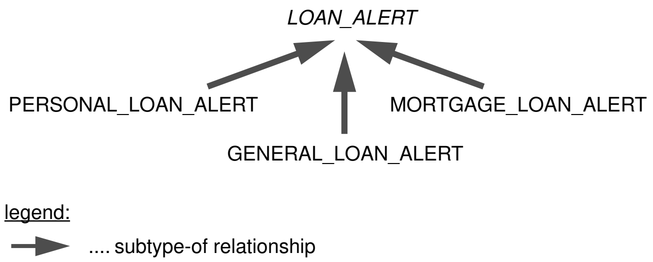


Figure 8.20: Inheritance tree of situation types

relationships represent the arcs (cf. Section 5.1). Further, we distinguish between concrete and non-concrete situation types. The former can be instantiated, the latter cannot be instantiated. The names of non-concrete situation types are depicted in a *slanted sans serif* type style corresponding to the names of non-concrete object types.

The running example about loan applications is extended by loans to show more details about the specialization of situations. After a loan application is granted (either by the clerk or the by the board) the loan is established. The loan object gets all necessary information from the loan application in order to be processed by the bank. The installment is computed only to derive the creditworthiness of the borrower. The loan need not be paid back exactly in the monthly installments computed for the loan application. Nevertheless, the bank wants to have an alert signaled if a borrower seems not to be able to pay back the loan at all or in the agreed term.

Example: Figure 8.20 presents the inheritance tree of non-concrete situation type *LOAN_ALERT* and its concrete subtypes PERSONAL_LOAN_ALERT, GENERAL_LOAN_ALERT, and MORTGAGE_LOAN_ALERT.

The graphical representation of situation types differs from that one of object types only in the small filled rectangle in the upper left corner of event-type diagrams and in the two small filled rectangles, one in the upper left and one in the upper right corner of interval-type diagrams. The subtype-of relationship is again depicted in a bar at the bottom labeled subtypeOf.

Example: Figure 8.21 depicts the diagrams of all object types and situation types extending the running example. Consider the situation types MORTGAGE_LOAN_ALERT, GENERAL_LOAN_ALERT, and PERSONAL_LOAN_ALERT, which are subtypes of situation type *LOAN_ALERT*. All object types are again local to the global object type. The local-to relationships are not depicted if they reference the global object type.

Remember from Section 5.1 the following differences between situation types and object types: Besides attributes and general relationships, a situation type may define contributing-situation relationships and a local-to relationship.

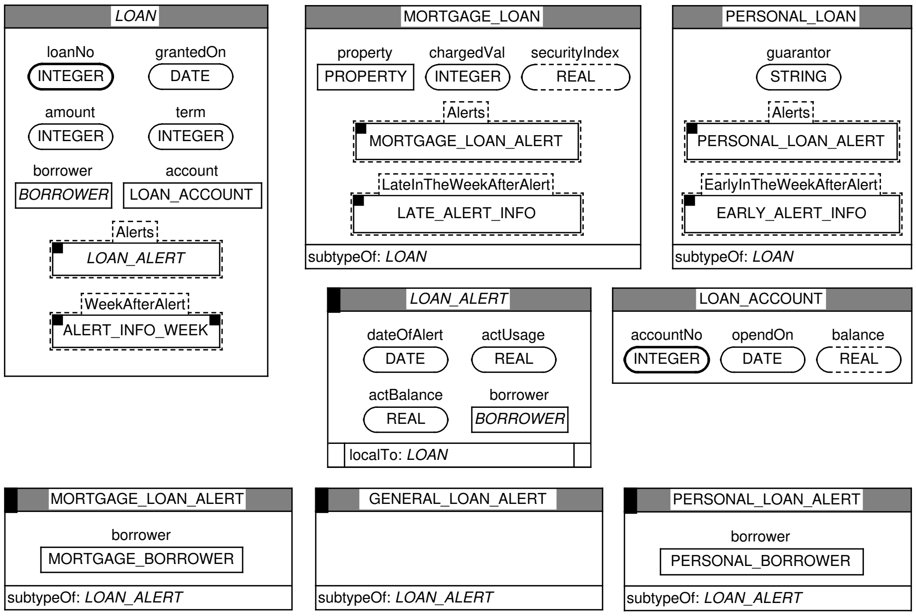


Figure 8.21: Further object types and situation types

Corresponding to the specialization of object types, we distinguish between the redefinition, the extension, and the refinement of situation types. Since situation types are very similar to object types, the following discussion is limited to the differences between the specialization of situation types and object types. Only those consistency rules that do not follow immediately from the specialization of object types will be shown. Remember also the general consistency rules about specialization from Sections 8.1.2 and 8.1.3.

The conflicts that arise because of the specialization of situation types correspond also to the conflicts that arise because of the specialization of object types. They can be detected at compile-time and need not be resolved at run-time. Therefore, conflict resolution schemes need not be discussed for the specialization of situation types.

An activity-event class collects activity events, which denote the completion of an activity (cf. Section 5.2). The structure of such an activity event is defined by the activity specification. An activity-event type defines a general relationship for each object port, an attribute for each data port of the activity, and a local-to relationship to the object to which the activity belongs. Specializing the activity specification (cf. Section 8.3.2) specializes implicitly the activity-event type, which will be discussed also in the following subsections.

Redefining Situation Types

The properties of situation types are introduced at a single situation type as a single-valued or a multi-valued property. The multiplicity of a property may never be changed.

The following consistency rules restrict the redefinition of a situation type's properties at a subtype.

- The *rule for redefinition of relationship domains* says that the type domain of an inherited relationship property (general relationship, contributing-situation relationship, local-to relationship) may be restricted only to a direct or indirect subtype of that domain type defined for the inherited relationship property at the supertype.

  Example: Domain type *BORROWER* of general relationship borrower at situation type *LOAN_ALERT* is specialized to MORTGAGE_BORROWER at situation type MORTGAGE_LOAN_ALERT (Figure 8.21).

Redefining the activity specification at subtypes redefines also the activity-event type associated with this activity.

Example: Consider activity create at object type *LOAN_APP* (Figure 8.16) and at object type MORTGAGE_LOAN_APP (Figure 8.17). Redefining the domain of port b of activity create at object type *LOAN_APP*, object type *BORROWER*, by subtype MORTGAGE_BORROWER redefines also the domain of the general relationship b of the associated activity-event type MORTGAGE_LOAN_APP_ACTIVITY_CREATE, which inherits b from its supertype LOAN_APP_ACTIVITY_CREATE (not depicted).

Extending Situation Types

Like object types, situation types can be extended arbitrarily by introducing properties at subtypes provided that the following consistency rule, which corresponds to the rule of immediate definition of object-defining properties, is obeyed.

- The *rule of immediate definition of local-to relationships* requires that exactly one local-to relationship is defined for each situation type that constitutes the root of an inheritance tree. Local-to relationships must not be added at a subtype.

  Example: The local-to relationship of general loan alerts, mortgage-loan alerts, and peronal-loan alerts is already introduced at root type *LOAN_ALERT* (Figure 8.21).

Extending an activity specification by introducing ports at subtypes extends also the activity-event type associated with the activity.

Example: Consider the extension of activity create at type MORTGAGE_LOAN_APP (Figure 8.17). Introducing port p with domain type PROPERTY extends also the associated activity-event type MORTGAGE_LOAN_APP_ACTIVITY_CREATE (which is a subtype of LOAN_APP_ACTIVITY_CREATE) with a general relationship p with domain type PROPERTY (not depicted).

Refining Situation Types

The structure of situation types may be refined in the same way as the structure of object types. An *abstract property* of a situation type is also neither single-valued nor multi-valued, does not define a domain type, and may never be key or a part of it. The refinement of situation types is constrained by the following consistency rule, which corresponds to the refinement rule for object-defining properties.

- The *refinement rule for local-to relationships* says that a local-to relationship must not refine an abstract property of a situation type.

Examples on the refinement of situation types are omitted as they are analogous to examples on the refinement of object types.

Refining an activity specification refines also the associated activity-event type.

Example: Consider the refinement of activity create of type CUSTOMER depicted in Figure 8.19. Refining the port adr refines also the abstract property adr of the associated activity-event type CUSTOMER_ACTIVITY_CREATE (not depicted).

8.4.2 Specializing Object Types with Situation Classes

The specialization of object structure has been discussed already in Section 8.2 but without situation classes as properties of object types. In this section, we do not repeat all rules about the specialization of object structure but present only rules for the specialization of situation classes of object types.

Remember the three different kinds of situation classes as defined in Section 5.2: situation class with local members, situation class with indirectly local members, and situation class with non-local members. The term "situation class" used in the following refers to either one of the three kinds of situation classes. Therefore, the rules defined for situation classes have to be applied to each kind.

Specializing an activity specification specializes the associated activity-event type defined with the activity (cf. Section 8.4.1). For example, redefining the port domain of an activity redefines also the domain of the corresponding relationship of the associated activity-event type. Specializing object behavior has an effect on the activity-event classes and the state-interval classes that exist for each activity and for each state (cf. Section 5.2). For example, extending the behavior by introducing a new activity introduces also an associated activity-event class. An *activity-event class* (an event class collecting events that denote the completion of an activity) exists for each basic activity. A *state-interval class* (an interval class collecting intervals that denote the time period for which an object resides in a state) exists for each basic state. Situation classes for abstract activities and abstract states will be introduced below. The impact of specializing the object life cycle on these situation classes will be also discussed shortly in the following subsections, which treat the redefinition, the extension, and the refinement of object types with situation classes.

Redefining Object Types with Situation Classes

Like relationships and attributes, situation classes (situation classes with local members, situation classes with indirectly local members, and situation classes with non-local members) are introduced at an object type as single-valued or multi-valued property.

Following rules apply to the redefinition of situation classes.

- The *rule for redefinition of member types of situation classes* says that the member type of an inherited situation class may be restricted only to a direct or indirect subtype of that member type defined for the inherited situation class at the supertype.

  Example: Consider Figure 8.21. Member type *LOAN_ALERT* of situation class Alerts of object type *LOAN* is redefined at subtypes PERSONAL_LOAN and MORTGAGE_LOAN to member types PERSONAL_LOAN_ALERT and MORTGAGE_LOAN_ALERT, respectively.

- The *rule for redefinition of situation diagrams of logical situation classes* requires that each situation (event or interval) that occurs for an object viewed as an instance of a supertype occurs also for the object viewed as an instance of a subtype, i.e., the set of situations selected or generated for an object by the situation diagrams given with a situation class at a supertype is a subset of the situations selected or generated for the object by the situation diagrams given with the situation class at a subtype.

  This rule is enforced in that, analogously to permissions, where only disjunctive terms have been added at subtypes, situation diagrams introduced at subtypes for some situation class are added to the situation diagrams inherited for that situation class.

  Since situation diagrams determine the extension of logical situation classes, they have to be specialized in such a way that this rule is ensured. The specialization of situation diagrams is discussed in Section 8.4.3.

Extending Object Types with Situation Classes

Like attributes and relationships, situation classes may be introduced at subtypes in order to extend an object type. The same rules as defined for attributes and relationships apply also to situation classes.

Extending an object life cycle by introducing activities and states at subtypes introduces also the associated activity-event classes and state-interval classes at the subtype.

Example: The life cycle of object type *LOAN_APP* (Figure 8.5) is extended at subtype MORTGAGE_LOAN_APP (Figure 8.6) with activity registerMortgage and with state registered. As a consequence, activity-event class RegisterMortgage_Events and state-interval class Registered_Intervals are added, too.

Refining Object Types with Situation Classes

Situation classes cannot be refined themselves, but they may be part of a refinement, i.e., they may refine an abstract property.

As already mentioned above, an activity event is raised when executing a basic activity is completed and a state interval is generated when an object enters a state; the state interval ends when the object leaves the state.

Situation classes exist also for abstract activities and object states, which we call abstract activity-interval classes and abstract state-interval classes to distinguish them from basic activity-event classes and basic state-interval classes mentioned above. The terms "abstract" and "basic" denote if such a situation class corresponds to an abstract or to a basic feature. Opposed to other abstract features like abstract properties, abstract activity-interval classes and abstract state-interval classes may be used within expressions as they represent actually interval classes for abstract activities and states but the classes themselves are not abstract.

An *abstract activity interval* covers the time period between "starting the abstract activity" by invoking one of its source activities and "ending the abstract activity" by invoking one of its sink activities. The situation class handled for an abstract activity contains intervals and not events as the situation class handled for a basic activity. Since the interface of an abstract activity is not specified by means of an activity-specification diagram, an abstract activity does not define any structure (attributes representing data ports, general relationships representing object ports) besides the occurrence period and the local-to relationship, which refers to the object on which the abstract activity is "executed." The interface of basic activities is specified by means of an activity-specification diagram. Their event type defines an attribute for each data port and a general relationship for each object port besides the local-to relationship and the attribute for the occurrence time.

Example: Consider the refined life cycle of object type *LOAN_APP* in Figure 8.13. There exists an abstract activity-interval with the name Decide_Intervals, which corresponds to the abstract activity decide. Such an interval is opened, for example, if the application is forwarded to the board and is ended when the board grants or rejects the application.

If an abstract activity is refined according to the alternative refinement primitive and all its subactivities are either basic or again refined according to the alternative refinement primitive, an event class could collect the events signaling the completion of a subactivity instead of an interval class. Nevertheless, to treat all abstract activities uniformingly, an interval class is associated also with such an activity.

An *abstract state interval* covers the time period between the start of a basic state interval of a source state and the end of a basic state interval of a sink state.

Example: Consider again the refined life cycle of object type *LOAN_APP*. An abstract state-interval class InPreparation_Intervals exists for state inPreparation. The intervals start with the start of state created and end with the end of state incomeChecked.

Resolving Conflicts

A violation of the rule for redefinition of situation diagrams of logical situation classes cannot occur, since situation diagrams of a situation class are only extended by adding situation diagrams at subtypes.

Activity events and state intervals occur for an instance of a subtype at least at those points in time at which they occur for the instance of a supertype. This is ensured by the rules for redefinition of permissions, prohibitions, and postconditions (cf. Section 8.3.2). For example, if executing an activity on an object o is permitted if o is viewed as instance of object type O, it has to be permitted also if o is viewed as instance of object type O', a subtype of O.

8.4.3 Specializing Situation Diagrams

Situation diagrams, which have been introduced in Chapter 5, determine the extension of the logical situation class, with which they are associated. Situation diagrams represent a condition over the history of time and the database. Obligations use situation classes to determine the points in time when something is obligatory to do. As obligations represent sufficient conditions, situation diagrams, which determine the extension of the situation classes used by obligations, have to represent also sufficient conditions over the history. Consequently, situation diagrams may be weakened only at subtypes such that each situation that occurs for an object viewed as instance of a supertype occurs also for the object viewed as instance of a subtype. This is ensured in that situation diagrams of a situation class are only extended by adding situation diagrams at subtypes. Moreover, situation diagrams may be incrementally defined by introducing abstract situation diagrams at subtypes. Abstract situation diagrams are merely a design aid and are actually never evaluated.

Situation diagrams, which have to obey the member type of the situation class with which they are associated and the rule for redefinition of situation diagrams of logical situation classes, are similar to activity realizations, which have to obey the activity specification and the redefinition rule for permissions, prohibitions, and postconditions.

Several situation diagrams may be specified for a single logical situation class. Since situation diagrams represent features of object types like properties, activities, or states, and since several situation diagrams of a logical situation class have to be distinguished for specialization, they have to be referred to by a unique name. In the following, we consider only situation classes with a single situation diagram. Therefore, names of situation diagrams are omitted. Nevertheless, if more than one situation diagram are associated with a situation class, they can be assumed to have a unique identifier like a unique number.

Since only concrete situation types may be instantiated, a generating situation diagram may only define a concrete situation type as result type. Of course, the result type has to be either the member type of the situation class with which the situation diagram is associated or a direct or indirect subtype of the member type.

Example: Figure 8.22 depicts the situation diagram defined for situation class Alerts at type *LOAN*. The situation diagram generates a events of the concrete type GENERAL_LOAN_ALERT. The semantics of the situation diagram will be explained below.

Extending Situation Diagrams

As an object type can be extended by introducing additional properties at a subtype, it may also be extended by introducing additional situation diagrams at subtypes. Introducing new situation diagrams at subtypes ensures implicitly the rule for redefinition of situation diagrams of logical situation classes because the situation diagrams introduced at the subtype can raise only additional situations.

Example: Figure 8.22 depicts the situation diagram defined for situation class Alerts at type *LOAN*. The situation diagram generates an event of type GENERAL_LOAN_ALERT when 75 percent of the agreed term $(0.75 * t * 4WK)$[4] have passed and if more than 50 percent of the loan amount are still not payed back $(b/a > 0.5)$.[5]

[4] A loan's term is always given in a number periods of four weeks.

[5] The amount and the balance hold negative numbers. The amount represents the maximum amount which the borrower is allowed to overdraw the loan account. The account is usually negative until the borrower has not completely paid back the loan.

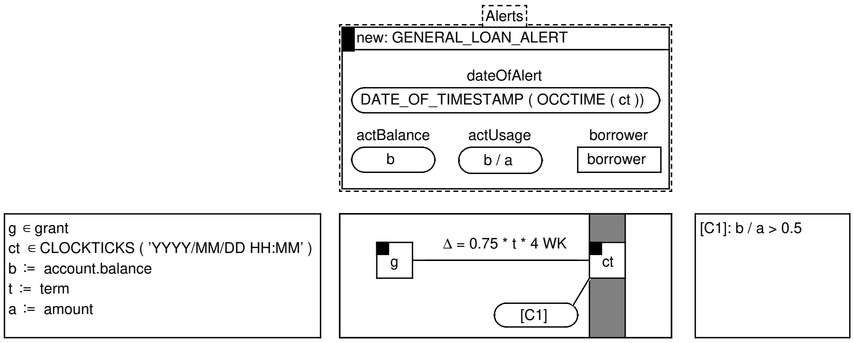


Figure 8.22: Situation diagram for situation class Alerts at object type *LOAN*

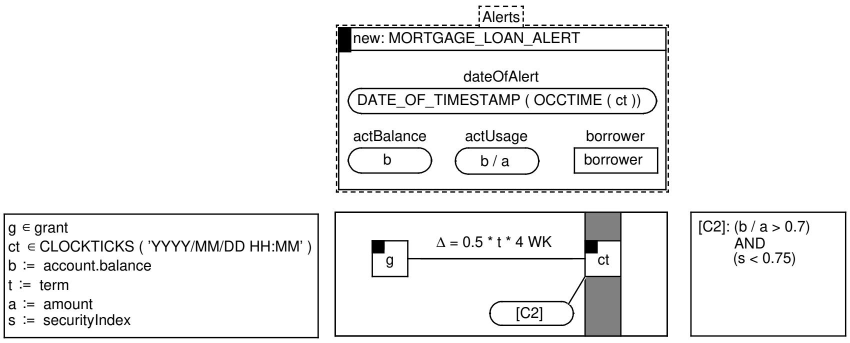


Figure 8.23: Additional situation diagram for situation class Alerts at object type MORTGAGE_LOAN

Figure 8.23 depicts an additional situation diagram of situation class Alerts at subtype MORTGAGE_LOAN. The situation diagram raises an additional alert for a mortgage loan for a mortgage loan when already half of the term ($0.5 * t * 4WK$) have passed, if more than 70 percent of the loan amount are still not payed back ($b/a > 0.7$), and if the security index is lower than 0.75 ($s < 0.75$). The security index of a mortgage loan is a derived attribute, which is computed as the ratio of charged value of the mortgage property to the loan amount (ABS(chargedVal/amount)).

Although situation diagrams may be only added at subtypes and, therefore, ensure the rule for redefinition of situation diagrams of logical situation classes for a logical situation class S_1, this rule may be violated by another logical situation class S_2. The situation diagram associated with S_2 might select or generate situations for S_2 by requiring some or all situations of S_1 not to occur. If the number of situations in S_1 increases downwards the inheritance tree, the number of situations in S_2 decreases, and, thus, S_2 violates the rule for redefinition of situation diagrams of logical situation classes.

Therefore, we forbid to "negate" the occurrence of logical situations in situation diagrams. The occurrence of situations could be negated by specifying logical conditions, in which the boolean function NOT is applied directly or hidden to conditions over logial situations or over its properties. For example, a nested situation diagram is "negated" (i.e., required to return an empty set) by means of NOT [X] and, thereby, the nested situation diagram with name [X] is required to return an empty set. Further, the occurrence of logical situations could be negated by applying a condition on the temporal adjacency of logical situations, which allows to formulate conditions like "the first response after the last offer" or any response after the last offer." This concept hides negation of the occurrence of the involved logical situations and, thus, is also forbidden if applied to logical situations. Of course, the occurrence of basic situations may be "negated."

Refining Situation Diagrams

Refining situation diagrams means to specify step by step when an event exactly occurs or over which time period an interval exactly lasts. The number of points in time at which an event might occur or over which an interval might last is reduced by the refinement.

An *abstract situation diagram* returns a set of *abstract intervals*, which define no structure except the occurrence period. The occurrence time or the occurrence period of the situations resulting from situation diagrams that refine an abstract situation diagram have to lay within the occurrence period of the abstract intervals.

The graphical representation of abstract situation diagrams differs from conventional situation diagrams as follows: In an abstract selecting situation diagram, the border of the situation box whose situation variable determines the abstract result situations is dashed, and the result box does not contain a situation variable. In an abstract generating situation

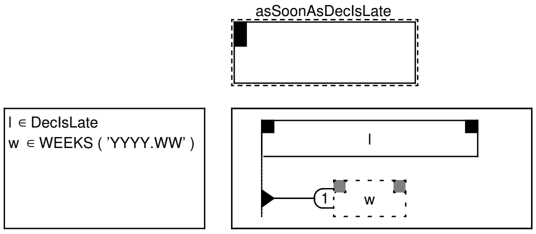


Figure 8.24: Abstract situation diagram for situation class asSoonAsDecIsLate at object type *LOAN_APP*

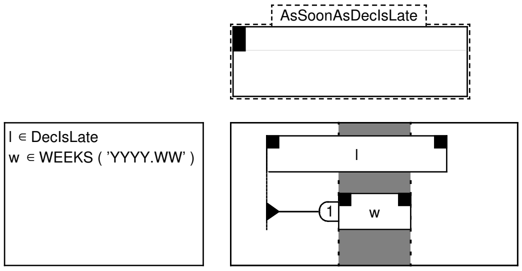


Figure 8.25: Abstract situation diagram for situation class AsSoonAsDecIsLate at object type *LOAN_APP*

diagram, the border of the occurrence bar, which generates the abstract result situation is dashed. Since abstract situations are of no particular type but define only an occurrence period, no situation type is depicted within the result box. Selecting and generating abstract situation diagrams may be used for all kinds of situation classes (i.e., situation classes with local, indirectly local, or non-local members) because they select or generate only imaginary (abstract) situations.

Example: Consider the abstract selecting situation diagram of situation class asSoonAsDecIsLate at object type *LOAN_APP* in Figure 8.24. The situation box of the resulting week w is depicted with a dashed border, and no situation variable is depicted in the result box. This abstract situation diagram describes that in a further refinement of this situation diagram the resulting situation has to occur within the first week after the decision about a loan application has become late. Assume the decision about a loan application to be late when either two or five weeks (depending on whether the borrower has applied for a personal loan or for a mortgage loan) have passed since the credit rating has been determined.

Consider the abstract generating situation diagram of situation class AsSoonAsDecIsLate at object type *LOAN_APP* in Figure 8.25. The occurrence bar is depicted with a dashed border, and no situation type is depicted in the result box. This situation diagram defines the same semantics as above except that it does not select but generate the abstract situations.

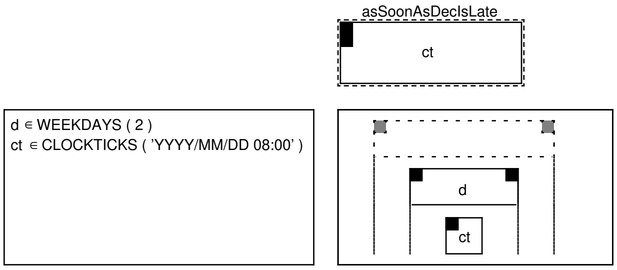


Figure 8.26: Basic situation diagram for situation class asSoonAsDecIsLate at object type PERSONAL_LOAN_APP

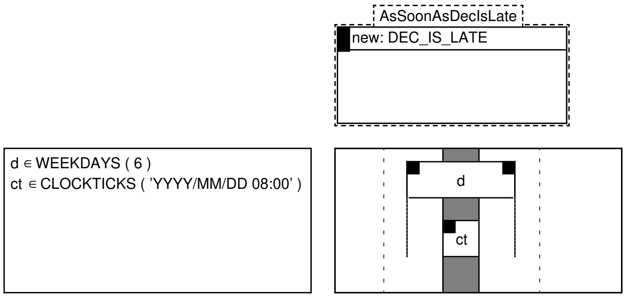


Figure 8.27: Basic situation diagram for situation class AsSoonAsDecIsLate at object type MORTGAGE_LOAN_APP

A selecting situation diagram that refines an abstract situation diagram depicts an *abstract situation box* within a time line of its condition box (or within a time line of each condition box if the situation diagram has several alternative branches). A generating situation diagram that refines an abstract situation diagram depicts an *abstract occurrence bar* within its condition box (or within each condition box if the situation diagram has several alternatives).

In addition, following rules have to be obeyed if abstract situation diagrams are refined.

- The *rule of corresponding situation classes for abstract situation diagrams and situation diagrams in their refinements* requires that both the abstract situation diagram and its refining situation diagram have to be associated with the same situation class.

  Example: Consider the abstract situation diagram depicted in Figure 8.24 and its refinement depicted in Figure 8.26. Both situation diagrams are associated with situation class asSoonAsDecIsLate.

- The *rule of abstract situation satisfaction* says that in the case of event classes, the events resulting from the situation diagram that refines an abstract situation diagram have to occur at least once within each interval resulting from the abstract situation diagram; in case of interval classes, the intervals resulting from the situation diagram that refines the abstract situation diagram have to occur at least once within each interval resulting from the abstract situation diagram.

Example: Consider the situation diagrams in Figures 8.26 and 8.27. They refine the abstract situation diagrams depicted in Figures 8.24 and 8.25, respectively. Both situation diagrams refine the abstract situation diagram defined at object type *LOAN_APP* at the subtypes, PERSONAL_LOAN_APP and MORTGAGE_LOAN_APP, respectively.

For a personal-loan application, an event is raised already at eight o'clock (ct) on Monday (d) in the first week after the decision has become late. For a mortgage-loan application, an event is raised at eight o'clock (ct) on Friday (d) in the first week after the decision has become late. Both situations occur exactly once within the abstract situation (i.e., the first week after the decision has become late).

Remember also the rules defined for abstract features in general. E.g., the up-to-basic refinement rule for abstract situation diagrams of concrete object types requires that each abstract situation diagram of a concrete object type O is either refined at O or inherits a refinement from some supertype of O.

Resolving Conflicts

The rule for redefinition of situation diagrams of logical situations cannot be violated, since situation diagrams are only "extended" by adding new situation diagrams at subtypes (cf. Section 8.4.2). Further, abstract situation diagrams represent only a design aid and are actually never evaluated. Thus, the refinement rules are not checked and therefore cannot be violated.

8.4.4 Specializing Rules

Rules implement obligations. Obligations represent sufficient conditions, which say that an action has to be performed when a certain condition over the history of time and the database is satisfied. Sufficient conditions may be weakened only at subtypes. Consequently, the situation diagrams of the rule-triggering event class and the rule conditions may be weakened only at subtypes. The specialization of situation diagrams has already been discussed in Section 8.4.3. The specialization of rule conditions will be discussed here. The correct specialization of obligations is called "decision consistency." Since rules are specified by means of activation diagrams, the term "activation consistency" could be used instead of decision consistency (cf. Section 8.4).

Rules are bound to object types like properties, activities, states, situation classes, and situation diagrams. They may be specialized again by redefinition, by extension, or by

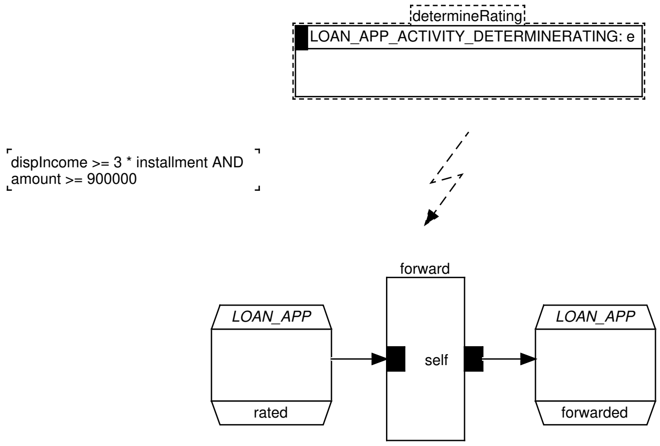


Figure 8.28: Activation diagram Forward1 of object type *LOAN_APP*

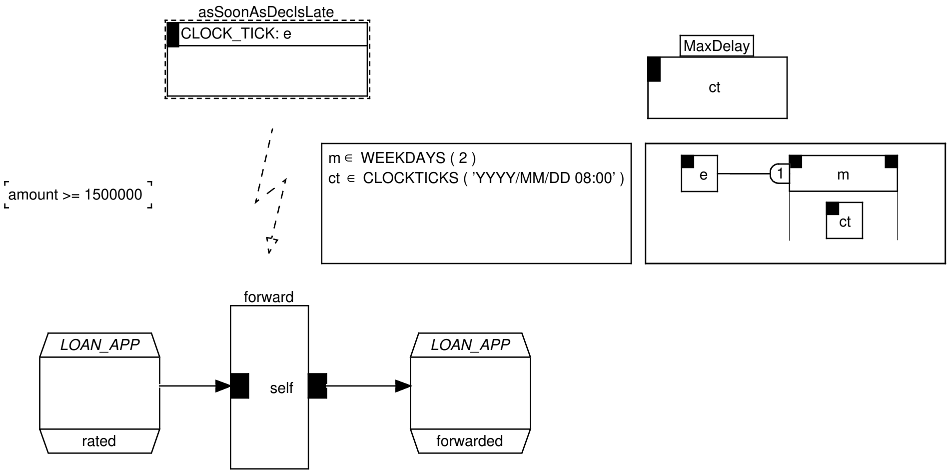


Figure 8.29: Activation diagram Forward2 of object type *LOAN_APP*

refinement. All three kinds of specialization have to consider the weakening of rules at subtypes, corresponding to the specialization of situation diagrams.

Note that rules may never have an abstract activity as rule action. If an abstract activity would be defined to be a rule action, it is not clear, which basic activity should to be performed if the rule fires. Therefore, it is not meaningful to consider rules with abstract activities as actions.

Redefining Rules

Like all other features of object types, a rule is introduced at an object type O and is inherited by each direct or indirect subtype of O. Certain parts of a rule may be redefined at a subtype by obeying the following rules.

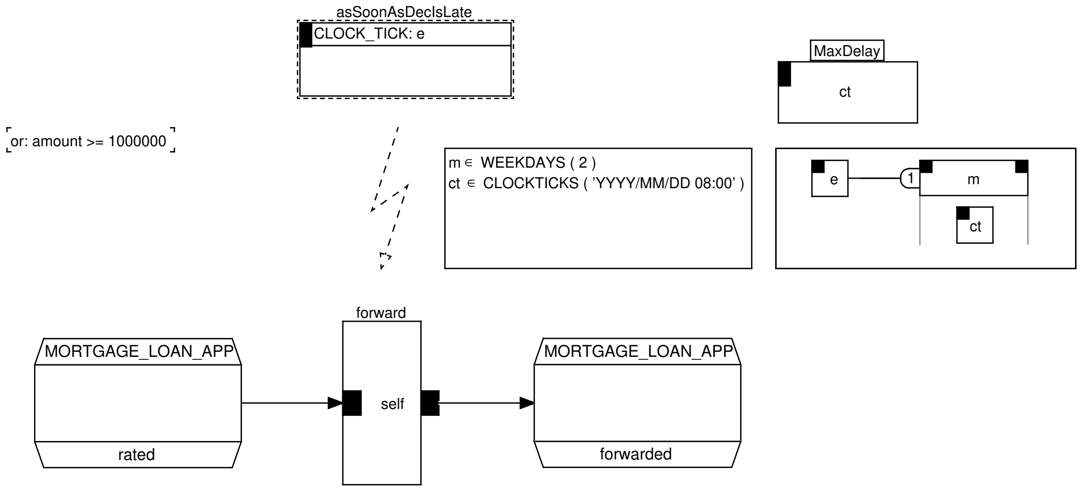


Figure 8.30: Activation diagram Forward2 of object type MORTGAGE_LOAN_APP

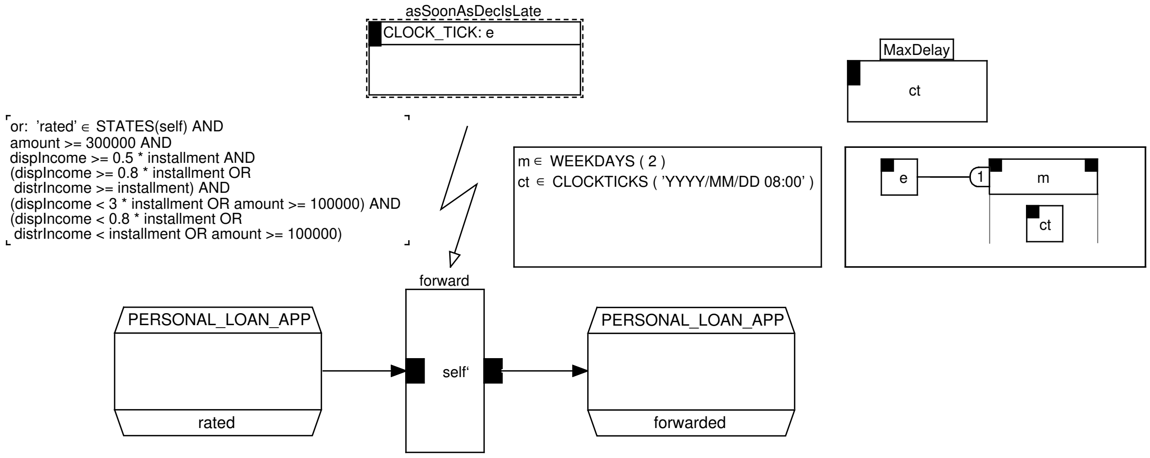


Figure 8.31: Activation diagram Forward2 of object type PERSONAL_LOAN_APP

- The *rule of forbidden redefinition of rule actions* requires that the rule action of an inherited rule R at the subtype corresponds to the rule action of R at the supertype. Although rule R must have the same activity as rule action at the subtype and at the supertype, the activity itself may be specialized at the subtype.

  Example: Rules Forward1 and Forward2, which are depicted in Figures 8.28 and 8.29, are specified at type *LOAN_APP* and are inherited by subtypes MORTGAGE_LOAN_APP and PERSONAL_LOAN_APP. Both rules determine when a loan application is fowarded to the board.

  Rule Forward2 defines the same action, activity forward, at object type *LOAN_APP* (Figure 8.29) and at object type MORTGAGE_LOAN_APP (Figure 8.30).

- The *rule of forbidden redefinition of rule-triggering event classes* requires that the rule-triggering event class of an inherited rule R at the subtype corresponds to the rule-triggering event class of R at the supertype. Of course, the event class and its situation diagrams may be specialized at the subtype.

  Example: Rule Forward2 defines the same rule-triggering event class, situation class asSoonAsDecIsLate, at object type *LOAN_APP* (Figure 8.29) and at object type MORTGAGE_LOAN_APP (Figure 8.30).

- The *rule of forbidden redefinition of maximum delay* requires that the situation diagram that determines the maximum delay of an inherited start-rule R at the subtype corresponds to the situation diagram that determines the maximum delay of R at the supertype.

  Example: The situation diagram that determines the maximum delay of rule Forward2, which is inherited from object type *LOAN_APP* (Figure 8.29), is not modified at object type MORTGAGE_LOAN_APP (Figure 8.30). The maximum delay occurs at eight o'clock on the Monday following the triggering event.

- The *rule for redefinition of rule conditions* requires that if the rule condition is satisfied when a rule-triggering situation s occurs for an object viewed as instance of a supertype, the rule condition has to be satisfied also for the object viewed as instance of a subtype. Situation s has to occur also for the object viewed as instance of a subtype because of the rule for redefinition of situation diagrams of logical situation classes.

  Since rule conditions are part of obligations, which represent sufficient conditions, they may become only weaker at the subtype. To simplify the specification of weaker conditions, the condition term specified at the subtype (at label or) is enlarged disjunctively with the inherited rule condition at the supertype.

Example: The rule condition of rule Forward2 at object type *LOAN_APP* (Figure 8.29) restricts the automatic forwarding upon this rule for loans with an amount greater than or equal to 1,500.000 ATS. The rule condition of the redefined rule Forward2 at object type MORTGAGE_LOAN_APP (Figure 8.30) is weaker because it is already true for loans with an amount greater than or equal to 1,000.000 ATS. Since the condition term at the subtype is enlarged disjunctively with the condition inherited from the supertype, specifying the condition term 1000000 <= amount < 1500000 at type MORTGAGE_LOAN_APP instead of amount >= 1000000 would be sufficient.

- The *rule for redefinition of definite/experimental rules* says that an experimental rule may become a definite rule at a subtype but *not* vice versa.

  Opposed to definite rules, experimental rules include implicitly the prestate condition (P^{pre}_{prim}) of the action activity's primary port into the rule condition. It is assumed in the following discussion that the rule condition (irrespective of the rule being definite or experimental) does not include explicitly P^{pre}_{prim}. If included explicitly, there is no difference between definite and experimental rules except that the latter have P^{pre}_{prim} included two times.

  1. If an experimental rule is redefined towards a definite rule, the rule condition itself is weakened because P^{pre}_{prim} is discarded from the rule condition. However, the rule does not fire more often at a subtype; only the exception event indicating a not enabled object self is raised more often.
  2. If a definite rule would be redefined towards an experimental rule (which is not allowed here), the rule condition would be strengthened. Nevertheless, the rule itself is not fired less often at the subtype, but the exception events are raised less often.

  Example: Rule Forward2 is experimental at type *LOAN_APP* (Figure 8.29). Its specialization at type PERSONAL_LOAN_APP (Figure 8.31) is definite. The action's permission is included explicitly into the rule condition (including P^{pre}_{prim}).

- The *rule of fobidden redefinition of do/start rules* says that a start-rule at a supertype must not become a do-rule at a subtype *and* vice versa.

Extending Rules

The active behavior of object types may be extended by introducing rules at subtypes. Introducing a new rule at a subtype ensures decision consistency because additional obligations are introduced for instances of subtypes.

The obligations of an object viewed as an instance of a supertype can be enlarged for the object viewed as instance of a subtype (1) by introducing a new rule at the subtype or

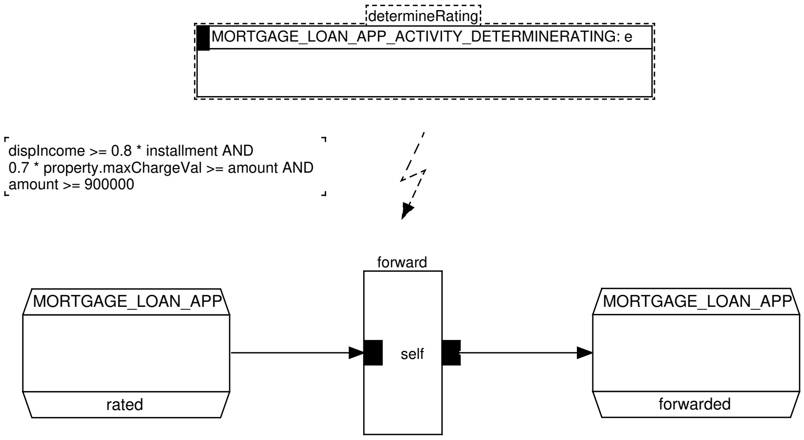


Figure 8.32: Activation diagram Forward3 of object type MORTGAGE_LOAN_APP

(2) by redefining a rule inherited from the supertype so that the rule fires more often. Redefining a rule inherited from a supertype is only useful if the obligation to be enlarged invokes the same action as its enlargement.

Example: The rules Forward1 and Forward2, which are depicted in Figures 8.28 and 8.29, are specified at type *LOAN_APP* and are inherited by the subtypes MORTGAGE_LOAN_APP and PERSONAL_LOAN_APP. Rule Forward2 is redefined at both subtypes (Figures 8.30 and 8.31).

Rule Forward3, which is depicted in Figure 8.32, is introduced at object type MORTGAGE_LOAN_APP. The rules Forward1, Forward2, and Forward3 together determine when a mortgage-loan application is fowarded automatically to the board.

Refining Rules

Refining rules means first to restrict the points in time at which a rule has to be triggered and then to determine the exact triggering time by a refinement. Situation diagrams are refined in a similar way. Abstract situation diagrams return intervals during which the events of the refinement have to occur or in which the intervals of the refinement must be contained.

An *abstract rule* defines a rule-triggering interval class instead of a rule-triggering event class. Like a conventional (basic) rule, an abstract rule defines an activity as action and may define a rule condition. Further, an abstract rule has to be local to the object type to which the activity specified as action belongs. Different to basic rules, the properties of the interval type of the rule-triggering interval class may be used only within the rule condition; the ports of the action activity are not mapped, since the rule is abstract and, as such, never executed.

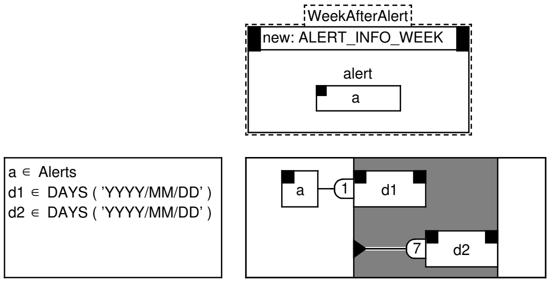


Figure 8.33: Situation diagram for situation class WeekAfterAlert of object type *LOAN*

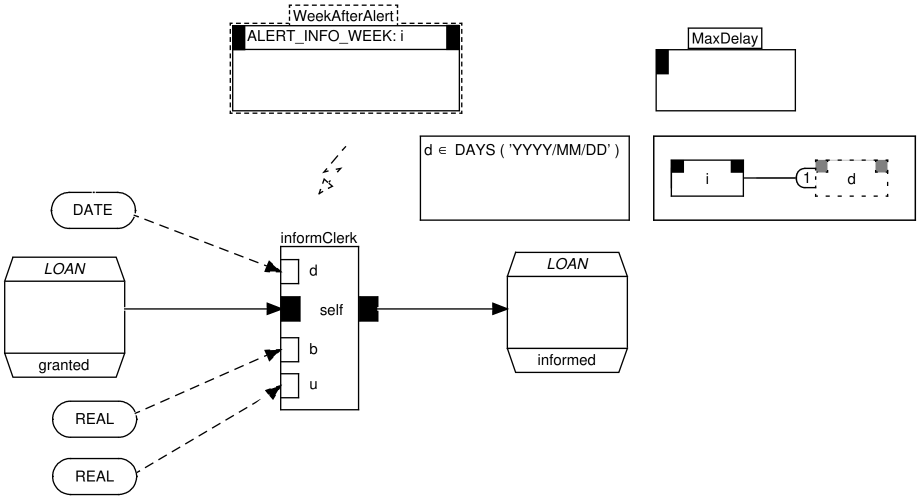


Figure 8.34: Abstract activation diagram Inform of object type *LOAN*

The graphical representation of an abstract rule differs from the representation of a basic rule only in the different representation of the rule-triggering interval class and of the rule-triggering event class. The former depicts two filled rectangles, one in the upper left and one in the upper right corner, the latter depicts a single filled rectangle in the upper left corner. An activation diagram that depicts an abstract rule is also called "abstract."

Example: Consider the following information policy: After an alert has been raised upon a loan, a clerk has to be informed about this alert. Short-term loans (e.g., a personal loan) have to be processed more efficiently than long-term loans (e.g., a mortgage loan). The general policy for all loans is to inform the clerk in the first week after an alert has been raised.

The situation diagram in Figure 8.33 generates a seven-days interval after each alert. These weeks are collected in situation class WeekAfterAlert at object type *LOAN*.

Figure 8.34 depicts abstract rule Inform, which implements the information policy by invoking activity informClerk upon the occurrence of some event within an interval of its triggering *interval* class WeekAfterAlert. For a moment, ignore the situation

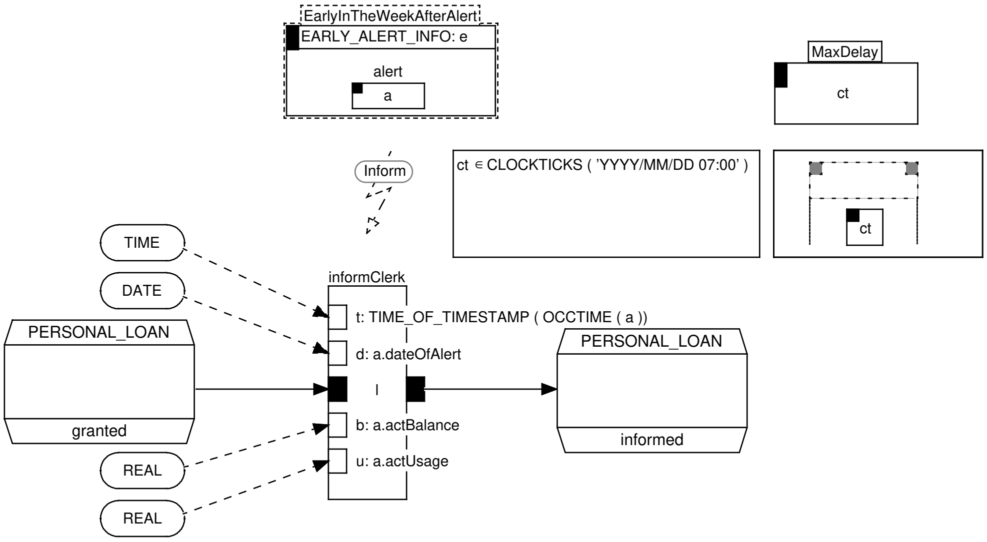


Figure 8.35: Activation diagram InformEarly of object type PERSONAL_LOAN

diagram determining the maximum delay for rule Inform; it will be discussed later.

An activation diagram for a rule that refines an abstract rule depicts the name of the abstract rule in a rounded rectangle placed in the middle of the activation arrow.

Example: Rule InformEarly (Figure 8.35) of object type PERSONAL_LOAN refines abstract rule Inform (Figure 8.34), which is defined at object type LOAN and inherited by subtype PERSONAL_LOAN. The semantics of this rule will explained in the successive examples.

The following consistency rules have to be obeyed if abstract rules are refined.

- The *rule of corresponding action activities for abstract rules and rules in their refinements* requires that both the abstract rule and its refining rule have the same activity as action.

  Example: Abstract rule Inform (Figure 8.34) and its refining rule InformEarly (Figure 8.35) define activity informClerk as action. Ignore the situation diagram determining the maximum delay for rule InformEarly; it will be discussed later.

- The *rule of satisfaction of rule-triggering situations of abstract rules* says that at least one rule-triggering situation of a rule refining an abstract rule has to occur within each rule-triggering interval of the abstract rule.

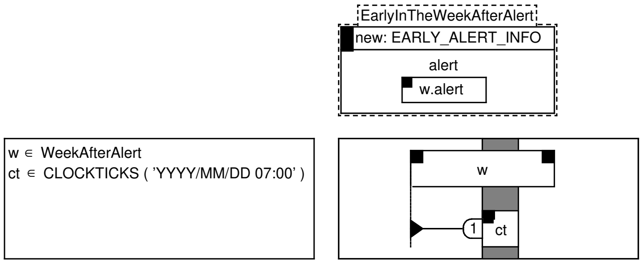


Figure 8.36: Situation diagram for situation class EarlyInTheWeekAfterAlert of object type PERSONAL_LOAN

Example: One event (i.e., the first seven-a.m. clock tick in the week after an alert) in situation class EarlyInTheWeekAfterAlert, the triggering-event class of rule InformEarly, occurs in each week after an alert. The situation diagram of class EarlyInTheWeekAfterAlert is depicted in Figure 8.36.

Rules can be refined also "indirectly" by refining the situation diagram of the rule-triggering event class instead of the rule itself. One might argue that refining rules indirectly is to prefer to a separate concept like abstract rules. However, if rules are refined indirectly, situation classes are "bound" to rules and thus can hardly be reused by other situation diagrams or other rules (cf. Chapter 9). Thus, abstract rules represent a meaningful concept for the specialization of obligations.

Unlike do-rules, start-rules define a maximum delay for being executed. The maximum delay is specified by means of a situation diagram, which returns a single calendar event, whose occurrence time has to be known when the rule-triggering event occurs. This situation diagram may be refined like a conventional situation diagram except that the situation diagram that refines an abstract situation diagram may return only one event occurring within the interval of the abstract situation diagram.

Example: Abstract start-rule Inform of type *LOAN* (Figure 8.34) says that the clerk has to be informed until some point in time of the following day. The abstract situation diagram determining the maximum delay returns the abstract day interval following the week after the alert.

Basic start-rule InformEarly of type PERSONAL_LOAN (Figure 8.35) refines the abstract situation diagram determining the maximum delay. The maximum delay of the basic rule occurs exactly once (at seven o'clock) within the abstract maximum delay of the abstract rule. The refinement says that the rule implementing the information policy has to be executed successfully for a personal loan until seven o'clock of the following day.

The consistency rules defined for abstract features in Section 8.1.3 have to be applied to abstract rules as well as to abstract situation diagrams determining the maximum delay.

E.g., the up-to-basic refinement rule for abstract features requires that each abstract feature is refined at a concrete object type or inherits a refinement from a supertype.

Resolving Conflicts

The rule for redefinition of rule conditions represents for rules, what the rule for redefinition of permissions represents for activites. Since the inherited conditions are not replaced but enlarged disjunctively, this rule cannot be violated.

The rule of satisfaction of rule-triggering situations of abstract rules is violated if for an interval i of the rule-triggering interval class of an abstract rule, no situation s of the rule-triggering situation class C of a rule R that refines the abstract rule occurs where (1) the occurrence time of s is covered by i if C contains events (R is a basic rule) or where (2) the occurrence period of s lays within the occurrence period of i if C contains intervals (R is an abstract rule again). The exception event raised upon this violation is inserted into class r_ruleRefViolation where r is the name of the refining rule R. The violation of this rule can be detected at the end of each interval i.

If for an abstract maximum-delay interval i, no maximum-delay situation s occurs for the rule that refines rule r where the occurrence time of s is covered by i (the situation diagram is basic) or where the occurrence period of s lays within the occurrence period of i (the situation diagram is again abstract), then an exception event is inserted into class r_maxDelayRefViolation. This violation can be detected at the end of the maximum delay interval i.

8.5 Related Work

Specialization is a well-known concept in conceptual modeling of object-oriented databases. The core concept, inheritance, is often used for different purposes. As already discussed in Section 8.1, in conceptual design, the concept of inheritance should be used only for *conceptual* specialization of object structure and passive and active behavior and not, for example, for code sharing.

Many object-oriented design methods like OMT [RBP$^+$91], OOSA [EKW92], OOAD [Boo94] define besides the inheritance of object structure also the inheritance of object behavior. Opposed to the approach discussed in this chapter, these methods define the inheritance of object behavior only at object-type level by saying how an object type's operations are inherited from the supertype and how their interface can be overridden at the subtype.

The core concepts of behavior specialization are based mainly on the work of Kappel, Schrefl, and Stumptner [KS94, KS96b, Sch90, SS95, SS97]. Related work on behavior specialization can be found there. The distinction between necessary and sufficient conditions and thus between permission and prohibition is due to Wieringa et. al [WWMD91]. The specification of permissions and prohibitions has been taken up in combination with Object/Behavior Diagrams first by Bichler, Kueng, and Schrefl in [BS95, KS95].

The ideas for the specialization of obligations (and thus of business rules) are also due to Wieringa et. al [WWMD91] where the inheritance of deontic concepts is discussed. Bichler and Schrefl used the concept of obligations and formed the general approach to the specialization of active behavior in object-oriented modeling in [BS95].

Specifying business rules in object-oriented schemas has already been discussed in the diverse literature. Some of the conceptual languages for modeling rules also allow to specify rules that are local to object types (e.g., "targeted rules" in Chimera [CBFP95, CFPT96, MPC96]). Nevertheless, not all of them deal with the specialization of rules, in particular, with the inheritance of rules.

Silva and Carlson [SC96] define rules in A/OODBMT locally to object types and define their specialization (similarly to the inheritance of object behavior) only at object-type level by saying how an object's rules are inherited from the supertype and how their components (event, condition, and action) may be overidden at the subtype. Unlike Active Object/Behavior Diagrams, A/OODBMT allows to override the condition and the action of an inherited rule at a subtype without any restriction. The inherited rule's event cannot be specialized at subtypes. Silva and Carlson allow rules to be invalidated at subtypes by falsifying the condition and by nullifying the action. This violates a design heuristic that is given by Riel [Rie96] for overriding passive behavior in object-oriented design but may be applied also to the active behavior of objects. This heuristic says that it should be illegal to override a method with a NOP (a method that executes no operations).

Chapter 9

Design Guidelines

Contents

Insufficient methodological support in analysis and design is encountered by Simon and Kotz-Dittrich [SKD95] as one of the main problems of using active database technology. Active Object/Behavior Diagrams try to fill this need by providing a homogenous set of graphical modeling concepts for conceptual active object-oriented database design and associated design guidelines.

The use of modeling concepts such as provided by Active Object/Behavior Diagrams is restricted by a set of consistency rules. Consistency rules address the syntactic correctness (e.g., life-cycle states may be connected only with life-cycle activities but not with life-cycle states) and the semantic correctness (e.g., the prestate of an activity port must belong to the object type that serves as port domain) of a database schema. Consistency rules are either guaranteed by the modeling concepts or have to be obeyed by the designer. They have the designer stick to a correct usage in order to achieve a consistent database schema, i.e., a database schema that obeys the consistency rules associated with the modeling concepts.

In addition to consistency rules, guidelines help the designer to achieve a "high-quality" database schema; a database schema is of high quality if it is comprehensible and maintainable. 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.

The guidelines discussed in this chapter deal only with the design of active behavior. Guidelines for designing object structure and passive behavior can be found, for example, in [Rie96].

The remainder of this chapter is structured as follows. Section 9.1 gives a short introduction into design guidelines. Section 9.2 presents several design guidelines for active object-oriented database design and presents how they are supported by Active Object/Behavior Diagrams and other active database design languages. Section 9.3 surveys a further concept to support the conceptual design of active databases: rule patterns.

9.1 On the Notion of Design Guidelines

Active object-oriented database design gathers more semantics in the database schema than, for example, object-oriented database design (cf. Chapter 3). In addition to object-oriented database schemas, active object-oriented database schemas include rules to describe the active behavior of objects. Putting additional information into a database schema requires additional modeling concepts. The more modeling concepts a design method provides, the more intertwined become the concepts.

The designer has to create a conceptual database schema that meets the requirements identified during the analysis phase by applying the modeling concepts provided by the design method. Although the designer obeys the consistency rules, he may produce different database schemas each of them meeting the requirements. If each resulting conceptual database schema meets the requirements, how can they differ from each other? If the designer obeys all consistency rules, he or she yields a database schema that ensures the syntactic and semantic correctness defined by the consistency rules. But, modeling concepts can usually be applied in different ways without violating a consistency rule and, thereby, still allow to meet the requirements.

The experienced database designer determines the quality of a (consistent) database schema by checking if guidelines for a "good" design are enforced. Riel [Rie96] has established a set of guidelines for object-oriented design and has called them "design heuristics." These guidelines do not represent hard rules; they represent rather warning mechanisms that ring if violated, but may be ignored. Riel says that the experienced designer maintains a "subconscious" list of guidelines based on his or her experience and applies them during the design process.

The aim of using guidelines for conceptual database design is to achieve a comprehensible and maintainable database schema. 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 schemas resulting from logical database design, whereas high performance and low storage costs determine the quality of database schemas resulting from physical database design. However, Batini et al. [BCN92] suggest to normalize also conceptual database schemas to remove redundancy.

If design guidelines are ensured implicitly by the modeling concepts, they help the designer to achieve a high-quality database schema when applying the modeling concepts (*implicit design guidelines*). If design guidelines are not ensured implicitly by the modeling concepts, the designer is free whether to design the database schema according to the guidelines (*explicit design guidelines*). In addition, developing the database schema may be supported by a software tool that implements design guidelines even if these guidelines are not included in the modeling concepts. The more powerful the concepts used, the more important it is to have a tool supporting the designer in checking the consistency rules and the design guidelines.

9.2 Guidelines for Active Object-Oriented Database Design

As already mentioned in the introduction of this chapter, we consider only guidelines for designing the active behavior of objects. Guidelines for designing object structure or passive behavior have been already set up, for example, by Riel in [Rie96]. Additional guidelines for active behavior may also be derived from the guidelines defined by Riel.

The following design guidelines are not limited to database schemas developed with Active Object/Behavior Diagrams. Of course, these design guidelines represent the result of modeling the active behavior of objects by means of Active Object/Behavior Diagrams, but they may be applied to different active object-oriented design methods. The design guidelines are not grouped into implicit and explicit guidelines because a single guideline may be ensured implicitly by the concepts provided by a certain design method, whereas it has to be obeyed explictly using another design method.

The design guidelines are introduced in the following sections in the same way: First, the need for a guideline is motivated. Then, a potential solution is discussed, a guideline derived, and an example given. Finally, we show how the guidelines are enforced using Active Object/Behavior Diagrams and other conceptual design methods, in particular, using A/OODBMT [SC96] and Chimera [CBFP95, CFPT96]. A/OODBMT and Chimera have already been introduced in Section 3.6.

The remainder of this section is structured as follows. Section 9.2.1 advises which sort of object behavior to model with active or with passive modeling concepts. Section 9.2.2 analyzes the consequences of specifying rules locally to object types or globally. Section 9.2.3 presents the advantages and disadvantages of "small" and "big" rules. Section 9.2.4 presents guidelines for specializing the active behavior of objects.

9.2.1 Active versus Passive Behavior

Two kinds of behavior are distinguished in active object-oriented database design: active behavior and passive behavior. Active behavior is modeled by means of rules (triggers), passive behavior is modeled by means of operations (activities, methods) and states.

Usually, business rules are modeled by the active behavior, and business procedures are modeled by the passive behavior of business objects. But, business rules may be understood in a broader or in a narrower sense,[1] so that it is again not clear what is (or should be) modeled actually by active behavior or by passive behavior.

The problem of maintaining a large set of rules has been articulated in the field of Artificial Intelligence first and has also been realized in the field of active databases [KS93]. A first step to reduce the number of rules is to single out which elements of the business area have to be modeled by means of active behavior and which elements of the business area have to be modeled by means of passive behavior.

Appelrath et al. [ABJZ96] have identified the problem whether to implement behavior by means of rules or by means of conventional programming, but did not weigh this positively or negatively. Kappel and Schrefl [KS96a] have analyzed how rules could be applied in active object-oriented database design. They have examined the following design elements and suggest to model only a single element of the business area by means of rules. The examples used in the remainder of this chapter deal again with loan applications, which have been introduced in Chapters 7 and 8.

1. Static integrity constraints have a "restrictive" nature. A transaction that violates a static integrity constraint cannot commit. Static integrity constraints are neither part of the active behavior nor part of the passive behavior, but they are part of the object structure, which they constrain (cf. Section 3.1.1).

   Example: The condition that the loan amount of a loan application always is lower than zero represents a static integrity constraint. Consequently, the loan amount is always represented as a negative number.

   Nevertheless, checking whether static integrity constraints are satisfied, may be implemented by means of rules during logical design. Nevertheless, static integrity constraints represent "restrictions" at the conceptual level and thus belong to object structure.

[1] Business rules in the narrower sense are rules on business policies, business rules in the broader sense comprise also rules on business procedures and integrity rules (cf. Section 3.1.1).

2. Dynamic integrity constraints, which constrain the execution of operations, have also a restrictive nature like static integrity constraints. A transaction that tries to perform an operation without satisfying the corresponding dynamic integrity constraints (precondition and postcondition) cannot commit. Dynamic integrity constraints are part of the passive behavior.

   Example: The condition that a loan application with a high credit rating and a low loan amount may not be forwarded to the board—such a loan may be granted already by a clerk—represents a dynamic integrity constraint, which is expressed by means of preconditions.

   A logical database schema may implement checking pre- and postconditions by means of rules or by means of conventional program statements. At the conceptual level, pre- and postconditions are always part of the passive behavior.

3. How values or objects of derived properties are actually derived, is specified declaratively by means of a derivation expression. A derivation expression belongs to that object type whose property it derives. Consequently, derivation expressions are part of the object structure.

   Example: The monthly payment (the installment) for a loan is computed as the loan amount increased by a certain interest rate and divided by the number of months (the agreed term) the loan is granted.

   The evaluation of derivation expressions can be implemented by means of rules during logical design. For example, derivation expressions are evaluated every time when the derived properties are accessed and if the values or objects that are needed for the evaluation have changed since the preceding evaluation.

4. Users (organizational units) must be authorized to access data and to invoke operations. Access constraints—they have not been considered so far in this work—have also a restrictive nature like integrity constraints because users (organizational units) either have the right (are permitted) or do not have the right (are forbidden) to perform certain tasks in the database (cf. Section 3.1.1).[2]

   Example: Only clerks who have experience in managing loans are allowed to decide about a loan application. They may grant or reject a loan or forward it to the board.

   Commonly, authorization concepts are subject to workflow modeling. Although they might be implemented by means of rules during logical design, they are specified declaratively at the conceptual level of workflow modeling (see, for example, [Bic97]).

[2] Note that the permissions and prohibitions used in Active Object/Behavior Diagrams do not represent authorization constraints. They represent simply the sufficient and necessary preconditions of activities (cf. Section 7.2.5).

5. The order in which work steps may be performed on business objects is usually expressed by some sort of state-transition diagram or Petri net. Their states and transitions represent the passive behavior.

   Example: The work steps to treat a loan application, like creating the application, checking the income, determining the credit rating, deciding about the application, and, last but not least, informing the applicant about the decision, are organized in a Petri net with intermediate states.

6. Obligations describe when to perform a certain task. They are modeled by means of rules and therefore form the only active behavior at the conceptual level.

   Example: Forwarding a loan application automatically to the board when the decision is late represents an obligation and is therefore modeled by means of a rule.

   Note that the concept of obligation as it is used herein does not correspond exactly to an obligation as it is defined in deontic logic (cf. Section 7.2.5). For example, we do not consider organizational aspects (authorization). Nevertheless, if obligations include organizational information (e.g., a certain user is obliged to do something), they may be modeled with a rule-like concept, too.

After this discussion we can give the first design guideline, which says which elements to model as passive behavior and which elements to model as active behavior.

> GUIDELINE 1
> *Use rules to specify only obligations to act. Obligations define an object's active behavior.*

The above enumeration reflects the division of the conceptual model into object structure with static integrity constraints, passive behavior with dynamic integrity constraints (permissions, prohibitions, and postconditions), and active behavior (obligations) as proposed in Section 3.1.1.

The events provided by Active Object/Behavior Diagrams are always viewed from the external user's perspective. Therefore, events and rules are not appropriate to model integrity constraints at all. For this, events would have to be detected and rules would have to be processed within the constraint-violating transactions; the rules would have to prevent the transactions from committing. The concepts of Active Object/Behavior Diagrams ensure implicitly Guideline 1.

A/OODBMT provides two kinds of rule concepts: static rules and dynamic rules. Static rules are used for specifying static integrity constraints. Dynamic rules are used for specifying exceptional semantics and dynamic integrity constraints, but also for specifying obligations. Since A/OODBMT does not provide other concepts but rules to model integrity constraints etc., Guideline 1 cannot be obeyed. Modeling integrity constraints by means of rules leads to a large set of rules. If these rules are managed also globally, their behavior is hard to comprehend (see Section 9.2.2).

Integrity constraints are specified declaratively also in Chimera. Active rules are generated automatically during logical design. For this, Chimera provides the two intra-transactional coupling modes "immediate" and "deferred." Consequently, rules are always executed within the event-signaling transaction. Guideline 1 must be obeyed explicitly using Chimera. Further, Chimera does not provide temporal events. Obligations expressing that an action has to be performed at certain points in time cannot be modeled.

Herbst et al. [HKMS94] use the term "business rule" in a broader sense because they assume also static and dynamic contraints to be business rules. Then, it is difficult to separate rules that represent obligations from all other rules, e.g., rules that maintain integrity constraints. As a consequence, Guideline 1 is hard to ensure unless different concepts for specifying integrity constraints are provided.

Further, Kappel and Schrefl [KS96a] suggest the following procedure to define the behavior of objects.

> GUIDELINE 2
> *First, design the passive model ensuring all integrity constraints, then make it active by specifying rules.*

The active behavior always represents a supplement to the already defined passive behavior. The passive model must provide all relevant operations being executed in transactions, which are invoked by the user, and the model's consistency must be ensured by integrity constraints. The rules added afterwards should determine only when a transaction (invoking an operation provided by the passive model) has to be performed without any intervention by a user. Thereby, the rules replace the user.

This guideline helps the designer to distinguish between active and passive behavior. Integrity constraints must be ensured by the passive behavior but also by the active behavior because it is based on the passive behavior. Using rules to model integrity constraints disallows to distinguish clearly between active and passive behavior.

Guideline 2 must be ensured explicitly using Active Object/Behavior Diagrams, by A/OODBMT, and by Chimera. Before specifying rules, object structure and passive behavior have to be developed such that they ensure all integrity constraints. Then rules should be specified on top of the passive model by executing "public" operations (consistency-preserving operations) on objects. But note that Chimera allows to specify also rules that perform operations which are not necessarily executed on an object. Rules may invoke also global database procedures and transaction commands like a rollback. Further, the coupling modes provided by Chimera allow rule actions to be executed only within the event-triggering transaction, and thus allow rules to implement integrity maintenance.

9.2.2 Local versus Global Events and Rules

Managing a large set of rules represents the most severe problem of active database design and active database systems. The larger the set of rules, the more difficult to comprehend the behavior of rules. This dilemma has already been addressed already in the field of Artificial Intelligence but also in the field of active databases [KS93], for example, by Simon and Widom in [SKD95] or by Kappel and Schrefl in [KS96a].

Modeling only obligations to act by rules (as suggested by Guideline 1) helps to reduce the number of rules in conceptual database design.[3] Although Guideline 1 is obeyed, a large set of rules may exist also at the conceptual level.

A large set of rules becomes more comrehensible by dividing it into smaller sets of rules that belong semantically together. Conventional active design methods and conventional active database systems (not object-oriented) may split up their global set of rules into several subsets. Active object-oriented design methods have another, much better ability to cope with a large number of rules: Rules may be specified locally to objects or globally. But which rules should be specified locally to which object? In an object-oriented model, each operation that is performed on an object is specified at the corresponding object type or is inherited from some supertype. As a rule describes when a certain operation (the rule's action) is performed, the rule is best specified locally to that object type on whose instances the rule's action is invoked.

GUIDELINE 3
Specify a rule locally to that object type on whose instances the rule invokes its action.

This guideline requires that only a single operation is specified as rule action. If a rule invokes operations on instances of different object types, the rule cannot be attached to a single object type (see also Section 9.2.3).

Example: Assume an operation on a single account that debits the account with service charges, credits the account with the interest yield, and prints a statement about it. The corresponding rule invoking this operation at the end of every third month is specified locally to the account's object type.

[3]Nevertheless, if additional rules are used in logical design, for example, to implement the maintenance of integrity constraints, a small set of rules at the conceptual level may become large at the logical level.

Guideline 3 is ensured implicitly by Active Object/Behavior Diagrams. A rule may invoke only an activity that is local to the same object type to which the rule is local. A “global” rule has to be specified locally to the global object type, which represents the database and of which only one instance exists.

Rules in A/OODBMT are always specified locally to object types. Consequently, Guideline 3 represents an implicit guideline also for A/OODBMT.

In Chimera, rules that are specified locally are called targeted rules, rule that are specifed globally are called untargeted rules [MPC96]. As targeted rules can invoke also global database operations besides attribute modifications, Guideline 3 has to be obeyed explicitly. Further, the rules' action part may hold also a sequence of operations. Guideline 3 has to be obeyed explicitly by specifying only one operation that is invoked on the object to which the rule is targeted. The differences to Active Object/Behavior Diagrams and to A/OODBMT are probably due to the relational touch of Chimera. Rules specified in Chimera are translated automatically into triggers of the relational database system Oracle (Oracle, Inc.) when a logical database schema is generated. Triggers in this database are always global. To fill the need for structuring the set of global rules, modularization techniques [BCP96] have to be given to divide complex rules into disjoint strata, which can be evaluated individually (see also Section 9.2.3).

Summarizing the effects of Guideline 3, a rule is restricted to perform an operation only on instances of that object type to which it is attached. Now, which events may trigger such a rule? Corresponding to rule actions, the rule-triggering events may also be specified either locally to object types or globally. Then, events occur either locally at some object or globally in the database.

If rules are triggered by events that occur at a different object than that one for whose object type the rule is specified, the dependencies between rules of different object types can be recognized only by studying the rule specifications (e.g., the activation diagrams if the rules are specified by means of Active Object/Behavior Diagrams). If the set of rules is very large, the inter-dependencies of active behavior become incomprehensible.

GUIDELINE 4
The events triggering a rule should occur locally to the instances of that object type to which the rule is local.

If rules are triggered only because of events that occur locally to the objects on which the rule invoke their actions, the dependencies between the active behavior of different object types can be recognized already at object-type level by inspecting the specified rules and events.

Example: The rule that forwards automatically the loan application to the board if the decision about the loan is late is specified locally to the object type of the loan application. The event describing when the decision has become late has to occur also at the loan application.

Active Object/Behavior Diagrams treat situations (events and intervals) as first-class objects. We distinguish between situation types and situation classes. Like conventional properties, situation classes may be specified locally to object types. The triggering event class of a rule must be local either to the object type O at which the rule is specified or to a direct or indirect supertype of O so that O inherits the event class from its supertype. Activity events are automatically local to the object to which the activity belongs. All other user-defined events may be local to any object. Guideline 4 is again ensured implicitly.

A/OODBMT does not allow a rule to react "directly" upon an event specified locally to the object type at which the rule is specified. The events used for triggering rules are always specified globally and are given a global name. Therefore, Guideline 4 cannot be obeyed.

Chimera does not treat events as first-class objects; events cannot be specified independently from rules. A rule either uses a basic event or defines a composite event within the rule's event part using the composite event language introduced in [MPC96]. If a rule is targeted to a class, only events regarding that class are considered for triggering, otherwise events regarding any class in the database can appear in the event part of the rule [MPC96]. For targeted rules, Guideline 4 is ensured implicitly.

If rules are attached to object types and if events occur locally to objects, specializing object types requires also to specialize rules and events. Specializing rules and events means to modify inherited rules or events at subtypes. See Section 8.4 for details about specializing rules and events and Section 9.2.4 for further design guidelines on this topic.

If work steps have to be performed on several objects at once, the work steps are usually executed collectively on all involved objects. Today's business uses this technique to save resources and to increase performance. Active object-oriented database design can make use of this technique by obeying the following guideline.

Thereby, it makes no difference whether a rule should be triggered on a number of components of the same kind or on several different components. Using the terminology of Active Object/Behavior Diagrams, we need not distinguish between a number of objects that are referenced by a single multi-valued component-of relationship and objects that are referenced by several component-of relationships.

GUIDELINE 5
Instead of triggering rules individually for a number of components of an object o, trigger an operation on o that invokes the desired operations on its components.

Obeying Guideline 5 reduces the number of triggered rules, which have to be managed by the active database system, and thus increases the performance (cf. "set-oriented production rules" in [WF90]).

Example: Remember again the operation on a single account that debits the account with service charges, credits the account with the interest yield, and prints a statement about it. This operation has to be executed at the end of every third month. Instead of triggering a rule for each account object, a single rule is triggered at the bank object that invokes this operation on each account of the bank. A bank that manages several kinds of accounts (loan accounts, savings accounts, current accounts, etc.) may invoke the above operation on on all different kinds of accounts at once.

Such operations and the rules that invoke them can be specified in Active Object/Behavior Diagrams, A/OODBMT, and Chimera. Of course, Guideline 5 has to be obeyed explicitly. In Active Object/Behavior Diagrams, we use compound activities (cf. Section 4.4.4) to execute operations on different kinds of objects (i.e., objects referenced by different component-of relationships) unless the desired operation is included already in the object life cycle. In today's business, operations that are performed collectively on several objects are usually triggered at certain points in time. This semantics cannot be modeled using Chimera because Chimera does not support temporal events and rules that are performed in their own top-level transaction.

9.2.3 Small versus Big Rules

Different "public" operations (consistency-preserving operations that may be invoked by the user) may be invoked upon the same event. Some active conceptual database languages (e.g., Chimera) allow to specify rules holding several operations in their action part. These operations are all triggered when the rule-triggering event occurs and if the rule condition is satisfied. Other languages (e.g., Active Object/Behavior Diagrams) allow to specify only a single operation as rule action. Since a single operation may invoke several other operations, the same semantics could be expressed as with specifying several operations as rule action.

Let us call a rule invoking (directly or indirectly) several "public" operations a *rule with a big action* ("big rule") and a rule invoking (directly or indirectly) only a single "public" operation a *rule with a small action* ("small rule"). Using small rules to model the same semantics as with big rules usually increases the number of rules. Using big rules decreases the number of rules, but may make the effects of triggered rules be less comprehensible because several operations are performed upon a single event.

GUIDELINE 6
Specify several rules that invoke different operations but are triggered by the same events, which are coincidentally identical but may be different in the future.

Example: At the end of every third month, each bank account is debited with the service charges, credited with the interest yield, and a statement is printed about it.

If these operations are always executed together (i.e., all operations are invoked upon the occurrence of the same event) and if all these operations are executed together also for special accounts like loan accounts or current accounts, a big rule invoking these operations can be specified.

If, sometimes later, one of these operations may be invoked upon the occurrence of a different event (e.g., bank statements should be printed at the end of every month rather than at the end of every third month), then several small rules should be specified invoking these operations independently. If one of these operations has to be invoked upon the occurrence of a different event for special accounts (e.g., a bank statement for a current account is printed after each deposit), again several small rules should be specified invoking these operations independently.

Active Object/Behavior Diagrams do not allow to specify rules with several operations as rule action but provide the concept of compound activities, which allow to invoke several activities (cf. Section 4.4.4). Guideline 6 must be obeyed explicitly.

Like Active Object/Behavior Diagrams, A/OODBMT allows to specify only a single operation in the action part of a rule. The operation has to be understood by the object at whose object type the rule is specified. Guideline 6 must be also obeyed explicitly.

Unlike Active Object/Behavior Diagrams and A/OODBMT, Chimera allows to specify several operations in the rule's action part. Nevertheless, Guideline 6 must be obeyed explicitly.

Operations that produce the same effect in the database if they are executed in any order, are said to be *independent*. Otherwise, they are said to be *dependent*. The order in which dependent operations are executed has to be specified by some control structure, which may be implemented either by another operation or in the action part of a rule provided that the rule language allows to specify several operations in the action part (e.g., Chimera).

A big rule with dependent operations should never be replaced by a set of small rules—one rule for each operation—that encode the order in which their operations are executed in their triggering events.

GUIDELINE 7
Do not implement the execution order of different rule actions by means of events. Specify events always independently from rules to allow different rules to use them.

If events are misused for defining the order in which the rule actions are executed, they become more complex and are bound to a single rule. As a consequence, the events can hardly be reused to trigger another rule.

Example: Assume the operations accounting (debiting and crediting an account) and printing a statement about an account to be dependent, each of them invoked by a single rule. If an account has to be debited and credited before a statement about it is printed at the end of every third month, the event triggering the rule that invokes the printing has to occur at the end of every third month but immediately after the account has been debited and credited. This event is rather complex and is difficult to reuse by other rules because it implements the execution order of the corresponding rule's action.

Although Active Object/Behavior Diagrams collect events used for triggering rules in event classes, which are always specified independently from rules at an object type, the execution order of the rule's action may be expressed in the event class's situation diagram. Thus, Guideline 7 has to be obeyed explicitly.

In A/OODBMT, events are always specified globally and thus may be reused by different rules. Nevertheless, the execution order of the rule's action may be encoded in the event specification. Guideline 7 must be obeyed explicitly.

Guideline 7 cannot be obeyed using Chimera because events cannot be specified independently from rules. The composite event language for specifying complex ("composite") events [MPC96] extends only the event part of the rule definition language.

Guidelines 6 and 7 dealt with several operations (whether dependent or independent) that have to be executed upon the occurrence of a single event. Contrary, a single action may have to be executed upon the occurrence of several different events. Now, let us define another kind of big rules and small rules. A rule triggered upon two or more different events is called a *rule with a big event* ("big rule"). A rule triggered by a single event is called a *rule with a small event* ("small rule"). The involved operation may be invoked either by a single big rule or by several small rules. Specifying big rules keeps the number of rules lower than it would be if each big rule is replaced by several small rules. But sometimes small rules are preferable over big rules.

GUIDELINE 8
Specify several rules that are triggered by different events but invoke the same operation, if the triggering events are likely to change or if the triggering events differ for subtypes.

A big rule should be specified at an object type O only if the contributing events that trigger the big rule are not likely to change and if all events that contribute to the triggering of the big rule at O contribute also to the triggering of the rule at the subtypes of O. Small rules are better to maintain than big rules.

Example: Remember the business rules about the forwarding of loan applications in Chapter 8. (1) A loan application has to be forwarded immediately after the credit rating has been determined if the loan amount and the credit rating are rather high. (2) A loan application has to be forwarded when the decision (whether to grant, to reject, or to forward the application) about the application is late.

These two business rules may be implemented either by a single big rule triggered upon a complex event or by two separate small rules triggered by different events. Since the events are likely to change (e.g., applications should not be forwarded any more immediately after the rating has been determined), the two business rules should be modeled by two separate rules. Further, the time period after which the decision about a loan application is late differs for personal-loan applications and for mortgage-loan applications. Therefore, the two business rules should be modeled by two different rules again. See Section 8.4.4 for the representation of the rules in Active Object/Behavior Diagrams.

Guideline 8 has to be obeyed explicitly using Active Object/Behavior Diagrams, A/OODBMT, and Chimera.

9.2.4 Specializing Situations and Rules

The active behavior of objects may be specialized like the passive behavior or the structure of object types. If object types are specialized, the situations (events and intervals) and the rules that are specified locally to these object types have to be considered for specialization, too.

GUIDELINE 9
If an operation is invoked on objects of different object types by a rule, specify the rule and its triggering events at the object types' common supertype and specialize them accordingly at the subtypes by overriding the inherited rule or the inherited triggering events or by adding new rules or triggering events.

Guideline 9 is well-known for specializing object structure and passive behavior. Riel [Rie96] defines such a guideline by advising to factor the commonality of data and behavior as high as possible in the inheritance hierarchy. But such a guideline has not been proposed by current active database design methods. Business rules that are common to different object types should be specified already at their common supertype and should be specialized appropriately at subtypes.

Example: Mortgage loan applications as well as personal loan applications have to be forwarded automatically and immediately after the rating has been determined and if the amount is rather high and the borrower's disposable income is also high enough. Since both kinds of loan applications have to be forwarded, the corresponding rule and the triggering event should be specified already for a general loan application.

Active Object/Behavior Diagrams, A/OODBMT, and Chimera have to follow explicitly this guideline. Specializing business rules in Active Object/Behavior Diagrams is discussed in detail in Section 8.4. A/OODBMT allows to specialize only rules but not to specialize events; events are always specified globally (cf. Section 3.6). Chimera provides targeted rules, which are attached to object types. It does not become clear from the available literature [CBFP95, CFPT96, MPC96] how targeted rules may be specialized. Since rule events are not specified separately, they cannot be specialized separately from rules provided that rules can be specialized at all.

Attaching situations and rules to any object type within the inheritance hierarchy again helps to divide the global rule set into local rule sets, whose rules' behavior is easier to comprehend (cf. Section 9.2.2). If the inheritance hierarchy is narrow and deep—this has already been suggested for object-oriented design, e.g., in [Rie96]—and if rules and situations are attached to object types, rules and situations are distributed over the inheritance hierarchy and their behavior becomes again more comprehensible.

Situations and rules are specialized in Active Object/Behavior Diagrams as follows. Situations and rules that are specified locally to some supertype are specialized by overriding them at a subtype. Overriding inherited situations and rules has to follow consistency rules as defined for object structure and passive behavior (cf. Chapter 8). The specialization rule for obligations requires that if performing an operation is obligatory for an object viewed as instance of a supertype it has to be obligatory also for an object viewed as instance of a subtype. Contrary to Active Object/Behavior Diagrams, A/OODBMT allows rules to be freely overridden at a subtype. Events—A/OODBMT provides only events but not intervals—are always specified globally.

The remainder of this section introduces guidelines that are established especially for Active Object/Behavior Diagrams. They cannot be applied to A/OODBMT or to Chimera because they are based on concepts that are currently provided only by Active Object/Behavior Diagrams.

A condition that is inherited from some supertype is usually specialized by replacing it with the condition specified at the subtype. Consistency rules that restrict a condition to be either strengthened or weakened at the subtype are hard to check at compile-time (for a general underlying model as it may be specified by means of Active Object/Behavior Diagrams) if the inherited condition is simply replaced by its specialized condition at the subtype.

The condition inherited from the supertype should not be replaced by a condition specified at the subtype, but it should be enlarged automatically with the condition specified at the subtype: (1) If a condition has to be strengthened at subtypes (e.g., the activity postcondition), the condition inherited from the supertype should be enlarged conjunctively with the condition specified at the subtype. (2) If a condition has to be weakened at subtypes (e.g., the activity permission, the activity prohibition, the rule condition), the condition inherited from the supertype should be enlarged disjunctively with the condition specified at the subtype.

Now, we give a guideline about specifying a condition at a subtype, which enlarges either conjunctively or disjunctively the condition inherited from the supertype.

> GUIDELINE 10
> *If necessary conditions inherited from a supertype are added conjunctively to the additional conditions specified at the subtype, the additional conditions at the subtype should not overlap with the inherited conditions. If sufficient conditions inherited from a supertype are added disjunctively to the additional conditions specified at the subtype, the additional conditions at the subtype should not overlap with the inherited conditions.*

Guideline 10 says that that part of the condition added at the subtype should not be satisfied, if the inherited part of the condition is already satisfied. This guideline may be applied to all parts of database design where conditions are specified textually, i.e., to object structure (static integrity constraints), to passive behavior (pre- and postconditions of activities), and to active behavior (rule conditions).

Example: Consider an arbitrary sufficient condition C_T specified at type T as $x > 5$. This condition has to be weakened at type T', a subtype of T. If the condition specified at T is added disjunctively to the condition specified at T', it is sufficient to specify $2 \leq x \leq 5$ at T' in order to achieve $x \geq 2$ for $C_{T'}$, which represents a weakened specialization of C_T.

If Guideline 10 is obeyed, exactly that part of a condition can be illustrated that strengthens or weakens the inherited condition in the type hierarchy. This makes the inheritance hierarchy and the conditions specified within this hierarchy better to comprehend and facilitates also testing the design because it is easier to recognize whether a condition specified at the supertype or at the subtype is satisfied.

Although conditions are specialized by enlarging inherited conditions conjunctively or disjunctively with the conditions specified at subtypes in Active Object/Behavior Diagrams, the designer has to obey explicitly this guideline. Since A/OODBMT and Chimera do not enlarge the conditions in this way, they cannot make use of Guideline 10. The object-oriented programming language Eiffel [Mey92b] uses this technique to specialize the pre- and postconditions of operations ("routines"). If a precondition is specified once with the keyword "require" it may be extended (weakened) at a subtype only with the keyword "require else." If a postcondition is specified once with the keyword "ensure" it may be extended (strengthened) at a subtype only with the keyword "ensure then." Guideline 10 has to be obeyed also explicitly using Eiffel.

9.3 Further Design Support

Design guidelines and design patterns support the designer in different ways. Design guidelines can highlight a problem in facet of a design while patterns can provide a solution [Rie96]. Kappel et al. introduce in [KRSRS96] *rule patterns* as a 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 providing predefined application-independent components. Rule patterns are organized in sets of patterns, which are collected in a pattern library.

Patterns may also be introduced for Active Object/Behavior Diagrams by parametrizing situation diagrams and activation diagrams. Situation diagrams holding input parameters could be specified in advance. The input parameters and the properties of the "meta object type" to which such a parametrized situation diagram is attached can be used to formulate conditions in the situation diagram. Activation diagrams define already patterns for their execution semantics. Rules defined by an activation diagram represent either start-rules or do-rules and are either experimental or definite. Start-rules may again define patterns for its maximum execution delay.

Chapter 10

Conclusion

Contents

This chapter gives a brief overview of the work presented in this thesis. Section 10.1 summarizes the main achievements and the major contributions of Active Object/Behavior Diagrams to conceptual active object-oriented database design. Section 10.2 shortly discusses an editor for drawing Active Object/Behavior Diagrams. Section 10.3 enumerates some points left open in this thesis and outlines ideas for further work.

10.1 Summary

Modeling business rules at the conceptual level requires a different set of concepts than currently provided by active object-oriented database systems or by recent approaches to active object-oriented database design. After examining the basic features of active object-oriented database systems in Chapter 2 and after analyzing current approaches to active object-oriented database design in Chapter 3, we have identified the requirements on event languages and on the semantics of rule execution necessary for modeling business rules from an external user's perspective (cf. Section 3.3).

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], a graphical object-oriented design method for modeling the structure and the passive behavior of objects, with Situation/Activation Diagrams [LOS96, LOS97] for modeling business rules, which are represented by the active behavior of objects.

Object/Behavior Diagrams. Object/Behavior Diagrams (Chapter 4) have been introduced by Kappel and Schrefl in [KS91a] and extended later on in various directions such as local referential integrity constraints, specialization rules, etc. (cf. [Kap91, KS91b, KS92, KS94, KS96b, Sch90, SS95, SS97]).

Object/Behavior Diagrams provide a uniform graphical representation of the structure of objects (by means of *object-type diagrams*) and of the passive behavior of objects (by means of behavior diagrams, which are, for better readability, split into *life-cycle diagrams*, *activity-specification diagrams*, and *activity-realization diagrams*), 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 defined as properties of objects, and therefore they are generally referred to as local object classes.

We have extended the original definition of Object/Behavior Diagrams by introducing object-class diagrams and query diagrams. 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 in OQL-like structure in order to determine the values or objects of a derived attribute or relationship.

Situation/Activation Diagrams. Situation/Activation Diagrams are integrated seamlessly into Object/Behavior Diagrams. They consist of situation diagrams and of activation diagrams. *Situation diagrams* [LOS96] (Chapter 5) allow to model declaratively when a situation (i.e., an event or interval) occurs by means of a graphical query diagram with an OQL-like structure. A situation is said to occur if the occurrence time of a situation that is selected from other situations or that is generated upon the occurrence of other situations is the current system time. Situation diagrams are based upon query diagrams but provide additional concepts for the temporal comparison of situations. 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. *Activation diagrams* [LOS97] (Chapter 6) allow to model ECA rules, which are triggered upon the occurrence of an event by performing the action, provided that the condition is satisfied. Situation diagrams populate the event classes whose events trigger rules, which are specified by means of activation diagrams. Rules are always local to the object type to which the rule's event class and the activity that serves as the rule's action belong.

The concepts provided by Situation/Activation Diagrams meet the requirements for modeling business rules at the conceptual level (cf. Section 3.3):

1. Situations are viewed from the external user's perspective, i.e., events and intervals

are visible after the transaction in which they are raised commits. All actions performed within a transaction receive the same time-stamp and not the time-stamp denoting when they are processed within the transaction.

2. Events and intervals are treated as first-class objects. We distinguish between situation classes collecting situation occurrences and situation types describing the structure of situation occurrences.

3. Calendar situations are predefined in the past as well as in the future. Business rules often refer to calendar situations in the future.

4. Our time model abstracts from the granularity of the time axis defined by the internal system clock. E.g., having minutes as the smallest temporal unit ("chronon") is sufficient for many business applications. But notice that events occurring in any order within a chronon thereby are recognized to occur simultaneously because they carry the same time-stamp.

5. Situation diagrams are specified declaratively because end users usually prefer a declarative language over an operational one, similarly as QBE and SQL are better accepted by end users than Relational Algebra.

6. Rules specified by means of activation diagrams are executed reliably (i.e., transactions performing rule executions are restarted automatically if they are aborted due to serialization failures) and controllably (i.e., exception events are reported if exceptional situations occur during rule execution).

7. Rules are either executed asynchronously to the triggering event or immediately to the triggering event in their own top-level transaction and not in the event-triggering transaction. They are executed in a "temporally faithful" manner in the sense that user transactions an transactions performing rule executions are serialized in an order which rule modelers and system designers intuitively expect.

Active Object/Behavior Diagrams Revisited. The concepts of Active Object/Behavior Diagrams have been extended in Chapter 7 to make the concepts of specializing passive and active behavior more intelligible.

Wieringa et al. [WWMD91] treat the inheritance of integrity constraints and distinguish two kinds of preconditions required for the execution of operations: necessary preconditions and sufficient preconditions. A *necessary precondition* must be satisfied in order to invoke an operation at all but does not guarantee that it is completed successfully. A satisfied *sufficient precondition* guarantees that an operation is executed successfully and therefore is much stronger than a necessary condition. Before their revision, Active Object/Behavior Diagrams have allowed to specify only necessary preconditions but not to specify sufficient preconditions for activities. The distinction between the two kinds of preconditions has allowed to combine the benefits of two strategies in behavior specialization, which have been assumed to be conflicting views of the world [ES94]: covariance and contra-variance (see below for more details).

Introducing the concepts of necessary and sufficient preconditions has brought major advantages to Active Object/Behavior Diagrams. Business experts are more familiar with the notions permission, prohibition, and obligation than with the concepts precondition, situation, and rule. Therefore, the former notions are better be applied for modeling the business domain. Permission, prohibition, and obligation represent also the core concepts in a branch of philosophical logic, called deontic logic. Deontic logic is a modal logic that deals with the actual as well as the ideal behavior of systems in terms of permission, prohibition, and obligation [WM93]. As Active Object/Behavior Diagrams allow to specify the behavior of objects, which may describe a system or are part of a system, we have borrowed the terms *permission*, *prohibition*, and *obligation* from deontic logic to denote sufficient preconditions of activities, negated necessary preconditions of activities, and rules.

Specialization. Inheritance is the basic concept used in all object-oriented design methods (and thus also in Active Object/Behavior Diagrams) for specializing object structure and object behavior. Inheritance defines a relationship between two object types, where one object type, called the subtype, inherits the structure and the behavior from the other one, called the supertype.

Chapter 8 has shown how object structure, passive behavior, and active behavior can be specialized by *redefinition* (i.e., overriding an object type's features at subtypes—its features are properties, activities, states, situation classes, situation diagrams, and rules), by *extension* (i.e., adding new features at subtypes), and by *refinement* (i.e., replacing abstract features by newly introduced features).

We have required that only the object types that represent leaves of an inheritance tree can be instantiated as required also in [KS96b]. This does not limit the power of Object/Behavior Diagrams, but the design of object types becomes more comprehensible. Objects may be viewed as if they were instances of some supertype in order to check their correct specialization.

The basic rule for specializing inherited conditions at subtypes is that sufficient conditions may become only weaker at subtypes, whereas necessary conditions may become only stronger at subtypes [WWMD91]. Consequently, distinguishing between sufficient precondition (permission) and necessary precondition (negated prohibition) of activities has allowed us to follow both the contra-variance rule and the covariance rule with the specialization of activities, although contra-variance and covariance are seen as views of two conflicting worlds [ES94]. The contra-variance rule allows that preconditions (i.e., sufficient preconditions) become weaker and that postconditions (i.e., necessary postconditions) become stronger at subtypes. The covariance rule allows that preconditions (i.e., necessary preconditions) as well as postconditions (i.e., necessary postconditions) become stronger at subtypes. As a further consequence, strong invocation consistency [SS95] is required for the specialization of object life cycles. Strong invocation consistency ensures both observation consistency, which is in the line of covariance, as well as weak invocation consistency, which is in the line of contra-variance.

Obligations are assumed to be sufficient conditions as they say when something is to be (ultimately) done. Therefore, they may become only weaker at subtypes, which means that what is obligatory for an object viewed as instance of a supertype has to be obligatory for the object viewed as instance of a subtype. Since obligations are modeled by means of rules and their triggering situation classes, rule conditions and situation diagrams—the latter populate the triggering situation classes—may become only weaker at subtypes.

Whether object structure, passive behavior, or active behavior is specialized correctly is determined by a set of consistency rules for each kind of specialization (redefinition, extension, and refinement).

Design guidelines. The usage of modeling concepts such as provided by Active Object/Behavior Diagrams is restricted by a set of consistency rules. Consistency rules are either guaranteed by the modeling concepts or have to be obeyed by the designer. They have the designer stick to a correct usage in order to achieve a consistent database schema.

Even if all consistency rules are obeyed, different designers may produce different consistent database schemata. In addition to consistency rules, guidelines should help the designer to achieve a high-quality database schema, i.e., a comprehensible and maintainable database schema. Active Object/Behavior Diagrams together with design guidelines fill the need for the methodological support in analysis and design issued by Simon and Kotz-Dittrich [SKD95].

Primarily, Active Object/Behavior Diagrams and the design guidelines given in Chapter 9 help to avoid a large set of rules by specifying rules locally to object types. Further, a clear distinction between active and passive behavior is achieved by modeling only obligations to act by means of rules and not, for example, integrity constraints, which are specified declaratively at the conceptual level. Nevertheless, declaratively specified integrity constraints may be maintained by means of rules at the logical level. Further guidelines help to decide whether to use "small rules" or "big rules." Finally, guidelines support the designer to specialize the active behavior, for example, by advising to factor the active behavior common to several object types as high as possible in the inheritance hierarchy.

10.2 Tool Support

The graphical modeling concepts provided by Active Object/Behavior Diagrams could be drawn by hand. Although Active Object/Behavior Diagrams provide high-level concepts, they are very expressive and thus complex. Therefore, using them 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.

Designing a conceptual database schema by means of Active Object/Behavior Diagrams is supported by a graphical editor, the so-called *AOBD editor*. A prototype 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 syntactical correctness of the diagrams is ensured by disallowing to apply the components of diagrams in an invalid way. The semantical 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. The semantic correctness of such expressions is not yet checked.

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.). Currently, the AOBD editor supports Active Object/Behavior Diagrams as described in Chapters 4, 5, and 6. Extended specialization concepts as described in Chapter 8 are not yet supported.

10.3 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.

[1] We thank Honeywell for making a DoME license available for us at no cost.

- 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. Sections 3.6 and 9.3).

- 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.

- 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.

- Three kinds of specialization are currently distinguished: redefinition, extension, and refinement. Once an abstract feature (property, activity, state, situation diagram, rule) is refined at an object type, the refinement must not be modified at some subtupe. Yet the features in the refinement may be further refined. Extending the refinement of an abstract feature at some subtype requires additional consistency rules and may be considered in the future.

- In contrast to situation diagrams, query diagrams cannot be specialized at all. Situation diagrams represent a sufficient condition over the history of time and the database. Sufficient conditions may become weaker at subtypes, which means that all situations that occur for an object viewed as instance of a supertype have to occur for the object viewed as instance of a subtype. Query diagrams determine "only" the value(s)/object(s) for an object's properties and do neither represent necessary nor sufficient conditions. In order to allow to specialize query diagrams, derived properties should be given a range of valid results besides their type domain. The range may be more and more restricted at subtypes. Then, a query diagram may be specialized in any way as far as its result remains within the range given for the property with which it is associated.

- Incorporating new concepts as suggested above requires additional design guidelines that help to achieve a high-quality database schema if the new concepts are used. Nevertheless, additional guidelines may also be found without using the new concepts. A further practical application of Active Object/Behavior Diagrams may help to find further design guidelines.

- Active Object/Behavior Diagrams represent a high-level diagramming technique for the design of conceptual database schemata. The concepts are high-level enough that some of them can be used also for workflow modeling if certain aspects like organizational units, authorization, etc. are considered (cf. [Bic97]).

- The AOBD-editor prototype currently supports "only" the concepts presented in Chapters 4, 5, and 6. The specialization of object structure, passive behavior, and active behavior as described in Chapter 8 is not yet supported. A full-fledged AOBD editor should incorporate the specialization concepts, provide "compile-time" consistency checking as far as possible, supervise the enforcement of design guidelines, and provide the ability to generate automatically a logical database schema including "run-time" checking of consistency. The concepts for mapping the conceptual schema specified by means of Active Object/Behavior Diagrams into a logical schema have already been developed (cf. [Obe97]).

List of Figures

List of Tables

Bibliography

[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 3rd 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. In Bunemann and Jajodia [BJ93], pages 99 – 108.

[Aqv86] L. Aqvist. Deontic Logic. In Gabbay and Guenther [GG86], chapter II.11, pages 607–714.

[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.

[Bic97] P. Bichler. *Conceptual Modeling of Authorization in Workflow Systems*. Dissertation (in preparation), University of Linz, Austria, 1997.

[BJ93] P. Bunemann and S. Jajodia, editors. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC*, volume 22 of *SIGMOD RECORD*. ACM Press, June 1993.

[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.

[BM91] E. Bertino and L. Martine. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, 24(4), April 1991.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications (2nd edition)*. Benjamin Cummings, 1994.

[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.

[BS95] P. Bichler and M. Schrefl. Inheritance of Business Rules. In C.Ẽckert et al., editor, *7. Workshop "Grundlagen von Datenbank Systemen"*, Hildesheimer Informatik-Berichte, June 1995.

[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.

[Car88] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation, Academic Press*, 76, 1988.

[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.

[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.

[CKTB94] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. H. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. Technical Report UF-CIS-TR-94-023, University of Florida, Computer and Information Sciences, May 1994.

[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.

[CM93] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language For Active Databases. Technical Report UF-CIS-TR-93-007, University of Florida, Computer and Information Sciences, March 1993.

[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.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985. Not used in article: aoodb.

[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.

[DMW94] F. Dignum, J.-J. Meyer, and R.J. Wieringa. Contextual Permission—A Solution to the Free Choice Paradox. In A.J.I. Jones and M. Sergot, editors, *Proc. of the 2nd Int. Workshop on Deontic Logic in Computer Science (DEON)*, pages 107–180. Norwegian Research Center for Computers and Law, 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.

[EKW92] D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-Oriented Systems Analysis—A Model-Driven Approach*. Yourdon Press, 1992.

[ES94] S. Even and M. Sakkinen. The safe use of polymorphism in the O_2C database language. Technical Report 021, GOODSTEP ESPRIT-III project No. 6115, July 1994.

[Esw76] K. Eswaran. Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated Data Base System. Research Report RJ1820, IBM, August 1976.

[FG90] G. Friederich and G. Gottlob. Inferenzstrategien. In G. Gottlob, T. Früwirth, and W. Horn, editors, *Expertensysteme*, Springers Angewandte Informatik, chapter 6, pages 117–150. Springer, 1990.

[FW95] R. Feenstra and R.J. Wieringa. Validating Database Constraints and Updates Using Automated Reasoning Techniques. In B. Thalheim, editor, *Proc,. of the Workshop on Semantics in Databases*, pages 24–32, 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.

[GG86] D. Gabbay and F. Guenther, editors. *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic.* D. Reidel, 1986.

[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. In Stonebraker [Sto92], pages 81–90.

[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.

[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.

[Har91] W. Harrison. Contravariance for the rest of us. *Journal of Object-Oriented Programming*, pages 10–18, November 1991.

[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 Computer Aided 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.

[HKRSR97] R. Hiebl, G. Kappel, S. Rausch-Schott, and W. Retschitzegger. A Tour on the TriGS Active Database System - Architecture and Implementation. Technical Report 6/97, Univeristy of Linz, Department of Information Systems, 1997.

[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.

[Jon93] D. Jonscher. Extending Access Control with Duties—Realized by Active Mechanisms. In B.M. Thuraisingham and C.E. Landwehr, editors, *Database Security, VI: Status and Prospects (A-21)*, pages 91–111. Elsevier Science (North-Holland), 1993.

[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.

[KD93] A. Kotz-Dittrich. Adding Active Functionality to an Object-Oriented Database System—A Layered Approach. In W. Stucky and A. Oberweis, editors, *Datenbanksysteme für Büro, Technik und Wissenschaft*, pages 54–73. Springer, 1993.

[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.

[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 Informaion 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. Not used in article: aoodb.

[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.

[KS95] P. Kueng and M. Schrefl. Spezialisierung von Geschäftsprozessen am Beispiel der Bearbeitung von Kreditanträgen. *HMD: Theorie und Praxis der Wirtschaftsinformatik*, 32(185):78–94, September 1995.

[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. (in German).

[KS96b] G. Kappel and M. Schrefl. *Objektorientierte Informationssysteme—Konzepte, Darstellungsmittel, Methoden*. Springer, 1996. (in German).

[KS96c] J. Krogstie and G. Sindre. Utilizing Deontic Operators in Information Systems. *Requirements Engineering*, 1(4):210–237, 1996.

[KT97] W. Kraus and T. Thalhammer. Design and Implementation of a Graphical Editor for Active Object/Behavior Diagrams. Master's thesis, Universität Linz, Austria, 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.

[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.

[Mai93] T. Maibaum. Temporal Reasoning over Deontic Specifications. [MW93b], chapter 7, pages 141–202.

[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.

[Mey92a] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.

[Mey92b] B. Meyer. *Eiffel—The Language*. Prentice Hall Object-Oriented Series. Prentice Hall, 1992.

[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.

[MW93a] J.-J.Ch. Meyer and R.J. Wieringa. Deontic Logic: A Concise Overview. [MW93b], chapter 1, pages 3–16.

[MW93b] J.-J.Ch. Meyer and R.J. Wieringa. *Deontic Logic in Computer Science: Normative Systems Specification*. Wiley, 1993.

[Obe97] W. Obermair. *Active Object-Oriented Database Design—From Conceptual Design to Logical Design*. Dissertation (in preparation), Universität Linz, 1997.

[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.

[RN95] S.J. Russel and P. Norvig. *Artificial Intelligence—A Modern Approach.* Prentice Hall, 1995.

[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.D̃os 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.

[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.

[Sto92] M. Stonebraker, editor. *Proceedings of the 1992 ACM SIGMOD International Conference of Data, San Diego, CA*, volume 21 of *SIGMOD Record*. ACM Press, June 1992.

[SW95] A.P̃. Sistla and O. Wolfson. Temporal Triggers in Active Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):471 – 486, June 1995.

[Tai96] A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[Tan92] A.K. Tanaka. *On Conceptual Design of Active Databases*. PhD thesis, Georgia Institute of Technology, 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.

[Urq86] A. Urquhart. Many-valued Logic. chapter III.2, pages 71–116. D. Reidel, 1986.

[VGD97] A. Vaduva, S. Gatziu, and K.R. Dittrich. Investigating Termination in Active Database Systems with Expressive Rule Languages. Technical Report 97.03, Institut für Informatik, Universität Zürich, April 1997.

[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].

[Wei93] H. Weigand. Deontic Aspects of Communication. [MW93b], chapter 11, pages 259–273.

[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.

[WM93] R.J. Wieringa and J.-J.Ch. Meyer. Applications of Deontic Logic in Computer Science: A Concise Overview. [MW93b], chapter 2, pages 17–40.

[WMW89] R.J. Wieringa, J.-J.Ch. Meyer, and H. Weigand. Specifying Dynamic and Deontic Integrity Constraints. *Data & Knowledge Engineering*, 4(2):157–190, 1989.

[WWMD91] R.J. Wieringa, H. Weigand, J.-J.Ch. Meyer, and F.P.M. Dignum. The Inheritance of Dynamic and Deontic Constraints. *Annals of Mathematics and Artificial Intelligence*, 3(2–4):393–428, 1991.

[WZ88] P. Wegner and S.B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing and K. Nygaard, editors, *Proc. of the European Conference on Object-Oriented Programming*, LNCS 322, pages 55–77. Springer, 1988.

Curriculum Vitae

Personal Record

Name:	Dipl.-Ing. Peter Lang
Date of Birth:	September 20, 1972
Place of Birth:	Ried im Innkreis
Citizenship:	Austria
Home Address:	A-4040 Linz, Baumgärtelstraße 3/2

University Education

1990–95: Study of computer science at the Johannes Kepler University of Linz, Austria

1995–97: Ph.D. study at the Johannes Kepler University of Linz, Austria

Professional Experience

1992–95: Tutor for courses in programming languages at the Johannes Kepler University of Linz

1993–95: Design and development of relational database applications at the Programmierfabrik Hagenberg

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