Conceptual Design of Secure Workflow Systems

An Object-Oriented Approach to the Uniform Modeling of Workflows, Organizations, and Security

DISSERTATION
zur Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

Eingereicht an der Johannes Kepler Universität Linz
Institut für Wirtschaftsinformatik
Forschungsschwerpunkt 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 Bichler**

Linz, im Mai 1998

Eidesstattliche Erklärung

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

Linz, am 7. Mai 1998

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die zum Gelingen dieser Arbeit beigetragen haben. Mein ganz besonderer Dank gilt Michael Schrefl. Er hat durch seine Kritik und seine wertvollen Anregungen einen unersetzlichen Beitrag zum Gelingen dieser Arbeit geleistet. Bei Gerti Kappel bedanke ich mich recht herzlich für die Übernahme der Zweitbegutachtung.

Weiters möchte ich mich bei allen Kolleginnen und Kollegen am Institut für Wirtschaftsinformatik der Universität Linz (Forschungsschwerpunkt Data & Knowledge Engineering), Heidelinde Kneissl, Peter Küng, Peter Lang, Ingeborg Liedlbauer, Werner Obermair, Günter Preuner, Thomas Prückler, Brigitte Röck, Margit Rudelstorfer und Roman Schendl, für das sehr angenehme Umfeld, in dem diese Arbeit entstehen konnte, bedanken.

Besonderer Dank gebührt meiner Familie, vorallem meiner Mutter Marianne, die viel zu früh verstorben ist, und meinem Vater Helmut. Sie haben mich über all die Jahre hinweg unterstützt. Ein besonders großes Dankeschön gilt meiner Verlobten Roberta für ihr Verständnis und ihre liebevolle Unterstützung.

Kurzfassung

Der konzeptuelle Entwurf von Workflow-Systemen umfaßt die Modellierung von organisationalen Prozessen, der Struktur von Organisationen, sowie von Sicherheitsanforderungen. Wir stellen ein umfassendes, konzeptuelles Workflow-Model vor, das für die frühen Phasen des Entwurfs von Workflow-Systemen mit hohen Sicherheitsanforderungen gedacht ist. Das Workflow-Modell folgt einem einheitlich objekt-orientierten Ansatz.

Die *Prozesse von Organisationen* sind aufgrund der sich ständig ändernden Marktbedingungen häufigen Änderungen unterworfen. Wir stellen eine neuartige Schema-Architektur für die Modellierung von organisationalen Prozessen vor. Die sogenannte „Zwei-Schemen-Architektur" trennt Wissen über organisationale Prozesse entsprechend seiner Herkunft in *externes* Wissen (z.B.: natürliche Fakten und Gesetze) und *internes* Wissen (organisatorische Regelungen). Da Organisationen keinen, zumindestens keinen direkten, Einfluß auf externes Wissen haben, führen die meisten Änderungen an organisationalen Prozessen ausschließlich zu Änderungen des internen Wissens. Die „Zwei-Schemen-Architektur" unterstützt in einem hohen Maße die Wiederverwendung von Wissen über organisationale Prozesse und erlaubt eine leichte und schnelle Anpassung von organisationalen Prozessen an geänderte Rahmenbedingungen.

Die *Struktur von Organisationen* wird traditionell aufgrund von Geschäftsfunktionen gestaltet. Die Strukturen moderner Organisationen sind um deren Prozesse gestaltet, wodurch flachere Strukturen entstehen, die nur mehr wenige Hierarchie-Stufen aufweisen. Handlungsträger werden in netzwerkartige Gruppen zusammengefaßt, die schnell an geänderte Rahmenbedingungen angepaßt werden können. Wir unterstützen die Modellierung von hierarchischen und netzwerkartigen Organisationsstrukturen.

Die *Sicherheitsanforderungen* von organisationalen Prozessen beschreiben im wesentlichen, an welche Handlungsträger welche Zugriffsrechte ausgestellt werden dürfen, in welcher Form bestimmte Zugriffsrechte repräsentiert werden müssen (z.B.: Magnetkarten) und wie diese Zugriffsrechte zu verwalten sind. Grundlegende Anforderungen an die Sicherheit in Workflow-Systemen sind: (1) Handlungsträger dürfen Arbeitsschritte von organisationalen Prozessen nur dann ausführen, wenn ihr Handeln durch entsprechende Zugriffsrechte autorisiert ist. (2) Zugriffsrechte, die den Schutzbedingungen der jeweiligen organisatorischen Prozesse widersprechen, dürfen nicht gewährt werden. Wir beschreiben die Sicherheitsanforderung eines Workflow-Systems in einem *Autorisierungsschema*, mit dessen Hilfe ein Workflow-Management-System sicherstellen kann, daß nur Zugriffsrechte existieren bzw. vergeben werden, die den Sicherheitsanforderungen organisationaler Prozesse genügen.

Abstract

The conceptual design of workflow systems comprises the modeling of organizational processes, organization structures, and security requirements. We present a comprehensive, conceptual workflow model that is to be used in early phases of the design of workflow systems that have high demands on security. The workflow model follows a uniform object-oriented approach.

The *processes of organizations* need often be adapted to changed requirements in the business environments of organizations. We propose a novel schema architecture for the modeling of organizational processes. The so-called "two-schema architecture" separates, according to its origin, knowledge on organizational processes into external knowledge, e.g., natural facts and law, and internal knowledge, i.e., organizational commitments. Since organizations have no influence on external knowledge—at least no direct influence—, most adaptions of organizational processes cause only changes to internal knowledge. The "two-schema architecture" supports to a high degree the reuse of knowledge on organizational processes and eases the adaption of organizational processes to changed requirements in the business environments of organizations.

The *structure of organizations* is typically organized around business functions of organizations. The structure of modern organizations is organized around the processes of organizations. As a result, the vertical hierarchy of organizations is flattened. Typically, actors work in network-like groups, which may be easily adapted to changed requirements. We support the modeling of both forms of organization structures, i.e., hierarchies and networks.

The *security requirements* of organizational processes say to which actors authorizations can be issued, in which form particular authorizations must be represented (e.g., key cards), and how authorizations that have been issued to actors must be maintained. Basic security requirements for workflow systems are: (1) Accesses of actors that do not possess appropriate authorizations must be denied. (2) Authorizations that are in conflict with the security requirements of organizational processes must not be issued. We specify the security requirements of a workflow system in an *authorization schema.* A workflow management system can use such an authorization schema as a filter to exclude authorizations that are illegal wrt. the protection requirements of organizational processes from its authorization base.

Contents

Part I

Introduction

Chapter 1

Motivation

Contents

Workflow management systems (WFMSs) allow to specify, enact, monitor, and coordinate the flow of work items within distributed office environments (e.g., [EN96, GHS94]). Although workflow technology has existed for decades, i.e., first prototype systems have been developed in the late 1970s, industry strength WFMSs are not well established (cf. [EN96]). That is, future research effort is required before WFMSs will mature and can serve as an underlying commodity service of critical business applications. This introductory chapter motivates the work presented in this thesis and classifies it within a conceptual framework.

This chapter is organized as follows. Section 1.1 reviews the historical development of workflow technology and identifies relevant issues for future research in the workflow area. Section 1.2 discusses the workflow-oriented design of information systems. In fact, it builds a conceptual framework that allows to classify the objectives of the thesis presented in Section 1.3. Section 1.4 concludes this chapter by reviewing related work.

Note: WFMS products referred to in this thesis are either trademarks and/or registered trademarks of the respective owners.

1.1 History of Workflow Management Systems

Workflow management systems (WFMSs) have evolved from diverse origins (cf. [JB96, KB95, Wor95]), refer to Georgakopoulos *et al.* [GHS94] or White and Fischer [WF94] for a list of workflow products. Many of the existing systems have evolved from *database management systems*, *document imaging systems*, *electronic mail systems*, *office information systems*, or *software process management systems*. Each of these approaches is offering a variety of strengths and did considerably contribute to the development of workflow technology. However, the fact that each of these approaches had its own terminology and its own interfaces impeded an even faster progress of workflow technology.

Aware of the hindrance of missing standards in workflow technology, vendors of WFMSs founded the *Workflow Management Coalition* (WfMC) in 1993. The mission of the WfMC is to promote the use of workflow technology through the establishment of *standards* (cf. [Wor95]). Currently, standards for workflow terminology and for system architecture and interfaces are available (cf. [Wor96]). Their purpose is to build a common "workflow language" and to improve the interoperability and interchange among workflow products.

The vision of the WfMC is that the standards-based approach allows users of workflow technology to combine the strengths of different workflow products in one infrastructure (cf. [Wor95]). The vision of Sheth [She97] goes a step further and requires for future workflow technology that the integration of currently disparate technologies supporting organizational processes, e.g., database systems, groupware systems, workflow systems, will happen "within the context of a broader and more unified modeling and enactment framework with adequate attention to all key components [...], rather than as a patchwork, interfacing, or interoperability of independent technologies".

Office information systems (OISs), e.g., OfficeTalk (cf. [EN80, Ell82]), are frequently referred to as the first WFMSs (e.g., [Bus96, EN96, JB96, Joo96]). OISs were unsuccessful because of two reasons (cf. [EN96]): First, necessary technologies were either not commonly available in offices, e.g., networking, or were not socially accepted, e.g., personal computers. Second, both the enactment systems as well as the process models of OISs were too rigid to increase productivity of organizations. Today, key technologies are available as well as socially accepted. However, contemporary workflow technology is still said to be too inflexible to meet the requirements of dynamic and evolving organizations of today.

In fact, many leading workflow researchers address in their research agendas the improvement of the flexibility of "next generation" WFMSs as a main issue of future research (e.g., [AS94, EN96, She97]). This means, future WFMSs must allow organizations to easily adapt their processes to changes in their environments. Frequently, buzzwords such as "dynamic change" (e.g., [BPS97a, EK95]), "process evolution" (e.g., [CCPP95]), "process adaptability" (e.g., [She97]), and "process reuse" (e.g., [She97]) occur in this context. Joosten [Joo96] mentions further that more comprehensive modeling techniques are needed, since "many modeling techniques today capture only one or few aspects of organizational processes".

Another key issue for future WFMSs is the development of appropriate *authorization models* (cf. [AS94, Joo96]). Contemporary WFMSs pay little or no attention to authorizations. In fact, they determine who *can* access a workflow operation (or workstep) based on the *responsibility* of actors (cf. [JB96, p. 183]). The assumption that actors who are responsible to perform certain worksteps are at the same time authorized to do so is an oversimplification of matters that impedes the development of secure workflow systems. As Joosten states (cf. [Joo96]): "... an organization will not tolerate a workflow system that schedules work to persons who are not authorized to do that work." That is, future WFMSs must support the modeling of responsibilities *and* authorizations of actors to perform worksteps. Otherwise, workflow systems cannot avoid unauthorized executions of worksteps, which is a primary requirement for secure workflow systems.

Abbott and Sarin did present an approach to *predict* the future evolution of workflow technology. They proposed to compare the evolution of workflow technology to that of database technology (cf. [AS94, p. 115]): "Over time, we expect workflow architecture and technology to mature and grow in a pattern similar to DBMS architecture and technology." Their hypothesis is based on the observation that both technologies, WFMSs as well as DBMSs, are fundamental in that they underlie a whole spectrum of business applications.

Jablonski and Bussler [JB96] published a classification system of workflow technology that is based on the hypothesis of Abbott and Sarin, assuming that WFMSs will evolve in phases similar to DBMSs. They identified three phases: the experimental phase, the conceptual phase, and the professional phase (cf. Figure 1.1). In the experimental phase, both academic as well as commercial prototype systems were developed to collect experiences with the modeling and enactment of processes. *Currently, we are in the conceptual phase.* In this phase, conceptual models, architectures, and first design methodologies will be developed. In the professional phase, there will be a proliferation of tools for the development of workflow applications. Abbott and Sarin predict that finally WFMSs will mature and serve as an underlying commodity service for organizations.

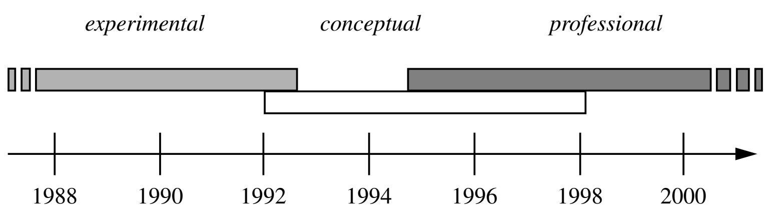


Figure 1.1: Phases of WFMS evolution (source: [JB96, p. 14])

In the conceptual phase of the evolution of DBMSs, consensus has been reached on the decomposition of the design process into phases, i.e., database design has been decomposed into conceptual design, logical design, and physical design (cf. [BCN92]), leading to a reduction of the complexity of the overall database design process. Furthermore, several conceptual models have been proposed for the conceptual design of databases. A conceptual model is a language used to build conceptual schemas, which are high-level descriptions of the structure of databases independent of particular DBMS products. The independence of a conceptual database schema from a particular DBMS product has several advantages (e.g., [BCN92]). For example, conceptual design results survive a change of the target DBMS used to implement a database. Note: The Entity-Relationship model (cf. [Che76, TYF86]) is certainly the best known conceptual model for the conceptual design of databases.

Summing up, we emphasize that future research effort shall be put in the development of more comprehensive and more flexible workflow models that explicitly distinguish between authorizations and responsibilities of actors. This means, future workflow models must be flexible enough to cope with the dynamic and evolutionary character of modern organizations and must consider authorization as an integral part of workflow allowing to specify and implement secure workflow systems. Looking at the evolution of database technology, workflow technology could benefit from a separation of the overall design process into phases conceptual, logical, and physical design.

1.2 Information-Systems Design

An *information system* of an organization supports the storage, retrieval, and processing of information on elements from the business area (or "universe of discourse"). Note: We are interested in computer-based information systems.

The development of an information system is a complex activity that requires several steps: feasibility study, requirements collection and analysis, design, prototyping, implementation, validation and testing, and operation (cf. Figure 1.2). We concentrate on the *design of information systems*, which is concerned with the specification of the structure and behavior of relevant elements of the business area of the information system.

1.2.1 An Object-Oriented View of the Business Area

Object-orientation is often referred to as being a new way of thinking about the world. A way that is more natural to humans. Actually, object-orientation views the world as a system of interacting objects. An *object* is a model of some real-world entity, be it a conceptual or a physical entity. Objects have structure and behavior (cf. Figure 1.3).

Inter-related Structured Objects and Static Integrity Constraints. The structure of objects is determined by a set of attributes and a set of relationships. Attributes are descriptive features of objects, e.g., the name of a person. Relationships are references to other objects. In the real-world, references between objects can have different semantics. For example, an object may be a component of a related object. Static integrity

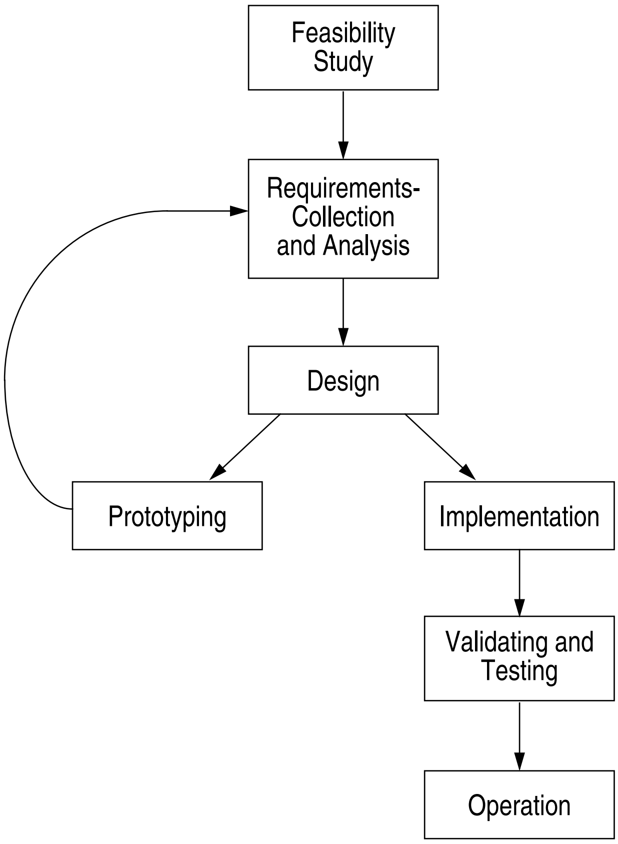


Figure 1.2: Information systems life-cycle (source: [BCN92, p. 5])

constraints express additional static information. They restrict attributes to hold only a limited range of data values and relationships to refer only to a limited range of objects.

Object Behavior and Dynamic Integrity Constraints. The behavior of objects is determined by a set of operations that can be executed on these objects. Operations may be simple reads or updates of attributes and relationships of objects as well as complex computations. Dynamic integrity constraints express additional behavioral information. They specify conditions that must be satisfied before the execution of an operation or after the execution of an operation, and conditions that restrict the order in which operations may be executed.

Note: Leading workflow researchers argued that the everything-is-an-object view of the real-world is too low-level for workflow design (e.g., [GHS94]). The arguments have been that designers must redesign object types that frequently occur in the business area (cf. business area, below) in each project from scratch. In the following, we offer a high-level classification of elements of the typical business area of workflow systems. But remember that all elements found in the business area are objects with structure and behavior. For

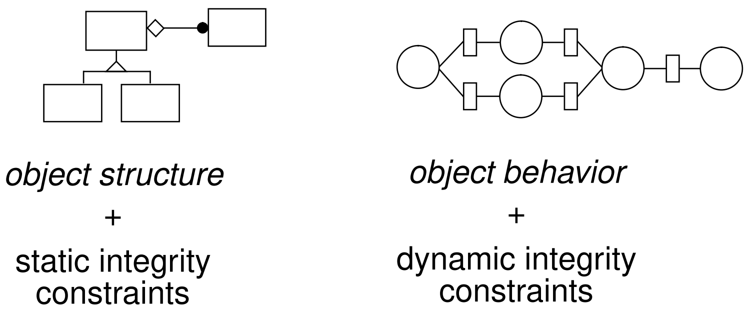


Figure 1.3: Object-oriented view of the business area

each category of elements found in the business area, a predefined conceptual object-type will be presented in Part III of this thesis. Designers can specialize these conceptual object-types to develop workflow schemas.

1.2.2 Elements of the Business Area

The elements of the business area of workflow systems are (cf. Figure 1.4): information objects, organizational processes, inter-related organizational objects such as actors and organizational units, and protection requirements as well as authorizations.

Information Objects. Information objects (sometimes called "entities", "data objects" or simply "data") carry structural information about the business area in attributes and relationships, and behavioral information in operations that can be invoked on them. Essentially, all elements found in the business area carry information about the business area. We classify each object that does not qualify for anyone of the other categories of elements of the business area as an information object.

Organizational Processes. Organizational processes are essential resources of organizations. Each organization carries out a set of organizational processes to achieve its objectives, e.g., offer services or goods to customers. An organizational process is a purposeful collection of activities that add a value for the customer of the organizational process. To carry out the activities of an organizational process, several actors—humans and machines—collaborate.

Inter-Related Organizational Objects. Organizational objects are distinguished into actors and organizational units. Actors play a central role in information systems, since they perform activities of organizational processes. That is, actors are the *active* entities within an organization that help to meet the objectives of the organization. Actors are grouped together into organizational units, organizational units are grouped together into overall organizations thereby determining the physical structure of an organization. Furthermore, each organization defines a control structure determining who is responsible to coordinate or supervise the work of others.

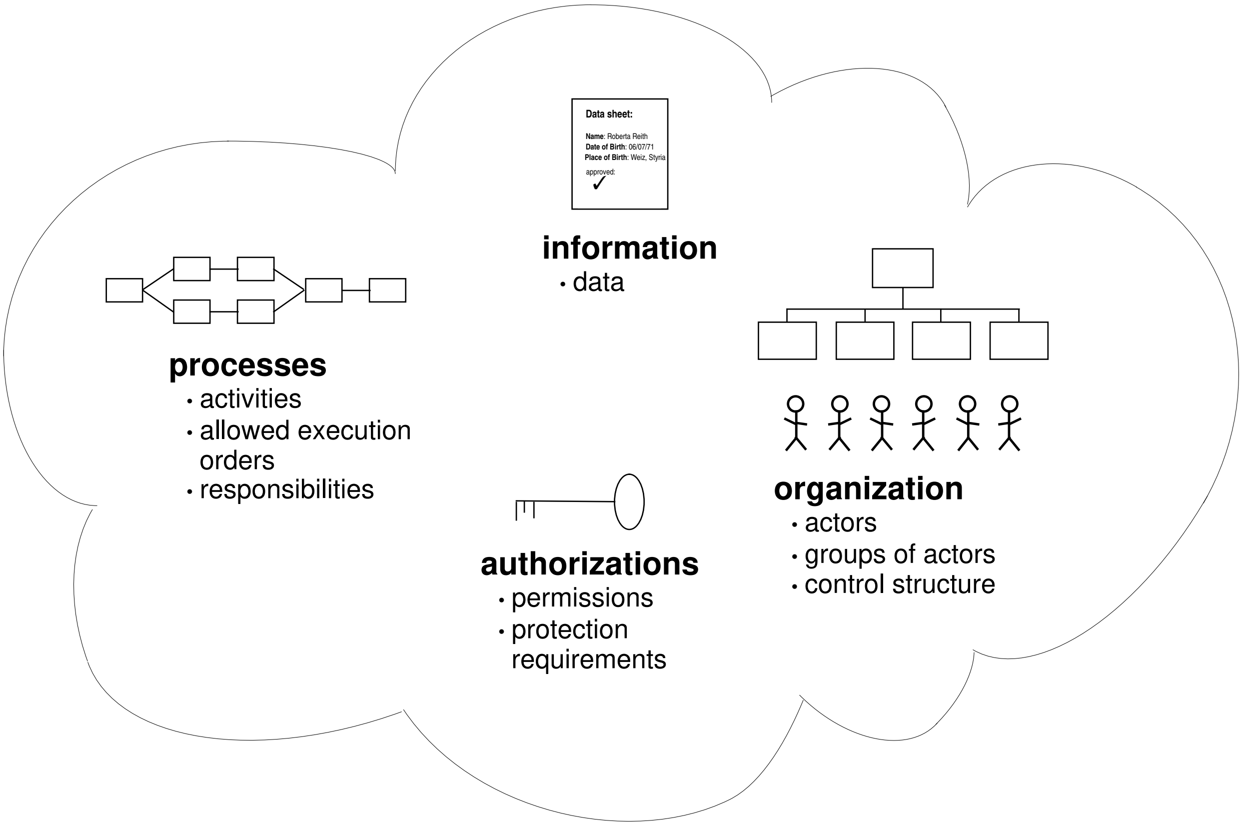


Figure 1.4: Elements of the business area

Authorization Model. Authorizations determine the behavior of an organization in that they express the permissions of actors of an organization to perform particular activities of processes. Whereas organizational regulations determine the responsibilities of actors (obligations), authorizations determine the authority of actors (permissions). In fact, an actor assigned as a responsible actor of a particular activity of an organizational process may perform that activity only if she or he is authorized to do so. Consequently, a permission to act is a prerequisite for an obligation to act. Furthermore, an authorization model must represent the protection requirements of process activities, i.e., authorizations to invoke a particular process activity may not be granted to every kind of actor.

1.2.3 Workflow-Oriented Design of Information Systems

The design of an information system is driven by the implementation strategy selected for the realization of the information system (cf. [JB96]). Basically, information systems can be realized solely by application programs (application-oriented implementation strategy). Traditionally, the design of application programs has followed a function-oriented approach. With the availability of object-oriented programming languages and design methods, the design of application programs may follow an object-driven approach now.

Database systems allow to take out the definition, storage, manipulation, and retrieval of objects from application programs. That is, information systems may be realized by database systems and application programs together (database-oriented implementation strategy). Consequently, the design of an information system is divided into two (possibly parallel) design tasks (cf. [BCN92]), i.e., database design and application-program design, if a database-oriented implementation strategy has been selected.

Workflow-oriented implementation is a novel implementation strategy for information systems. Workflow systems allow to take out the process logic of application programs. That is, information systems may be realized by workflow systems, database systems, and application programs together (workflow-oriented implementation strategy). Consequently, the design of an information system is divided into three (possibly parallel) design tasks, i.e., workflow design, database design, and application-program design, if a workflow-oriented implementation strategy has been selected. Together, workflow design, database design, and application-program design must completely specify the information system (cf. Figure 1.5).

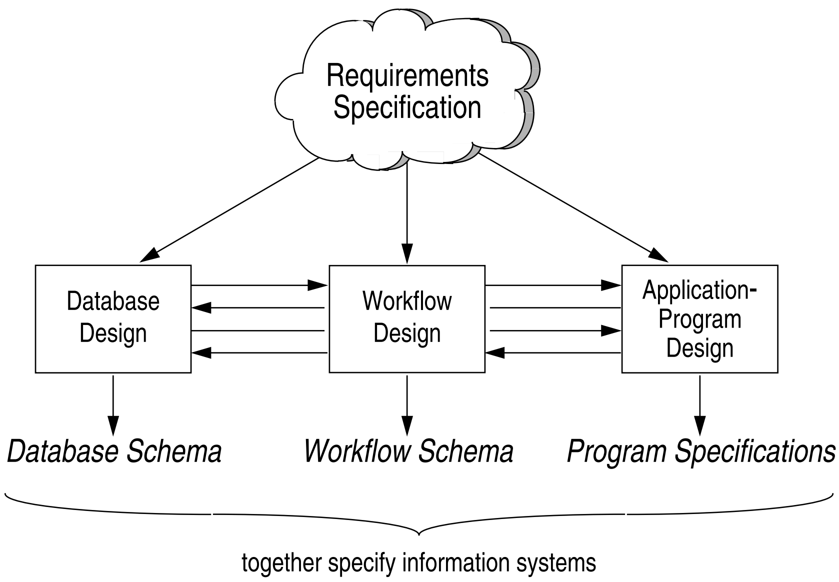


Figure 1.5: Workflow-oriented design of information systems

Workflow design. Workflow design is concerned with the specification of organizational processes, inter-related organizational objects, and protection requirements as well as authorizations. To reduce the complexity of workflow design, we further subdivide the design of workflows into three (possibly parallel) design tasks: process design, organization design, and authorization design.

- *Process design* is concerned with the specification of relevant process activities[1], the specification of allowed execution orders on these process activities, and the specification of actors that are responsible to perform the process activities.

- *Organization design* is concerned with the specification of actors, groups of actors, and physical as well as control structures among actors and groups of actors of an organization.

- *Authorization design* is concerned with the specification of the protection requirements of organizational processes and the specification and maintenance of authorizations that determine the behavior of an organization, i.e., who is authorized to perform what process activities.

Database design. Database design is concerned with the specification of object structures, static integrity constraints, and (depending on the class of DBMS intended to use) object behavior and dynamic integrity constraints. Furthermore, database design is concerned with the specification of transactions. Transactions are sequences of operations that transfer a database from one consistent state into another consistent state.

Application-program design. Application-program design is concerned with the specification of business functions that perform transactions on objects within associated databases and the design of user interfaces[2] that allow actors to interact with the application. In the context of workflow-oriented information systems, actors typically use individual functions of application programs during the execution of activities of organizational processes. That is, in the ideal case application programs are collections of functions that can be configured to the needs of workflow-oriented information systems.

1.2.4 Workflow Design Process

During the design of an information system the problem statement and the requirements are translated into specifications of each component of the system. This means, the design phase of an information system must produce a workflow schema, a database schema, and application-program specifications, if a workflow-oriented implementation strategy has been selected for the realization of the information system (cf. Figure 1.5).

The design of larger information systems is rather complex and requires decisions at very different levels. Experiences in the area of database design showed that for larger information systems, the problem-statement and the requirements cannot be directly translated

[1]Note: The specification of data that flow as parameters into and out of individual process activities is delegated to database design.

[2]Note: In fact, the design of *human-computer interfaces* has emerged as a separate discipline on its own. Since specific software tools are available for the design and implementation of user interfaces (e.g., [Mye97]), user interfaces are taken out of application programs.

into a particular database language. To master the complexity of the design phase, the design of databases has been further divided into phases (cf. [BCN92]).

To master the complexity of the workflow design process, we propose to further divide the design of workflows into phases, too. Due to the tight relation between workflow systems and database systems, i.e., database systems maintain objects processed in workflows, we aim at coordinating their design processes such that it can be easily tested that database schemas and workflow schemas are mutually consistent and complete. Consequently, we propose to further divide the design of workflows into phases conceptual design, logical design, and physical design (cf. Figure 1.6), in analogy to the well-known phases of the database design process.

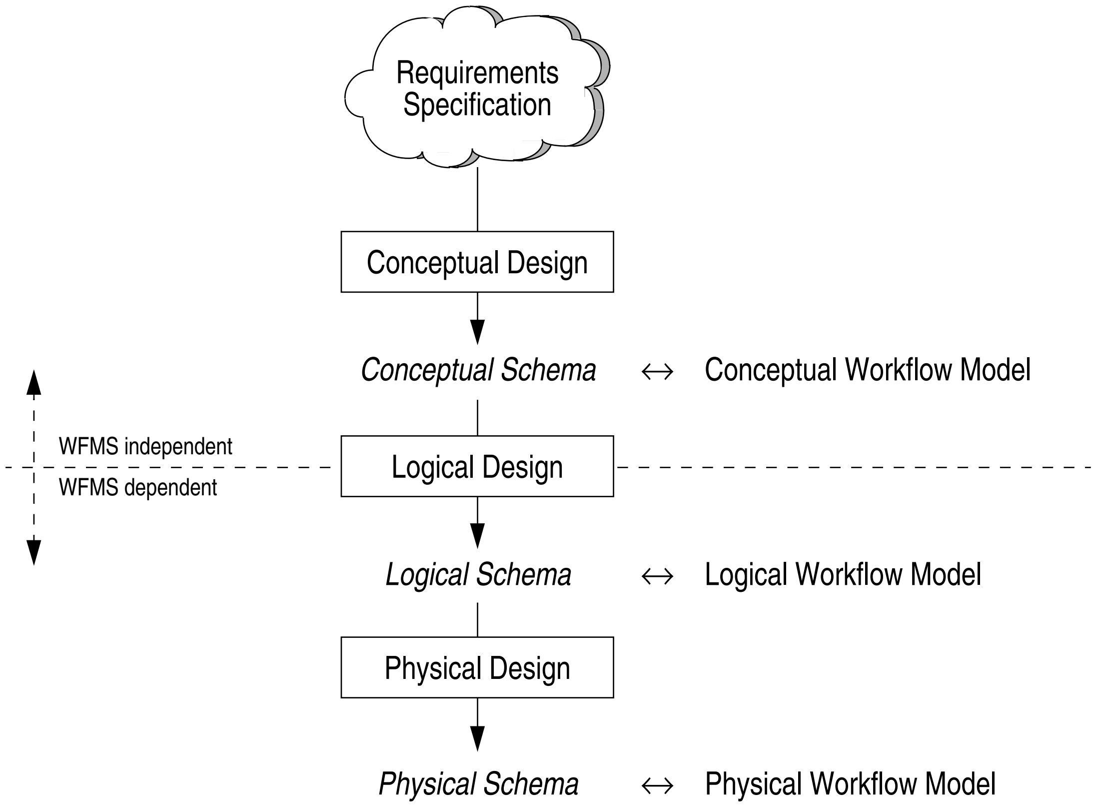


Figure 1.6: Workflow design phases

1. *Conceptual design* translates the problem statement and the requirements into a conceptual workflow schema, which is a high-level description of the real-world semantics of a workflow system, independent of a particular WFMS that will be used to realize the workflow system. A conceptual model is a language used to describe conceptual schemas. A conceptual workflow schema constitutes the fundament for the communication between end users and workflow designers to clarify the requirements of the information system. Therefore, a conceptual model uses some expressive

and easy-to-understand language, usually a set of diagrams, to represent conceptual schemas.

2. *Logical design* translates the conceptual workflow schema into a logical workflow schema, which is a description of a workflow system that can be directly mapped to the implementation model of a particular WFMS. A logical model is a language used to describe logical schemas.

3. *Physical design* defines physical aspects of a workflow systems to improve their performance. For example, decisions taken during physical design are concerned with storage structures and access methods used to effectively access relevant objects, e.g., actors, authorizations, process data, and distribution to raise performance and availability of workflow systems.

1.3 Objectives of the Thesis

> The objective of this research is to develop an authorization model for the conceptual design of secure workflow systems. This authorization model shall be embedded into a workflow model that is comprehensive and flexible wrt. change.

In the sequel we review the individual objectives addressed in the given mission statement in more detail.

Design of Secure Workflow Systems. Basic principles for the design of secure information systems are: (1) *Access rights of actors must be considered as independent concept.* To ensure security in information systems, unauthorized accesses must be detected and denied. Modeling access rights of actors as an independent concept is a prerequisite to allow information systems to distinguish unauthorized accesses from authorized accesses. (2) *"Access restrictions should be defined at the highest possible level where their semantics are explicit"* (cf. [FLPG93, p. 135]). In the context of workflow systems this means that access restrictions should be defined at the level of workflow operations (or worksteps). (3) *The protection requirements of information systems must be formalized.* Another basic principle for the design of secure information systems is that the protection requirements of information systems must be formalized such that only access rights that are in accordance with these protection requirements are issued to actors of information systems.

Conceptual Design of Workflow Systems. Conceptual design of information systems is focused on semantic aspects of the business area rather than on implementation details. A basic feature of conceptual design and of conceptual models is that they rely on a basic set of modeling concepts, which are easy to understand for professional information systems developers as well as for end users. Since specifications of information systems expressed using the language of a conceptual model, e.g., a conceptual workflow schema, constitute the fundament for the communication between end users and information systems developers, conceptual models rely on expressive languages, typically graphical notations, to represent specifications of information systems.

Batini *et al.* [BCN92, pp. 29f] state that conceptual models should possess the following qualities: expressiveness, simplicity, minimality, and formality. A conceptual model is *expressive* if it offers a rich set of modeling concepts. Conceptual models that are expressive allow to represent a large variety of integrity constraints directly in schemas. A conceptual model is *simple* if schemas that are built using that model are easy to understand, both to professional designers and end users. Simplicity and expressiveness are often conflicting goals. A conceptual model is *minimal* if every modeling concept supported in that model has a distinct meaning wrt. every other modeling concept. A conceptual model is *formal* if schemas that are built using that model represent a formal specification of reality. Frequently, conceptual models offer a graphical notation for the design of schemas. In such cases, conceptual models should possess the following qualities, too: graphic completeness and ease of reading. A conceptual model is *graphically complete* if all its modeling concepts have a graphic representation. A conceptual model is *easy to read* if each modeling concept is represented by a graphical symbol that can be clearly distinguished from all other graphical symbols.

Comprehensive Workflow Model. A comprehensive workflow model has to support the specification of all relevant elements of the business area of workflow systems, i.e., processes, inter-related organizational objects, and protection requirements as well as authorizations (cf. workflow design, Subsection 1.2.3).

Flexible Workflow Model. Organizations must frequently adapt their processes to changes in their business environment to stay competitive. An important quality of a workflow model is thus that it offers concepts to support the adaption of workflow systems to changed requirements. Furthermore, a flexible workflow model should support the extendibility and maintenance of workflow systems as well as the reuse of design results.

1.4 Related Work

Research in the workflow area has been dominated by WFMS vendors for a long time. Not surprisingly, vendors of workflow product concentrated on technological aspects with the purpose of bringing workflow tools to the market (cf. [Joo96]). Scientific research on technological aspects has been conducted mainly by database research groups. They address topics such as extended transaction models (e.g., [Hsu93, She93]), and interoperability and scalability (e.g., [AAAM97]). Only few research has been reported on workflow design methodologies and on conceptual design notations so far. Some models for the conceptual design of workflows have been presented by Casati *et al.* [CCPP95], Ellis [Ell93], Joosten [Joo94, Joo95]. However, these conceptual models concentrate on the process modeling aspect (below). That is, these models do not offer concepts to model groups of actors, physical and control structures among actors and groups of actors, or protection requirements of processes and authorizations, which are inherent aspects of workflows (cf. workflow design, Subsection 1.2.3). Recently, research has been reported on security in workflow systems (e.g., [AH96, EPQ94, TS96]).

Related research areas such as *business-process modeling* (e.g., [HC93, Oul95]) and *enterprise modeling* (e.g., [Sch98a, Sch98b, Ver96]) are influential for the development of design methodologies and design notations. Fundamental to business-process modeling as well as to enterprise modeling is to take a process-centric view of organizational behavior and to consider most elements of the business area. Nevertheless, modeling of protection requirements and authorizations is neither considered in business-process modeling nor in enterprise modeling.

In the remainder of this section we review related work in the area of conceptual design of workflow systems, i.e., modeling of *processes*, *organizations*, and *authorizations*. Due to the tight relation between workflow systems and database systems, i.e., workflow operations access and manipulate objects stored in database systems, we also shortly review related work in the area of data modeling. Since we do not consider application-program design in this thesis, we do not review related work in this area.

1.4.1 Process Modeling

The modeling of processes is concerned with the specification of relevant process activities, the specification of allowed execution orders on these process activities, and the specification of actors that are responsible to perform the process activities (cf. process design, Subsection 1.2.3). In the following, we characterize different approaches that WFMSs follow to model the process logic of workflows and discuss the basic conceptual models behind these approaches. Thereafter, we characterize the capabilities of WFMSs to specify responsible actors for process activities.

Comparison of Conceptual Process Modeling Approaches

Activity-based approaches. As the name indicates, the basic concept of activity-based approaches is the activity (sometimes referred to as task). An activity represents a set of operations to be performed by a particular actor on a certain business case as a unit. Many commercial WFMSs provide activity-based workflow models (cf. [GHS94]). Activity-oriented process models can be further distinguished into state-based and event-based process models (cf. [Aal96]).

- *State-based process models* represent real-world processes by sets of tasks *and* sets of states, i.e., states between tasks are explicitly modeled. Examples of conceptual state-based process models are: (a) Petri nets (cf. [Mur89, Pet77]) and (b) Statecharts (cf. [Har88, HLN$^+$90]). Several prototype WFMSs use Petri nets to model processes, e.g., *INCOME/STAR* (cf. [OSS94]) and *Melmac* (cf. [DGW94, DGSZ94]), or derivates of Petri nets, e.g., *OfficeTalk* (cf. [Ell82]) uses so-called Information Control Nets (ICNs) to model processes (cf. [Ell93]). The prototype WFMS *MENTOR* (cf. [WWW+97]) uses statecharts to model processes.

- *Event-based process models* represent real-world processes by sets of tasks suppressing states between tasks. Instead of using states, inter-task dependencies are modeled directly by relationships between tasks. The basic conceptual event-based process model is the Activity Network. That is, a process consists of a set of activities connected by precedence relationships (cf. [Wor96]). Frequently, special nodes are used to model alternative or parallel splitting (joining) of the control flow of processes. Many commercial WFMSs use activity networks to model processes, e.g., *FlowMark* (IBM) (cf. [LR94]).

Conversation-based approaches. Conversation-based approaches are based on the speech-act theory developed by Austin and his student Searle. Conversation-based approaches model a workflow as a set of connected workflow loops, where each workflow loop models a task of a workflow as a four-phase conversation between a customer and a performer. The four phases of a conversation are "proposal", "agreement", "performance", and "satisfaction" (cf. [GHS94, MMWFF92]). The commercial WFMS *Action Workflow* (Action Technologies, Inc.) (cf. [MMWFF92]) uses a conversation-based approach.

Object-oriented approaches. Object-oriented methodologies have been used successfully for the analysis, design, and implementation of information systems in general. However, object-oriented methods do not provide explicit support for process modeling. As Georgakopoulos *et al.* [GHS94] state, the problem is that designers typically would have to define workflow-model-specific object types from scratch. They propose to address this problem by supporting designers with predefined workflow-model-specific types, e.g., types "workflow process" and "workstep". The commercial WFMS *InConcert* (XSoft Advanced Information Technology, Inc.) (cf. [MS93]) combines object-orientation with the activity-based approach.

Modeling of Responsible Actors of Process Activities

The set of actors responsible to perform a particular process activity is determined by evaluating a so-called *assignment rule* associated with the process activity (cf. [BJ94, BJ95]). Bussler [Bus96] investigated three commercial WFMSs, i.e., *COSA* (Software-Ley), *FlowMark* (IBM), and *WorkParty* (Siemens Nixdorf), wrt. their capabilities of modeling assignment rules. Bussler states that all investigated systems offer a non-extendible, fixed language to specify assignment rules and only one of the systems, i.e., *COSA* (Software-Ley), allows to access information on business cases within assignment rules. In short, the results of Bussler can be best described by his statement: "The most interesting question is why WFMS allow to model arbitrary workflows [i.e., process logic] but not arbitrary organization structures or assignment rules of any form."

1.4.2 Organization Modeling

The modeling of organizations is concerned with the specification of actors, groups of actors, and physical as well as control structures among actors and groups of actors of an organization (cf. organization design, Subsection 1.2.3). Typically, the knowledge on organizational objects and organizational relationships is maintained in an organizational database. Thus, the modeling of organizational knowledge corresponds to the design of a specific database schema, called *organization schema*. This means, concepts known for the conceptual design of database schemas in general can be applied in the same way to the conceptual design of organization schemas (cf. data modeling, Subsection 1.4.4). In fact, most WFMSs, commercial as well as academic prototype systems, do offer a predefined, static organization schema, which is not extendible by workflow designers. This means, modeling of a particular organization, i.e., modeling of organizational objects and organizational relationships, is accomplished by instantiating the predefined organization schema. That is, depending on the database technology used to maintain organizational objects either tuples are inserted into relations (Relational DBMS) or instances of predefined object types are created (Object-Oriented DBMS).

There are several approaches that are more flexible. For example, *ORM* (Organization & Resources Model) (cf. [Rup90, RW94, Rup97]), a separate software product used by the commercial WFMS *WorkParty* (Siemens Nixdorf) to define and maintain the organizational database, allows to extend its predefined organization schema to the requirements of a particular organization in that subtypes of the predefined object types are introduced.

So far the most flexible approach has been proposed by Jablonski and Bussler for the academic prototype system *MOBILE* (cf. [JB96]) and by Dittrich for the academic prototype system *DOPAS* (cf. [Dit95]). These prototypes offer a so-called *dynamic organization-schema* (as opposed to *static organization-schemas* offered by most WFMSs, above). That is, *MOBILE* and *DOPAS* allow workflow designers to develop arbitrary organization schemas. Both systems offer two basic concepts, i.e., organizational objects and organizational relationships. In fact, these concepts are offered in the form of predefined meta types.

A relevant contribution to the modeling of organizations has been presented by Esswein (cf. [Ess93]). Esswein proposed to model positions (or job functions) within an organization by a set of objects called *roles* that are organized into so-called *role hierarchies*. All objects in the same role hierarchy represent the same position, which is identified by the root of the role hierarchy. This model allows to easily cope with the evolution of positions in organizations. That is, if new areas of responsibility (and authority) are to be added or old ones are to be removed, new roles are added or old roles removed, respectively, from a role hierarchy.

1.4.3 Authorization Modeling

The modeling of authorizations is concerned with the specification of the protection requirements of organizational processes and the specification and maintenance of authorizations that determine the behavior of an organization, i.e., who is authorized to perform what process activities (cf. authorization design, Subsection 1.2.3). Contemporary WFMSs, both commercial systems as well as academic prototypes, pay little or no attention to authorizations. In fact, they determine who *can* access a workflow operation (or workstep) based on the *responsibility* of actors (cf. [JB96, p. 183]). That is, contemporary WFMSs assume that actors who are responsible to perform certain worksteps are at the same time authorized to do so. A comparison of the commercial workflow systems *COSA* (Software-Ley), *FlowMark* (IBM), and *WorkParty* (Siemens Nixdorf) conducted by Bussler [Bus96] shows that these systems consider authorizations only for some administrative tasks such as access to worklists of other actors or control of process execution, and for modeling tasks such as the design of process schemas and organization schemas.

Much research on conceptual authorization models has been conducted in the area of database security, refer to Castano *et al.* [CFMS94] for a review of this research. Under the topic "authorization models for next-generation databases" conceptual authorization models for object-oriented databases have been presented (cf. [ADG+92, BW94, FLPG93, RBKW91, Thu89]). Fundamental for object-oriented systems is that the state of objects can only be manipulated by invoking methods on the objects. Consequently, authorization models for object-oriented databases consider methods as relevant access modes. Authorization models for workflow systems must consider workflow operations as relevant access modes and have to deal with allowed execution orders defined on these operations by a workflow process. Recently, authorization models for workflow systems have been presented by Atluri and Huang (cf. [AH96]), and by Thomas and Sandhu (cf. [TS93, TS94]).

Moffet and Sloman (cf. [MS88, MS91]) investigated the source and proper distribution of authority in organizations. They state that a formal authorization model must adequately represent the control structure within an organization such that the authority of each employee of an organization can be re-traced back to the owners of the organization. This research emphasizes that the quality of a formal authorization model depends on the quality of the organization model used to represent actors of a system.

Role-based access control (RBAC) (cf. [SCFY96]) emerged as a new kind of access control. Its focus is on simplifying the administration of authorizations in large information systems used in commercial environments. RBAC is based on the observation that in many organizations the authority (and responsibility) of employees is determined due to the roles these employees play within the organizations. This means, authorizations are assigned to roles and the authorizations of individual employees are determined by assigning employees to roles. This extra level of indirection simplifies administration of authorizations considerably. Assume a new employee enters an organization. The only administrative tasks required to determine the appropriate authorizations (and responsibilities) of the new employee are to assign the employee to the roles she or he shall play within the organization. No individual authorizations must be assigned to the new employee. This research again indicates how tight organization modeling and authorization modeling are related.

1.4.4 Data Modeling

The modeling of data is concerned with the specification of object structures, static integrity constraints, and (depending on the class of database management system intended to use) object behavior and dynamic integrity constraints (cf. database design, Subsection 1.2.3). Although database design is not considered a portion of workflow design (cf. workflow design, Subsection 1.2.3), there is a tight relation between workflow design and database design, since workflow operations access and manipulate objects maintained in database systems.

Much research has been conducted on the conceptual design of database schemas. Certainly the best-known model for conceptual design of database schemas is the Entity-Relationship (ER) model (cf. [Che76]). Under the term Extended Entity-Relationship (EER) model several extensions of the original ER model have been presented (e.g., [TYF86]). Semantic data models have been presented in [HM81, HK87]. Recently, the *Unified Modeling Language* (UML) (cf. [R$^+$97]) has been proposed for the design of conceptual schemas for object-oriented databases. UML unifies the object-oriented design methods of Booch (cf. [Boo94]), Jacobson (cf. [JCJO92]), and Rumbaugh *et al.* (cf. [RBPE91]).

Conceptual workflow models that consider data modeling as an integral part of workflow modeling use the ER model or an EER model to design the structure of relevant data objects. For example, *INCOME/STAR* (e.g., [OSS94]), a model and environment for the development of software processes, uses an EER model to design the structure of data objects processed by workflow operations.

Summing up, we note that contemporary workflow technology is still immature. In Section 1.1 we identified relevant issues for future research, i.e., the development of more comprehensive and more flexible workflow models that explicitly distinguish between authorizations and responsibilities. In the following, we state in which areas of workflow modeling (i.e., process modeling, organization modeling, and authorization modeling) we build on concepts that are state-of-the-art in workflow technology and in which areas we extend the state-of-the-art by introducing new concepts.

Process Modeling. There exist very powerful models that allow to specify nearly arbitrary process logics, e.g., Petri nets, Activity Networks (cf. process modeling, above). We rely on a Petri-net-based approach to process modeling. However, existing process models do not offer support for the evolution of processes. *To support the evolution of organizational processes, we present a novel schema architecture for workflow systems.* In short, the basic principle of the so-called two-schema architecture is to separate the knowledge on organizational processes into knowledge of external and internal origin. External knowledge on organizational processes is modeled in business-process schemas, internal knowledge in workflow-process schemas. Since organizations have no direct influence on external knowledge on organizational processes, e.g., natural facts or law, they may only change internal knowledge on organizational processes to adapt to changed requirements in their business environments. The main benefit of the two-schema architecture is that most changes are reflected only in the workflow-process schema. As a consequence, the business-process schema can be reused in most cases when organizational processes must be changed.

Organization Modeling. The organization modeling capabilities of contemporary WFMSs are rather limited (cf. [Bus96]). Most contemporary WFMSs offer a fixed organization schema that cannot be extended by workflow designers in order to capture semantics of individual organizations. *We extend the state-of-the-art in that we offer a semantic data-modeling approach that allows to define arbitrary organization schemas.* In fact, we offer a set of predefined conceptual-types of organizational entities that workflow designers can specialize when developing organization schemas.

Authorization Modeling. Contemporary WFMSs do not consider authorizations as an independent concept. In fact, they determine who can access a process activity based on the responsibilities of actors (cf. [JB96, p. 183]). *We extend the state-of-the-art in that we consider authorizations and responsibilities as related but independent concepts.* That is, we explicitly distinguish between permissions and obligations of actors to perform activities of organizational processes. This separation of authorizations and responsibilities is a prerequisite for the design of secure workflow systems. Furthermore, we extend the state-of-the-art in conceptual modeling of security aspects of workflow systems in that *we model protection requirements of activities of organizational processes as part of workflow schemas.* So-called authorization schemas capture the security requirements of workflow systems.

We offer a uniform, object-oriented approach to the conceptual design of workflow schemas. The conceptual workflow model we present builds on an object-oriented data model that has originally been introduced for the conceptual design of database schemas. The advantages of this approach are: First, the same modeling concepts can be used to model organizational processes, inter-related organizational objects, and authorizations. Second, the boundaries between database design and workflow-system design are blurred, since the underlying object-oriented data model can be used to develop database schemas.

Chapter 2

Overview

Contents

This chapter is organized as follows. Section 2.1 gives a survey of the workflow modeling concepts presented in this thesis by means of a small modeling example. Section 2.2 discusses briefly the organization of the thesis.

2.1 Survey of Workflow Modeling Concepts

Modeling of workflows is concerned with the specification of organizational processes, interrelated organizational objects, and protection requirements as well as authorizations (cf. workflow-oriented design of information systems, Subsection 1.2.3). In this section we survey the basic modeling concepts offered by the conceptual workflow model presented in this thesis. This section serves two purposes: First, it aims at building a basic understanding of the modeling concepts offered by the conceptual workflow model. Second, it functions as an index into Parts II and III of the thesis, where the modeling concepts offered by the conceptual workflow model are discussed in detail. That is, this section gives a quick survey of the basic modeling concepts of our conceptual workflow model and guides the reader to places, where more detailed information on these basic modeling concepts and on advanced modeling concepts can be found.

The conceptual workflow model we present in this thesis offers a uniform, object-oriented approach to the design of workflow schemas. The conceptual workflow model builds on the data model and the graphic notation of Object/Behavior Diagrams (cf. [KS91, KS96b]), which have originally been introduced for the conceptual design of object-oriented database schemas. Object/Behavior Diagrams will be substantially revised and extended in this thesis. The resulting data model together with its graphic notation serves as a common fundament on which we build all submodels of the overall workflow model. This means, the same object-oriented modeling concepts can be used to model all elements of the business area that are relevant for workflow systems, i.e., organizational processes, inter-related organizational objects, and protection requirements as well as authorizations. As a consequence, we present common modeling concepts first and continue with presenting concepts that are specific for the modeling of organizational processes, inter-related organizational objects, and protection requirements as well as authorizations, respectively. That is, Subsection 2.1.1 (*Data Modeling*) presents basic concepts that can be used to model all kinds of elements from the business area as inter-related structured objects together with their behavior. Basic concepts that are specific for the modeling of organizational processes are presented in Subsection 2.1.2 (*Process Modeling*), basic concepts that are specific for the modeling of inter-related organizational objects are presented in Subsection 2.1.3 (*Organization Modeling*), and basic concepts that are specific for the modeling of protection requirements and authorizations are presented in Subsection 2.1.4 (*Authorization Modeling*). Figure 2.1 illustrates which subsection covers the modeling of what elements of the business area. In fact, the organization of this section resembles the organization of Parts II and III. Part II presents basic data modeling and workflow modeling concepts, Part III presents advanced modeling concepts.

2.1.1 Data Modeling

This subsection surveys basic concepts of an object-oriented data model for the modeling of inter-related structured objects and their behavior. The object-oriented data model serves as a fundament for the conceptual workflow model presented in this thesis. Contrary to many other object-oriented data models (e.g., [Boo94, JCJO92, RBPE91]), we clearly distinguish between object types and object classes. An object type specifies the structure and behavior of its instances, an object class serves as a container, collecting a set of objects of a particular object type. Each object is instance of a particular object type and is member of one or several object classes.

An *object type* specifies the structure and behavior of its instances, called *objects*. To specify the structure of objects means to specify their properties, i.e., attributes and relationships. Attributes model descriptive features of objects, relationships model references to other objects. Each property is specified by a name, a domain, a multiplicity, and a cardinality. The domain, the multiplicity, and the cardinality of a property model static integrity constraints. The domain of a property determines the kind of values a property may hold. The domain of an attribute property is given by a data type, e.g., INTEGER or STRING, the domain of a relationship property by another object type. The multiplicity and the cardinality of a property constrain the number of values a property may hold. The multiplicity determines whether a property is single-valued or multi-valued, the cardinality

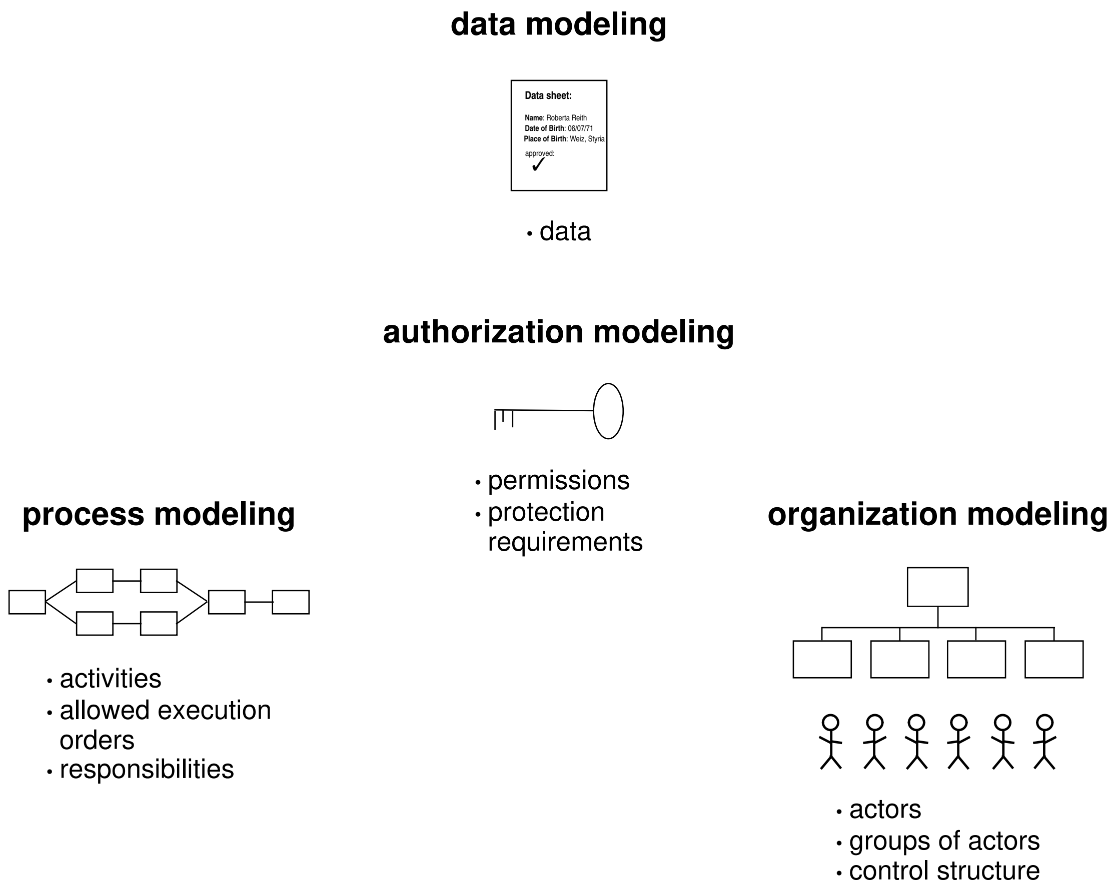


Figure 2.1: Modeling of elements of the business area

determines the exact number of values a property may hold. The structure of objects of a particular object type is depicted within an object-type diagram. An *object-type diagram* is a box showing the name of the represented object type in a shaded bar at its top. Properties are depicted by nested graphical symbols. Attributes by rounded rectangles, relationships by ordinary rectangles.

Example: Figure 2.2 shows a type diagram of object type RESERVATION. Each reservation has three single-valued properties. An attribute resNo with a value from domain INTEGER and two relationships guest and room to values from domains GUEST and ROOM, respectively. Relationship guest is mandatory, i.e., its cardinality is [1:1] (Note: A cardinality of [1:1] is the default cardinality for properties and is thus not shown in object-type diagrams.). Relationship room is optional, i.e., its cardinality is [0:1].

To specify the behavior of objects means to specify operations and allowed execution orders on these operations. An operation, called *activity*, is specified by a name, a set of input ports, a set of output ports, a precondition, and a postcondition. Input ports represent formal input parameters of an activity, output ports formal output parameters. Like a property of an object, a port of an activity is specified by a name, a domain, a

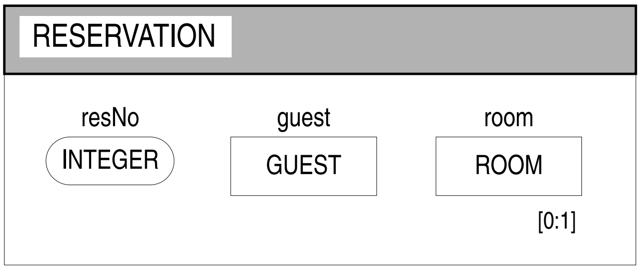


Figure 2.2: Object type RESERVATION

multiplicity, and a cardinality. The precondition and the postcondition of an activity model dynamic integrity constraints. The precondition of an activity is a boolean condition that must be satisfied immediately before the activity is executed, the postcondition a boolean condition that must be satisfied immediately after the activity has been executed. A *life-cycle* determines allowed execution orders of activities defined for instances of a particular object type. It models allowed execution orders of activities by a set of states, a set of activities, and a set of arcs connection activities to states and vice versa. The basic principle behind life-cycles stems from Petri nets (cf. [Mur89, Pet77]). A life-cycle is depicted within a life-cycle diagram. A *life-cycle diagram* shows an activity by a named rectangle and a state by a rectangle with trapezoids at its top and bottom. The top trapezoid shows the name of the object type at which the life-cycle has been defined, the bottom trapezoid the name of the state.

Example: Figure 2.3 shows the life-cycle defined for instances of object type RESERVATION. After a reservation has been issued, a guest checks in at the respective hotel. She or he will stay with the hotel for several days. Before a customer checks out, she or he has to pay for the stay. After check-out, a guest's reservation will be archived. (Note: For simplicity of the example, we do not consider cancellations of reservations.)

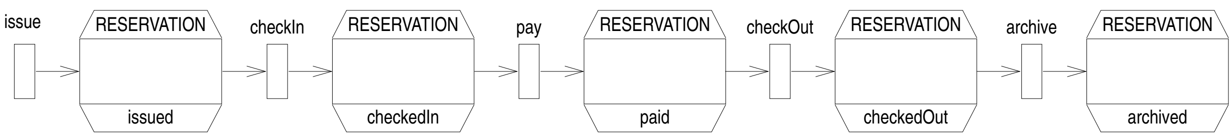


Figure 2.3: Behavior at RESERVATION

An *object class* collects a set of objects as its *members*. The members of an object class are instances of a particular object type, called the *member type* of an object class. The member type of an object class specifies the structure and behavior of the members of the object class. Object classes may be organized in *class hierarchies*, which are modeled by subclass-superclass relationships. The set of members of a subclass must be a subset of the set of members of its superclass. Moreover, an object class may specify other object classes as domains for relationships defined for its members. This domain is called class domain to distinguish it from the domain specified for a relationship at an object type, which is called type domain. Class domains represent referential integrity constraints. The specifications of class domains are depicted within class-domain diagrams. Note: Object classes are objects, too. That is, each object class is instance of a particular object type, called *class type*, that defines its structure and behavior.

Example: Figure 2.4(a) shows the specification of object class Reservations and its subclass VIP-Reservations. Both object classes are instances of class type RESERVATIONS, which defines object type RESERVATION (cf. Figure 2.2) as their member type. (Note: The member type of an object class is not explicitly shown in its graphical specification. One must consult the specification of the respective class type or a class-domain diagram that shows the object class (below)). Furthermore, both object classes are multi-valued, which is indicated by the double-headed arrows in the side bars of the rectangles illustrating their specifications.

Figure 2.4(b) shows a class-domain diagram with object classes Reservations, VIP-Reservations, Guests, and VIP-Guests. Object class Reservations collects instances of object type RESERVATION (cf. Figure 2.2). It restricts the domain of relationship guest to object class Guests. Object class VIP-Reservations is a direct subclass of object class Reservations. It collects reservations that have been issued to VIP guests. That is, object class VIP-Reservations restricts the inherited class domain of relationship guest to object class VIP-Guests. Object class VIP-Guests is a direct subclass of object class Guests.

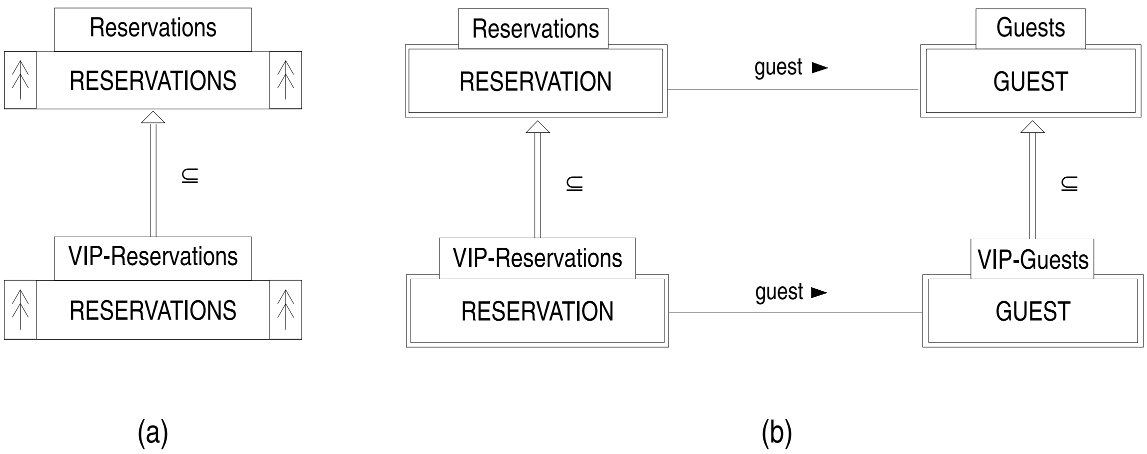


Figure 2.4: Classes of reservations

The conceptual design of object-oriented database schemas is discussed in detail in Chapters 3 and 7 (*Data Modeling*). Chapter 3 presents a simple but complete object-oriented data model that considers the basic set of modeling concepts surveyed above. Chapter 7 extends the basic data model with advanced features. In short, the advanced data model adds type specialization, semantic relationships, local classes, and advanced concepts for the synchronization of the behavior of different objects. Type specialization allows to organize object types into type hierarchies, where subtypes inherit the structure and behavior defined at their supertype. Semantic relationships allow to express the different semantics of relationships between objects in the real-world. For example, a semantic relationship allows to express that an object is a component, a constituent, or a role of another object. Furthermore, the advanced data model proposes to maintain objects as members of classes that are local to (i.e., are properties of) other objects. Since then classes are no longer global schema objects but are properties of objects, they are called local classes. In fact, each object o_1 is member of a local class that is a property of that object o_2 of which o_1 is a component. That is, local classes build on two fundamental concepts, object classes and object composition, requiring that all objects are organized into a so-called composite-object hierarchy. A proper synchronization of the behavior of interacting objects is a prerequisite for successful interactions. Behavior synchronization constraints are either specified with activities or states. Synchronization constraints specified with activities require either that objects passed to the activities reside in particular states before and/or after their executions or that activities are invoked only by certain other activities. Synchronization constraints specified with states determine dependencies between states of different objects. For example, a synchronization constraint might require that an object resides in a particular state s_1 while a related object resides in another state s_2.

2.1.2 Process Modeling

This subsection surveys basic concepts of the conceptual design of process schemas of workflow systems. The respective conceptual process model builds on the object-oriented data model presented above. It offers a novel, two-schema architecture for the design of process schemas. The two schemas are the business-process schema and the workflow-process schema.

We propose a *two-schema architecture* for WFMSs, which supports the evolution of organizational processes. The novel schema architecture for WFMSs separates the knowledge on organizational processes into external knowledge and internal knowledge. External knowledge determines a frame within which an organization may operate, e.g., business rules based on natural facts or law (cf. [Her95, HK94]). *Internal knowledge* restricts this frame and determines the way an organization actually runs its business, i.e., business rules based on organizational commitments. The two-schema architecture builds on the fact that organizations may change their internal business rules to react to changes in their business environments but have no—at least no direct—influence on business rules of external origin. The two schemas of organizational processes are the business-process schema and the workflow-process schema. A *business-process schema* consists of a set of business-process types and represents external knowledge on organizational processes, a *workflow-process schema* consists of a set of workflow-process types and represents internal knowledge on organizational processes. In the sequel we survey basic concepts for the modeling of organizational processes in the context of the proposed two-schema architecture.

A *business-process type* is an object type (cf. data modeling, above) that specifies the structure and behavior of its instances, called business cases. Business cases represent individual occurrences of organizational processes. The structure defined at business-process types determines which data is stored for business cases (attributes) and to which other objects the business cases hold references (relationships). The behavior defined at business-process types, called *business processes*, determines which activities can be invoked in what order on business cases. We model business processes using the same concepts as we use to model the behavior of ordinary objects, i.e., activities and life-cycles (cf. data modeling, above). The difference between business processes, i.e., the behavior of business cases, and the behavior of ordinary objects is a semantic one. The behavior of ordinary objects, e.g., information objects, does not necessarily produce a value for some customer—at least not directly—and is thus not recognized by a designer as a business process.

When designing business processes only external knowledge on organizational processes is considered. A business process is completed by a workflow process, which models internal knowledge on an organizational process. Together a business process and a workflow process do model an organizational process as its currently performed in a particular organization. Since internal knowledge on organizational processes restricts the frame expressed by external knowledge, business processes must offer a variety of alternatives to achieve the objectives of an organizational process and pose as weak restrictions as possible on the invocation of process activities, e.g., weak preconditions of process activities, parallel execution of process activities where possible. A workflow process can restrict a business process in that it considers only a subset of the alternatives offered by a business process, defines stronger preconditions for process activities, groups several business-process activities into a single workflow-process activity, requires sequential execution where the business process allows parallel execution of activities, etc.

Example: The processing of reservations is an important organizational process in hotels. The life-cycle shown in Figure 2.3 shows how a particular hotel does process its reservations. As a consequence, the life-cycle must be re-designed to serve as a business process in the context of the proposed two-schema architecture for WFMSs. Figure 2.5 shows the respective business process defined at business-process type RESERVATION.

The re-designed life-cycle allows for more flexibility in handling the payments of guests. Handling the payment of a guest is now a two-step process that starts with billing the guest and finishes with the payment of the guest. Furthermore, it is now possible that a guest is billed for her or his reservation and pays at any time after the reservation has been issued. Payment is not restricted to check-out time. The business process shown in Figure 2.5 may serve as a fundament for several workflow processes that specify how particular hotels do actually process reservations. Two possible workflow processes that build on the reservations business-process are shown in Figure 2.6.

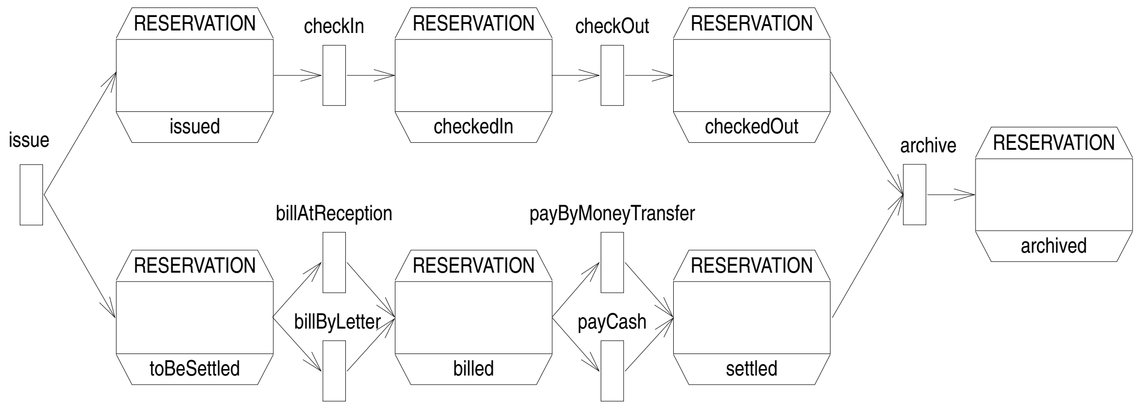


Figure 2.5: Business process at RESERVATION

A *workflow-process type* is an object type (cf. data modeling, above) that specifies the structure and behavior of its instances, called workflow objects. In addition, a workflow-process type specifies behavior, called *workflow process*, that is not intended for its instances but for business cases that are assigned to it for processing. In fact, business cases may be assigned to a workflow object for processing if the business process defined at the type of the business cases and the workflow process defined at the type of the workflow object together specify that organizational process of which the business cases are occurrences. Whereas the structure and behavior of workflow objects is specified like the structure and behavior of ordinary objects (cf. data modeling, above), advanced concepts for the modeling of workflow processes are introduced. A workflow process is specified by a set of containers of business cases, called *boxes*, a set of workflow-process activities, called *worksteps*, and a set of arcs connecting boxes to worksteps and vice versa.

Example: Figure 2.6 shows two workflow processes. Each one specifies how a particular hotel (or kind of hotels) does actually handle reservations. Both workflow processes build on the business process shown in Figure 2.5.

Figure 2.6(a) shows a simple workflow process for the handling of reservations. After a reservation has been issued, a guest may check in at the respective hotel. At the time a guest checks out, she or he is handed out the bill at the reception and has to pay the bill in cash. That is, billing, payment, and checking-out a guest is performed within a single workstep (cf. workstep CheckOut_BillAtReception&PayCash). Finished reservations will be archived. (Note: As a notational convention the names of worksteps start with a capital letter. Furthermore, we name worksteps after the business-process activities they invoke to emphasize the inter-relationships between business processes and workflow processes.)

Figure 2.6(b) shows a workflow process for the handling of reservations that allows to make all-inclusive offers that are to be paid before check-in. That is, when issuing a reservation for a guest, the guest is immediately sent the bill for his future stay by letter. Before a guest may check-in at a hotel, she or he has to pay the bill by money transfer. At the end of the stay, a guest checks out. Finally, finished reservations will be archived.

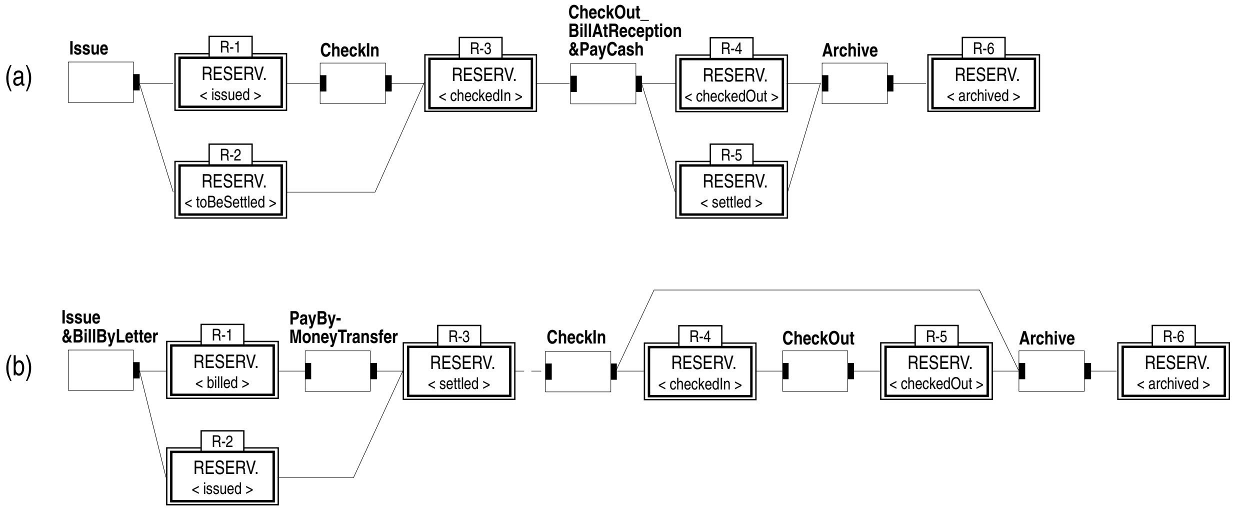


Figure 2.6: Workflow processes defined for reservations

An *assignment rule* determines a set of responsible actors for a workstep of a workflow process. For each workstep of a workflow process there is an assignment rule. We model assignment rules as queries that retrieve sets of responsible actors from a database that maintains knowledge on actors and groups of actors (cf. organization modeling, below).

The fundamental concepts of the two-schema architecture for WFMSs are presented in Section 4.2. The conceptual design of business-process schemas and workflow-process schemas is discussed in detail in Chapters 4 and 8 (*Process Modeling*). A business-process schema is modeled relying solely on concepts offered by the object-oriented data model presented in Chapter 3 and Chapter 7, respectively. Section 4.3 and Section 8.2, respectively, review the use of these concepts for the modeling of business-process schemas. Section 4.4 presents a simple process model that considers the basic set of concepts for the modeling of workflow processes as surveyed above. Section 8.3 extends the basic process model with advanced features for the modeling of workflow processes. In short, the advanced process model adds nested workflow processes and specialization of workflow processes. Nested workflow processes allow to decompose the specifications of organizational processes into a set of workflow processes. A nested workflow process is embedded into another workflow process, called its superordinate workflow process. In fact, a nested workflow process is a workflow process that specifies part of the organizational process that its superordinate workflow process specifies. Type specialization allows to organize object types into type hierarchies, where subtypes inherit the structure and behavior defined at their supertype (cf. data modeling, above). Since workflow-process types may also be organized into type hierarchies, inherited workflow processes may be specialized, too. We discuss criteria for a consistent specialization of inherited workflow processes. Specialization of workflow processes allows for an incremental design of workflow processes.

2.1.3 Organization Modeling

This subsection surveys basic concepts of the conceptual design of organization schemas of workflow systems. The respective conceptual organization model builds on the object-oriented data model presented above. It offers specific concepts for the object-oriented modeling of actors, groups of actors, and control structures among actors and among groups of actors (cf. Figure 2.7).

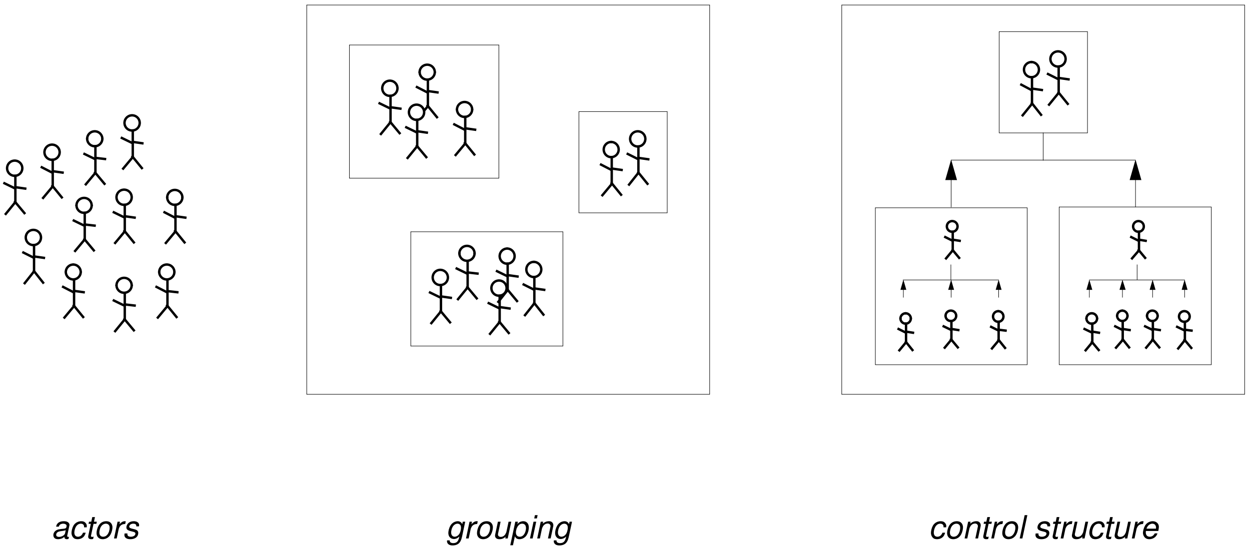


Figure 2.7: Organizational entities

Each actor of organization is modeled by one agent object and by several position objects. An *agent* models a particular real-world actor. Agent objects act as mediators of real-world actors. That is, real-world actors perform worksteps within a workflow system through agent objects. A *position* models a kind of "abstract actor" of an organization. Each position object belongs to a particular organizational unit and determines the authority and the responsibility of that individual agent that is assigned to that position at a particular time. An agent that is assigned to a position is said to occupy that position. An agent may occupy several positions within an organization. Historically, the concept of positions has been introduced to make organizations independent from individual actors that join and leave the organizations (e.g., [LARB$^+$91]). For example, assigning the agent object representing an actor that did just join an organization to a particular position object determines the authority and responsibility of the new actor as well as the organizational unit within which that actor has to fulfill her or his responsibilities. Figure 2.8 illustrates how real-world actors are represented within workflow systems.

The *physical structure* of an organization determines how positions are grouped together into organizational units and how organizational units are grouped together into an overall organization. That is, an *organizational unit* models a group of actors, where actors are represented by position objects, or a group of other organizational units.

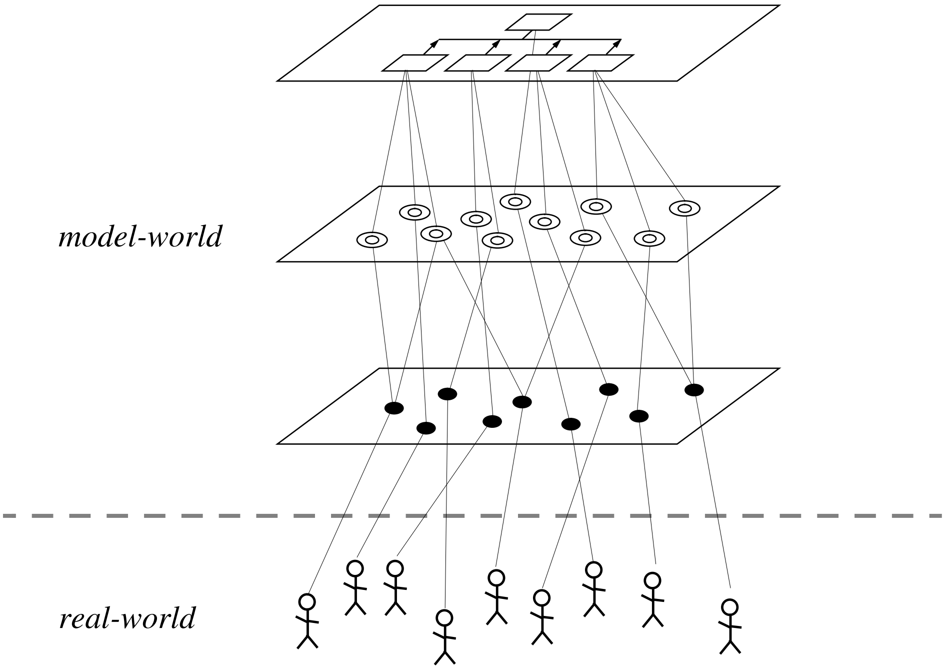


Figure 2.8: Organization modeling concepts

Example: Figure 2.9(a) shows a type diagram of organizational-unit type RECEPTION. A reception groups together several positions. A manager position (cf. relationship head), several receptionist positions (cf. relationship receptionists), and one workflow-administrator position (cf. relationship wfAdmin). The responsibility of a workflow administrator is to perform some administrative tasks of the reservation workflow-process of a reception (not shown). Note: Workflow-process type RESERVATION-WF, which has been given as domain of relationship reservation-wf, defines the workflow process depicted in Figure 2.6(a) for the processing of reservations.

Figure 2.9(b) shows a physical-structure diagram of organizational units of type RECEPTION. It considers only positions and organizational units.

The *control structure* of an organization models an unbroken connection between the owner or the shareholders of an organization and each position within that organization. The control structure of an organization serves as a communication channel along which authority and responsibility may be delegated to and removed from individual positions within an organization. The control structure of an organization can take on different forms, e.g., *hierarchy* is such a control structure. In an organizational unit with a hierarchical control structure, all managerial tasks are within the responsibility of a single position, called *supervisor*. The supervisor position of an organizational unit is superordinate to all other positions within the same organizational unit and to positions in subordinate organizational units.

Example: Figure 2.10 shows a control-structure diagram of organizational units of type

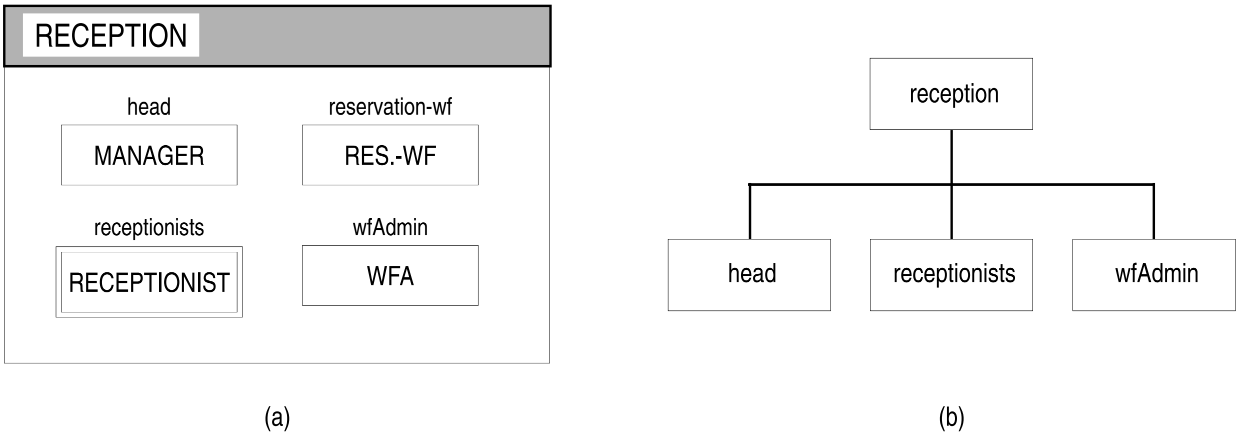


Figure 2.9: Structure of receptions

RECEPTION (cf. Figure 2.9). The head of a reception, i.e., the actor that occupies the manager position of a reception, is the supervisor of all other members of a reception. That is, actors that occupy the workflow-administrator position or a receptionist position are subordinate to the head of that reception within which they occupy a position.

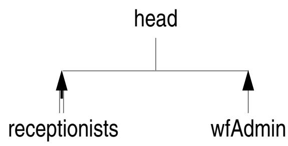


Figure 2.10: Hierarchical control structure of receptions

The conceptual design of organization schemas is discussed in detail in Chapters 5 and 9 (*Organization Modeling*). Chapter 5 presents a simple organization model that considers the basic set of concepts for the modeling of organization schemas as surveyed above. Chapter 9 extends the basic organization model with advanced features for the modeling of organization schemas. In short, the advanced organization model adds the network control structure, substitution plans, and role modeling of positions and agents. The *network control structure* allows to model self-managed organizational units. Self-management means that all actors of an organizational unit may participate in decisions. Typically, a self-managed organizational unit has a *coordinator* whose job is either to guide the members of the organizational unit in making decisions, to make decisions on behalf of the members of the organizational unit, or a mix of both. We model a coordinator of a self-managed organizational unit by a position that can be occupied by all or at least by most members of that organizational unit. A *substitution plan* determines which agents are to be selected to perform particular worksteps on business cases if the regular, responsible agents are not available. Furthermore, the advanced organization model makes extensive use of the role concept introduced in Chapter 7. For example, each position can be modeled by a set of objects, called *organizational roles*, that are organized in a role-hierarchy. Each organizational role determines part of the authority and responsibility of the position it represents. Since roles may be acquired and abandoned dynamically, the authority and responsibility of a position can be augmented or diminished dynamically by acquiring or abandoning, respectively, organizational roles.

2.1.4 Authorization Modeling

This subsection surveys basic concepts for the conceptual design of authorization schemas of workflow systems. The respective conceptual authorization model builds on the object-oriented data model presented above. It offers specific concepts for the modeling of the protection requirements of organizational processes as well as concepts for the object-oriented modeling of authorizations.

An *authorization schema* of a workflow system is a formal description of the protection requirements of the organizational processes realized by the workflow system. An authorization schema consists of a set of authorization interfaces, a set of authorization constraints, and a set of authorization-contract types. The purpose of an authorization schema is to act as a filter for illegal authorizations and thus to ensure the integrity of information processed by a workflow system.

An *authorization interface* contains a set of activities of an organizational process that have similar protection requirements. The protection requirements of a set of activities of an organizational process that belong to an authorization interface f are determined in that so-called actor domains are defined at f. An *actor domain* is specified by an authorization constraint. An *authorization constraint* is a sentence of a deontic constraint language that distinguishes four deontic modalities: permission, obligation, prohibition, and discretion. As a consequence, we can give specific connotations to actor domains. Since we assume a closed-world policy for workflow systems, we are mainly interested in two deontic modalities, i.e., permissions and obligations. For example, associating deontic modality P (permission) with an actor domain D defined at an authorization interface f says that a security administrator *may* authorize actors that are elements of domain D to invoke operations that belong to f. Associating deontic modality O (obligation) with an actor domain D defined at an authorization interface f says that a security administrator *must* authorize actors that are elements of domain D to invoke operations that belong to f. Authorization constraints can be defined at the object type and at the object class level. Note: In the context of the two-schema architecture for workflow systems (cf. process modeling, above), actors may not invoke business-process activities directly because in doing so they would circumvent internal business rules. Consequently, we define authorization interfaces only on workflow-process activities (or worksteps).

Example: Figure 2.11 shows authorization interface CheckIn defined on worksteps of the workflow process shown in Figure 2.6(b). In fact, authorization interface CheckIn contains a single workstep, the workstep CheckIn. The actor domains defined at authorization interface CheckIn say security administrators *may* issue authorizations to invoke worksteps that belong to authorization interface CheckIn to guests, but they *must* issue respective authorizations to receptionists of a hotel.

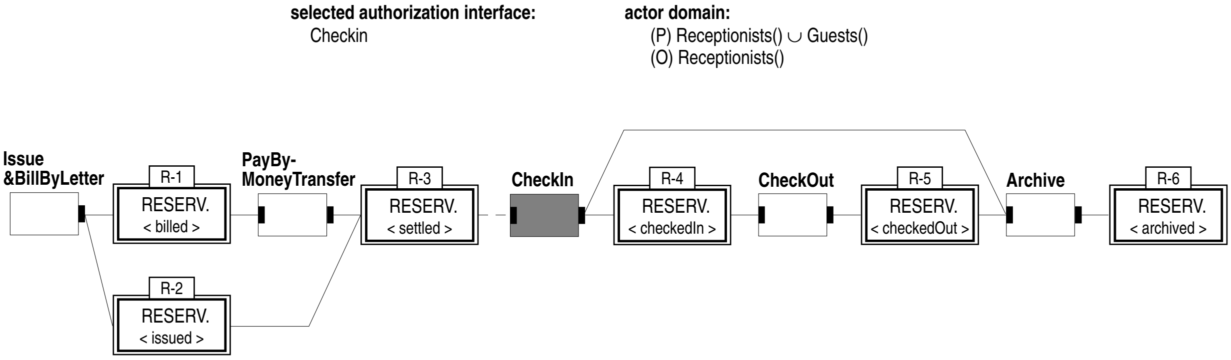


Figure 2.11: Authorization interface CheckIn

An *authorization contract* models an authorization. Each authorization contract is a formal agreement between two or more parties, where one or several parties, called suppliers, commit themselves to offer one or several services to the remaining parties, called clients. The suppliers of authorization contracts are organizational processes, the services are process activities, and the clients are actors. Authorization contracts are first-class citizens. That is, each authorization contract is an object with structure and behavior (i.e., an authorization contract is instance of a particular object type), and is member of one or several object classes. Types of authorization contracts are called authorization-contract types, classes of authorization contracts are called authorization-contract classes. This means, the authorization model allows for a high-level semantic description of the security semantics of a workflow system abstracting from a particular realization of authorizations in a concrete WFMSs, e.g., the realization of authorizations as tuples of an access matrix. Indeed, all semantic modeling concepts offered by the object-oriented data model presented before (cf. data modeling, above) are available to a security designer, too. As a consequence, a security designer can model authorizations at the conceptual level in the way she or he understands it, e.g., as key cards.

Example: Figure 2.12 shows a type diagram of authorization-contract type GUEST-CARD. Our hotel issues special key cards, called guest cards, to its guests. A guest card authorizes its holder, a guest, to perform *several* worksteps of the reservation workflow-process on reservations that have been issued for the guest (cf. relationships client, services, and supplier). Which worksteps of the reservation workflow-process might be assigned as values to relationship services of a guest card depends on the current authorization policy of an organization that is expressed by means of authorization constraints and authorization interfaces (above). Furthermore, a guest card has three attributes cardNo (card number), expDate (expiration date), and isValid. The boolean attribute isValid is a *derived* property (indicated by a dashed-line border), which means that its value is not stored but computed from other property values.

Indeed, the value of attribute isValid is computed each time the authorization control sub-system of a workflow system has to validate an access of a guest. It evaluates to true if the guest card has not expired yet and the guest aimed at invoking an allowed workstep on a reservation that has been issued for her or him. It evaluates to false otherwise. This means, guest cards model conditional authorizations.

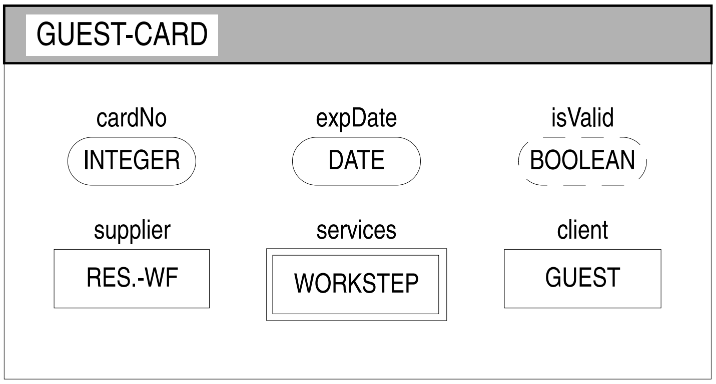


Figure 2.12: Authorization-contract type GUEST-CARD

The design of authorization schemas is a novel approach to the conceptual design of protection requirements and authorizations within information systems. In traditional information systems the responsibility to determine whether authorizations are legal wrt. the protection requirements of an organization is with the security administrators of the information system (cf. [CFMS94, p. 30]).

The conceptual design of authorization schemas is discussed in detail in Chapters 6 and 10 (*Authorization Modeling*). Chapter 6 presents a simple authorization model that considers the basic set of concepts for the modeling of the protection requirements of organizational processes and the modeling of authorizations as surveyed above. Furthermore, Section 6.4 discusses basic principles of the administration of authorizations. It emphasizes the influence of knowledge on the structure of an organization on the administration of authorizations. Basically, a manager may delegate authority to subordinate actors and may delegate authority on resources that belong to the same organizational unit as the manager or to subordinate organizational units. A fundamental principle is, however, that managers do not administer authorizations on their own but delegate this responsibility to a designated security administrator (separation-of-duty principle). Chapter 10 extends the basic authorization model with advanced features for the modeling of authorization schemas. In short, the advanced authorization model adds local-authorization constraints and role constraints. In addition, the advanced authorization model is extended to cope with features of the advanced process model in particular and of the advanced object-oriented data model in general. *Local-authorization constraints* specify local classes of positions (or agents) as actor domains of activities of organizational processes. A local-authorization constraint says (depending on the deontic modality associated with the specified actor domain) that an authorization to invoke a particular activity on an occurrence of a particular organizational process may be, must be, may not be, or must not be issued to a particular position (or agent) if both the position (or agent) and the occurrence of the organizational process belong to the same organizational unit or organization. *Role constraints* either determine legal configurations of roles that represent a particular actor within a workflow system or restrict the activation of roles representing a actor during sessions. Furthermore, Section 10.3 discusses the administration of authorizations in the context of organizational units with a network control structure.

2.2 Organization of the Thesis

The thesis is organized into four parts. Part I (*Introduction*) contains introductory material. Part II (*Basic Workflow Modeling*) and Part III (*Advanced Workflow Modeling*) present an object-oriented data model for the conceptual design of workflow schemas. The presentation of the material on the conceptual workflow model has been separated into two parts to keep the presentation clear, concise, and understandable. As the titles of Part II and Part III indicate, Part II presents basic concepts for the conceptual design of workflow schemas, Part III advanced concepts. Part IV (*Conclusion*) summarizes the main achievements of the research presented in this thesis and highlights some issues for future research. The remainder of this thesis is organized as follows.

Part II (*Basic Workflow Modeling*) presents basic concepts for the conceptual modeling of workflows. The uniform approach to the modeling of workflows is reflected also in the organization of the chapters of Part II and Part III. Each of these chapters starts with a section that gives a management definition of the fundamental modeling concepts presented in the respective chapter. The sections on schema design discuss in detail the modeling of inter-related structured objects and object behavior (*data modeling*), the modeling of processes (*process modeling*), the modeling of inter-related organizational objects (*organization modeling*), and the modeling of protection requirements as well as authorizations (*authorization modeling*), respectively. The section on schema design may be preceded or followed by a section discussing specific aspects of data modeling, process modeling, organization modeling, or authorization modeling, respectively. E.g., architectural decisions are presented in such sections. Each chapter in Part II and Part III is concluded by a short summary.

- Chapter 3 (*Data Modeling*) presents a basic data model for the conceptual design of object-oriented database schemas. The presented data model serves as a common fundament of all submodels of the conceptual workflow model presented in this thesis.

- Chapter 4 (*Process Modeling*) presents a two-schema architecture for workflow systems and discusses the modeling of organizational processes in the context of this novel schema architecture for workflow systems. That is, the modeling of business-process schemas as well as the modeling of workflow-process schemas is discussed.

- Chapter 5 (*Organization Modeling*) discusses the modeling of actors, groups of actors, and physical as well as control structures among actors and among group of actors.

Furthermore, the importance of a properly defined control structure among the actors of an organization for a formal authorization model is discussed.

- Chapter 6 (*Authorization Modeling*) discusses the modeling of protection requirements of organizational processes as well as the modeling of authorizations. It proposes to use the concept of schema design known from data modeling for the modeling of protection requirements and authorizations, too. Furthermore, it discusses basic concepts for the administration of authorizations in workflow systems.

Part III (*Advanced Workflow Modeling*) presents advanced concepts for the conceptual modeling of workflows. Chapter 7 presents an object-oriented data model that offers advanced features for the conceptual design of database schemas in particular and workflow schemas in general. As a consequence, Chapters 8 to 10 review these concepts in the respective context and may add additional features specific to process modeling, organization modeling, or authorization modeling, respectively. Below, we mention for Chapters 8 to 10 only advanced modeling concepts that are specific for the respective modeling context.

- Chapter 7 (*Data Modeling*) extends the basic object-oriented data model presented in Chapter 3 with advanced features such as type specialization, semantic relationships, local object classes, and additional concepts for behavior synchronization.

- Chapter 8 (*Process Modeling*) extends the basic workflow-process model presented in Chapter 4 with advanced features such as nested workflow processes and specialization of workflow processes.

- Chapter 9 (*Organization Modeling*) extends the basic organization model presented in Chapter 5 with advanced features such as a network control structures, substitution plans, and roles for the modeling of positions and agents.

- Chapter 10 (*Authorization Modeling*) extends the basic authorization model presented in Chapter 6 with advanced features such as local-authorization constraints and role constraints. In addition, it extends the basic authorization model to cope with the extensions of advanced workflow-process model.

Part IV (*Conclusion*) concludes the thesis. It consists of a single chapter.

- Chapter 11 (*Summary and Future Work*) summarizes the main achievements of the research presented in this thesis and discusses main issues for future work.

Part II

Basic Workflow Modeling

Chapter 3

Data Modeling

Contents

A *data model* is a collection of concepts that can be used to describe given realities. The process of describing a given reality is called *data modeling*. A description of a given reality that has been developed using a given data model is called *schema*. A *conceptual data model* is a data model that is used during the early phases of the design of information systems. A conceptual data model offers concepts that are found in reality and allows to represent given realities at a high level of abstraction. The concepts of a conceptual data model typically build on abstraction mechanisms that allow a designer to understand and classify given realities (cf. [BCN92]), and have a graphic notation that allows to model given realities by sets of diagrams that are easy to understand by professionals and by end users.

We present a conceptual data model together with a graphic notation for the design of object-oriented schemas. The modeling concepts it offers build on the following abstraction mechanisms: classification, association, aggregation, decomposition, and specialization. The presentation of the conceptual data model is separated into two parts. This chapter presents modeling concepts that build on the abstraction mechanisms classification and association. The presentation of the conceptual data model is completed by the material presented in Chapter 7. Note: The conceptual data model we present in this chapter builds on Object/Behavior Diagrams (cf. [KS91, KS96b]), which have originally been introduced for the conceptual design of object-oriented database schemas.

This chapter is organized as follows. Section 3.1 gives a management definition of the fundamental modeling concepts presented in this chapter. Section 3.2 presents basic modeling concepts of our conceptual data model. This conceptual data model together with its graphic notation serves as a fundament on which the process model (cf. Chapter 4), the organization model (cf. Chapter 5), and the authorization model (cf. Chapter 6) of the our conceptual workflow model have been built. Section 3.3 summarizes this chapter.

3.1 Fundamental Modeling Concepts

The core concept presented in this chapter is the *business object*. A business object[1] (or object) is a model of some real-world entity, be it a conceptual or a physical entity. Business objects have structure and behavior. The conceptual structure of business objects is determined by a set of attributes and a set of relationships. The behavior of business objects is determined by a set of activities and by allowed execution orders on these activities. We present an object-oriented data model together with a graphic notation that is an extension of Object/Behavior Diagrams [KS91, KS96b], which have originally been introduced for the conceptual design of object-oriented database schemas. The data model serves as a common fundament on which we build all submodels of the overall workflow model that is presented in this thesis. This means, process objects (cf. Chapter 4), organizational objects (cf. Chapter 5), and authorization objects (cf. Chapter 6) are just specific categories of business objects. As a consequence, the set of modeling concepts presented in this section can be used to model all kind of elements of the business area (cf. Subection 1.2.2) ensuring a uniform modeling of the business area. Modeling concepts that are specific for the modeling of particular kinds of elements of the business area are presented in subsequent chapters.

Business object: A *business object* (or object) is a model of an entity from the business area. A business object may represent, for example, a business process, a workflow process, a person, or an authorization. Each business object is instance of a business-object type and is member of one or several business-object classes.

Attribute: An *attribute* models a descriptive feature of a business object. Together with relationships, attributes describe the conceptual structure of business objects.

Relationship: A *relationship* models a reference between business objects. Together with attributes, relationships describe the conceptual structure of business objects.

Behavior: The *behavior* of business objects is determined by a set of activities. Activities may change the state of business objects, i.e., values of attributes and relationships, and interact with other business objects. Furthermore, allowed execution orders may be specified on activities.

[1]Note: We use the term *business object* to give emphasize to the specific "universe of discourse", also called *business area* by information-systems engineers.

Business-object class: A *business-object class* (or object class) collects a set of business objects, called its *members*. A business-object class is itself instance of a specific business-object type, called *class type*, and may itself be member of one or several other business-object classes. Each class type specifies a business-object type, called *member type*, as member domain for its instances. The member type defined at a class type specifies individual structure and behavior of the members of its instances. Collective structure and behavior of the members of a business-object class, i.e., structure and behavior that concern the members of a business-object class considered as a unit, are specified at the type of the business-object class. This means, besides individual structure and behavior of its instances, a class type specifies collective structure and behavior for the members of its instances.

Business-object type: A *business-object type* (or object type) defines the structure and behavior of its instances, called business objects. A business-object type may serve as member type of one or several class types.

Summing up, it is fundamental to state that all modeling concepts presented in this chapter for the modeling of business objects apply in the same way to all kinds of business objects, i.e., business-process objects, workflow-process objects, organizational objects, and authorization objects. Modeling concepts that are specific to particular kinds of business objects will be introduced in subsequent chapters.

3.2 Design of Object-Oriented Schemas

This section presents a data model for the conceptual design of object-oriented database schemas together with its graphic notation. The presented data model serves as a fundament on which the process model, the organization model, and the authorization model of the overall workflow model presented in this thesis are built. This means, all concepts presented for the conceptual modeling of business objects apply to the conceptual modeling of process objects, organizational objects, and authorization objects in the same way unless stated otherwise in subsequent chapters.

The model of business objects as presented in this section relies on the model of objects introduced by Kappel and Schrefl [KS91, KS96b] for Object/Behavior Diagrams. Their conceptual model has been extended in various aspects: First, exact *cardinalities* can be specified with relationships, with ports of activities, and with object classes restricting the allowed number of members. Second, object classes are considered as first-class citizens.

A *business-object schema* consists of a set of business-object types and a set of business-object classes. Each business-object type defines the structure and behavior of its instances, called *business objects*. A business-object class collects a set of business objects of a specific type as its members. As a notational convention, we write names of object types in capital letters. The names of object classes start with a capital letter.

An *instance* of a business-object schema consists of a set of business objects. Each business object is instance of one business-object type and may be member of one or several business-object classes. Business objects are related to and communicate with other business objects.

The remainder of this section is organized as follows. Subsection 3.2.1 discusses the modeling of the structure of business objects, Subsection 3.2.2 the modeling of the behavior of business objects, and Subsection 3.2.3 the modeling of classes of business objects.

3.2.1 Modeling the Structure of Business Objects

The structure of business objects (or *objects* for short) is determined by a set of *properties*. Each property is specified by a unique *name*, a *domain*, and a *multiplicity*. The multiplicity of a property determines whether it is *single-valued* or *multi-valued*. The exact number of values an object may hold for a certain property is specified by the property's *cardinality*[2]. Note: Since in most cases the meaning is understood from the context, we use the term 'property' to denote properties as well as their definitions. Only in situations that are ambiguous we use the term 'property definition' to denote the definition of a property. (Whereas a property belongs to an object and has a name and a value (instance level), a property definition belongs to an object type and specifies the name, the domain, the multiplicity, and the cardinality of a property of instances of the type to which it belongs (schema level).)

Properties are abstract concepts. They describe the *conceptual structure* of objects, not their physical (or internal) structure. This means, the decision whether a property shall be stored or shall be computed is a physical one and is not relevant during conceptual design. Consequently, properties must not be confused with the concept of instance variables as found in diverse object-oriented models (e.g., [GR83]). Rather, each property is to be interpreted as a public retrieval operation that returns either one instance of the property's domain (single-valued property) or a set of instances of its domain (multi-valued property). These public retrieval operations are named after the properties. In addition, hidden update operations are associated with properties which may be invoked by public update operations specified in the life-cycle of an object type (cf. Section 3.2.2). In the case of a single-valued property, the hidden update operation takes one instance of the property's domain as an argument. Its name is formed by adding a prefix set to the name of the property starting with a capital letter, e.g., the hidden update operation setReservations would be associated with a single-valued property reservation. In the case of a multi-valued property, two hidden update operations are associated with the property. Both take a set of instances of the property's domain as an argument. One allows to *add* a set of instances of the property's domain to the current set of values, the second allows to *remove* a set of instances from the current set of values. The names of these update operations are formed by adding the prefixes add and remove, respectively, to the name of the property starting with a capital letter, e.g., the hidden update operations addReservations and removeReservations would be associated with a multi-valued property reservations.

[2]Object/Behavior Diagrams in their original form (cf. [KS91, KS96b]) do not allow to specify the cardinality of object properties.

Properties are distinguished into *attributes* and *relationships*. Attributes model descriptive features of objects. Their domain is some predefined *data type*, e.g., DATE, INTEGER, REAL, STRING. Their values are instances of a data type, called *data values*. Data values are printable, e.g., the number 17 is an instance of data type INTEGER. Relationships model references between instances of object types, i.e., relationships are directed. Their domain is another user-defined *object type*. Their values are instances of an object type, called *objects*. Objects are identified by a non-printable *object identifier*.

As mentioned before, relationships are *directed*[3]. This allows to decide already during conceptual design whether a binary relationship shall be directly accessible at objects of only one of the participating object types (one directed relationship) or at objects of both of the participating object types (two directed relationships).

Graphically, the structure of the instances of an object type is depicted in a type diagram. A *type diagram* is a box that consists of a header[4], and a body. The header, a shaded bar, shows the name of an object type. The body shows the specifications of properties of instances of an object type. Properties are depicted by different graphical symbols. An attribute property is depicted by a rounded rectangle, a relationship property by a rectangle. The name of a property is placed above, its domain within the graphical symbol representing it.

Example: Figure 3.1 shows a type diagram of object type ROOM-RESERVATION. Each room reservation has attributes resNo and noOfReminders with values from domain INTEGER, attributes arrivalDay, departureDay, and settlementDay with values from domain DATE, attributes amountDue, amountPaid, and balance with values from domain REAL, and attribute discount with a value from domain % (percentage). Furthermore, each reservation has relationships category, guest, room, and services to values from domains CATEGORY, GUEST, ROOM, and SERVICE, respectively.

To emphasize a direct semantic correspondence between two directed relationships, they may be declared to be *inverses*. Assume relationship r_1 of object type O_1 and relationship r_2 of object type O_2 are declared to be inverses. Then O_2 must be the domain of r_1 and O_1 must be the domain of r_2. Moreover, if relationship r_1 of an object o_1 (of type O_1) references an object o_2 (of type O_2), then relationship r_2 of o_2 must reference o_1 and vice versa. Two relationships are declared to be inverses by annotating at least one of the relationships. A relationship is annotated by giving the keyword inverseTo, the name of the domain type, and the name of the relationship to which the relationship is inverse. Graphically, the annotation that a relationship r_2 is inverse to a relationship r_1 of object type O_1 can be depicted (optionally) in the type diagram of object type O_1 within the graphical symbol representing relationship r_1.

[3]In Chapter 7, the concept of aggregation will be used to model *undirected* relationships. That means, undirected relationships are modeled as instances of an object type formed by means of aggregation of the object types participating in the relationship.

[4]When type hierarchies are introduced, type diagrams will be extended by an optional footer showing the subtype-of relationship of an object type to a supertype, if one exists.

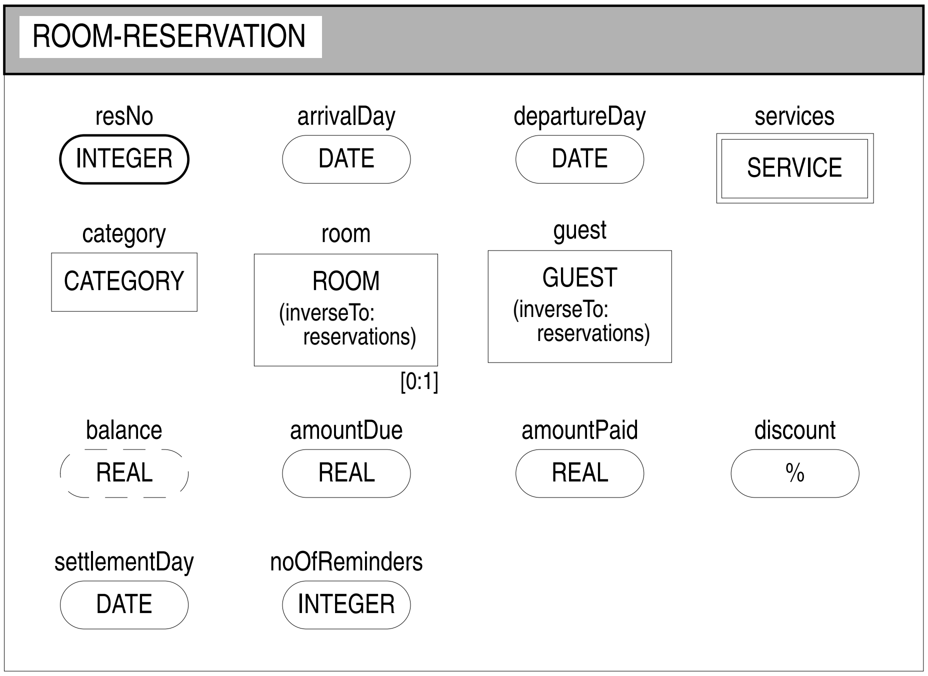


Figure 3.1: Object type ROOM-RESERVATION

Example: Relationship guest is inverse to relationship reservations defined at object type GUEST. Relationship room is inverse to relationship reservations defined at object type ROOM (cf. Figure 3.1 and Figure 4.3).

The properties of an object are either single-valued or multi-valued. A *single-valued* property holds at each time either no or exactly one instance of its domain, a *multi-valued* property holds a set of instances of its domain. The exact number of instances a property may hold at a time can be further determined by specifying the cardinality of the property. The *cardinality* v of a property is a specified by two integers v_{min}: v_{max}, denoting the minimum cardinality and the maximum cardinality, respectively. For single-valued properties the maximum cardinality must be one, the minimum cardinality can be either zero or one (the default). For multi-valued properties the maximum cardinality can be any integer greater then one (the default is positive infinity), the minimum cardinality can be equal to zero (the default) or any integer greater than zero and less than or equal to the maximum cardinality. In either case, however, a minimum cardinality of zero denotes an *optional* property, one greater than zero a *mandatory* property.

In a type diagram, a graphical symbol representing a single-valued property is depicted with a single-line border, one representing a multi-valued property with a double-line border. The cardinality of a property is depicted in square brackets placed below and aligned to the right inner border of the graphical symbol representing it. Note: The star symbol $*$ denotes positive infinity.

Example: Single-valued relationship room of room reservations is optional, i.e., its cardinality is [0:1] (cf. Figure 3.1). Note: In general, the desired room category is determined if a room reservation is issued, rooms are assigned to reservations later on, e.g., at check-in time. All other single-valued properties are mandatory, i.e., their cardinality is [1:1]. Multi-valued relationship services is optional, i.e., its cardinality is [0:∗]. Note: Default cardinalities are not shown in Figure 3.1.

Single-valued properties may be defined to be *key properties*. Key properties play a similar role in object-oriented models as their counterparts—primary keys—do in the Relational model, i.e., they allow to uniquely identify an instance of an object type. But contrary to the Relational model, where usage of primary keys is mandatory, usage of key properties is *optional*, since object-oriented systems maintain an *implicit key*, they object identifier. Key properties have to be initialized at object creation and cannot be modified later on. In a type diagram, a graphical symbol depicting a key property is shown with a bold-line border.

Example: Each room reservation is identified by the value of its key attribute resNo (cf. Figure 3.1).

The value of a property may be *derived*. A derived property has an associated derivation expression. A derivation expression is a query that may either be specified textually, e.g., using OQL (cf. [Atw93, CB97]), or graphically. Lang *et al.* [LOS96b, Lan97, Obe98] have introduced *query diagrams* as a graphical means to specify OQL-like queries. In a type diagram, a graphical symbol depicting a derived property is shown with a dashed-line border.

Example: The balance of a room reservation (cf. Figure 3.1) is the difference between the amount due for a reservation, reduced by an eventual discount granted to a guest, and the amount already paid for it. A negative balance says that a guest has not (fully) cleared the amount due yet. The corresponding derivation expression is: amountPaid - (amountDue * (1 - discount)).

3.2.2 Modeling the Behavior of Business Objects

The behavior of business objects (or *objects* for short) is determined by a set of *activities*, by *allowed execution orders* on these activities, and by their *interactions* with other objects. The behavior of the instances of a business-object type can be modeled at different levels of detail. Abstracting from complete specifications, a *life-cycle* identifies activities that represent public update operations and allowed execution orders of these activities. It is depicted in a *life-cycle diagram*. A *state specification* represents a state in the life-cycle of objects. It is depicted either in a life-cycle diagram, an activity-specification diagram, or in a separate *state-specification diagram*. An *activity specification* gives formal input and output parameters of an activity. It is depicted in an *activity-specification diagram*. An *activity realization* represents the implementation of an activity. It is depicted in an *activity-realization diagram*. This means, the behavior of the instances of a business-object type can be depicted using one life-cycle diagram, several state-specification diagrams, several activity-specification diagrams, and several activity-realization diagrams. Together these diagrams are called the *behavior diagram* of a specific business-object type (cf. [KS91, KS96b]).

Life-Cycle

A *life-cycle* models the public update behavior of instances of an object type using a set of *states*, a set of *activities*, and a set of directed *arcs* connecting activities to states and vice versa. As a notational convention, we start names of states and activities with a small letter. The names of states and activities must be unique within an object type. The principle idea behind life-cycles stems from Petri nets (cf. [Mur89, Pet77]). States correspond to places of Petri nets, activities to transitions. Like transitions are connected to places, activities are connected by a set of directed arcs to their pre- and poststates which represent necessary pre- and postconditions of activities, respectively. Consequently, *a life-cycle determines allowed execution orders on activities.*

The execution semantics of a life-cycle relies on the execution semantics of Petri nets (cf. [Mur89, Pet77]). Contrary to Petri nets, however, where a transition is fired automatically if all prestates contain a token, an activity in a life-cycle must be explicitly invoked for an object that resides in every prestate of the activity. During the execution of an activity on an object, the object resides in an implicit state named after the activity. Such a state is referred to as *activity state*. Note: This is different to Petri nets, where transitions occur instantaneously. Each instance of an object type resides, at any particular point in time, in one or several states. To record its current states, every object has a multi-valued property inStates which is implicitly specified at each object type.

Each life-cycle defined for objects of a particular type must at least consist of one create activity[5], one destroy activity[6], and one state that is poststate of the create activity and prestate of the destroy activity.

Graphically, the life-cycle of the instances of an object type is depicted in a life-cycle diagram. A *life-cycle diagram* shows a set of states, a set of activities, and a set of arcs connecting activities to states and vice versa. A state is depicted by a rectangle with trapezoids at its top and bottom. The top trapezoid contains the name of the object type of the instances 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.

Example: Figure 3.2 shows a life-cycle diagram for object type ROOM-RESERVATION. Once a room reservation has been issued, the customer may either cancel the reservation or check in at the hotel. Customers that cancel their reservations are charged with a cancellation fee, the amount charged depends on when their cancellation arrives at the hotel. Paying the cancellation fee settles the liability of a customer. The

[5] A *create activity* is an activity with no prestate, it creates a new instance of an object type.

[6] A *destroy activity* is an activity with no poststate, it can be used to destroy an object.

reservation will then be archived. If a customer checks in, she or he can consume diverse services. After her or his check-out, the hotel bills a guest for the services consumed during her or his stay. Once the bill has been paid for, a reservation gets archived. If a payment is overdue, a guest will receive one or several reminders. After a period of being archived, room reservations may be destroyed.

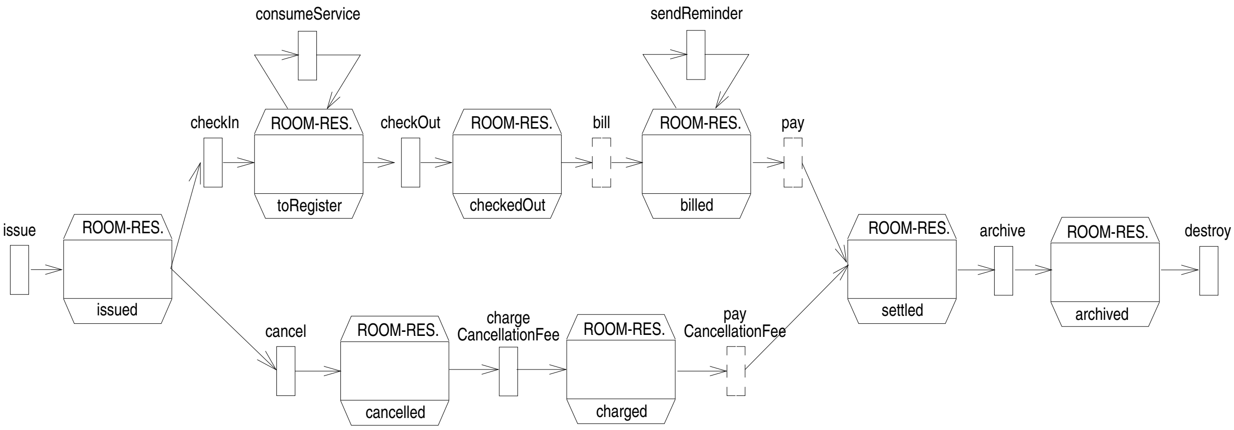


Figure 3.2: Life-cycle at ROOM-RESERVATION

The life-cycle of an object type can be developed by stepwise refinements of activities and states starting from one create activity, one state, and one destroy activity. Activities and states that have been refined are called *abstract*[7], those that have not *basic*. Each activity and each state can be refined into a life-cycle. To ensure consistent refinements of activities and states, we require that sources and sinks of their refinements[8] must also be activities and states, respectively, and that each refinement must have at least one source and one sink.

A major problem when refining activities or states of life-cycles is to ensure that if an object leaves a state in the refinement of one activity or state and enters a state outside the refinement, the object leaves the refinement entirely. Schrefl (cf. [Sch91]) proposed a set of structured refinement primitives for activities and states that ensure that an object leaves the refinement of an activity or state always entirely. The refinement primitives allow for a sequential, parallel, alternative, or iterative refinement of activities and states. Structured refinement of life-cycles aims at improving understandability, reuse, and maintenance of object life-cycles. Similar as structured programming aims at improving the quality of programming code by avoiding the use of GOTOs, structured refinement of life-cycles allows only sources and sinks of a refinement be connected by transfer arcs to elements outside the refinement. Elements in a refinement that are neither sources nor sinks may be connected

[7]Note: Currently, activities and states are implicitly declared to be abstract by refining them. In Chapter 7 an extension will be presented that allows to explicitly declare activities and states to be abstract, although no refinement has been given yet.

[8]Sources of a life-cycle are elements, i.e., activities or states, with no incoming arcs. Sinks of a life-cycle are elements with no outgoing arcs.

to elements outside the refinement only by read arcs. Note: Although every life-cycle given as an example in this thesis has been developed by applying structured refinement primitives, we do not require that life-cycles are necessarily developed by structured refinement primitives.

Objects reside in basic *and* in abstract states. An abstract state represents a process in the life-cycle of objects. The process is started if a source activity of the abstract activity is invoked, and is terminated if sink activities complete. An abstract state represents a period in the life-cycle of objects. The period is started if a source state of the abstract state is entered, and is terminated if sink states are left.

In a life-cycle diagram, activities and states that have been refined are shown with a dashed-line border. Alternatively, the refinement of an activity or state may be expanded in the original life-cycle diagram or in a separate one. In the former case, the refined activity or state may be superimposed to its refinement.

Example: Figure 3.3(a) shows an alternative refinement of activity bill defined for room reservations. A hotel may bill its guests at the reception or it may send the bill by letter or fax. Figure 3.3(b) shows an alternative refinement of activity pay defined for room reservations. A hotel may accept different forms of payment (cash, credit card, or money transfer) or it may delegate settlements to a collection agency.

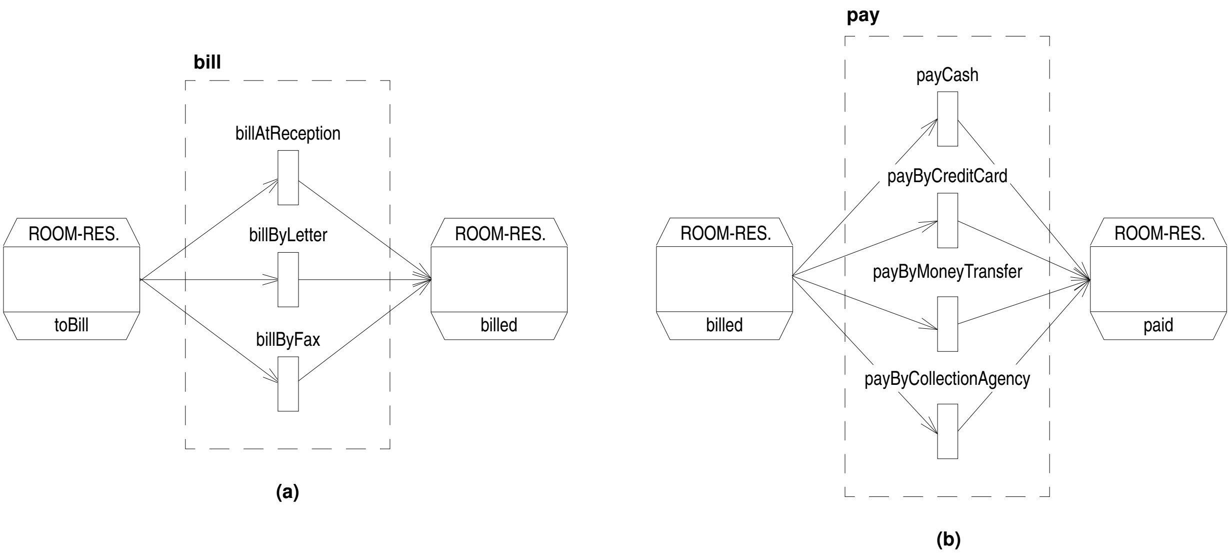


Figure 3.3: Refinements of activities (a) bill and (b) pay

State Specification

A *state specification* models a period in the life-cycle of instances of a particular object type by a *name* and a *domain*. The names of states must be unique within an object type. The domain of a state corresponds to the object type at which the state has been defined. In the advanced model (cf. Chapter 7), the specification of states will be extended in that constraints may then be associated with states either as preconditions, invariants, or postconditions.

Graphically, a state specification in its simple form, i.e., a state specification considering only the name and the domain of a state, can be shown in a life-cycle diagram, in an activity-specification diagram, or in a separate state-specification diagram. State specifications in their simple form are always depicted as states are depicted in life-cycle diagrams (above). A state specification in its extended form, i.e., a state specification considering also preconditions, invariants, and postconditions associated with a state (cf. Subsection 7.2.2), is only shown in a state-specification diagram.

Example: The specification of state issued of room reservations is depicted in the life-cycle diagram shown in Figure 3.2. The name of the state is shown at the bottom of the graphical symbol representing the state, the domain at its top.

Activity Specification

An *activity specification* models the interface of an activity by a *name*, a set of *input ports*, and a set of *output ports*. Input and output ports represent formal input and output parameters of an activity. Like properties, every port is specified by a *name*, a *domain*, and a *multiplicity*, which determines whether the port is *single-valued* or *multi-valued*. The names of ports must be unique within an activity. Each port is either an input port, an output port, or both. The exact number of values to be assigned to (to be returned by) a port of an activity is specified by the port's *cardinality*[9].

The domain of a *data port* is a data type, the domain of an *object port* is an object type. The domain of an object port is further restricted to a set of states in the life-cycle of the domain of the port. States connected to an input port p are called prestates of p, states connected to an output port poststates. Separate sets of prestates and poststates are connected to ports that are both input port as well as output port. An input port may be connected to a prestate either by a transfer arc or a read arc. A *transfer arc* indicates that the object assigned to the input port has to reside in the connected state before activity execution but must not reside therein after activity execution unless the port is also an output port and the state is contained in the port's poststates. 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 after activity execution. An output port may be connected to a poststate only by a transfer arc.

The ports of an activity are either single-valued or multi-valued. A *single-valued* port takes (returns) either no or exactly one instance of its domain, a *multi-valued* port a set of instances of its domain. The exact number of instances a property takes or returns can be further determined by specifying the cardinality of the port. The *cardinality* v of a port is a specified by two integers v_{min}: v_{max}, denoting the minimum cardinality and the maximum cardinality, respectively (cf. cardinality of properties in Subsection 3.2.1). The

[9] Object/Behavior Diagrams in their original form (cf. [KS91, KS96b]) do not allow to specify a cardinality with ports of activities.

default cardinality for single-valued ports is [1:1], for multi-valued ports [0:∗]. This means, single-valued ports are mandatory by default, multi-valued ports optional.

Every activity has a distinguished, mandatory single-valued object port, the *primary port*. The primary port is an output port in the case of a create activity, an input port and an output port in the case of an inner activity, and an input port in the case of a destroy activity. It identifies the object type to which the activity belongs, i.e., the domain of a primary port is the object type for which the activity is defined. Pre- and poststates of a primary port correspond to those specified in the life-cycle of instances of the domain of the primary port.

The formal return value of an activity is represented by a *return port*. Each output port may serve as return port. The default return port of an activity is its primary port.

A *precondition* and a *postcondition* can be specified with every activity. Preconditions represent restrictions on input parameters of activities, postconditions restrictions on their output parameters. Prestates connected to input object ports are part of the precondition of an activity, poststates connected to output object ports of an activity are part of its postcondition. If some port p is both, input as well as output port, of an activity, its postcondition may refer to the value of port p at the time the activity was invoked using the pseudo function *old(p)*. Prestates and poststates of an activity are specified graphically, the rest of its pre- and postcondition is specified textually using an OQL-like language (cf. [Atw93, CB97]).

The execution semantics of the activities of a life-cycle is determined by two port assignment rules. The *input port assignment rule* determines under which conditions an activity t can be invoked: (1) Every input port p of t is assigned the correct number of values (data values or objects), which is specified by the cardinality of p. (2) Every value assigned to an input port p must belong to the domain of p. (3) Every object assigned to an input port p must reside in every prestate of p. (4) The precondition of t is satisfied. The *output port assignment rule* determines under which conditions an activity t can complete successfully: (1) Every output port p of t is assigned the correct number of values, which is specified by the cardinality of p. (2) Every value assigned to an output port p must belong to the domain of p. (3) Every object assigned to an output port p must reside in every poststate of p. (4) The postcondition of t is satisfied. Note: It is assumed that activity realizations (below) do not change assignments of values to input ports of activities.

Graphically, the specification of an activity is depicted in an *activity-specification diagram*. It represents an activity specification by a rectangle showing the name of the activity above and a square for each port of the activity. Single-valued ports are depicted by squares with a single-line border, multi-valued ports by squares with a double-line border. Squares representing input ports are aligned to the left inner border of the rectangle, squares representing output ports to the right outer border. Consequently, a port which is both, an input port and an output port, is represented by two squares, which are aligned to the same height. The cardinality of a port is depicted in square brackets inside the rectangle representing an activity aligned to the same height as the square(s) representing the port. Note: Usually, default cardinalities of ports are not shown. The primary port of an activity is depicted in black, its return port is emphasized by superimposing a semi-circle to the square representing the return port.

The domain of a data port is depicted by a rounded rectangle showing the name of the data type inside. The domain of an object port is represented by a set of states which are depicted as in life-cycle diagrams. Transfer arcs connecting object ports to states are depicted by solid lines. Read arcs connecting object ports to states are depicted by dashed lines—like arcs connecting data ports to their domain. The names of ports are associated with the arcs connecting a port with its domain. Pre- and postcondition of an activity that are specified textually are shown as annotations below the rectangle representing the activity. The keywords precondition and postcondition are used to distinguish pre- and postcondition of an activity.

Example: Figure 3.4 shows an activity-specification diagram for activity billAtReception. Its pre- and poststates correspond to those of the life-cycle diagram. Parameter discount may hold a data value of type % (percentage). The precondition specified textually says that the discount granted to a guest must be between zero and ten percent. All ports of activity billAtReception are single-valued and mandatory. Note: As an exception default cardinalities of ports are shown in Figure 3.4.

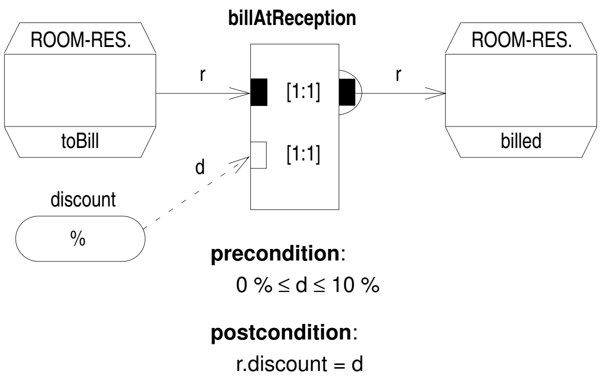


Figure 3.4: Specification of activity billAtReception

Activity Realization

An *activity realization* models the implementation of a basic activity. Activity realizations follow the object-oriented paradigm according to which the implementation of an operation has only direct read and write access to properties of the object on which it is performed—usually referred to via the pseudo variable *self*. To access or change a property of another object, an activity must send an appropriate message to this object. Message sends are modeled using activity invocations. The control flow of activities is modeled relying on structured diagrams. Activity realizations are not further discussed in this thesis. Note: The reader interested in activity realizations and their graphical representation, i.e., activity-realization diagrams, is referred to [KS91, KS96b] for information.

3.2.3 Modeling Classes of Business Objects

A *business-object class* (or *object class* for short) collects a set of business objects, called its *members*. Object classes are themselves instances of object types, called class types, and may themselves be members of several other object classes. That is, object classes are first-class citizens in the object model presented in this thesis. This means, object classes may have individual structure and behavior like general business objects. Moreover, since each object class acts as representative for the set of its members, an object class may model collective structure and behavior of its members[10], i.e., structure and behavior that concern the members of an object class considered as a unit. In addition, an object class may specify another object class as domain for each relationship that is defined for all its members.

Class Type

Each business-object class is instance of a specific business-object type, called *class type*. A class type specifies a particular business-object type, called *member type*, as member domain for its instances, called *object classes*. A class type plays a dual role: First, it specifies individual structure and behavior of its instances, like an ordinary business-object type (cf. Subsection 3.2.1). Second, it specifies collective structure and behavior of the members of its instances. Note: Individual structure and behavior of the members of the instances of a class type are specified at the member type of the class type. Collective structure and behavior of the members of an object class is structure and behavior that concern the members of an object class considered as a unit. Collective structure and behavior of members of object classes are specified at the class type of object classes in form of a set of *class properties* and a set of *class activities*.

Class properties are specified in the same way as individual properties of objects are specified at particular object type. Class activities denote the class behavior and are specified like individual activities of objects are specified. However, no valid execution sequences of class activities are specified in form of a life-cycle. Consequently, no prestates and no poststates will be specified with class activities.

As mentioned before, class properties express characteristics of the members of an object class considered as a unit. Frequently, their values are derived evaluating aggregate functions such as average, minimum, and maximum on the values of instance properties of the current set of members of an object class (cf. [MMWM93]). Appropriate derivation expressions are given by queries which may either be specified textually using OQL (cf. [Atw93, CB97]) or be specified graphically using query diagrams (cf. [LOS96b]).

Graphically, the structure of the instances of a class type is represented by a class-type diagram. A *class-type diagram* depicts the member type in a bar labeled hasMembers, located at the bottom of the diagram. The bar has a double-line border saying that,

[10] Object/Behavior Diagrams in their original form (cf. [KS91, KS96b]) do not represent object classes as first-class citizens. Consequently, Object/Behavior Diagrams in their original form do not allow to model class properties and class activities of class members.

in general, every object class of this type can have multiple members of the given type. Note: Whether an instance of a class type is a singleton-class or not, is specified for each object class individually (below). Class properties are depicted like instance properties in object-type diagrams.

Example: Figure 3.5 shows a type diagram of class type ROOM-RESERVATIONS. Members of object classes that are instances of type ROOM-RESERVATIONS must be instances of object type ROOM-RESERVATION (cf. Figure 3.1). They have a derived class-attribute NoOfReservations. The derivation expression for this class attribute is COUNT(hasMembers) or COUNT(*), i.e., the value of class attribute NoOfReservations is determined by evaluating the function COUNT on the current set of members of an object class of type ROOM-RESERVATIONS.

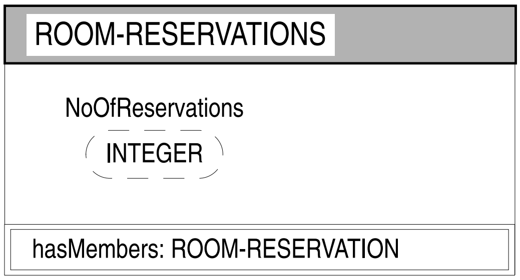


Figure 3.5: Class type ROOM-RESERVATIONS

Specification of Object Classes

Each object class is specified by a *name*, a *domain*, a list of *superclasses* (possibly empty), a list of *subclasses* (possibly empty), a *multiplicity* determining whether the object class is *single-valued* or *multi-valued*, and is either *basic* or *derived*. The name of an object class must be unique within the system, the domain is given by a class type (above). If an object class c is a derived class, a derivation expression is specified to determine the extension of c. The exact number of members an object class may hold is specified by the *cardinality* of the object class[11]. Moreover, a flag specifies whether an object class is the *base class* of the object type specified as member domain for the object class. Note: The member domain is specified at the type of an object class.

Object classes are specified *globally*, i.e., an object class is specified by introducing a global variable name[12] with the appropriate class type as domain. At system-startup time an instance of the given class type will be created and bound to the global variable. Variable names bound to object classes are treated like variables bound to constants in programming languages.

[11] Object/Behavior Diagrams in their original form (cf. [KS91, KS96b]) do not allow to specify the cardinality of object classes.

[12] This means, that every object class can be addressed using its name by all objects within a system.

Object classes may be organized in *class hierarchies*, which are specified by subclass-superclass relationships between object classes. More precisely, we denote a set of object classes which are related via subclass-superclass relationships a class hierarchy. Every object class may have one or several superclasses and may have one or several subclasses. That means, a class hierarchy forms a *semi-lattice*. The set of members of a subclass must be a subset of the set of members of its superclass. If a subclass has more than one superclass, its set of members must be a subset of the intersection set of the sets of members of its superclasses. Note: Each subclass must be instance of the same class type as its superclass(es). In Chapter 7 this restriction will be relaxed in the realm of type hierarchies. Then, the type of a subclass must be identical to the type of its superclass(es) or it must be a direct or indirect subtype thereof.

An object class is either a *basic* class or a *derived* class. To become (stop being) a member of a basic class, an object (a member) must be explicitly inserted into (removed from) that object class. To be a member of a derived class, an object must be in the result set of a *query* specified as derivation expression for that object class. The set of members of a derived class is determined by evaluating a query against sets of members of other (basic or derived) object classes. Note: Queries must be recursion-free, i.e., the set of members of an object class c must not be determined based on the set of members of c itself. A query may either be specified textually using OQL (cf. [Atw93, CB97]) or may be specified graphically using query diagrams (cf. [LOS96b]).

Object classes are either single-valued or multi-valued. A single-valued class, usually called *singleton class*, is either a basic or a derived class that is designated to hold only a *single member*. Note: If a singleton class is introduced as the base class of an object type O, there may only be a single instance of O at any particular time. In any case, singleton classes are a means to emphasize single instances of object types. A multi-valued class is either a basic or a derived class that holds a set of members. The exact number of members an object class may hold at a time can be further determined by specifying the cardinality of the property. The *cardinality* v of an object class is specified by two integers v_{min}: v_{max}, denoting the minimum cardinality and the maximum number of members an object class may hold. Cardinalities are specified for object classes as they are specified for properties of objects (cf. Subsection 3.2.1), except that the default minimum cardinality for single-valued classes is zero, not one.

Object classes function as *object warehouses*. This means, object classes can be used to maintain the extension of an object type. A special object class, called *base class*, must be associated with every object type. Base classes do automatically maintain the extension of object types. That is, each instance of an object type O will be automatically inserted as member into the base class c of O at the time it is created and it will be automatically removed from c at the time it is destroyed. Consequently, each instance of an object type is member of the base class of the object type of which it is instance during its lifetime. The base class of an object type must be a basic class.

A basic class must either be the base class of an object type or must be a direct subclass of some other basic classes. A derived class may be direct subclass of classes of any kind.

Graphically, an object class is specified by a rectangle showing the global variable name to which the object class will be bound at the top and the name of the class type inside the rectangle. The rectangle is shown with side-bars. Within the side-bars a single-headed arrow is shown if the specified object class is a singleton class, a double-headed arrow is shown otherwise. The cardinality of an object class is depicted in square brackets placed below and aligned to the right inner border of the graphical symbol representing it. Note: Usually, default cardinalities of classes are not shown. A base class is emphasized by depicting its graphical symbol with a bold-line border. A derived class is depicted with a dashed-line border. A subclass-superclass relationship between two object classes is shown by a double-line arc marked with the mathematical symbol for a subset relationship ($\subseteq$).

Example: Figure 3.6 shows the specifications of object classes Room-Reservations, Suite-Reservations, and King's-Suite-Reservation. Object class Room-Reservations of class type ROOM-RESERVATIONS is the base class of object type ROOM-RESERVATION. Note: Object type ROOM-RESERVATION is the member type of instances of class type ROOM-RESERVATIONS (cf. Figure 3.5).

Object class Suite-Reservations, a subclass of object class Room-Reservations, is a derived class. It contains all room-reservations that concern rooms of our hotel that belong to category 'Suite'. Figure 3.7 shows a query diagram (cf. [LOS96b]) that determines the extension of object class Suite-Reservations. Object class King's-Suite-Reservation, a subclass of object class Suite-Reservations, is a singleton class (cf. the single-headed arrows in the side-bars of the rectangle denoting the domain for global variable name King's-Suite-Reservations). The single member of object class King's-Suite-Reservations will be the next reservation for the only King's Suite of our hotel. Since object class King's-Suite-Reservations is a subclass of a derived object-class it must be a derived object-class, too. (Note: The query diagram that determines the extension of object class King's-Suite-Reservation is not shown.) Note: As an exception default cardinalities of classes are shown in Figure 3.6.

As mentioned before, object classes are defined *globally*. This means, every object class can be addressed using its name from everywhere within a system. Consequently, the members of some object class of a system are accessible from everywhere within that system, too. Although there may be objects which are of interest to a whole system, frequently objects will be of interest only to a single component of a system. Maintaining such objects in global object classes would make the system hard to survey. Kappel and Schrefl have introduced *local object classes* (cf. [KS92, KS96b]) to model such situations adequately. A local object class is an object class belonging exclusively to one component of a system. This means, if we specify an object class with name c at some object type O, there is not a single (global) object class c but there will be one object class c for every instance o_i of O. This situation has severe consequences on the class model of Object/Behavior Diagrams as presented in this thesis. Object classes cannot longer be created at system-startup time. Instead, a local class c must be created whenever a new instance of object type O is created. Note: Local object classes will be introduced in Chapter 7.

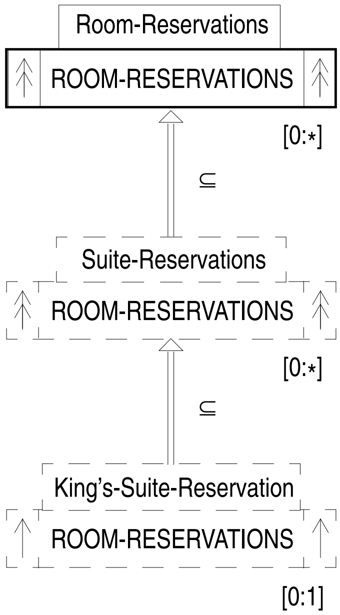


Figure 3.6: Graphical specification of object classes

Class-Domain Constraints

An object class may specify other object classes as domains for relationships defined for its members. More precisely, each object class may specify an object class as domain for every relationship that is defined for all its possible members[13]. This domain is called *class domain* to distinguish it from the domain specified for a relationship at an object type, which is called *type domain*. Note: The member type defined at the type of an object class that has been specified as class domain of a relationship r must be identical to the type domain of r. If no class domain is defined explicitly for a relationship r at an object class c, the class domain of r is implicitly defined by the *base class* of that type O that has been given as type domain of r[14]. Class domains represent *referential integrity constraints*. That is, each member of an object class may be related via a relationship r only to objects which are members of the class domain of r.

A *class-domain constraint* consists of a source class c_{source}, a relationship r, a target class c_{target}, and a cardinality constraint v (optional). The source class c_{source} identifies the set of objects for which the domain of relationship r is constrained. Relationship r must be defined for objects identified by c_{source}. Target class c_{target} identifies the set of objects serving as domain for relationship r of objects identified by c_{source}. The cardinality constraint v constrains the number of objects of the target class c_{target} objects of the source class c_{source} may reference via relationship r at a particular time. If no cardinality constraint v is

[13]Note: A relationship is defined for all possible members of an object class if it is defined at the member type that has been specified at the type of the object class.

[14]Note: The base class of an object type O maintains the *active domain* of O (e.g., [HK87]). The active domain of an object type O at time t is the set of all instances of O that exist at time t.

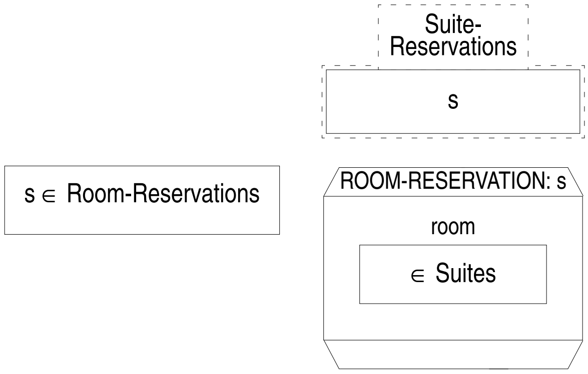


Figure 3.7: Query diagram for derived class Suite-Reservations

defined explicitly for a relationship r in a class-domain constraint, it is implicitly defined by the cardinality constraint specified for r at the member type of the type of the source class. If a cardinality constraint v is specified explicitly, it must be identical to or restrict the cardinality constraint v_M specified for relationship r at the member type M of the type of the source class. In general, a cardinality constraint v' restricts another cardinality constraint v, if $v'_{min} \geq v_{min}$ and $v'_{max} \leq v_{max}$ and if either $v'_{min} > v_{min}$ or $v'_{max} < v_{max}$.

Every subclass inherits class-domain constraints from its superclass[15]. It may specialize its superclass by refining inherited class-domain constraints and by introducing new class-domain constraints. A class-domain constraint c'_{source}: r: c'_{target}: v' refines a class-domain constraint c_{source}: r: c_{target}: v if c'_{target} is a subclass of c_{target}, c'_{source} is a subclass of c_{source}, and v' is identical to or restricts v.

Graphically, a class-domain constraint is specified in a *class-domain diagram*. Since class-domain constraints are constraints on the members of a class, the graphical symbols representing the object classes in a class-domain diagram emphasize the member domain aspect of object classes. That is, an object class is depicted by a rectangle showing the variable name to which it is bound at the top and the name of the member type inside the rectangle. Note: The member type of an object class is defined at the type of an object class and may be restricted by a member-type constraint at the specification of an object class. The rectangle has a single-line border if the depicted object class is a singleton class, it has a double-line border otherwise. Like in class diagrams (see above), a basic class is depicted with a solid-line border, a derived class with a dashed-line border. A base class is emphasized by a bold-line border.

A class-domain constraint is depicted by showing both the source class as well as the target class and connecting their graphical symbols by a simple line representing the relationship to be constrained. The name of the relationship is associated a direction indicator (a small arrow-head) pointing to the class domain (the target class). A second name associated with a relationship line represents an inverse relationship. Cardinalities are depicted in square brackets placed below the line representing the constrained relationship. They

[15]Note: Inheritance of class-domain constraints does not occur at the class-type level, it happens at the object-class level.

are aligned to the target of a relationship. Note: Showing [1:1] cardinalities is optional.

Example: Figure 3.8 shows a class-domain diagram with object classes, Room-Reservations, Suite-Reservations, King's-Suite-Reservation, Rooms, Suites, Guests, VIP-Guests, and King. Note: Base classes are emphasized by a bold-line border.

Object class Rooms collects instances of object type ROOM. Object class Suites, a derived subclass of object class Rooms, collects that subset of members of object class Rooms that are assigned to room category 'Suite'. (Note: The derivation expression of object class Suites is not shown.)

Object class Room-Reservations collects instances of object type ROOM-RESERVATION. It restricts the domain of relationship room to object class Rooms. Object class Suite-Reservations, a derived subclass of class Room-Reservations, collects instances of object type ROOM-RESERVATION. It restricts the inherited class domain of relationship guest to object class VIP-Guests, and that of relationship room to object class Suites. The derivation expression determining the set of members of object class Suite-Reservations has been specified by a query diagram shown in Figure 3.7. Object class King's-Suite-Reservation, a derived subclass of class Suite-Reservations, is a singleton class. It has either no member or contains the next reservation for the single King's Suite of our hotel. Object class King's-Suite-Reservation restricts the inherited class domain of relationship guest to object class King, i.e., the hotel may not let the King's Suite to anyone else than the King.

Object class Guests collects instances of object type GUEST and restricts the domain of relationship reservations to members of class Room-Reservations. Object class VIP-Guests is a basic subclass of object class Guests. Whether a guest of our hotel becomes a member of object class VIP-Guests is decided on by an authorized person. That means, guests must be explicitly inserted into object class VIP-Guests. Object class King is a subclass of object class VIP-Guests. It is a singleton class, representing the fact that there may only be one King.

Relationships reservations and room are optional, their cardinalities are [0:*] and [0:1], respectively. Relationship guest is mandatory, i.e., its cardinality is [1:1] (not shown).

Example: Figure 3.8 shows the graphical specification of a set of class-domain constraints. Below, a textual specification of these class-domain constraints is given. Note: As an exception, default cardinalities are given in the textual specifications of the class-domain constraints below.

```
Room-Reservations:guest:Guests: [1:1]
Room-Reservations:room:Hotel-Rooms: [0:1]
Guests:reservations:Room-Reservations: [0:*]
Suite-Reservations:guest:VIP-Guests: [1:1]
King's-Suite-Reservation:guest:King: [1:1]
```

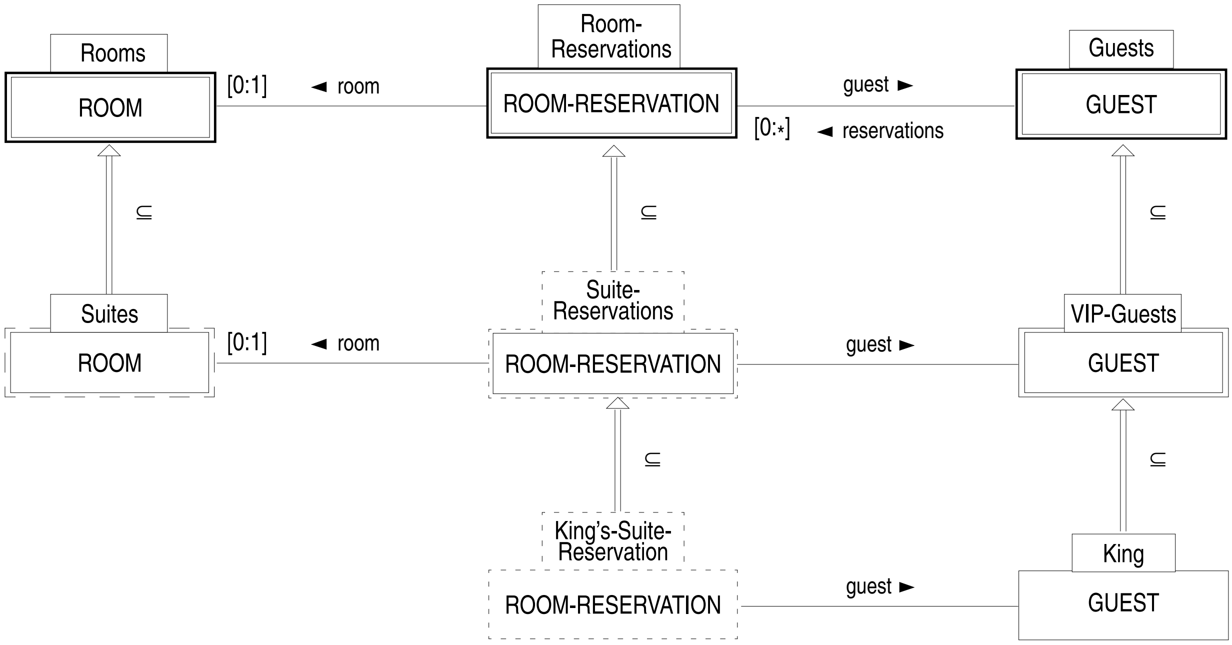


Figure 3.8: Class-domain diagram

3.3 Summary

In this chapter we presented basic concepts for the conceptual design of object-oriented schemas. First, we gave a management definition of the fundamental concepts presented in this chapter to build a common terminology. Then, a basic set of modeling concepts of our conceptual data model together with their graphic notation were presented. The presented modeling concepts build on known abstraction mechanisms: classification and association. Objects that share common properties are collected into object classes (*classification*). The elements of an object class are called its members. A specific object type, called the member type of an object class, specifies the structure and behavior of the members of an object class. The elements of an object type are called its instances. An object is instance of exactly one object type and is member of one or several object classes. The structure of objects is specified by a set of attributes and a set of relationships. Attributes model descriptive features of objects, relationships model references to other objects (*association*). The behavior of objects is specified by sets of activities and by allowed execution orders on these activities. The object-oriented data model together with its graphic notation serves as a fundament for all submodels of the workflow model presented in this thesis. This means, all elements in the business area, e.g., authorizations, processes (cf. Subsection 1.2.2), are represented as objects.

Chapter 4

Process Modeling

Contents

Workflow management systems (WFMSs) specify, enact, and maintain organizational processes. We propose a two-schema architecture for workflow systems that considers two distinct models of organizational processes. A *business process* is a model of an organizational process that considers only knowledge on the process that is external to an organization. That is, business processes consider only knowledge on organizational processes that is expressed in form of external business rules, e.g., business rules based on natural facts or law, and do abstract from knowledge expressed in form of internal business rules, i.e., knowledge that builds on organizational commitments. A *workflow process* is a model of an organizational process as it is currently performed in a particular organization. This means, a workflow process considers both, knowledge on an organizational process expressed in form of external business rules as well as knowledge expressed in form of internal business rules. However, a workflow process does not represent external knowledge on an organizational process directly. Instead, it delegates the representation of external knowledge on an organizational process to a business process that models the same organizational process.

This chapter is organized as follows. Section 4.1 gives a management definition of the fundamental modeling concepts presented in this chapter. Section 4.2 presents a two-schema architecture for workflow systems. Section 4.3 discusses the design of business-process schemas. In fact, business-process schemas are developed relying exclusively on modeling concepts offered by the object-oriented data model that has been presented in Chapter 3. Section 4.4 discusses the design of workflow-process schemas. First, it reviews object-oriented concepts (cf. Chapter 3) in the realm of the modeling of workflow-process objects. Second, it presents an extension of the object-oriented data model and the graphical notation presented in Chapter 3 for the conceptual design of workflow processes. Section 4.5 summarizes this chapter.

4.1 Fundamental Modeling Concepts

The core concept presented in this chapter is a *two-schema architecture* for workflow systems. In the context of a two-schema architecture for workflow systems, organizational processes are represented by two related but separate models, a business process and a workflow process. Whereas a business process shows how work could be done in general, a workflow process shows how work is actually performed within a particular organization. This means, a business process is a model of an organizational process considering knowledge that is external to an organization. A workflow process is a model of an organizational process considering knowledge that is internal to an organization. This means, an organizational process is completely represented by a business process and a workflow process together.

Organizational process: An *organizational process* is a purposeful collection of activities that add a value for the customer of the process. To carry out the activities of an organizational process, several actors—humans and machines—collaborate. Frequently, these actors will belong to different organizational units, i.e., organizational processes cross functional boundaries (cf. [Oul95]).

Business rule: A *business rule* is a statement about how the business is done, i.e., about guidelines and restrictions with respect to states and processes in an organization[1]

[1]Note: Lang *et al.* [LOS97] proposed situation/activation diagrams as a graphical means for the modeling of business rules in Object/Behavior Diagrams. They define business rules as "*rules on business policies* defining which transactions have to be performed at specific calendar times or which transactions have to be initiated upon the occurrence of specific, consistent database states". That is, situation/activation diagrams can be used to model a particular kind of behavioral business rules (cf. Section 4.2).

(cf. [Her95]). Business rules can be classified into external and internal ones. *External* business rules are based on ethics, culture, law, or natural facts. Organizations cannot—at least not directly—modify business rules which are of external origin. Rather, external business rules define a frame within which organizations may operate. Internal business rules, which express organizational commitments, are the means of an organization to adapt the frame to its individual requirements.

Two-Schema architecture: We propose a two-schema architecture for workflow systems that represents each organizational process by two related but separate models, a business process and a workflow process. A *business process* considers knowledge on organizational processes that is external to an organization (i.e., knowledge expressed in form of external business rules), a *workflow process* considers knowledge that is internal to an organization (i.e., knowledge expressed in form of internal business rules). The separation of the knowledge on organizational processes into two separate models is motivated by the following observations: First, a large number of changes to organizational processes are due to changes in the business environment of organizations, i.e., the market. Second, organizations react to this kind of changes by adapting their internal business rules, because organizations cannot change external business rules (e.g., business rules based on natural facts or law). In a two-schema architecture such changes are reflected only in the workflow process leaving the business process the same. This means, a two-schema architecture for workflow systems supports the evolution of organizational processes.

Business process: A *business process* models an organizational process considering only knowledge on the organizational process that is expressed in form of external business rules, i.e., business rules based on natural facts or law. Knowledge on the organizational process that is expressed in form of internal business rules, i.e., business rules based on organizational commitments, is modeled in a workflow process. A business process is defined as the behavior of a business-process object.

Workflow process: A *workflow process* models an organizational process as it is currently performed in a particular organization. This means, a workflow process considers both, knowledge on an organizational process expressed in form of external business rules as well as knowledge expressed in form of internal business rules. However, in the context of a *two-schema architecture*, a workflow process does not represent knowledge expressed in form of external business rules directly. Instead, a workflow process delegates the representation of external knowledge on an organizational process to a business process. A workflow process is defined in the behavior part of a workflow-process type.

Business-process activity: A *business-process activity* is a step in a business process. It is defined as an activity of business-process objects.

Business-process object: A *business-process object* is a special kind of business object (cf. Section 3.1). This means, each business-process object is instance of an object type, called business-process type, and is member of one or several object classes, called business-process classes. Unlike ordinary business objects, however, the behavior defined for a business-process objects models a business process (cf. business process, above). SYNONYMS: business case (cf. [BPS97a, KBKS96]), business-process instance (cf. [Wor96]), work item (cf. [Wor96]).

Workflow-process activity: A *workflow-process activity* is a step in a workflow process. SYNONYM: workstep.

Workflow-process object: A *workflow-process object* (or workflow object) is a special kind of business object (cf. Section 3.1). This means, each workflow-process object has structure and behavior like an ordinary business object. In addition, however, every workflow-process object has a designated relationship to a set of business cases of a specific type. Together the type of the business cases and the type of the workflow-process object define an organizational process. The type of the business cases, i.e., a business-process type, defines the business-process model of an organizational process, the type of the workflow-process object, i.e., a workflow-process type, the workflow-process model. This means, a workflow-process type specifies additional behavior in form of a workflow process which adapts the business process of the selected type of business cases to the current settings of a particular organization.

Workflow-process class: A *workflow-process class* is a business-object class (cf. Section 3.1) that collects a set of workflow-process objects as its members. SYNONYM: workflow-object class.

Workflow-process type: A *workflow-process type* extends ordinary business-object types (cf. Section 3.1) in that it specifies in addition to structure and behavior of its instances, behavior designated to business cases processed by its instances. This kind of behavior is called workflow process. SYNONYM: workflow-object type.

Summing up, a two-schema architecture for workflow systems considers knowledge on organizational processes in two related but separate models. A business process models knowledge on an organizational process expressed in form of external business rules, a workflow process models knowledge on an organizational process expressed in form of internal business rules. That is, an organizational process is completely represented by a business process and a workflow process together. The two-schema architecture supports the evolution of organizational processes. The two-schema architecture builds on the observation that organizations may only change internal business rules to adapt to changes in their business environments. As a consequence, many changes are reflected only in workflow processes leaving business processes the same.

Furthermore, it is fundamental to state that the modeling concepts presented in Chapter 3 apply in the same way to the modeling of business-process objects and workflow-process objects. For the modeling of workflow processes, however, the set of modeling concepts must be extended. The focus of Section 4.4 is on introducing a specific set of concepts for the modeling of workflow processes.

4.2 A Two-Schema Architecture for Workflow Systems

Every organization has a number of processes it carries out to achieve its business objectives (cf. [Oul95]). Which processes an organization carries out, in which way it carries them out, and foremost the business objectives themselves are determined by the *business policy* of an organization. The business environment of organizations, i.e., the *market*, forces organizations more or less often to change their business policy in order to be and to stay competitive. To implement such changes, information systems must be frequently adapted to changed requirements, too. Loucopoulos *et al.* [LTP91] pointed out that "the evolution of information systems is, to a large extent, due to changes in the business environment" and that improvements could be made in the maintenance and evolution of information systems if the knowledge expressed in business policies were represented explicitly. Therefore, recent approaches to the modeling of information systems, e.g., [DAHK91, HKMS94, KHS94, LTP91, MNP$^+$91], express the business policy of an organization in form of business rules.

We propose a *two-schema architecture* for the modeling of organizational processes for WFMSs, claiming that further improvements could be made in the maintenance and evolution of organizational processes if the knowledge on these processes expressed in business policies were not only represented explicitly in form of business rules, but, in addition, if business rules were classified into two categories according to their origin (cf. Figure 4.1). *External business rules*—e.g., business rules based on natural facts or law (cf. [Her95, HK94])—determine a frame within which an organization may operate. *Internal business rules*—i.e., business rules based on organizational commitments—restrict this frame and determine the way an organization actually runs its business.

The two-schema architecture builds on the fact that organizations may change their internal business rules to react to changes in their business environments but, usually, have no—at least no direct—influence on business rules of external origin. Therefore, our approach to the modeling of organizational processes represents these processes in two distinct but related schemas. A *business-process schema* represents knowledge on an organizational process expressed by external business rules, a *workflow-process schema* knowledge expressed by internal business rules (cf. Figure 4.1). Consequently, the two-schema architecture copes very efficiently with a large number of changes of organizational processes that are only reflected in workflow-process schemas leaving the underlying business-process schemas the same.

Business rules are declarative statements expressing business knowledge (cf. Section 3.1). Following Gale and Eldred [GE96], we distinguish business rules that express knowledge on the structure of entities from the business area from those that express knowledge on their behavior. *Structural business rules*, usually called integrity rules, express conditions and constraints on attribute values of entities and their relationships to other entities. Section 3.2.1 presents a high-level language to express structural business rules on objects, which model entities from the business area. *Behavioral business rules* control processes, i.e., they express conditions and constraints on the consistent invocation and completion

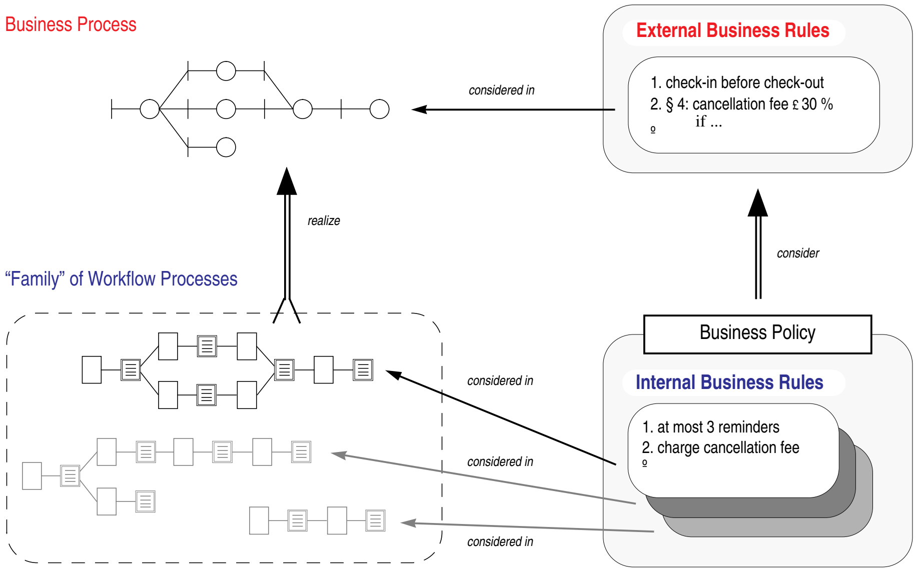


Figure 4.1: Basic concepts of the two-schema architecture

of activities performed on entities as well as on the allowed order of execution of these activities. Behavioral business rules, although expressed in form of *event-condition-action* rules by some authors (e.g., [Her95, HK95]), are *better represented by appropriate high-level concepts*, such as extended Petri nets, at the conceptual level (cf. [KS96a]). Section 3.2.2 presents an extended Petri net formalism that allows to express behavioral business rules. Obviously, the focus of the two-schema architecture is on behavioral business rules.

In the context of the two-schema architecture, each organizational process is modeled in form of a business-process schema and one or several workflow-process schemas, *separating behavioral business rules into external and internal ones.* The modeling of an organizational process for a WFMS that supports the two-schema architecture requires additional effort during the early phases of the information systems engineering cycle. During the first phase, called *enterprise planning*, goals and objectives as well as strategic business areas of an organization are described. During the second phase, called *business area analysis*, organizational processes that may be carried out to achieve the objectives of an organization are investigated. Both phases require additional effort, since the rationale behind organizational commitments, i.e., external business rules, must be extracted and must be made explicit. However, the additional effort more than pays off during the *maintenance* phase, which is, from the point of view of time, the longest and usually the most expensive phase of the information systems engineering cycle.

To design a business-process schema means to abstract from knowledge on organizational processes expressed in form of internal business rules, i.e., to abstract from organizational commitments which are subject to frequent changes caused by the dynamics of businesses. This means, a business-process schema describes an organizational process independent from the settings of any particular organization. This property of business-process schemas, called *organization independence*, allows that one business-process schema may serve as a fundament for a whole family of workflow-process schemas (cf. Figure 4.1) paying off the additional effort of the two-schema architecture. Due to organization independence, the various workflow-process schemas might even belong to different organizations. What, in detail, does it mean to design a business-process schema? To design a business-process schema means (a) to offer a *variety of alternatives* to achieve the purpose of the organizational process, (b) to allow for *parallelism* in the process where possible, i.e., to require sequentialization of process activities only if enforced by input-output dependencies or fundamental process logic, (c) to pose as weak restrictions as possible on the invocation of process activities, and (d) to allow for different realizations of process activities. The latter can be easily accomplished if the two-schema architecture is built on an object-oriented fundament. Note: Object-oriented system separate the specifications of activities form their realizations. Since realizations are hidden (*information-hiding*), they may be adapted to changed requirements without influencing clients. Furthermore, *late-binding* allows to even change realizations of activities while processes are carried out.

A workflow-process schema adapts a business-process schema, or part of it, to the *current set of internal business rules* relevant for the corresponding organizational process. The aim of a workflow-process schema is to represent the organizational process, described in general by the underlying business-process schema, as it is *currently* performed in a *particular* organization. To design a workflow-process schema thus means (a) that one may *select only a subset of the alternatives* offered by the business-process schema to achieve the purpose of the organizational process, (b) that one may *restrict possible execution orders of process activities* by enforcing additional sequentialization of business-process activities, (c) that one may *pose stronger restrictions on the invocation of business-process activities*, (d) that one may *restrict the invocation context of business-process activities in time and space* by calling two or more business-process activities in a single workflow-process activity[2], and (e) that one determines how the input for business-process activities is actually selected and how related external operations are performed.

In this thesis we show how Object/Behavior Diagrams, which have originally been introduced for the conceptual design of object-oriented database schemas (cf. [KS91, KS96b]), can be extended to a two-schema architecture. Each organizational process is described by two object types in the extended model of Object/Behavior Diagrams (cf. Figure 4.2). A *business-process type* specifies all data kept for business cases of an organizational process in its structure part, and specifies the business process partially representing an organizational process in its behavior part. The specification of the workflow process of an organizational process—which is subject to frequent changes since it considers internal business rules—is, however, moved from the business-process type to the related workflow-process type (cf. Figure 4.2) to improve maintenance and evolution of organizational processes[3]. A

[2]This concept is similar to that of behavior composition presented in [Kap91, KS96b] where selected operations and their order of execution are encapsulated to a new operation. In the two-schema architecture, however, the new operation is defined at a distinct type, the workflow-process type.

[3]Note: The aim of the two-schema architecture is to support changes in the "flow of work" in a flexible way. To achieve its purpose, the two-schema architecture represents each organizational processes by two models, a business-process model and a workflow-process model, separating behavioral business rules

workflow-process type specifies the workflow process representing an organizational process in its behavior part. The behavior part of a workflow-process type is extended to specify workflow processes by means of containers of business cases, called boxes, and workflow-process activities, called worksteps (cf. Subsection 4.4.4). Note: Each business case is represented as instance of a business-process type, which defines a business-process model of the organizational process according to which it is to be handled. A business case is assigned as member to a workflow object, which is instance of a workflow-process type that defines a workflow-process model of the organizational process according to which its members are to be handled. Consequently, an occurrence of an organizational process is represented by a business case that is member of one or several workflow objects.

In its structure part, each workflow-process type defines a special relationship for its instances. This so-called *has-member relationship* (cf. Subsection 4.4.1) relates a workflow object to its members, i.e., the set of business cases which are to be handled according to the workflow process defined for the workflow object.—Note: The workflow process defined at a workflow object and the business process defined at a related business-process type must be models of the same organizational process.—Relationships to other business processes are specified in the structure part of the business-process type, relationships to other workflow processes in the structure part of the workflow-process type. Note: In addition to the has-member relationship and behavior designated for members of its instances, each workflow-process type may specify individual structure and behavior of its instances, called workflow objects. Actually the behavior of workflow objects may itself represent an organizational process and may be designed for the two-schema architecture, too (Figure 4.2 gives a suggestion of the separation of the behavior of workflow objects.). This means, the concept of *the two-schema architecture is orthogonal* since it could be applied to workflow objects, too.

At runtime several business cases are assigned to a single workflow object. Assigning business cases to a workflow object means that their business process is to be carried out according to the additional restrictions imposed by the corresponding workflow process. The two-schema architecture relies on the basic principle of *delegation*. This means, *workflow-process activities may change the state of business cases only by invoking business-process activities*. Therefore, changes on business cases performed in workflow processes implicitly obey the constraints defined in the underlying business-process schema.

The two-schema architecture considerably improves the support of WFMSs for the maintenance and evolution of workflow-process schemas and the evolution of business cases to new workflow processes[4]. The two-schema architecture provides for *workflow transparency* which is a quality of models representing organizational processes that allows to dynamically change the workflow process without effecting the correctness of the busi-

into external and internal ones, respectively. Structural business rules, however, are *not* separated into external and internal ones in general. If such a separation is required, structural business rules could be considered as preconditions of business-process activities and workflow-process activities, respectively (cf. Subsection 4.3.2 and Subsection 4.4.4).

[4]Ellis and Keddara [EK95] coined the term *dynamic change* for the evolution of business cases to new workflow processes.

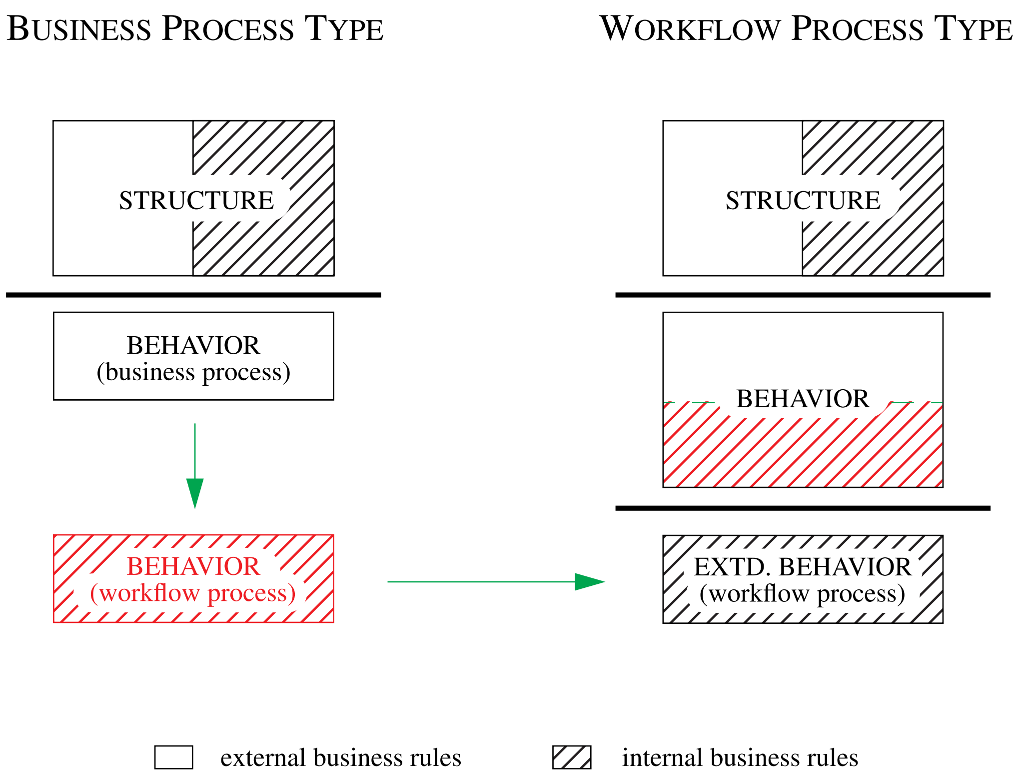


Figure 4.2: Object/Behavior Diagrams extended to a two-schema architecture

ness process it implements. The two-schema architecture allows to *verify* modifications of workflow processes against the business process they implement already at the schema level. In single-schema architectures modifications of workflow processes must be *validated* with respect to their environment.

The two-schema architecture supports the evolution of business cases to new workflow processes in a very flexible way. Business cases handled according to an old workflow process need not be migrated at once to the new workflow process (*static migration*) but can be migrated from time to time to the new workflow process making optimal use of both, the new and the old workflow process (*dynamic migration*). Business cases which have been migrated to a new workflow process need not be explicitly assigned to a particular position in the new workflow process (*static classification*) but qualify automatically for the right position according to their state (*dynamic classification*). Transition regulations can be specified if business cases do not qualify for a regular position in the new workflow process. The meaning of *workflow transparency* in this context is that a business case may be processed according to different workflow processes during its lifetime guaranteeing that, at any time, its business process is obeyed. Fundamental for providing workflow transparency is that business cases have no explicit relationship to the workflow object according to whose workflow process the are handled. In fact, only workflow objects have a relationship to the business cases that are handled according to its workflow process, not vice versa (above). For more information on the evolution of business cases to new workflow processes the reader is referred to [BPS97a, BPS97b].

Besides improving the support of WFMSs for the maintenance and evolution of workflow-process schemas and the evolution of business cases to new workflow processes, the two-schema architecture offers additional advantages:

- The *organizational memory grows* (cf. [KB95]), since additional information on organizational processes is made explicit which is otherwise hidden or only implicitly considered by internal business rules. External business rules are usually the rationale behind internal business rules. Making this knowledge on organizational processes explicit helps to improve maintenance and evolution of organizational processes.

- The separation of external and internal knowledge on organizational processes offers a potential for further separation and decentralization of analysis, design, implementation, test, and maintenance tasks.

- The two-schema architecture offers potential to *source out part of the development of organizational processes*. Since a business-process schema represents an organizational process as it could be performed by a number of different organizations in different business environments, its development may be sourced out of organizations. Note: Sourcing out the development of business-process schemas reduces the additional effort caused by the two-schema architecture. In the future, organizations might buy business-process schemas off the shelf, like a text editor or a spreadsheet application. These business-process schemas might be adapted to the requirements of the particular industry or trade in which organizations are doing their businesses.

> To make this vision come true, however, further research on the development of standards in the area of common object-models and common process-modeling paradigms must be conducted. An important contribution to the development of a common object-model has been made by the ODMG [Atw93], which recently published a new version of their standard object model [CB97]. Van der Aalst [Aal96] proposes Petri nets as a standard paradigm for the modeling of processes.

We conclude this section with a definition of workflow transparency:

> *Workflow transparency* denotes the quality of a workflow system to consider the specification of a business process independently from the workflow process(s) implementing it.

4.3 Design of Business-Process Schemas

A *business-process schema* consists of a set of business-process types and a set of business-process classes. Each business-process type defines the structure and behavior of its instances, called *business-process objects* (or *business cases*). A *business process* (cf. Section 4.1 and Section 4.2) is defined in the behavior part of a business-process type. A business-process class collects a set of business cases of a specific type as its members.

An *instance* of a business-process schema consists of a set of business cases. Each business case is instance of one business-process type and is member of one or several business-process classes. Business cases may be related to and may communicate with other business objects. However, in the context of the two-schema architecture for WFMSs (cf. Section 4.2) a business case *must not* have a user-defined relationship to a workflow object if it is handled according to the workflow process defined for that workflow object.

Business cases are business objects, too. This means, the concepts presented in Section 3.2 for the conceptual modeling of business objects can be applied in the same way to the modeling of business cases and business processes. Below, the concepts presented for the conceptual modeling of business objects are shortly reviewed in the realm of the conceptual modeling of business cases and the according business processes.

The remainder of this section is organized as follows. Subsection 4.3.1 discusses the modeling of the structure of business-process objects, Subsection 4.3.2 the modeling of the behavior of business-process objects, i.e., the modeling of business processes, and Subsection 4.3.3 the modeling of classes of business-process objects.

4.3.1 Modeling the Structure of Business-Process Objects

The structure of business-process objects (or *business cases*) is determined by a set of properties, i.e., by a set of *attributes* and a set of *relationships* (cf. Section 3.2.1). Whereas attributes model relevant data of business cases, relationships model references of business cases to other business objects, e.g., organizational objects (cf. Chapter 5) or other business cases. To guarantee the quality of *workflow transparency*, however, business cases must not hold a user-defined relationship to the workflow object(s) according to whose workflow process(es) they are handled (cf. Section 4.2). Instead, system-defined and system-maintained relationships are offered that allow each business case to refer to the workflow object(s) according to whose workflow process(es) they are handled currently.

Business cases that are handled according to the workflow process of a workflow object are assigned as *full members* to that workflow object. Besides being full members of a workflow object, business cases may be assigned as *affiliate members* to workflow objects. Whereas full members of a workflow object may be modified in a workflow process by invoking worksteps on them (cf. Chapter 4), no worksteps may be invoked on affiliate members. Affiliate members serve as inputs of worksteps and may be modified indirectly. An affiliate member is modified indirectly if a workstep invokes a business-process activity on a full member which directly or indirectly invokes a business-process activity on an affiliate member.

A *full-member-of relationship* models a directed relationship of a business case to the workflow object(s) according to whose workflow process(es) it is handled. Relationships inverse to full-member-of relationships are has-full-member relationships (cf. Subsection 4.4.1). An *affiliate-member-of relationship* models a directed relationship of a business case to the workflow object(s) in whose workflow process(es) it is handled indirectly (above). Relationships inverse to affiliate-member-of relationships are has-affiliate-member relationships (cf. Subsection 4.4.1). To provide for workflow transparency, no user-defined full-member-of or affiliate-member-of relationships may be defined with business-process types. Instead, a predefined full-member-of relationship with name fullMemberOf and a predefined affiliate-member-of relationship with name affiliateMemberOf are offered. Both relationships are multi-valued and return the sets of workflow objects to which a business case is assigned as full member and as affiliate member, respectively. The values of these relationships are automatically maintained.

Relationships between business-process objects model a network of the business processes an organization carries out to achieve its business objectives. Ould [Oul95] proposes to divide organizational processes into three broad types: core processes, support processes, and management processes. A *core process* concentrates on external customers of an organization. It directly adds value to the customer (cf. Section 4.1). A *support process* concentrates on internal customers of an organization. It might add value to the customer indirectly by supporting a core process. A *management process* concentrates on managing core processes and support processes.

Example: Figure 3.1 shows a type diagram of business-process type ROOM-RESERVATION. Processing of room reservations is a core process of our hotel, adding direct value to its customers, i.e., the guests (cf. Subsection 4.3.2 for the specification of the business process). Each room reservation is identified by the value of its key attribute resNo and is described by the values of further attributes arrivalDay, departureDay, settlementDay, discount, amountDue, amountPaid, balance, and noOfReminders. Each room reservation has relationships category and services to business objects of type CATEGORY and SERVICE, respectively. The former describes the desired room category and determines the price the guest has to pay for his stay, the latter describes the varying services a guest consumes during his stay. Furthermore, a room reservation has a relationship guest to an organizational object of type GUEST (cf. Chapter 5), and a relationship room to a business-process object of type ROOM. Business-process type ROOM (cf. Figure 4.3) defines a support process (cf. Subsection 4.3.2) that adds indirectly value to guests by servicing rooms.

Figure 4.3 shows a type diagram of business-process type ROOM. Each room is identified by the value of its key attribute roomNo. Furthermore, each room has relationships reservations and category to values from domains ROOM-RESERVATION and CATEGORY, respectively. Note: Relationship reservations is inverse to relationship room defined at object type ROOM-RESERVATION (cf. Figure 3.1).

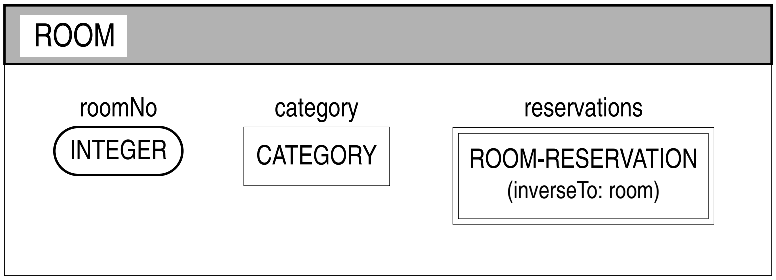


Figure 4.3: Business-process type ROOM

4.3.2 Modeling the Behavior of Business-Process Objects

The behavior of business-process objects, i.e., a *business process*, is determined by a set of *business-process activities*, by *allowed execution orders* on these activities, and by their *interactions* with other business objects. Behavior diagrams (cf. Section 3.2.2) are used to depict business processes.

To design a business process for a two-schema architecture means that a simple guideline must be followed when modeling organizational processes: *consider only knowledge on the organizational process expressed by external business rules* (cf. Section 4.2). Consequently, designing business-process schemas of an organizational process requires a basic working knowledge of the business policies effecting the organizational process to be modeled.

A *life-cycle* identifies business-process activities and their valid execution sequences. When designing a business process, a rich set of alternatives to meet the objectives of the process must be considered. Moreover, sequential execution of business-process activities may only be required if there exist input-output dependencies or if enforced by the underlying process logic.

Example: Obviously, the life-cycle of object type ROOM-RESERVATION shown in Figure 3.2 does not represent a business process. Too restrictive are the alternatives offered to meet the business objectives of the hotel, e.g., customers that cancel their reservation will always be charged a cancellation fee, and too restrictive are the allowed execution sequences of the activities, e.g., guests may only be billed after their check out.

Figure 4.4 shows a life-cycle representing a business process of business cases of type ROOM-RESERVATION. It has been redesigned for the two-schema architecture. The business process offers additional alternatives to meet the business objectives of the hotel, e.g., it is now possible to charge customers no cancellation fee if they cancel their reservations (cf. activity waivePayment), and allows for more flexibility in billing customers, e.g., it is now possible to bill customers at any time after their reservation has been issued. The business process presented in Figure 4.4 allows, for example, to make all-inclusive offers to be paid before check-in, to bill guests and have them pay the bill at checkout-time, or to send the bill at some time after the checkout. The increased flexibility in handling room reservations, however, makes additional business-process activities necessary. For example, if a customer cancels a reservation after having paid for it, a refund is to be paid to the customer.

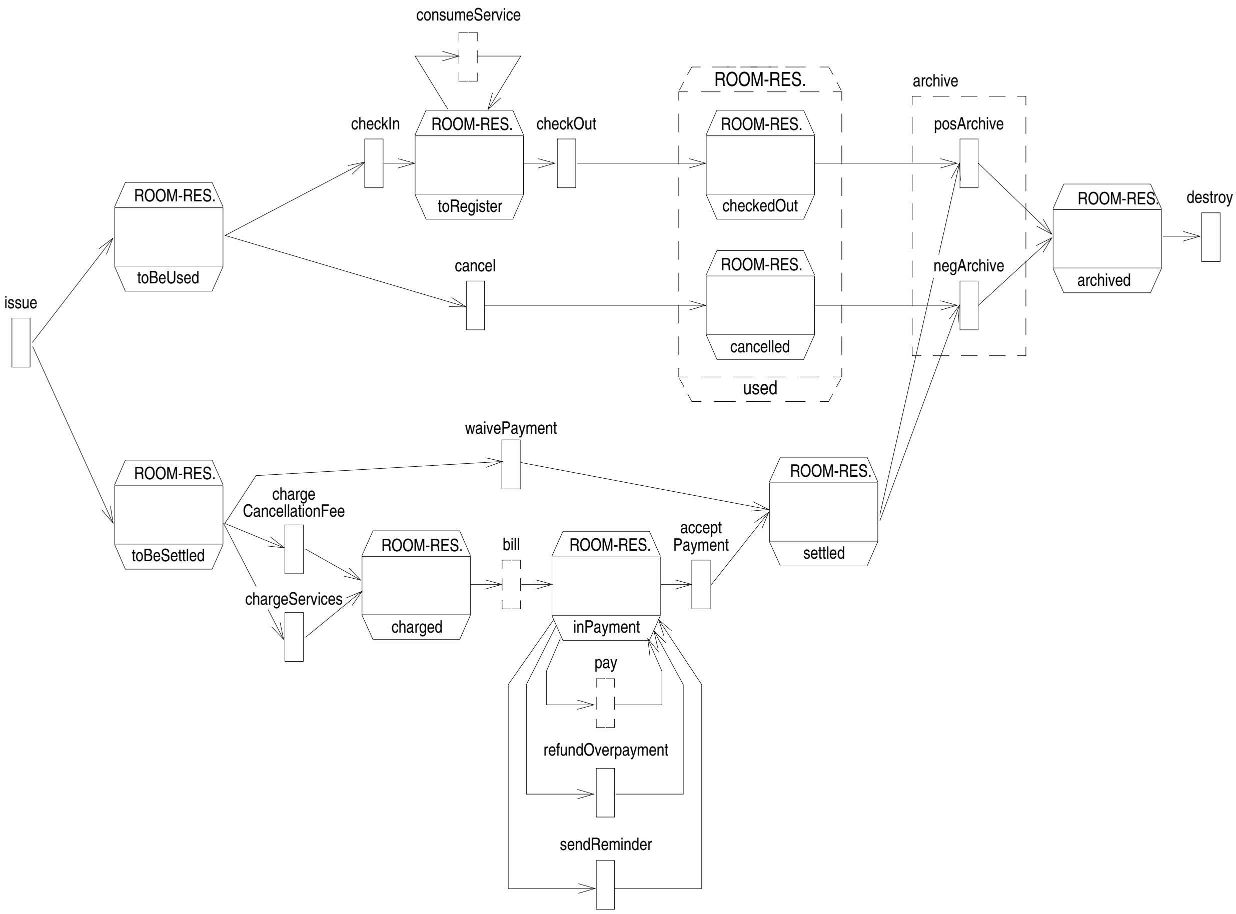


Figure 4.4: Business process at ROOM-RESERVATION

Out-sourcing has been mentioned as one potential offered by a two-schema architecture for workflow systems (cf. Section 4.2). This means, organizations might buy business-process models of their organizational processes off the shelf. To make this vision come true, however, the interoperability of WFMSs must be further increased. One requirement is the development of a common process modeling paradigm. Van der Aalst [Aal96], for example, proposed Petri nets as a standard paradigm for the modeling of processes. He mentioned "three good reasons" for using Petri nets as common process modeling paradigm. First, Petri nets allow to represent the business logic of organizational processes by a formal but also graphical language. Second, Petri nets allow, in contrast to many other modeling paradigms, to model the state of business cases explicitly. "A state-based description allows for a clear distinction between the enabling of a task and the execution of a task." Third, many analysis techniques are available for Petri nets (cf. [Mur89, Pet77]). Note: The principle idea behind life-cycles stems from Petri nets.

An *activity specification* gives formal input and output parameters of business-process activities and defines conditions on their invocation and completion. The specification of input parameters of business-process activities makes interactions with other business objects explicit. Note: We require that a business-process activity may only invoke activities on other objects it can address via its input parameters or via relationships of the object on which the business-process activity is performed. In addition, a business-process activity may create new objects—object types can be addressed via their global names—if it assigns them as values to relationships of the object on which it is performed or if it assigns them to its return port.

Example: Figure 3.4 shows an activity-specification diagram. It represents activity billAtReception of type ROOM-RESERVATION. Its specification must be changed for the two-schema architecture as it considers the internal business rule that the discount granted to a guest must be between zero and *ten* percent. The corresponding external business rule says that in general between zero and *fifteen* percent might be granted. The corresponding specification of business-process activity billAtReception is shown in Figure 4.5. Note: The business rules specifying a constraint on the discount granted to a guest are actually *structural* business rules specifying constraints on attribute discount of room reservations (cf. Figure 3.1). If structural business rules were used to represent the constraint on attribute discount, only the internal business rule could have been considered explicitly in the business-process schema. Mimicking structural business rules by behavioral business rules, however, allows to consider both business rules, the external one (below) as well as the internal one (cf. Subsection 4.4.4).

An *activity realization* is a model of the implementation of a business-process activity. Realization of business-process activities are not discussed in this thesis.

4.3.3 Modeling Classes of Business-Process Objects

A *business-process class* collects a set of business-process objects (or business cases) as its members. This means, business-process classes are like ordinary business-object classes

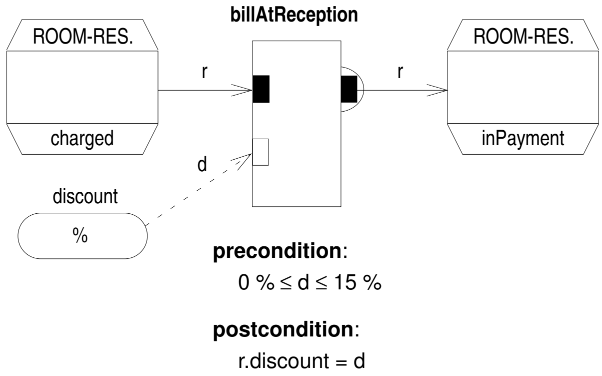


Figure 4.5: Specification of business-process activity billAtReception

(cf. Subsection 3.2.3) that collect instances of a business-process type as their members.

Example: Figure 3.5 shows the specification of class type ROOM-RESERVATIONS. Its instances collect members of business-process type ROOM-RESERVATION (cf. Figure 3.1) and have a derived class-attribute NoOfReservations.

Section 3.2.3 mentions *object warehousing* as one important function of object classes. That means, business-process classes can be used to maintain the extension of business-process types, i.e., to maintain business cases. Since each business case will be member of a base class (cf. Section 3.2.3), each business case can be accessed via some business-process class.

Object classes may be organized into *class hierarchies* (cf. Section 3.2.3) with base classes as their roots. Objects which are characterized by common properties at a higher level of abstraction, i.e., being members of the same superclass, may be distinguished by different properties at a lower level of abstraction, i.e., being members of different subclasses. In the basic model presented in this part of the thesis, all objects that are members of the same class hierarchy are instances of the same object type. Considering business cases this means that all business cases maintained in the same class hierarchy will be handled according to the same business process. One property that distinguishes business cases of the same type into different classes might be the different workflow process according to which they are handled.

Example: Figure 3.6 shows the specification of three business-process classes of type ROOM-RESERVATIONS organized into a class hierarchy. Business-process class Room-Reservations is a base class collecting all instances of business-process type ROOM-RESERVATION as its members. It has two derived subclasses Suite-Reservations and King's-Suite-Reservation. The query diagram showing the derivation expression that determines the extension of business-process class Suite-Reservations is shown in Figure 3.7.

For details on class types, on the specification of object classes, and on the modeling of class-domain constraints on business-process classes see Subsection 3.2.3.

4.4 Design of Workflow-Process Schemas

A *workflow-process schema* consists of a set of workflow-process types, a set of workflow-process classes, and a set of assignment rules. A workflow-process type specifies the structure and behavior of its instances, called workflow-process objects. Each workflow-process object has a designated relationship to a set of business cases that are instances of a specific business-process type. For this type of business-process objects, a workflow-process type specifies a workflow process. This means, in addition to the behavior of its instances, a workflow-process type specifies behavior of the members of its instances, i.e., a workflow process. Assignment rules determine responsible actors for worksteps of workflow processes.

An *instance* of a workflow-process schema consists of a set of workflow-process objects (or *workflow objects* for short). Each workflow object is instance of a particular workflow-process type and is member of one or several workflow-process classes. Workflow objects may be related to and may communicate with other business objects. Especially, each workflow object has a designated relationship to a set of business cases that are handled according to the workflow process it represents in its extended behavior part. Note: Fundamental to a two-schema architecture for workflow systems is that relationships between workflow objects and the business cases they process are uni-directional. More precisely, only workflow objects are explicitly related to the business cases that are handled according to their workflow process, not vice versa. Representing directed relationships between business cases and appropriate workflow objects explicitly would hinder the evolution of business cases to new workflow objects (cf. Section 4.2).

Workflow objects are business objects, too. This means, the concepts presented in Section 3.2 for the conceptual modeling of business objects can be applied in the same way to the modeling of workflow objects. Below, these concepts are shortly reviewed in the realm of conceptual modeling of workflow objects. In addition, we introduce concepts for the conceptual modeling of workflow processes. Note: Since the concepts of the two-schema architecture might be applied in an orthogonal way to workflow objects (cf. Section 4.2), the individual behavior of workflow objects might also be represented in form of a business-process model (cf. Section 4.3) and a workflow-process model (below).

The remainder of this section is organized as follows. Subsection 4.4.1 discusses the modeling of the structure of workflow objects, Subsection 4.4.2 the modeling of the behavior of workflow objects, and Subsection 4.4.3 the modeling of classes of workflow objects. Subsection 4.4.4 presents a conceptual model as well as a graphical language for the design of workflow processes. Subsection 4.4.5 discuss the modeling of assignment rules.

4.4.1 Modeling the Structure of Workflow Objects

The structure of workflow objects is determined by a set of properties, i.e., by a set of *attributes* and a set of *relationships*. Whereas attributes model relevant data of workflow objects, e.g., start of use, relationships model references of workflow objects to other business objects, e.g., to an organizational object representing the agent responsible for a workflow process.

Each workflow object has a designated relationship to a set of business cases of a specific business-process type. Business cases that are handled according to the workflow process defined at the type of a workflow object are called *full members* of the workflow object. Besides being full members of a workflow object, business cases may be related to a workflow object as *affiliate members*. Affiliate members of a workflow object are business cases that are indirectly used in worksteps, e.g., as input parameters. Relationships of workflow objects to full members and affiliate members are modeled by special semantic relationships that are specific for the two-schema architecture. Note: A workflow process defined for a workflow object o and a business process defined for full members of o together specify an organizational process as it is currently performed in a particular organization.

A *has-full-member relationship* models a directed relationship of a workflow object to a set of business cases. Each workflow object must have exactly one has-full-member relationship[5]. A has-full-member relationship must not be a derived relationship, i.e., business cases must be explicitly assigned as full members to the workflow objects that hold a model of the workflow process according to which they are to be processed. Relationships inverse to has-full-member relationships are *full-member-of relationships*. A full-member-of relationship models a directed relationship of a business case to the workflow object that holds a model of the workflow process according to which it is handled. To provide for *workflow transparency* (cf. Section 4.2), full-member-of relationships are not represented explicitly in business-process schemas (cf. Subsection 4.3). This means, business-process schemas should be designed independently from workflow-process schemas embedding them.

A *has-affiliate-member relationship* models a directed relationship of a workflow object to a set of business cases which are handled indirectly by the workflow object, e.g., are considered as input parameters of workflow process activities. More precisely, a workflow-process activity must not directly invoke a business-process activity on affiliate members of the underlying workflow object. Each workflow object may have one or several has-affiliate-member relationships. Business cases may either become affiliate member of workflow objects by explicit assignment or by evaluation of a derivation expression, i.e., has-affiliate-member relationships might be derived, too. Relationships inverse to has-affiliate-member relationships are *affiliate-member-of relationships*. An affiliate-member-of relationships models a directed relationship of a business case to a workflow object where the business case might serve as input parameters of workflow process activities. Like full-member-of relationships, affiliate-member-of relationships are not represented explicitly in business-process schemas (cf. Section 4.3).

Graphically, the structure of instances of a workflow object type is represented in an extended type diagram. Has-full-member relationships and has-affiliate-member relationships are depicted like ordinary relationships (cf. Subsection 3.2.1). To distinguish them from ordinary relationships, however, a special section of the type diagram, called *member section*,

[5]This restriction will be relaxed in the advanced model (cf. Chapter 8). It has been introduced, because this way the presentation of the comprehensive material on the synchronization of worksteps and workflow processes performed on different types of business cases can be postponed to Chapter 8. We hope this contributes to keep the presentation of the material of this chapter clear and concise.

is introduced. The member section of a workflow-object type diagram is separated from the "general" section by a single-line border. Actually a member section is separated into two subsections to further distinguish has-full-member relationships from has-affiliate-member relationships. The corresponding subsections are separated by a dashed-line border and are identified by the annotations full members and affiliate members, respectively.

Example: Figure 4.6 shows a type diagram of workflow-process type ROOM-RESERVATION-WF. Each room-reservation workflow is described by the value of attribute firstUse from domain DATE and by relationships room-wf and responsible to values from domains ROOM-WF and MANAGER, respectively. Relationship room-wf models a reference from a room-reservation workflow to a room workflow. The room-reservation workflow assigns rooms that are maintained by the related room workflow to the room reservations it handles. Note: Maintenance of rooms includes, for example, their servicing. In addition, each room-reservation workflow has a has-full-member relationship room-reservations to values from domain ROOM-RESERVATION (cf. Figure 3.1) and a derived has-affiliate-member relationship rooms to values from domain ROOM. The set of rooms which are affiliate members of a room-reservation workflow is identical to the set of full members of the related room workflow (cf. relationship room-wf), i.e., the derivation expression of the has-affiliate-member relationship rooms is room-wf.rooms, where rooms denotes the has-full-member relationship defined at workflow-process type ROOM-WF.

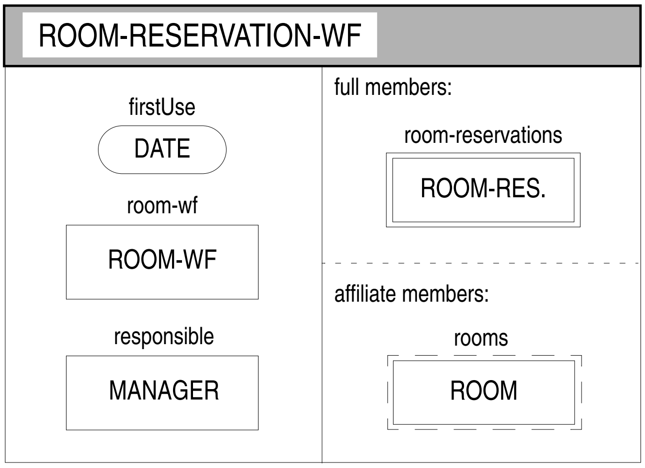


Figure 4.6: Workflow-process type ROOM-RESERVATION-WF

For each has-full-member relationship (has-affiliate-member relationship) defined for a workflow object, the set of business-process activities that may be directly (indirectly) invoked by its worksteps are specified. The set of business-process activities the worksteps of a workflow process may invoke is specified in form of life-cycles that are associated with has-full-member relationships and has-affiliate-member relationships. Actually, one or several life-cycles are associated with every has-full-member relationship and with every has-affiliate-member relationship. Such a life-cycle must be a consistent sub life-cycle (below) of the life-cycle specified at the domain of a has-full-member relationship or a has-affiliate-member relationship.

Comment 4.1 (Consistent Sub Life-Cycles)
A life-cycle $l' = (S', T', F', A', \Omega')$ is a consistent sub life-cycle of a life-cycle $l = (S, T, F, A, \Omega)$, where S' and S are sets of states, T' and T are sets of activities, $F' \subseteq ((S' \times T') \cup (T' \times S'))$ and $F \subseteq ((S \times T) \cup (T \times S))$ are sets of arcs, $A' \subseteq S'$ and $A \subseteq S$ are sets of initial states, and $\Omega' \subseteq S'$ and $\Omega \subseteq S$ are sets of final states, if $S' \subseteq S$, $T' \subseteq T$, $F' \subseteq F$, and if all prestates and poststates defined for an activity $t \in T'$ at a super life-cycle are prestates and poststates of t at a sub life-cycle, too, i.e., $\forall t \in T' \colon ((s,t) \in F \Rightarrow s \in S' \wedge (s,t) \in F') \wedge ((t,s) \in F \Rightarrow s \in S' \wedge (t,s) \in F')$. A general requirement for the consistency of a life-cycle $l = (S, T, F, A, \Omega)$ is that for each state $s \in S$ there must be at least one activity $t \in T$ that consumes from (produces into) s, unless s is a final state (an initial state), i.e., $\forall s \in S \colon (\exists t \in T \colon (s,t) \in F \vee s \in \Omega) \wedge (\exists t \in T \colon (t,s) \in F \vee s \in A)$. In addition, we require for sub life-cycles that each activity $t \in T'$ that consumes from (produces into) a state $s \in S'$ in a sub life-cycle must consume from (produce into) s in the according super life-cycle, too, i.e., $\forall t \in T' \colon (\forall s \in S' \colon (s,t) \in F' \Rightarrow (s,t) \in F)$ $\wedge (\forall s \in S' \colon (t,s) \in F' \Rightarrow (t,s) \in F)$. Informally, a life-cycle is a consistent sub life-cycle of another life-cycle, called super life-cycle, if it considers all "parallel branches" and at least one out of a set of "alternative branches" offered in that part of the super life-cycle it covers.

Note: Has-full-member relationships as well as has-affiliate-member relationships are a special kind of *uses*-relationships. They determine which part(s) of an organizational process a workflow process may cover. For example, this information could be used to easily test whether a business case assigned as full member to a specific workflow object is already member of another workflow object that covers the same or overlapping part(s) of the underlying business process.

To ensure consistency of workflow process execution, we require that a business case may not be member of different workflow objects at a time that cover common parts of the underlying business process (below). Graphically, a sub life-cycle specified for a has-full-member relationship or a has-affiliate-member relationship is shown in a separate life-cycle diagram (cf. Subsection 3.2.2). However, such a life-cycle diagram is annotated by the name of the type at which the relationship has been defined, the name of the relationship, and the keyword workflow for.

Example: Figure 4.7 shows the life-cycle associated with has-affiliate-member relationship room defined at workflow-process type ROOM-RESERVATION-WF. It is a sub life-cycle of the life-cycle defined for rooms (cf. Figure 8.4). It says that only activities occupy and free may be indirectly invoked by worksteps defined for instances of ROOM-RESERVATION-WF.

Relationships between workflow objects model a network of workflow processes an organization carries out to achieve its business objectives. As discussed in Subsection 3.2.1, we distinguish three broad types of organizational processes (cf. [Oul95]): core processes, support processes, and management processes. In the context of a two-schema architecture for workflow systems (cf. Section 4.2), *relationships between organizational processes might be considered at two distinct levels bearing different semantics.* If a relationship between

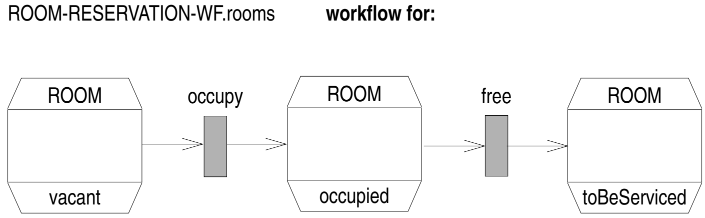


Figure 4.7: Sub life-cycle of ROOM

two types of organizational processes is represented in a business-process model, each business case of one type holds an individual relationship to one business case or to a set of business cases of the other type. If a relationship is represented in a workflow-process model, however, all business cases handled by an instance of the one type of the organizational process are related to all business cases handled by one instance or by a set of instances of the other type of organizational process. Now, if a relationship is represented in both models, in the business-process model as well as in the workflow-process model, the individual relationship between business cases will usually be determined during performing the workflow process. The domain of the individual relationship, however, will be restricted to the business cases treated by the corresponding workflow-process object.

Example: Relationship room-wf models a relationship of a core processes of workflow-process type ROOM-RESERVATION-WF to a support process of workflow-process type ROOM-WF. The relationship between the organizational processes of the hotel to handle room reservations and to maintain the rooms is represented in both models, in the business-process model (cf. Figure 3.1) and in the workflow-process model (cf. Figure 4.6). Actually, only the desired room category is determined at the time a room reservation is issued for a certain customer. A room of the appropriate category is selected at the time a guest checks in (see below) from the set of rooms handled by the related room workflow.

4.4.2 Modeling the Behavior of Workflow Objects

The behavior of workflow objects is determined by a set of *activities*, by *allowed execution orders* on these activities, and by their *interactions* with other objects. In other words, the behavior of workflow objects is modeled in the same way as the behavior of ordinary business objects (cf. Subsection 3.2.2).

Example: Figure 4.8 shows the life-cycle of workflow objects of type ROOM-RESERVATION-WF. After a room-reservation workflow object has been created, it resides in state suspended and must be activated explicitly. If it resides in state active, room reservations may be added to and removed from a room-reservation workflow object. Invoking activity suspend transfers a room-reservation workflow object again into state suspended. Before a room-reservation workflow object may be archived, it must be deactivated explicitly. Finally, a room-reservation workflow object may be destroyed.

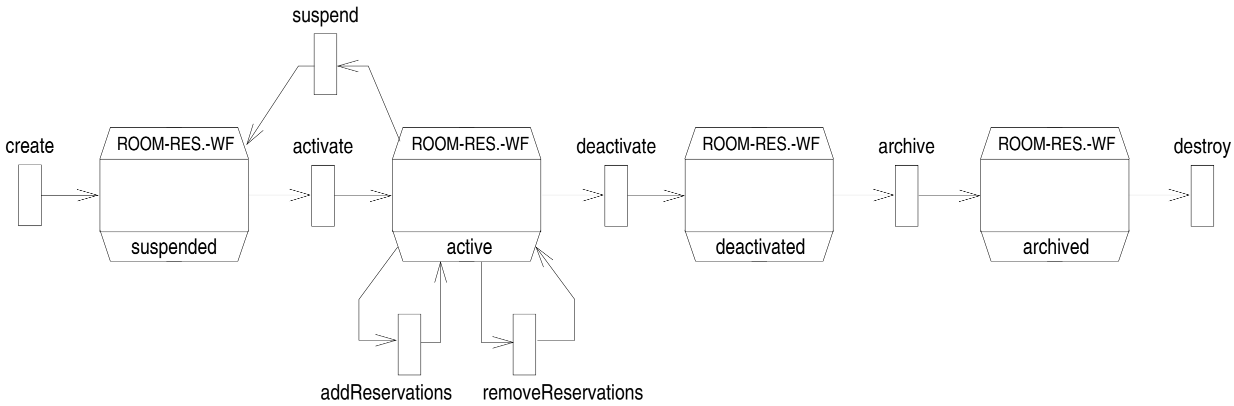


Figure 4.8: Life-cycle at ROOM-RESERVATION-WF

Note: In Part III a conceptual-type hierarchy will be presented. It offers a set of predefined conceptual types. A conceptual-type hierarchy allows a designer to build a conceptual schema of some business area in that she or he specializes, i.e., refines or extends, predefined conceptual types. One of the advantages of such a conceptual-type hierarchy is that user-defined types share common structure and behavior in a uniform way. Amongst other conceptual types, the conceptual-type hierarchy presented in Part III will offer a workflow-process type that specifies structure and behavior common to all user-defined workflow-process types.

If the behavior of a workflow object itself is a model of an organizational process that indirectly adds value to an external customer of an organization, e.g., a support process or a management process, the behavior of the workflow object might be designed for the two-schema architecture, too (cf. Section 4.2). In this case, the workflow object is itself considered as (full) member of another workflow object that defines the workflow process according to which the former is handled.

4.4.3 Modeling Classes of Workflow Objects

A *workflow-process class* collects a set of workflow objects as its members. Like with an ordinary business-object class (cf. Subsection 3.2.3), individual structure and behavior of its members are specified at the member type, a *workflow-process type*, of a workflow-process class, and collective structure and behavior of its members are specified at the class type of the workflow-process class.

4.4.4 Modeling of Workflow Processes

A *workflow process* is a model of an organizational process as it is *currently* performed in a *particular* organization. In a two-schema architecture an organizational process is not represented by a workflow-process model alone but its representation is split into two models. A workflow-process model considers both knowledge on an organizational process expressed in form of external business rules as well as knowledge expressed in form of internal business rules. However, a workflow-process model does not represent knowledge expressed in form of external business rules explicitly. Rather it delegates its representation to a business-process model of the organizational process (cf. Section 3.1).

Actually a workflow-process model of an organizational process is represented at different levels of detail. Abstracting from complete specifications, a high-level *workflow process* identifies worksteps and containers of business cases, indicating the actual flow of work. It is depicted in a *workflow-process diagram*. A *box specification* defines a container of business cases, which are full or affiliate members of a workflow-process object. It is depicted either in a workflow-process diagram, in a workstep-specification diagram (below), or in a separate *box-specification diagram*. A *workstep specification* defines input and output parameters as well as the set of business-process activities a workstep may invoke. It is depicted in a *workstep-specification diagram*. An *workstep realization* represents the implementation of a workstep. It is depicted in a *workstep-realization diagram*. This means, a workflow process can be fully depicted using one workflow-process diagram, several box-specification diagrams, several workstep-specification diagrams, and several workstep-realization diagrams. Together these diagrams are called the *workflow diagram* of a specific workflow-process type.

Workflow Process

A *workflow process* models an organizational process as it is *currently* performed in a *particular* organization using a set of *boxes*, a set of *workflow process activities* (or *worksteps* for short), and a set of directed *arcs* connecting workflow process activities to boxes and vice versa. As a notational convention, we start names of boxes and worksteps with a capital letter. Boxes serve as containers of business cases that are assigned as members to occurrences of the workflow process, and worksteps change the state of these business cases by invoking business-process activities and performing logically related external activities (such as collecting cash etc.).

The principle idea behind workflow processes stems from Petri nets (cf. [Mur89, Pet77]), similar as the principle idea behind business processes (or life-cycles in general (cf. Subsection 3.2.2)). Boxes correspond to places of Petri nets, worksteps to transitions. Like transitions are connected to places, worksteps are connected by a set of directed arcs to boxes. Boxes connected to a workstep with incoming arcs collect business cases on which the workstep may be performed. Immediately after the execution of a workstep on a particular business case completes, the business case must qualify for every box connected to that workstep with an outgoing arcs. Boxes represent necessary pre- and postconditions, respectively, of worksteps. A box may be connected to a workstep either by a *transfer arc* or a *read arc*. If it is connected to a workstep by a transfer arc and the workstep is performed on some business case, the workstep must change the business case (by invoking some business-process activity) such that it no longer qualifies for this box. If it is connected by a read arc, however, the workstep must not modify the business case such that it no longer qualifies for the box during or immediately after the execution of the workstep. Note: Outgoing arcs of a workstep may only be transfer arcs. Under which conditions a workstep may be invoked and complete will be discussed subsequently (cf. workstep specification, below).

A *workflow process* is a model of an organizational process that is built from a business-process model of the organizational process (cf. Sections 4.2 and 4.3). As a consequence, we define the consistency of a workflow process with respect to the underlying business process. For the sake of simplicity of the following consistency criteria, we assume that (a) a workflow process has only one type of members, i.e., only one has-full-member relationship is defined for a workflow object, (b) a separate workstep is defined for each business-process activity considered in a workflow process, and (c) the "underlying business process" is defined by the life-cycle given as constraint on the has-full-member relationship of a workflow object.

A workflow process has been defined consistently with respect to the underlying business process if it satisfies the following conditions (*Rule of Business-Process Consistency of Workflow Processes*): (1) It consists of at least one workstep w_i and of at least one box b_j that is connected to w_i either by an incoming arc or an outgoing arc. (2) Each workstep w_i defined for a business-process activity t_j is connected by incoming arcs (outgoing arcs) to a set of boxes that collect business cases residing in prestates (poststates) of t_j. More precisely, workstep w_i must be connected to a separate box for each prestate and for each poststate of t_j. Note: Additional boxes may be connected to a workstep by incoming read arcs. Thereby, allowed execution orders of activities of the business process underlying a workflow process may be further restricted. (3) Each box b_m collecting members of a workflow process that reside in a life-cycle state s_n has been connected by incoming arcs (outgoing arcs) to a workstep w_i, unless b_m collects members residing in a final state (an initial state). Informally, a workflow process is consistent wrt. the underlying business process, if it allows to handle business cases passed to it for processing such that they finally reside in final states of the underlying business process. This means, a workflow process must offer one workstep for each business-process activity t of the underlying business process, unless there has already been defined a workstep for an activity t' that is alternative to t.

A business case is treated according to a certain workflow process if it has been assigned to it as full member. A business case may be assigned to only one workflow process of the same type at a time. However, it may be assigned to workflow processes of different types as full member at the same time if the corresponding workflow processes cover different parts of the underlying business process, i.e., if the overall workflow-process model of an organizational process is split into several workflow-process models.

Graphically, a workflow process is depicted in a *workflow-process diagram*, which, similar as a life-cycle diagram provides an overall view of a business process, provides an overall view of a workflow process. Actually, a workflow-process diagram shows how business cases assigned as full members to a workflow process of some type flow between its boxes and its worksteps. A box is depicted by a rectangle with a double-line border. Its name is shown at the top of the rectangle, the name of the type of the business cases it collects as its members as well as the state in which these business cases must reside inside the rectangle, and the condition that members of a box must fulfill below the rectangle. A workstep is depicted by a rectangle showing its name above. The primary port of a workstep is indicated by black squares. A square representing a primary input port is aligned to the left inner border of the rectangle, a square representing a primary output port to the right outer border. Transfer arcs are depicted by solid lines, read arcs by dashed lines.

Example: Figure 4.9 shows the specification of a workflow process that handles room reservations (cf. Figures 3.1 and 3.2). It is defined at workflow-process type ROOM-RESERVATION-WF.

Once a room reservation has been issued, a customer may either cancel the reservation or check in at the hotel. Customers that cancel their reservation will be charged a cancellation fee (cf. workstep ChargeCancellationFee&Bill). After a customer did pay the cancellation fee, a reservation will be archived (cf. workstep NegArchive). Note: The refinement of box R-7 (cf. Figure 4.11), shows the workflow process that deals with the payment of both, cancellation fees and regular bills. The payment of a customer will be finished by invoking workstep AcceptPayment.

If a customer checks in, the customer can consume several services offered by a hotel during her or his stay. Usually, customers have to pay immediately at check-out (cf. workstep CheckOut_Charge_Bill&Pay). Thereafter a reservation will be immediately archived (cf. workstep PosArchive). If a customer has a credit rating higher than three, she or he may checkout without paying immediately (cf. incoming read arc connecting workstep CheckOut with box R-3; for the moment ignore the line connecting boxes R-2 and R-3). If a customer did not pay at check-out, she or he will receive a bill by letter or fax (cf. workstep ChargeServices&Bill), and the reservation will be archived (cf. workstep PosArchive) after their payment has been accepted (above).

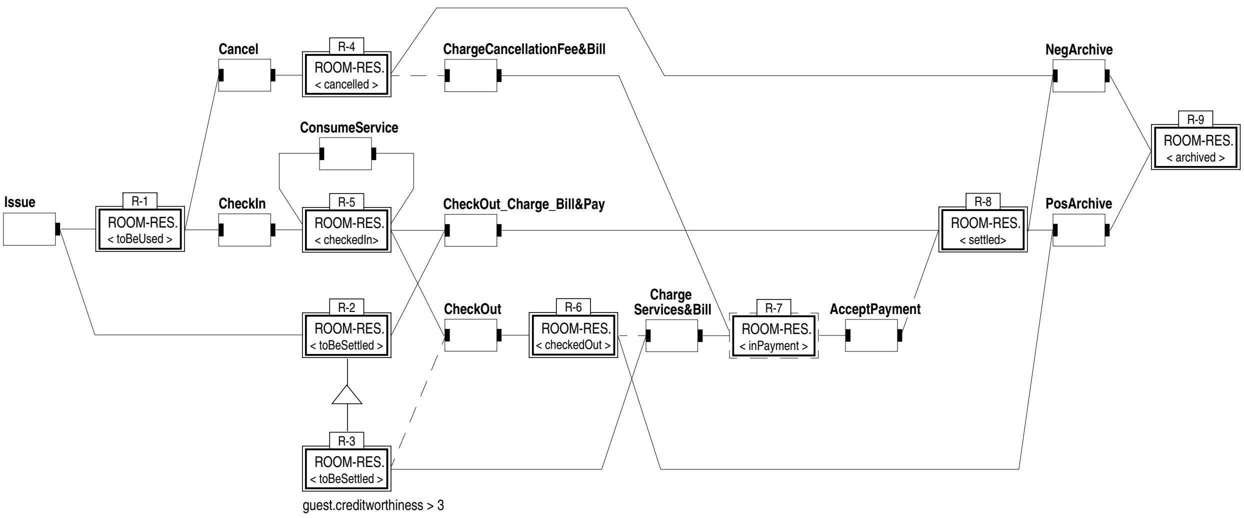


Figure 4.9: Workflow process at ROOM-RESERVATION-WF

A workflow process may handle business cases in a way that is more restrictive than their business process requires. For example, a workflow process might only support a selected subset of the alternatives offered by the business process to achieve the purpose of the represented organizational process (Note: A choice between alternatives may be even omitted at all by supporting only one alternative), it might restrict possible execution orders of process activities by enforcing additional sequentialization of business-process activities, and it might restrict the invocation context in time and space by invoking two or more business-process activities in a single workstep. Furthermore, different worksteps may invoke the same business-process activity. Usually, such worksteps will have different realizations or are connected to different boxes.

Example: According to business-process type ROOM-RESERVATION a guest may check out from a hotel irrespective of whether he has already paid or not (cf. Figure 3.2). Workflow process type ROOM-RESERVATION-WF requires that customers with a low credit rating are billed and have to pay immediately when they check out from the hotel (cf. Figure 4.9); thus workstep CheckOut_Charge_Bill&Pay (which invokes business-process activities checkOut, chargeServices, billAtReception, payCash or payByCreditCard, and thereafter acceptPayment) must be used. Workstep CheckOut (which invokes business-process activity checkOut) may be performed only on reservations of guests with a credit rating greater than three (cf. the read arc to box R-3). Notice that the workflow-process diagram does not show which business-process activities a workstep invokes. This information is depicted in a workstep-realization diagram (below).

According to business-process type ROOM-RESERVATION, one might choose between charging a cancellation fee for cancelled and yet unbilled reservations or not charging a cancellation fee (cf. business-process activity waiveSettlement in Figure 3.2). Workflow process type ROOM-RESERVATION-WF omits this alternative. Only workstep Charge-CancellationFee&Bill (which invokes business-process activities chargeCancellationFee and thereafter billByLetter or billByFax) may be performed on cancelled and yet uncharged reservations.

A workflow process can be developed by stepwise refinements of boxes and worksteps, similar as a life-cycle can be developed by stepwise refinements of states and activities (cf. Subsection 3.2.2). Since a workflow process need neither create nor destroy business cases, it may be developed by stepwise refinements starting from one box and optionally one create workstep or one destroy workstep, or both. Boxes and worksteps that have been refined are called *abstract*, those that have not *basic*. Each workstep and each box can be refined into a workflow process. To ensure consistent refinements of worksteps and boxes, we require that sources and sinks of their refinements must also be worksteps and boxes, respectively, and that each refinement must have at least one source and one sink. If a box or a workstep is refined into a workflow process, the underlying business-process model of the organizational process modeled must be considered. A workstep *must be* refined if alternative boxes are connected by incoming arcs to it. A box *must be* refined if the member state it defines is an abstract state. The result of a stepwise refinement process of worksteps and boxes of a workflow process must be a set of basic worksteps and a set of basic boxes, where each basic workstep is only connected to basic boxes and vice versa.

Note: Although the underlying business-process model of an organizational process must be considered, there need not be and may not be a one to one correspondence between the refinement steps of a workflow process and the refinement steps of the underlying business process.

Based on the idea of structured refinement primitives for life-cycles (cf. Subsection 3.2.2), we propose a set of structured refinement primitives for worksteps and boxes of workflow processes that ensure that a business case leaves the refinement of a workstep or box always entirely. Structured refinement of workflow processes allows only sources and sinks of a refinement be connected by transfer arcs to elements outside the refinement. Elements in a refinement that are neither sources nor sinks may be connected to elements outside the refinement only by read arcs. The refinement primitives allow for a sequential, parallel, alternative, or iterative refinement of worksteps and boxes (cf. Figure 4.10). Note: Although every workflow process given as an example in this thesis has been developed by applying structured refinement primitives, we do not require that workflow processes are necessarily developed by structured refinement primitives.

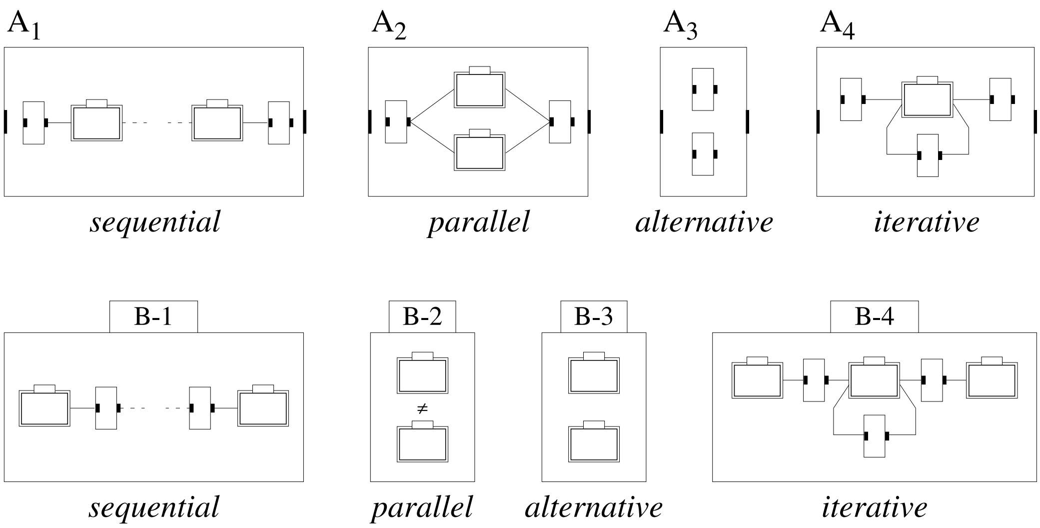


Figure 4.10: Refinement primitives for worksteps (top) and boxes (bottom)

In a workflow-process diagram, abstract boxes and abstract worksteps are shown with a dashed-line border. Alternatively, the refinement of a box or a workstep may be expanded in the original workflow-process diagram or in a separate one. In the former case, the refined box or workstep may be superimposed to its refinement.

Example: Figure 4.11 shows a stepwise refinement of box R-7 defined at workflow-process type ROOM-RESERVATION-WF. The workflow process in the refinement of box R-7 deals with payment of bills, with sending of reminders, and with refunding of overpayments. After a bill has been sent to a customer (cf. worksteps ChargeCancellationFee&Bill and ChargeServices&Bill), the customer's reservation qualifies for box R-711, which collects reservations with a negative balance, until the end of the agreed settlement day.

If a customer pays in time, i.e., on the settlement day at the very latest, she or he may pay by money transfer or by credit card (cf. workstep PayByMoneyTransferOrByCreditCard). Thereafter, a reservation qualifies either for box R-721, if the customer did clear the balance, or for box R-722, if the customer did overpay. If a customer did overpay, the overpayment will be refunded (cf. workstep RefundOverpayment). After refunding an overpayment, a reservation must qualify for box R-721.

If a customer did not pay in time, i.e., the settlement day of a reservation with a negative balance passed by, either a collection agency will be engaged to collect the outstanding amount or another reminder will be sent, depending on the number of reminders that have already been sent for a reservation. If already three reminders have been sent, a reservation qualifies for box R-7121 meaning that a collection agency will be engaged (cf. workstep PayByCollectionAgency; for the moment, ignore the shading and the double-line border of the graphical symbol representing this workstep). Otherwise a reservation qualifies for box R-7122 meaning that another reminder will be sent (cf. workstep SendReminder). (Note: If a reminder is sent, a new settlement day will be fixed. Consequently, a reservation will again qualify for box R-711 after a reminder has been sent.) Then, a reservation will qualify for box R-711 until the customer decides to clear the outstanding amount (cf. workstep PayByMoneyTransferOrByCreditCard) or the agreed settlement day again passes by. At the day after the settlement day, a reservation with a negative balance flows from box R-711 either to box R-7121 or to box R-7122. This means, reservations can move between boxes although no workstep is executed on them. Instead, an *event* may cause a move of reservations between boxes if a certain *condition* is satisfied. Such moves of business cases between boxes are specified by a special kind of arcs, the *transition arcs*, which are explained below. Figure 4.11 shows two transition arcs labeled with the event newday.

If the balance of a reservation has been cleared, it qualifies for box R-721. Then workstep AcceptPayment can be invoked to finish payment.

Business cases may move between boxes of a workflow process although no workstep is executed on them. Instead, an event might cause the transition if a certain condition holds. In such cases, two boxes are connected by a so-called *transition arc*. A transition arc is annotated with a textual specification of the event that might fire the transition, the condition, however, is not specified explicitly. The condition that causes a transition to occur can be inferred from the specifications of the boxes at the tail and the head of the transition arc. In a workflow-process diagram, transition arcs are depicted by arrows shown by a solid line. The event associated with a transition arc is shown as textual annotation.

Example: At the day after the settlement day, reservations that have not been cleared so far flow from box R-711 either to box R-7121 or to box R-7122, which is indicated by transition arcs (cf. Figure 4.11). The transition arcs are labeled with the event newday that may cause the transition of reservations that are members of box

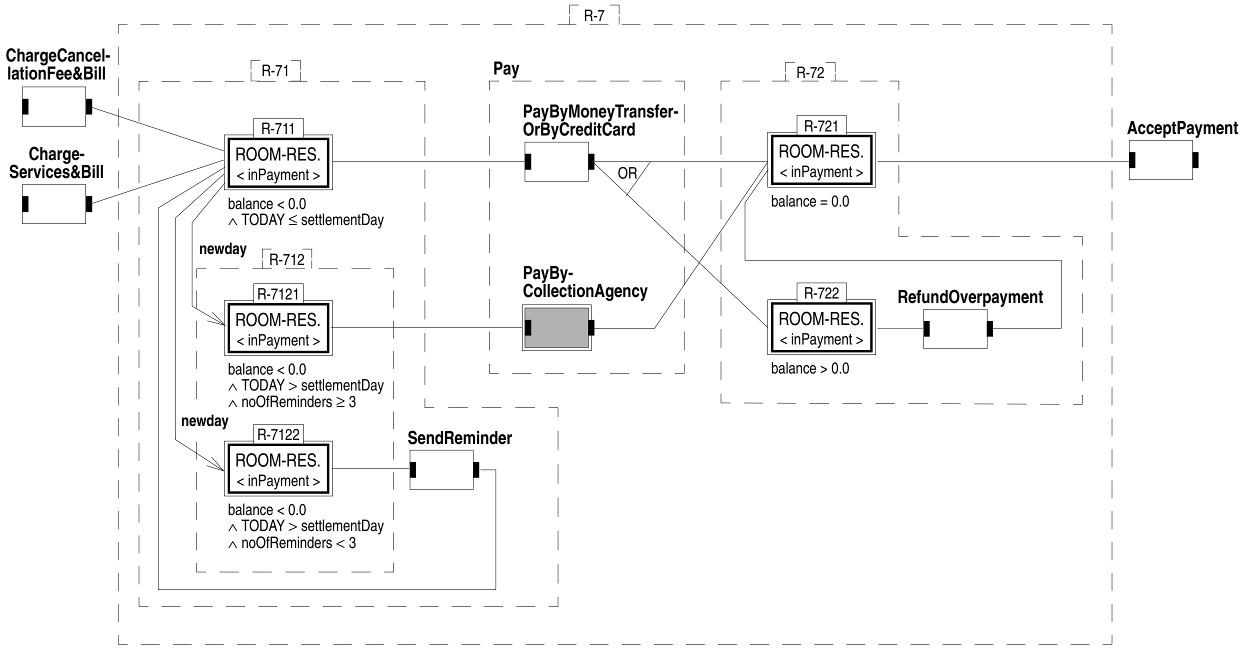


Figure 4.11: Refinement of box R-7

R-711. A reservation in box R-711 flows to box R-7121 if three reminders have already been sent. In this case, the condition inferred from the participating boxes is: settlementDay $<$ TODAY $\wedge$ noOfReminders $\geq$ 3. A reservation in box R-711 flows to box R-7122 if less than three reminders have been sent so far. In this case, the condition inferred from the participating boxes is: settlementDay $<$ TODAY $\wedge$ noOfReminders $<$ 3.

Box Specification

A *box specification* defines a container of business cases handled by a workflow process. Such containers, called boxes, are properties of workflow-process objects. A box is specified by a *name*, a *member type*, a *member state*, and a boolean *condition*. The name must be unique within a workflow-process type, the member type specifies the type of business cases a box may collect. It must correspond to the type domain of a has-full-member relationship of the workflow-process type if the box is connected to a primary port of a workstep; it must correspond to the type domain of a has-affiliate-member relationship of the workflow-process type otherwise. This means, boxes serve as containers of business cases that have been assigned as (full or affiliate) members to an occurrence of the workflow process of which the box is a property. The member state must be a life-cycle state defined at the object type given as member type of the box. It may be either a basic or an abstract state. A business case from the given member type belongs to a box of a workflow process if it is a member of the workflow process, if it resides in the given member state, and if it satisfies the given boolean condition. Notice that business cases are neither inserted explicitly into boxes nor are they removed explicitly from boxes. Box membership solely depends on the current life-cycle state, the current property values, and, possibly, the history of a business case. If a business case is assigned to a workflow process as a new member, the business case becomes a member of all boxes for which it currently qualifies.

Graphically, a box specification is shown in a workflow-process diagram, in a workstep-specification diagram (below), or in a separate box-specification diagram. Box-specifications are depicted in the same way in all kinds of diagrams (cf. graphical representation of workflow process, above).

Example: The member type of all boxes specified at workflow-process type ROOM-RESERVATION-WF (cf. Figure 4.9) is object type ROOM-RESERVATION (cf. Figure 3.1 and Figure 3.2). Each room reservation that is full member of a particular room-reservation workflow-process instance w qualifies for box R-2 of w if it resides in state toBeSettled.

Boxes may be organized into *specialization hierarchies* that are modeled by subbox-superbox relationships. A subbox must always contain a subset of the business cases that belong to its superbox. This is guaranteed in that a subbox “inherits” the member type, the member state, and the boolean condition from its superbox. A subbox may “specialize” its superbox by restricting the member type to a subtype of the inherited one, by restricting the member state to a state in the refinement of the inherited one if the inherited member state is an abstract state, and by adding additional constraints to the inherited boolean condition by means of conjunction. Subboxes of the same specialization are always *disjoint*. A box may, however, have different specializations. Graphically, a subbox-superbox relationship is depicted like a subtype-supertype relationship between object types (cf. Subsection 7.2.1. However, the graphical presentation of subbox-superbox relationships that share the same superbox coincide only if they belong to the same specialization of a superbox. Note: Showing inherited conditions of subboxes is optional.

Example: Box R-2 of a room-reservations workflow process (cf. Figure 4.9) contains all room reservations which reside in state toBeSettled (above). Subbox R-3 holds all room reservations which reside in box R-2 and whose guests have a credit rating greater than three, i.e., the value of attribute creditworthiness of the corresponding guest object is greater than three.

Workstep Specification

A *workstep specification* models the interface of a workstep by a *name*, a set of *input ports*, a set of *output ports*, a *precondition*, a *postcondition*, and a set of *life-cycles* that determine the set of business-process activities a workstep may invoke together with their prestates and poststates. A life-cycle that determines which business-process activities a workstep may directly or indirectly invoke on business cases assigned to its input ports must be a legal sub life-cycle (above) of the life-cycle specified as constraint on the corresponding has-full-member relationship or has-affiliate-member relationship of a workflow process (cf. Subsection 4.4.1). Different from business-process activities, postconditions of worksteps may be disjunctive.

Comment 4.2 (Behavior Composition with Worksteps)
Kappel and Schrefl (cf. [Kap91, KS91, Sch91, KS96b]) identify three phases of modeling the behavior of objects. Using *behavior refinement*, the life-cycle of objects of a particular type consisting of basic activities and basic states is developed by stepwise refinements of abstract activities and abstract states. Using *behavior composition*, basic activities are composed into more complex activities, called compound activities. A compound activity invokes basic activities or other compound activities. Using *behavior concretization*, a realization of activities is offered.

Modeling of worksteps is a form of *behavior composition*. Basic and compound business-process activities are invoked within a workstep. Unlike compound business-process activities, however, worksteps allow for interactive execution by an actor. Like compound business-process activities, *a workstep determines the temporal and organizational context(s) in which business-process activities may be invoked*. Since a workstep may invoke business-process activities on different objects of the same or of different types, it is a means to synchronize the behavior of objects. As a guideline, we require that worksteps may not introduce new synchronization points between the behaviors of different objects of the same type or of different types. This means, if activity t_1 defined at type O_1 and activity t_2 defined at type O_2, where O_1 may be identical to O_2, are grouped together into the same workstep s, then either a prestate or a poststate of t_1 is a prestate or a poststate of a secondary port of t_2, a prestate or a poststate of t_2 is a prestate or a poststate of a secondary port of t_1, there is an activity t_3 defined at type O_1 such that a prestate or a poststate of t_3 is a prestate or a poststate of a secondary port of t_2 and t_3 is a predecessor or a successor of t_1, or there is an activity t_4 defined at type O_2 such that a prestate or a poststate of t_4 is a prestate or a poststate of a secondary port of t_1 and t_4 is a predecessor or a successor of t_2.

Input and output ports represent input and output parameters of a workstep. Like ports of business-process activities, every port has a *name*, a *domain*, and is either *single-valued* or *multi-valued*. The names of ports must be unique within a workstep. Each port is either an input port, an output port, or both. The exact number of values to be assigned to (to be returned by) a port of a workstep is specified by the port's *cardinality*.

States connected to an input port p are called *prestates* of p, states connected to an output port *poststates*. Separate sets of prestates and poststates are connected to ports that are both input port as well as output port.

Ports either represent formal input and output data, i.e., some kind of internal data modeled by business objects, or informal input and output data, i.e., external data like a credit card. The *type domain* of a formal port is specified by a business-process type. The type domain is further restricted to a set of states in the life-cycle of the business-process type specified as domain of the port. States connected to an input port p are called *prestates* of p, states connected to an output port *poststates*. Separate sets of prestates and poststates are connected to ports that are both input port as well as output port. The *active domain* (cf. [HK87]) of a formal port is specified by a set of boxes. Separate sets of boxes are connected to a port that is both input port as well as output port. The member type of a box b connected to a port p must correspond to the type domain of p. The member state of b must correspond to a prestate of p if b is connected to p by an incoming arc, it must correspond to a poststate of p if b is connected to p by an outgoing arc. The type domain of a formal port may be specified indirectly by specifying its active domain, too. The domain of an informal port is specified by an informal data description. Note: A formal port is connected to states restricting the type domain of the port either by transfer arcs or read arcs. If a formal port is connected to a life-cycle state (type domain) by a transfer arc (read arc), then it must be connected to the box (active domain) that collects objects which reside in this life-cycle state by a transfer arc (read arc), too. Informal data ports are connected to their domains by transfer arcs. Note: Specifying a box as active domain of a port of a workstep is just a graphical specification of a precondition or postcondition of that workstep. Disjunctive boxes connected to an output port of a workstep are annotated by the keyword OR in workflow-process diagrams as well as in workstep-specification diagrams (below).

Every workstep has a distinguished mandatory single-valued formal port, the primary port. Boxes connected to the primary input port of a workstep must correspond to those boxes connected to a workstep by incoming arcs in a workflow process (above), boxes connected to the primary output port to those connected to a workstep by outgoing arcs. Boxes connected to a workstep by a transfer arc (read ard) in a workflow process, must be connected to the primary port of a workstep by a transfer arc (read arc) in a workstep specification, too.

Graphically, a workstep specification is depicted in a *workstep-specification diagram*. It depicts a workstep by a rectangle showing the name of the workstep above and a square for each port of the workstep. Single-valued ports are depicted by squares with a single-line border, multi-valued ports by squares with a double-line border. Squares representing input ports are aligned to the left inner border of the rectangle, squares representing output ports to the right outer border. Consequently, a port which is both, an input port and an output port, is represented by two squares, which are aligned to the same height. The cardinality of a port is depicted in square brackets inside the rectangle representing a workstep aligned to the same height as the square(s) representing the port. Note: Usually, default cardinalities of ports are not shown. The primary port of a workstep is depicted in black.

The domain of an informal data port is depicted by a rounded rectangle showing the informal data description below the rectangle. The type domain of a formal port is specified by a set of life-cycle states, which are depicted as in life-cycle diagrams (cf. Subsection 3.2.2), its active domain by a set of boxes, which are depicted as in workflow-process diagrams (above). A transfer arc connecting a port to its domain is depicted by a solid line, a read arc by a dashed line. Pre- and postcondition of a workstep that are specified textually are shown as annotations below the rectangle representing the workstep. The keywords precondition and postcondition are used to distinguish pre- and postcondition of a workstep. Below, the business-process activities a workstep may invoke are shown together with their pre- and poststates. Their specification is marked with the keyword workstep for.

Example: Figure 4.12 shows a workstep-specification diagram for workstep CheckOut-Charge_Bill&Pay. The boxes connected to the primary port of the workstep correspond to those already depicted in the workflow-process diagram (cf. Figure 4.9). That means, the type domain of the primary port is given by business-process type ROOM-RESERVATION. The active domain of the primary input port is given by boxes R-2 and R-5, the active domain of the primary output port by boxes R-6 and R-8. The secondary input ports indicate that, besides a reservation object, the workstep requires a key card of the guest and a means of payment. The secondary output ports indicate that the workstep returns a means of payment, i.e., the change, together with a receipt. That is, all secondary ports of workstep CheckOut_Charge_Bill&Pay represent informal data. The textual precondition says that the workstep may only be invoked on the last day of a guest's stay. The specification of the business-process activities the workstep may invoke indicates that not all forms of payment supported by a business process of type ROOM-RESERVATION (cf. Figures 3.2 and 3.3) can be used in workstep CheckOut_Charge_Bill&Pay. Only payments by cash or by credit card are allowed. Furthermore, a guest must be billed at the reception.

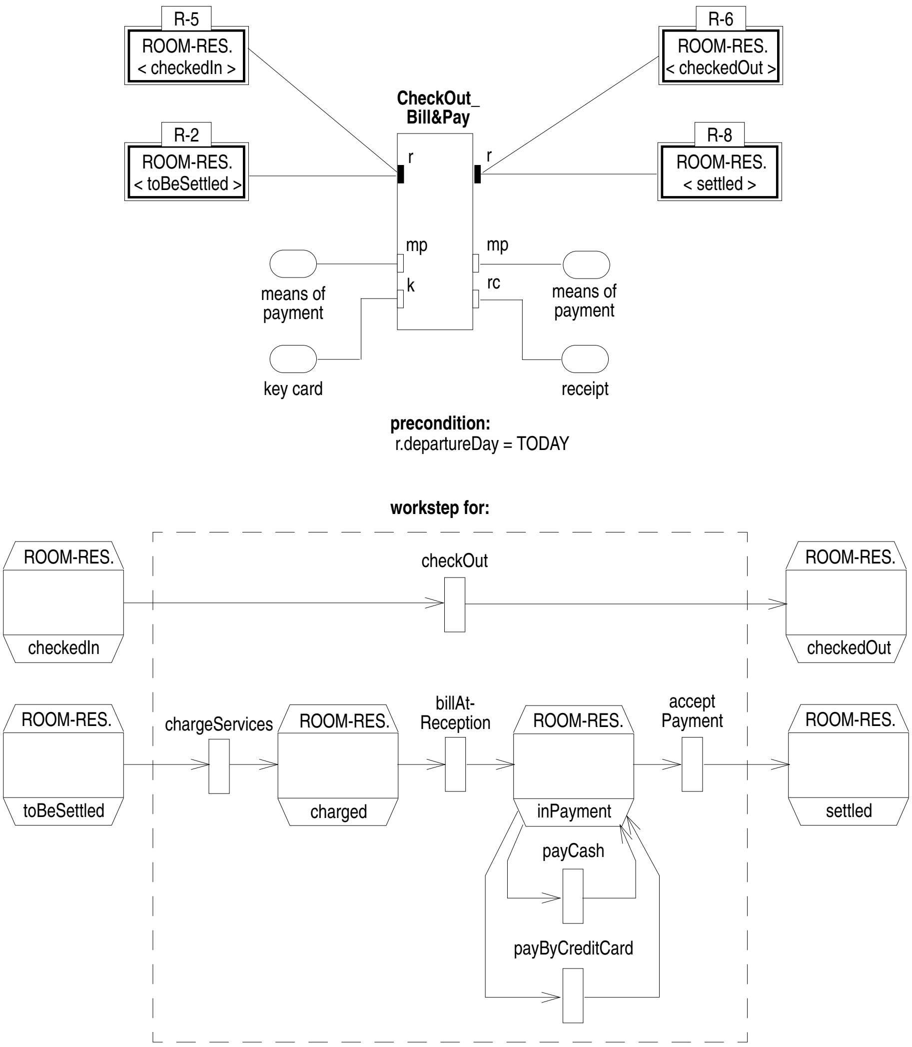


Figure 4.12: Specification of workstep CheckOut_Charge_Bill&Pay

The execution semantics of worksteps of a workflow process is determined by two port assignment rules. The *input port assignment rule* for worksteps determines under which conditions a workstep s can be started: (1) Every input port p of s is assigned the correct number of values (formal or informal data), which is specified by the cardinality of p. (2)

Every value assigned to an input port p must belong to the type domain of p. (3) Every object assigned to an input port p must belong to the active domain of p, i.e., it must be member of every box connected to p by incoming arcs. (4) The precondition of s is satisfied. (5) Neither workstep s nor a workstep that is alternative to s is currently active on the business case assigned to the primary port of s[6]. Note: The above input port assignment rule incorporates an optimistic strategy for the consistent execution of worksteps. This means, it aims at achieving a high degree of parallel execution of worksteps at the risk that conflicts might arise during the execution of worksteps. More precisely, the above input port assignment rule allows that two worksteps may be started although a conflict might arise during their execution with respect to an object or a set of objects assigned to a secondary port of one or both of the two worksteps. A pessimistic strategy aims at avoiding conflicts as far as possible. To avoid the above kind of conflicts the input port assignment rule could be extended by the following condition (optional): (6) No object assigned to a secondary port of s is also assigned to a secondary port of another instance of workstep s or to some port of an alternative workstep of s that is currently active[6] on some business case.

The *output port assignment rule* for worksteps determines under which conditions a workstep s can complete successfully: (1) Every output port p is assigned the correct number of values, which is specified by the cardinality of p. (2) Every value assigned to an output port p must belong to the type domain of p. (3) Every object assigned to an output port p must belong to the active domain of p, i.e., it must be member of every box connected to p by outgoing arcs. However, if two or more boxes connected to p are disjunctive, a business case must be member of only one of these boxes. (4) The postcondition of s is satisfied.

As mentioned before, we assume that information on the execution of worksteps is written to a trace. At least information on the instantiation of a workstep for a particular business case, the assignment of values to secondary ports of a workstep, and the completion or abort of a workstep must be written to the trace. Note: A workstep s is active on a business case c (assigned to the primary port of s) if an instance of s have been created for c and yet s did not complete or abort.

Workstep Realization

A *workstep realization* models the implementation of a basic workstep by a set of *containers*, a set of *operations*, and a set of directed *arcs* connecting containers to data and vice versa. Containers store values. Operations consume values stored in containers connected to their input ports, and produce values into containers connected to their output ports if they complete. For input ports and output ports of a workstep there are predefined containers. Operations are distinguished into: formal, informal, or control operations. Formal operations correspond to activities performed on business objects (cf. Subsection 3.2.2), informal operations to external operations such as collecting cash. Control operations are

[6]Whether a workstep is active on some business case or not, can be determined based on information on the execution of worksteps that is written to traces associated with workflow processes (below).

used to direct the data flow if necessary. Workstep realizations follow the object-oriented paradigm according to which the implementation of an operation has only direct read and write access to properties of the objects for which it is defined. Workstep realizations are not further discussed in this thesis. Note: The reader interested in workstep realizations and their graphical representation, i.e., workstep-realization diagrams, is referred to [BPS97b] for information.

4.4.5 Modeling of Assignment Rules

An *assignment rule* determines a set of responsible actors for a workstep of a workflow process. For each workstep of a workflow process there is an assignment rule. We propose to model assignment rules as queries that retrieve sets of positions and/or agents from a database that maintains knowledge on actors and groups of actors (cf. Chapter 5). The object-oriented data model presented in Chapter 3 offers a graphical notation for the specification of OQL-like queries. Specifying assignment rules as queries is a very flexible approach. It allows to specify assignment rules as arbitrary predicates on objects maintained within a workflow system. That is, assignment rules may depend on the state of organizational objects, business cases, workflow objects, and authorization contracts.

Assignment rules determine responsibilities of actors. Modeling of responsibilities of actors is closely related to the modeling of protection requirements of organizational processes and the modeling of authorizations of actors (cf. Chapter 6). Authorizations represent permissions to act, responsibilities represent obligations to act. Contemporary WFMSs oversimplify matters and determine who *can* access a workstep based on the *responsibility* of actors (cf. [JB96, p. 183]). This means, in contemporary WFMSs a responsibility to act implies the authority to act. In this thesis we clearly separate between authority and responsibility. An actor may perform a workstep s on a business case o only if that actor holds an appropriate authorization. This basic criterion must be satisfied even if an actor has been selected by an assignment rule as a responsible actor to perform some workstep on some business case. That is, no unauthorized access may occur. *To complete existing workflow technology, we present a powerful authorization model for the modeling of protection requirements of organizational processes and authorizations in workflow systems.* For the modeling of assignment rules, we adopt an approach that has been proposed by Bussler in [Bus96], i.e., we use a query language to specify assignment rules. We do not further discuss the modeling of assignment rules in this thesis due to space and time limitations. We leave a detailed discussion of the modeling of assignment rules as well as of the interdependencies between responsibilities and authorizations for future work.

4.5 Summary

In this chapter we presented a two-schema architecture for workflow systems. We started with a management definition of the fundamental concepts presented in this chapter, i.e., organizational processes and their representation within a novel two-schema architecture for workflow systems. The proposed *two-schema architecture* for workflow systems considers knowledge on organizational processes in two related but separate schemas, called the business-process schema and the workflow-process schema. A business-process schema consists of a set of business-process types, each one defining a so-called business process. A workflow-process schema consists of a set of workflow-process types, each one defining a so-called workflow process. The basic principle of the two-schema architecture for workflow systems is to separate knowledge on organizational processes according to its origin into external and internal knowledge (i.e., knowledge expressed in form of external and internal business rules, respectively). A *business process* is a model of an organizational process that considers only external knowledge. A *workflow process* is a model of an organizational process as it is currently performed in a particular organization. This means, a workflow process must consider both internal as well as external knowledge. However, a basic feature of the proposed two-schema architecture for workflow systems is that workflow processes do not specify external knowledge on organizational process directly. Instead, each workflow process is built on top of a business process. Together, a business process and a workflow process fully specify an organizational process. Note: An individual occurrence of an organizational process, called *business case*, is represented as instance of a business-process type. Each business case is assigned to an instance of a corresponding workflow-process type for processing. This ensures that internal business rules are adhered to when processing business cases.

The term *organization independence* was coined to denote the independence of business processes from knowledge on an organizational process that is based on commitments of a particular organization (cf. Section 4.2). The benefits of organization independence are: (a) *Process evolution*: Most changes to organizational processes are due to changes in the business environments of organizations. Since organizations have no direct influence on external business rules, organizations may only change internal business rules to adapt their processes to changed requirements. As a consequence, knowledge on organizational processes specified in business-process schemas is rather stable and thus reusable. (b) *Outsourcing*: Since business processes do not contain knowledge on organizational processes that is specific for a particular organization, business-process schemas might be developed by specialized organizations and be bought off the shelf in the future.

Chapter 5

Organization Modeling

Contents

Every organization is a dynamic system that carries out several processes to meet its objectives, e.g., offer services or goods to its customers. Usually, producing services and goods involves several activities and several actors within an organization. This means, there is a *division of labour*. To guarantee that the objectives of an organization are met, mechanisms and policies to *coordinate individual activities* must be employed. Organization theory (e.g., [Daf95, LARB+91]) knows different means to coordinate the work of actors within an organization. Models of organizational processes[1] help to coordinate the work of individual actors in that they show allowed execution orders of activities on business cases. We build two models of organizational processes to be enacted by a WFMS, i.e., a business process and a workflow process (cf. Chapter 4). Besides models of organizational processes, organization theory knows several structural means to coordinate work within an organization. The design of *positions*, the *grouping together of positions* into organizational units, the *grouping together of organizational units* into an overall organization, and the organization of positions and organizational units into a *control structure* are such structural means. In this chapter we will learn about the design of *organization schemas* for use in WFMSs.

[1] Organization theory calls models of organizational processes *rules* or *procedures* (e.g., [Daf95]).

This chapter is organized as follows. Section 5.1 gives a management definition of the fundamental modeling concepts presented in this chapter. Section 5.2 presents object-oriented concepts for the conceptual modeling of organization schemas. Section 5.3 identifies the source of authority in organizations and discusses how authority can be distributed among the actors of an organization in a disciplined way. Section 5.4 summarizes this chapter.

5.1 Fundamental Modeling Concepts

The core concept discussed in this chapter is *coordination*. This chapter is about mechanisms and policies to coordinate individual process activities performed by the *actors* of an organization such that the overall objectives of the organization are met.—An *actor* is someone who, i.e., a human, or something that, e.g., a piece of software, acts within an organization. Actors help to achieve the objectives of an organization by performing activities of organizational processes.—An important device to coordinate the activities of the actors of an organization has already been presented in Chapter 4, i.e., a conceptual model to represent organizational processes. The focus of this chapter is on structural means to coordinate the activities performed by different actors within an organization and on means to integrate models of actors and models of organizational processes.

Basically, the fundamental organization modeling concepts can be distinguished into two categories: *organizational objects* and *organizational relationships*.

Organizational objects. An *organizational object* is a model of an entity from a business area (or "universe of discourse"). We distinguish organizational objects into *agents*, *non-agents*, and *resources*. Agents model "active" entities from a business area, called *actors*. Non-agents and resources model "passive" entities from a business area. Non-agents are passive entities that help to coordinate activities performed by the same agent, e.g., positions, or by different agents, e.g., organizational units. Resources are passive entities that agents use or work on to meet the objectives of an organization.

Agent: An *agent* is a model of an actor. Agents are distinguished into human agents and software agents. *Human agents* model human actors, e.g., employees of an organization. *Software agents* model a kind of "virtual actors" that are realized by pieces of software.

Position: A *position* is a model of a kind of "abstract actor" of an organization. Each position determines the authority and responsibility of an agent occupying that position at a particular time. That is, a position determines a set of activities that an individual agent occupying that position at a particular time is permitted to perform (authority) and a set of activities that agent is obliged to perform (responsibility). The authority and responsibility of an individual agent is determined in that this agent is assigned to a specific position within an organization.

Positions are distinguished into single-agent and multi-agent positions: A *single-agent position* is occupied by a single agent who *alone* has to meet all responsibilities associated with that position. A *multi-agent position* is occupied by several agents who *together* have to meet the responsibilities associated with that position. For example, multi-agent positions allow to model positions that are occupied by several part-time workers at a time.

Organizational unit: An *organizational unit* is a model of a group of actors that has a specific area of responsibility within an organization. For example, divisions, departments, as well as teams of an organization are modeled as organizational units. Moreover, an organization itself is also modeled as an organizational unit.

Resource: A *resource* is an organizational object that is neither an agent nor a non-agent, e.g., a business case, a printer, a text editor. Agents use or work on resources to meet the objectives of an organization.

Organizational relationships. An *organizational relationship* links organizational objects. Organizational relationships determine the structure of an organization. The structure of an organization has one *physical dimension*, it determines how an organization is composed of its basic building blocks, which are positions, agents, teams, departments, divisions, etc., and several *logical dimensions*. In this chapter we discuss the the *control structure* of organizations as one of the logical dimensions of their structure. A control structure among the actors of an organization determines who coordinates the activities of whom such that the objectives of that organization can be met. *Hierarchy* is the most prominent structural mechanism to coordinate activities performed by different actors within an organization.

Physical structure: The *physical structure* of an organization determines how *positions are grouped* together into organizational units and how *organizational units are grouped* together forming the overall organization.

Control structure: Division of labour among different actors of an organization requires additional effort to guarantee that the overall goals of the organization are met, i.e., activities performed by different actors must be coordinated. Organizations install a control structure among its actors (organizational units), where selected actors (organizational units) are designated to coordinate the activities of others. *Hierarchy* is such a control structure. Actors (organizational units) designated to coordinate the activities of others are typically called “supervisors” in a hierarchical control structure. Note: *Network* is another control structure, which we will investigate in Chapter 9.

Organization chart: *Organization charts* are a tool that may be used to visualize *different* dimensions of the structure of an organization.

Summing up, we emphasize that the aim of designing the structure of an organization is to coordinate the activities of the organizational processes performed by individual actors. Positions are employed to coordinate activities performed by single actors. Organizational units and hierarchical relationships between positions and between organizational units are employed to coordinate activities performed by different actors. Models of organizational processes are another means to coordinate activities of actors (cf. Chapter 4).

5.2 Design of Organization Schemas

An *organization schema* consists of a set of organizational-object types and a set of organizational-object classes. Each organizational-object type defines the structure and behavior of its instances, called *organizational objects*, e.g., agents, positions, resources, or organizational units. An organizational-object class collects a set of organizational objects of a specific type, the member type of the class.

Organizational objects are business objects, too. This means, the concepts presented in Section 3.2 for the conceptual modeling of business objects can be applied in the same way to the modeling of organizational objects. In this section these concepts are reviewed in the realm of conceptual modeling of organizational objects and new concepts specific to the modeling of organizational objects are introduced, e.g., the hierarchy.

The remainder of this section is organized as follows. Subsection 5.2.1 discusses the modeling of the structure of organizational objects is discussed. Besides general concepts known for the modeling of business objects, concepts specific to the modeling of the structure of organizational objects are presented. Subsection 5.2.2 discusses the modeling of the behavior of organizational objects and Subsection 5.2.3 the modeling of classes of organizational objects in the realm of concepts presented in Section 3.2.

5.2.1 Modeling the Structure of Organizational Objects

The structure of organizational objects is determined by a set of *attributes* and a set of *relationships*. An attribute models a descriptive feature of an organizational object, e.g., its name, a relationship models a reference of an organizational object to one or several other objects.

Organizational Objects

We distinguish organizational objects into *agents*, *positions*, *organizational units*, and *resources*. Agents and positions represent actors of an organization, organizational units represent groups of actors of an organization, and resources represent things used or worked on by actors of an organization to meet the organization's objectives (cf. Figure 5.1 and Section 5.1). Note: Figure 5.1 shows meta information. To avoid confusion with regular diagrams depicted using the graphical notation of Object/Behavior Diagrams, we borrowed the graphical notation from OMT (cf. [RBPE91]) to depict the diverse categories of organizational objects and start names of relationships with a capital letter.

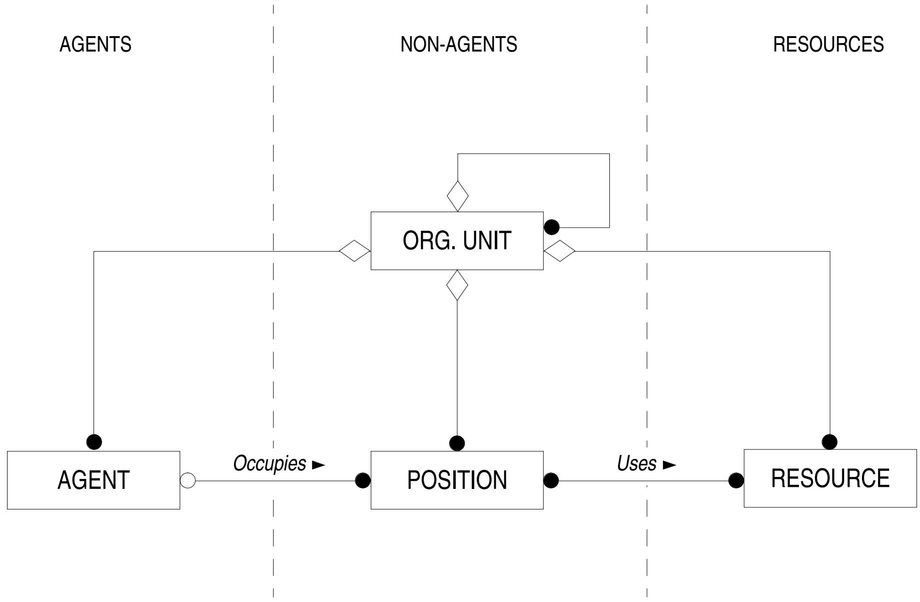


Figure 5.1: Conceptual relationships between organizational objects

Agents. An *agent* is a model of an actor. Agents are distinguished into *human agents* and *software agents*. Human agents represent humans, e.g., employees, software agents a kind of "virtual actors". That is, a software agent is a piece of software that is "smart" enough to perform activities of organizational processes, but has no real-world counterpart. Software agents are also called computational agents (cf. [Kir95]) or intelligent agents in the literature. Whereas software agents solely exist in model-worlds inside of WFMSs, an important function of human agents is to mediate between model-worlds and the real-world, i.e., *a human agent is a mediator through which a human can act inside a WFMS*. Before a human can act through an agent she or he has to connect to that agent. To avoid that humans connect to agents of others, mechanisms called *identification* and *authentication* are employed (cf. Section 6.2 and Subsection 10.2.4).

Each agent has a designated relationship that models a reference to the position it occupies within a particular organization (cf. positions,below). Traditionally, an agent occupies exactly one position within an organization. The conceptual model presented in this chapter is not that restrictive. It allows each agent to occupy an arbitrary number of positions within an organization (cf. Figure 5.1).

Note: To make the physical structure of organizations independent from individual actors, agents are usually modeled as components of organizational units representing the overall organizations for which the actors work, not as components of organizational units within which the actors occupy positions (cf. aggregation relationship between concepts 'agent' and 'organizational unit' in Figure 5.1 as well as the subsection on organizational relationships, below). To make the control structure of organizations independent from individual actors, the control structure among actors is modeled between positions (cf. positions, below), not between agents. Agents are embedded into the control structure of an organization in that they are assigned to positions (cf. organizational relationships, below).

Example: Figure 5.2 shows a type diagram of agent type EMPLOYEE. Each employee is identified by the value of its key attribute empNo and is further described by the values of attributes name, birthday, and hireDate. In addition, an employee has relationships address, spouse, and skills to values from domains ADDRESS, EMPLOYEE, and SKILL, respectively. Multi-valued relationship skills determines the set of skills acquired by an employee.

Furthermore, an employee has relationships workAsManager, workAsSecretary, and workAsAssistant to values from domains MANAGER, SECRETARY, and ASSISTANT, respectively. These relationships associate an employee with the position(s) the employee occupies within an organization. Relationships workAsSecretary and workAsAssistant are single-valued, relationship workAsManager is multi-valued, i.e., an employee may occupy several manager positions within an organization but only one secretary position and only one assistant position. Note: If an employee may not occupy different types of positions at the same time, an appropriate integrity constraint must be specified.

Relationship belongsTo with domain HOTEL models a reference from an employee object to the hotel for which the agent works (cf. physical structure of organizations, below).

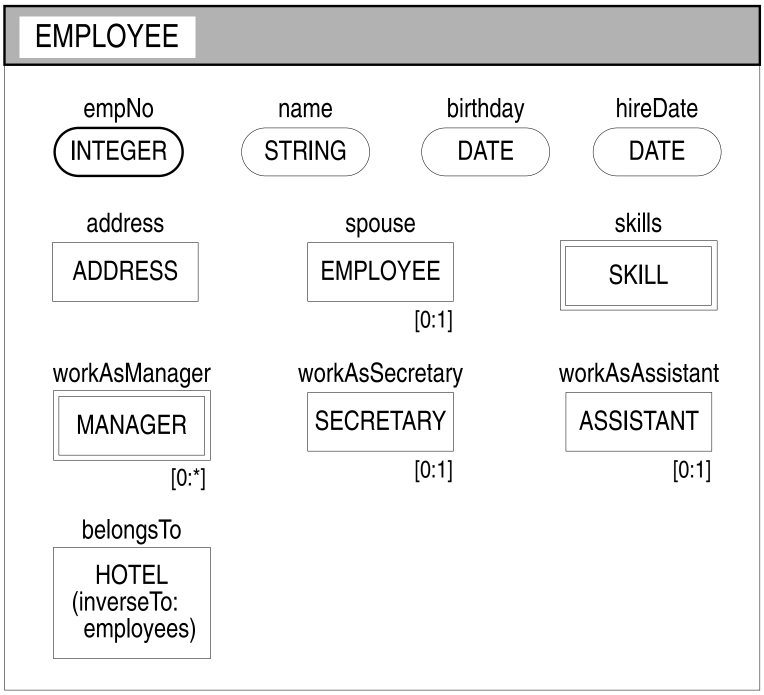


Figure 5.2: Agent type EMPLOYEE

Positions. A *position* models a kind of "abstract actor" of an organization. Historically, the concept of positions has been introduced to make the structure of organizations independent from individual actors (cf. [LARB+91]). In fact, a position determines the *authority*[2] and the *responsibility*[3] of an individual actor that is assigned to that position at a particular time. This means, authorizations and responsibilities of individual actors are determined according to the positions they occupy within an organization. The rank of an individual actor within the control structure of an organization is also determined by the position with which the actor is associated. An actor associated with a position is said to *occupy* that position. Each actor occupies one or several positions within an organization. A position that may be occupied by exactly one actor at a time is called *single-agent position*, a position that may be occupied by several actors at a time *multi-agent position*. An actor that occupies a position within an organizational unit uses resources of that organizational (cf. relationship 'Uses' between concepts 'Position' and 'Resource' in Figure 5.1). Note: Actors are evolving entities, i.e., actors may occupy different positions within the same and within different organizations over time.

Each position has a designated relationship called occupiedBy to one or several agents representing the individual actors occupying that position. Relationship occupiedBy is single-valued for single-agent positions, it is multi-valued for multi-agent positions[4]. For simplicity, we consider only single-agent positions in this chapter (cf. Figure 5.1).

Example: Figure 5.3(a) shows a type diagram of position type MANAGER. Each manager position is described by the value of attribute phoneNo. A manager position belongs to the manager's office of a hotel (cf. relationship belongsTo) and can be occupied by an agent of type EMPLOYEE (cf. relationship occupiedBy).

Figure 5.3(b) shows a type diagram of position type SECRETARY. Each secretary position is described by the values of attributes phoneNo and wordsPerMinute. A secretary position belongs to the manager's office of a hotel (cf. relationship belongsTo) and can be occupied by an agent of type EMPLOYEE (cf. relationship occupiedBy).

Figure 5.3(c) shows a type diagram of position type ASSISTANT. Each assistant position is described by the value of attribute phoneNo. An assistant position belongs to the manager's office of a hotel (cf. relationship belongsTo) and can be occupied by an agent of type EMPLOYEE (cf. relationship occupiedBy).

[2]Note: The authority of an actor occupying a position is determined by the set of authorizations assigned to that position (cf. Chapter 6).

[3]Note: The responsibility of an actor occupying a position is determined by the set of tasks assigned to that position, which the actor is obliged to perform. If activities with high inter-dependencies are carried out by the same actor, the overall coordination effort can be reduced, since intra-actor coordination is less expensive than inter-actor coordination.

[4]Note: Different agents that occupy the same multi-agent position share the *same* authorizations assigned to that position. As a consequence, auditing mechanisms, which monitor the utilization of the resources of a system by its users (cf. [CFMS94]), must write entries to their log files that consider the identity of both the position *and* the agent acting that position when a system resource is used.

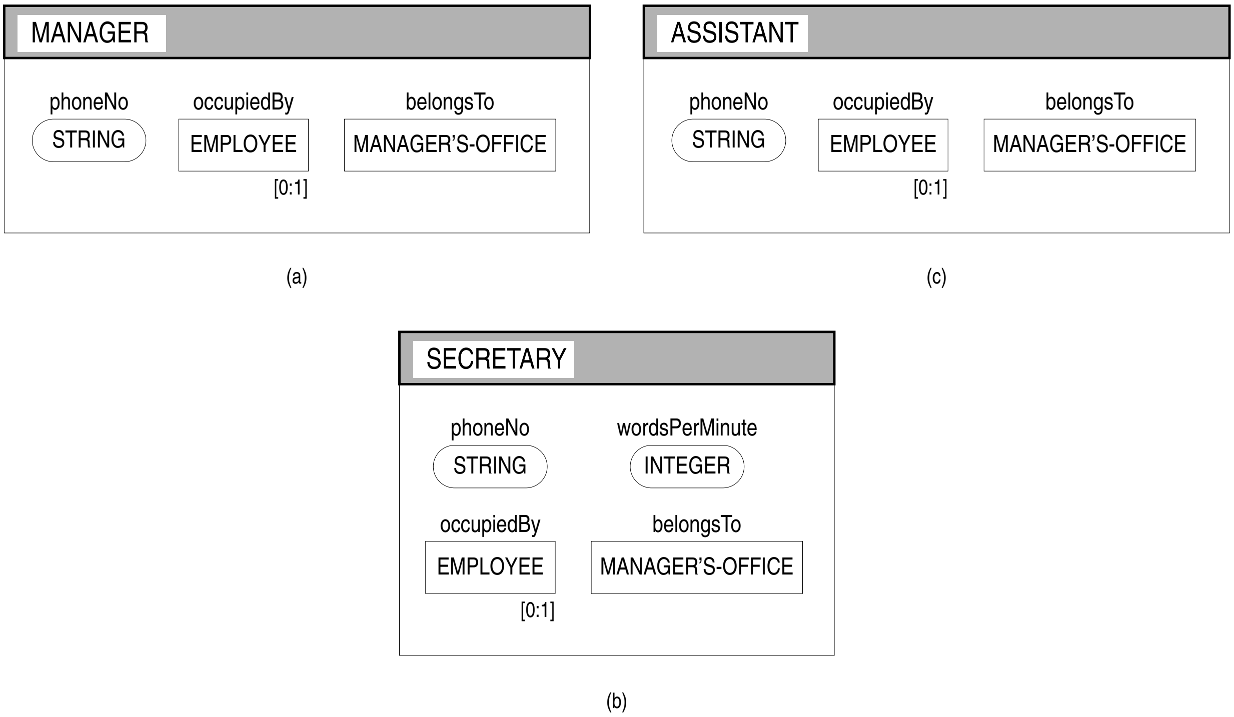


Figure 5.3: Position types MANAGER, SECRETARY, and ASSISTANT

WFMSs enact workflow processes. To notify actors about work to do, WFMSs associate so-called *worklists*[5] with agent objects. When a workstep is to be performed on a certain business case, a WFMS determines a set of responsible actors (cf. area of responsibility of actors, above) and notifies either one or several of them. WFMSs notify responsible actors in that they insert business cases to be processed in the worklists associated with the agent objects representing the responsible actors. Traditional WFMSs associate worklists directly with agent objects. As a consequence, such WFMSs may have to re-determine and re-notify responsible actors when actors leave an organization or move to other positions within the same organization. Otherwise it could happen that business cases contained in a worklist of an agent representing such an actor won't get processed. We assume that worklists are associated with position objects[6], making notification of actors independent from individual actors leaving or joining organizations, or changing positions within the same organizations. Of course, a well-defined *substitution plan* (cf. Subsection 9.2.1) must guarantee that substitute agents care for worklists associated with positions while they are not occupied by a regular agent.

[5] A *worklist* is a queue whose entries are business cases to be processed by the actor associated with that worklist.

[6] Note: Different agents that occupy the same multi-agent position (above) have access to the same worklist. A groupware system could be used to maintain a worklist in such a case.

Organizational units. An *organizational unit* may consist of a set of *positions*, a set of *agents*, a set of *resources*, and a set of other *organizational units* (cf. Figure 5.1). Positions are grouped together into organizational units, e.g., work groups, teams. Organizational units are grouped together into more complex organizational units, e.g., departments, divisions, or the overall organization, determining the physical structure of an organization (cf. physical structure of organizations, below).

Each organizational unit has a selected position (organizational unit) that is devoted to coordinate the activities performed by others. Such positions (organizational units) are determined by the control structure defined among the positions (organizational units) that belong to an organizational unit (cf. control structure of organizations, below).

Example: Figure 5.4 shows a type diagram of organizational-unit type HOTEL. A hotel is identified by the values of its key attributes name and address, and consists of a set of agents (cf. relationship employees with domain EMPLOYEE) and a set of organizational units, i.e., a manager's office, a marketing group, a finance-management group, a room-service group, and a reception (cf. relationships manager'sOffice, marketing, financeManagement, roomService, and reception with domains MANAGER'S-OFFICE, MARKETING-GROUP, FINANCE-MANAGEMENT-GROUP, ROOM-SERVICE-GROUP, and RECEPTION, respectively). The manager's office of a hotel is the supervising unit of the other organizational units of that hotel (cf. control structure of organizations, below). Note: For the moment, ignore that section of the type diagram marked by the keywords physical structure.

Figure 5.5 shows a type diagram of organizational-unit type MANAGER'S-OFFICE. A manager's office consists of a set of positions, i.e., a head, a secretary, and several assistants (cf. relationships head, secretary, and assistants with domains MANAGER, SECRETARY, and ASSISTANT, respectively). The head is the supervisor of the other agents assigned to positions within a manager's office (cf. control structure of organizations, below). A manager's office belongs to a particular hotel (cf. relationship belongsTo with domain HOTEL that is inverse to relationship manager'sOffice defined at type HOTEL). Note: For the moment, ignore that section of the type diagram marked by the keywords control structure.

Resources. Each *resource* belongs to a particular organizational unit of an organization and is used by agents acting positions in that organizational unit (cf. Figure 5.1).

Organizational relationships

Organizational relationships model a reference between organizational objects. Below, we focus on organizational relationships that determine the structure of an organization. The structure of an organization has several dimensions that may be modeled and investigated separately: The *physical dimension* determines the grouping together of positions into organizational units and of organizational units into the overall organization. The *control dimension* is a logical dimension of the structure of an organization. It determines a control

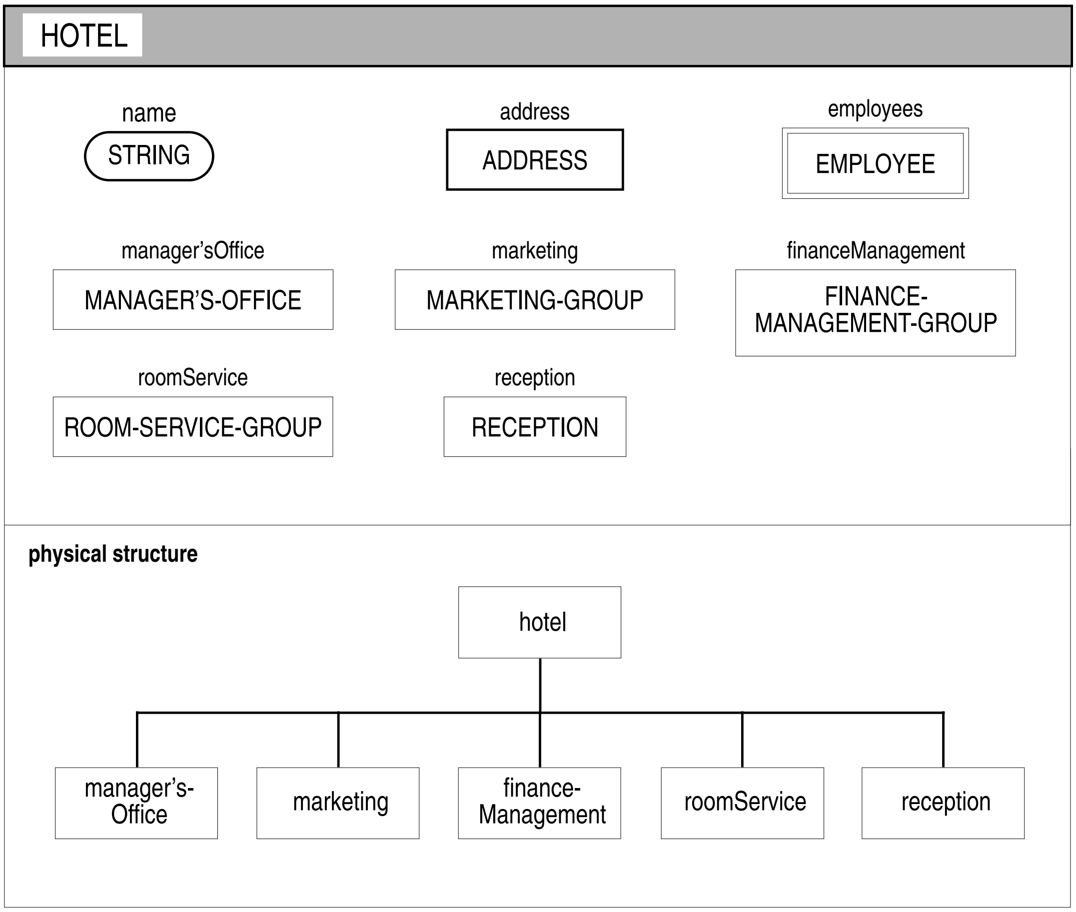


Figure 5.4: Organizational-unit type HOTEL

structure among the positions or among the organizational units grouped together into an organizational unit.

Physical structure of organizations. The basic building blocks of each organization are its actors. Performing activities of organizational processes they help to meet the objectives of an organization (cf. Section 5.1). We model each actor of an organization by exactly one agent object (cf. agents, above) and by one or several position objects. More precisely, a position does not model an individual actor of an organization but models the authority and responsibility of *some* actor of an organization (cf. positions, above). Moreover, it determines the *organizational context* of that abstract actor, i.e., an organizational unit within which an actor has to take over responsibility, as well as the *rank* of the abstract actor within that organizational context (cf. control structure of organizations, below). Positions help to make organizations independent from individual actors. The authority, the responsibility, the organizational context, and the rank of an individual actor of an organization can be determined in that the agent object representing that actor is assigned to a particular position. If an actor occupying a certain position within an or-

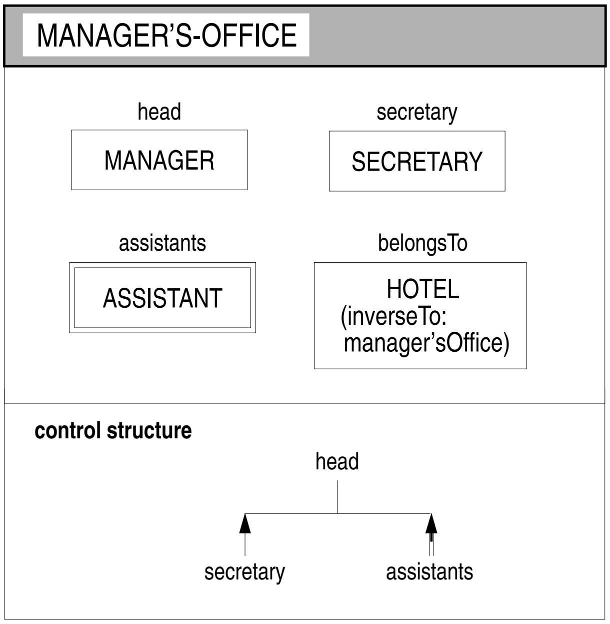


Figure 5.5: Organizational-unit type MANAGER'S-OFFICE

ganization leaves that organization or moves to another position within that organization, neither the physical structure of an organization nor its control structure (below) must be changed. The only thing to do is to occupy a vacant position with another actor, i.e., to assign another agent object to that position.

The physical structure of an organization is determined in that agents, positions, resources, and organizational units are grouped together forming an overall organization. Grouping together organizational objects organizes them into a part-whole tree. The root of such a tree represents an overall organization, intermediate levels represent organizational units, and leafs represent resources, positions, and agents. Resources are children of nodes representing either organizational units or overall organizations. Positions are children of nodes representing organizational units. Agents are children of nodes representing overall organizations. *Since agents to not belong directly to organizational units in which they act, the physical structure of organizational units need not be changed if an actor changes to another position within an organization or leaves an organization.* Note: Each agent may act in different organizational contexts if she or he occupies positions in different organizational units.

Graphically, part-whole trees are depicted in *physical-structure diagrams*. Nodes of a part-whole tree represent organizational units, positions, and agents and are depicted by graphical symbols with different shapes. Organizational units (overall organizations) are depicted by rectangles, positions by rounded rectangles, and agents by rounded rectangles that are shaded. The names of organizational units, positions, and agents are shown inside the graphical symbols representing them. If a position, an agent, or an organizational unit o belongs to an organizational unit u, then there is a solid line between the graphical symbols representing o and u, and the graphical symbol representing o is aligned below the graphical symbol representing u. Figure 5.6 schematically shows a part-whole tree of an organization. It is annotated with dashed lines indicating *occupies* relationships between agents and positions. Physical-structure diagrams are one form of the well-known organization charts. They show how an organization is composed of agents, positions, organizational units, and resources. Note: A part-whole tree may also be depicted partially in a type diagram showing an organizational-unit type U. In this case, only positions, agents, and organizational units that directly belong to organizational units of type U are considered in the part-whole tree. A separate section of the type diagram is devoted to the presentation of a part-whole tree in such cases. It is marked by the keywords physical structure.

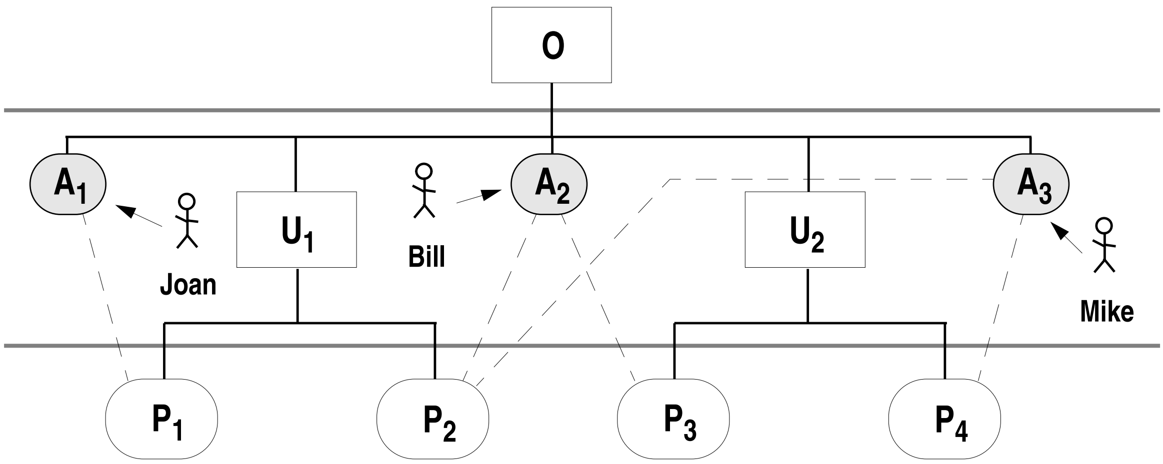


Figure 5.6: Part-whole tree of organizational objects (schematically)

Example: Figure 5.4 partially shows a part-whole tree of hotels. It shows only organizational units that directly belong to a hotel of type HOTEL.

Figure 5.7 shows the overall part-whole tree of hotels. A hotel consists of several agents, i.e., employees, and several organizational units, i.e., a manager's office, a marketing group, a finance-management group, a room-service group, and a reception. The organizational units consist of several positions. For example, the manager's office of a hotel consists of a head position, a secretary position, and several assistant positions. Note: Resources used by different organizational units are not considered in Figure 5.7.

Designers that model the physical structure of objects basically use two abstraction principles: decomposition and aggregation. *Decomposition* means that a complex system is divided into a set of simpler subsystems that are easier to understand. *Aggregation* means that a set of relative simple systems are assembled into a more complex system. The basic conceptual model presented in this part of the thesis offers only one kind of

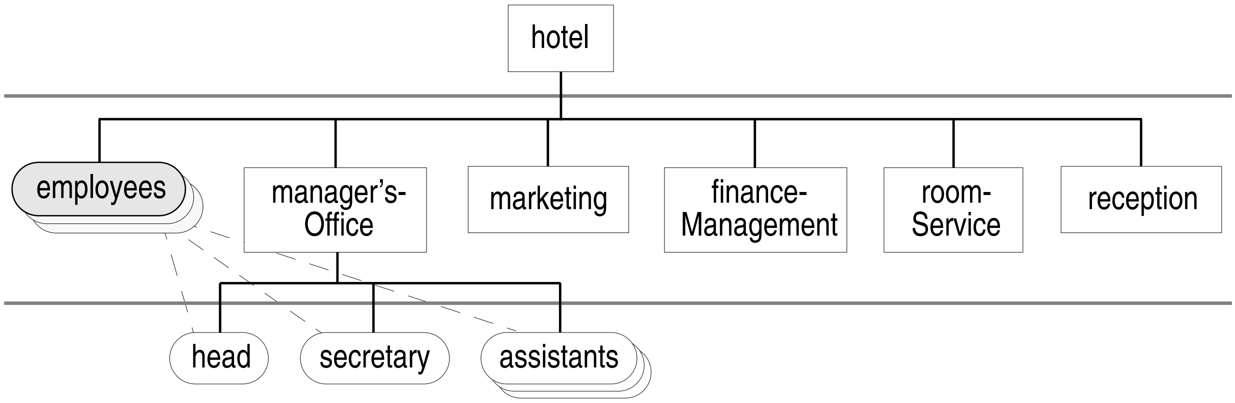


Figure 5.7: Part-whole tree of hotels

relationship to model references between composite objects and their components (decomposition) or references between aggregate objects and their constituents (aggregation) or "general" references between objects. That is, the concept of "relationships" presented in Subsection 3.2.1 is used to model all kind of references between objects. As a consequence, special semantics of references between objects cannot be captured in schemas and thus gets lost. The advanced conceptual model presented in Part III, will offer special types of relationships that include the semantics of references between objects, e.g., decomposition, aggregation, into schemas.

Example: The part-whole tree of hotels (cf. Figure 5.7) shows direct and indirect components of hotels that are organizational objects. Relationships between hotels and their components are defined at the types of the components and at the type of the hotels. For example, relationship belongsTo defined at agent type EMPLOYEE (cf. Figure 5.2) models for each employee a reference to the hotel for which the employee works. The inverse relationship employees has been defined at type HOTEL (cf. Figure 5.4).

The way in which positions are grouped together into organizational units has major influence on the effort necessary to coordinate the activities performed by actors occupying positions within an organization. *The basic principle is to group together positions with high inter-dependencies*, since intra-group coordination is less expensive than inter-group coordination. Positions and organizational units may be grouped together into other organizational units according to different criteria. We distinguish three criteria (e.g., [Daf95]): grouping by activity, output, or customer. *Grouping by activity* groups actors together who have similar skills and perform similar activities, e.g., all salespersons are grouped into a sales department. *Grouping by output* groups actors together who have different skills and work on the same product or service, e.g., all actors required to produce typewriters are grouped together into a single department. *Grouping by customer* groups actors together who have different skills and serve a certain customer or area, e.g., all actors required to serve the European market are grouped into a department.

Example: Organizational units are grouped together into hotels *by output.* That is, organizational units with different functions, i.e., a manager's office, a marketing group, a finance-management group, a room-service group, and a reception, are grouped together that work on the same service, offering accommodations to guests. Actors of a hotel are grouped together into organizational units *by activity.* For example, actors with marketing skills are grouped together into the marketing group of a hotel, those with finance-management skills into the finance-management group.

Control structure of organizations. Division of labour among different actors of an organization requires additional effort to guarantee that the overall goals of the organization are met, i.e., activities performed by different actors must be coordinated (cf. Section 5.1). Organizations install a control structure among its actors (organizational units) where selected actors (organizational units) are designated to coordinate the activities of others. Such *a control structure models an unbroken connection between the owner or the shareholders of an organization and each employee.*

The control structure of an organization serves as a *chain of command.* That is, a control structure corresponds to communication channels along which authority and responsibility, which originates at the owners (shareholders) of an organization, may be delegated to *and* removed from individual actors within an organization. To test legitimacy of authority and responsibility of actors within an organization, the chain of command must be unbroken such that it is possible to re-trace the authority and responsibility of individual actors back to the authority and responsibility of the owners (shareholders) at any time (cf. Section 5.3).

Hierarchy is such a control structure[7]. If *hierarchy* is used as a structural means to coordinate activities performed by different actors (organizational units) within an organizational unit, all managerial tasks are within the responsibility of a single actor (organizational unit), called *supervisor.* Typically, all members of an organizational unit, i.e., positions or other organizational units, are subordinate to the supervisor of that organizational unit. A hierarchical control structure between positions (organizational units) of an organizational unit can be fully specified by a so-called *subordination graph.*

A *subordination graph* $G_U = (V, E)$ is a rooted tree defined at an organizational-unit type U. It defines a hierarchy among positions or organizational units that belong to organizational units of type U (cf. physical structure of organizations, above). V, the set of vertices of G_U, is the set of *names* of positions or organizational units that belong to organizational units of type U. $E \subseteq V \times V$, the set of directed edges of G_U, is the set of subordination relationships between names of positions or organizational units contained in V. An edge $(v_1, v_2) \in E$ between positions (organizational units) v_1 and v_2 says that v_2 is subordinate to v_1. The root of a subordination graph G_U is the name of the supervisor position or organizational unit of organizational units of type U. If an organizational unit u of type U is itself subordinate to an organizational unit u' of type U', then the supervisor of u is subordinate to the supervisor of u'. If an organizational unit u of type U has

[7]Note: *Network* is another control structure, which we will investigate in Chapter 9.

no superordinate organizational unit, then the supervisor of u is the top position or the top organizational-unit in the control structure of the organization to which u belongs. Subordinate positions or organizational units are connected to the root of a subordination graph by a path of length n, where $n \geq 1$.

A subordination graph G_U need not be instantiated for organizational units of type U. To refer to subordinates or superordinates of positions and organizational units, predefined operations hasSubordinates and subordinateOf can be invoked on positions and organizational units. Invoking these operations on a position or an organizational unit o returns *names* of positions or organizational units that are subordinate or superordinate to o. These names are properties of the organizational unit to which o belongs. Since properties of objects are to be considered as public retrieval operations (cf. Subsection 3.2.1), a position or an organizational unit can refer to its subordinates or to its superordinate in that it applies the names of its subordinates or the name of its superordinate as retrieval operations to the organizational unit to which it belongs. This means, the knowledge specified in a subordination graph G_U fully determines the control structure among the positions or the organizational units that belong to organizational units of type U.

Graphically, a subordination graph is depicted in a *control-structure diagram.* A control-structure diagram depicts nodes of a subordination graph, i.e., names of positions and organizational units, as text, and edges as solid arrows. The targets of such arrows are the superordinates. An arrow is shown with a single line if the underlying relationship is single-valued, it is shown with a double line if it is multi-valued. The graphical presentations of different edges partially coincide if they share the same superordinate. Usually, subordinates are aligned below their superordinates. Control-structure diagrams are one form of the well-known organization charts. They show subordination relationships among the positions and among the organizational units of an organization. Note: A subordination graph may also be depicted in a type diagram showing the organizational-unit type at which the subordination graph has been defined. In this case, a separate section of the type diagram is devoted to the presentation of a subordination graph, which is marked by the keywords control structure.

Example: Figure 5.8 shows the subordination graph defined at organizational-unit type HOTEL (cf. Figure 5.4). The manager'sOffice, an organizational unit of type MANAGER'S-OFFICE (cf. Figure 5.5), is the supervisor of organizational units named marketing, financeManagement, roomService, and reception. Note: Invoking operation hasSubordinates on the manager's office of a hotel answers the names 'marketing', 'financeManagement', 'roomManagement', and 'reception'. Invoking operation subordinateOf on the marketing group, the finance-management group, the room-service group, or the reception of a hotel answers the name 'manager'sOffice'.

The subordination graph defined at organizational-unit type MANAGER'S-OFFICE is depicted in a separate section of the type diagram shown in Figure 5.5. The head, a position of type MANAGER, is the supervisor of positions named secretary and assistants. Note: Invoking operation hasSubordinates on the head of the manager's office of a hotel answers the names 'secretary' and 'assistants'. Invoking operation subordinateOf on the secretary or an assistant of the manager's office of a hotel answers the name 'head'.

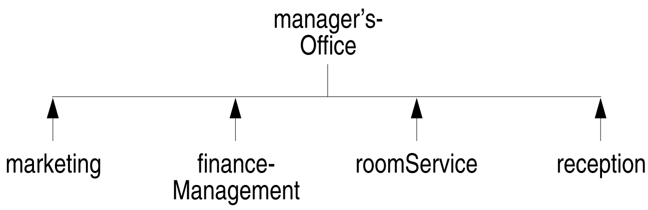


Figure 5.8: Subordination graph defined at HOTEL

5.2.2 Modeling the Behavior of Organizational Objects

The behavior of organizational objects is determined by a set of *activities*, by *allowed execution orders* on these activities, and by their *interactions* with other objects. In other words, the behavior of organizational objects is modeled in the same way as the behavior of ordinary business objects (cf. Subsection 3.2.2).

Example: Figure 5.9 shows the life-cycle of agent type GUEST. Instances of agent type GUEST represent guests of a hotel. After a guest agent has been created, the guest's creditworthiness is tested. If accepted, the guest object resides in state active. Provided a guest agent has been authorized appropriately, a guest may consume several services using its agent. If denied, the guest agent will be archived for later reference. From time to time, the creditworthiness of guests may be tested again. If guests do not pass this test their agent will be in-validated. This means, their agents become passive such that guests may no longer act through them. Passive agents may be archived or re-validated later on.

Note: If the behavior of an organizational object itself is a model of an organizational process that indirectly adds value to an external customer of an organization, i.e., is a model of a support process or a management process, the behavior of the organizational object might be designed for the two-schema architecture, too (cf. Section 4.2). In this case, an organizational object would be full member of some workflow object that defines the workflow process according to which the organizational object is handled. For example, an organizational process for staff development might be modeled as the behavior of agent types.

The behavior of organizational objects must not be confused with the behavior of an organization. The term *behavior of an organization* is frequently used to denote the processes an organization carries out to achieve its objectives. Appropriate concepts for the modeling of organizational processes have been presented in Chapter 4.

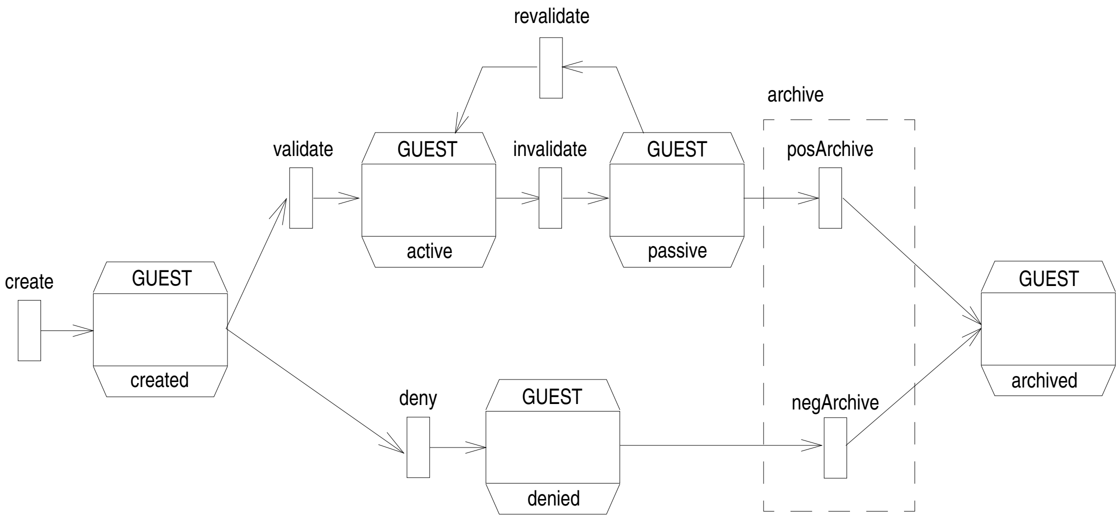


Figure 5.9: Life-cycle of agent type GUEST

5.2.3 Modeling Classes of Organizational Objects

An *organizational-object class* collects a set of organizational objects as its members. Note: Classes of organizational objects are modeled in the same way as classes of business objects (cf. Subsection 3.2.3).

Example: Figure 5.10 shows the specifications of agent classes Guests, VIP-Guests, King, Employees, Managers, and SeniorEmployees.

Agent class Guests is the base class of agent type GUEST and an instance of class type GUESTS. Agent class VIP-Guests is a basic subclass of agent class Guests. Whether a guest of our hotel becomes a member of agent class VIP-Guests is decided on by an authorized person. This means, guests must be explicitly inserted into agent class VIP-Guests. Agent class King is a subclass of agent class VIP-Guests. It is a singleton class, representing the fact that there may only be one King.

Agent class Employees is the base class of agent type EMPLOYEE and an instance of class type EMPLOYEES. Agent class Managers is a derived subclass of agent class Employees. It collects all employees that are assigned to a manager position. Agent class SeniorEmployees is also a derived subclass of agent class Employees. It collects all employees that have been with the organization for more than twenty-five years. (Note: The corresponding derivation expressions are not shown.)

5.3 The Source of Authority in Organizations

The source of authority and responsibility within an organization is with its owner (cf. [MS88, MS91]). The owner of an organization, which is either a single person or a set of shareholders, has authority over the use of resources. In many organizations ownership

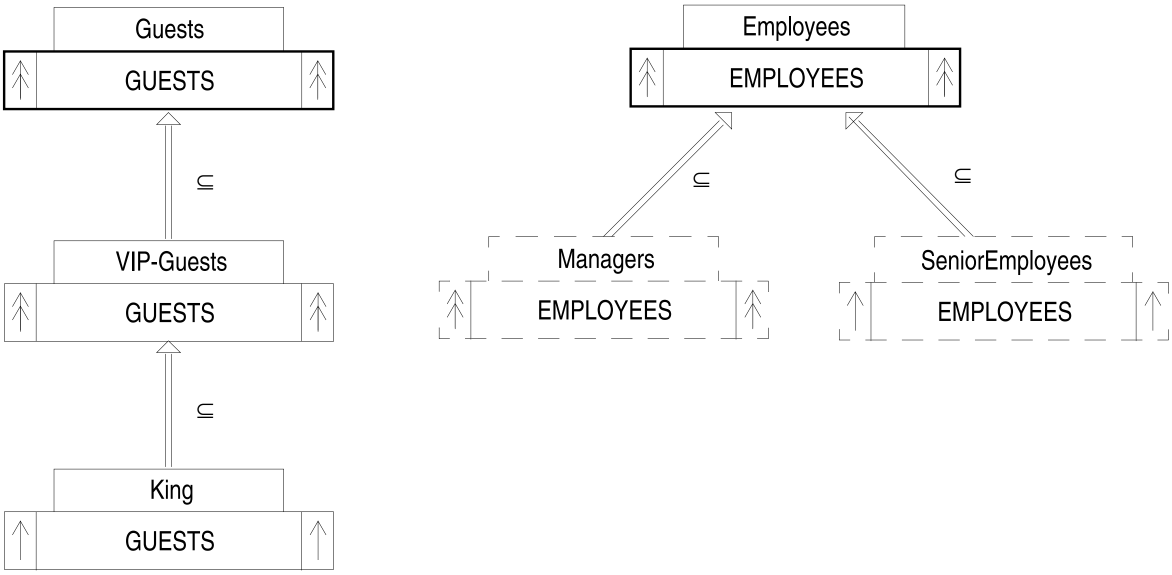


Figure 5.10: Specification of classes of organizational objects

is separated from management of resources. This means, the owner of an organization delegates authority *and* responsibility to the management. Delegating authority to the management does not mean, however, that the owner loses his authority. Rather, delegating authority to the management of an organization creates a new authority of the owner, the authority to revoke authority from the management to which she or he just did delegate it. Similarly, delegating responsibility to the management of an organization does not mean that afterwards the owner is free from any responsibility. Rather, a new kind of responsibility is created. It is an owner's responsibility to delegate responsibility to managers that have the capability to handle it. The same is true if a manager further delegates authority or responsibility to another employee. In general, an employee is responsible for performing some activities, whereas a manager is responsible to coordinate activities carried out by his subordinates, bring them into action according to their capabilities, and furnish them with necessary resources.

Authority as well as responsibility may not be delegated in an arbitrary way within an organization. Moffet and Sloman [MS88] use the term *organization domain* to denote the set of employees, usually defined by their positions, to whom a manager may delegate authority or responsibility, and the term *resource domain* to denote the set of resources over which authority or responsibility can be delegated to employees in the organization domain. *The organizational domain of a manager is determined by the control structure of an organization, the resource domain by the physical structure of an organization.* That is, modeling the control structure the physical structure of an organization is fundamental for a formal authorization system.

In a formal authorization system, there must be an *unbroken connection* between the owner or the shareholders of an organization and each employee of an organization, usually via the positions employees occupy. This connection represents an official communication channel along which authorizations may be delegated. Testing legitimacy of an employee's authorizations requires to re-trace the authorization back to the owner's (or the shareholders') authority.

5.4 Summary

In this chapter we presented basic concepts for the design of *organization schemas* for use in WFMSs. First, we gave a management definition of the fundamental concepts presented in this chapter, i.e., *organizational objects* and *organizational relationships. Agents* model actors of an organization, e.g., employees. *Positions* model a kind of "abstract actors" that help to make organizations independent of individual actors. To model that an actor occupies a position within an organization, the agent object representing that actor is related to the appropriate position object. Positions are grouped together into *organizational units*, organizational units into more complex organizational units, determining the physical structure of organizations. To offer an unbroken connection from the owners of an organization to each employee, a control structure is established among the positions and among the organizational units of an organization. This connection represents a chain of command along which authority and responsibility is delegated to individual employees within an organization. An unbroken connection between the owner of an organization and its employees allows to test the legitimacy of each employees authority and responsibility at any time and is thus a fundament for a formal authorization system.

Chapter 6

Authorization Modeling

Contents

Integrity of information maintained in information systems has been identified as the primary concern of information security in commercial environments (e.g., [CW87]). First, integrity of information requires that information is correct and complete. Traditionally, an information system uses a *schema* to determine the correctness and completeness of information it maintains. A schema is a collection of constraints that describe the structure and behavior of information and acts as a filter to exclude illegal information from an information system. Second, integrity of information requires that the information is protected from being modified by unauthorized actors. Traditionally, an information system engages an authorization manager to mediate requests of actors and to deny those requests for which no appropriate authorization is stored in the authorization base of the information system. However, an authorization manager cannot decide whether authorizations stored in the authorization base of an information system are legal wrt. the protection requirements of an organization. This means, an authorization manager cannot deny unauthorized requests of actors if illegal authorizations are stored in the authorization base of an information system. In traditional information systems, the responsibility to issue only authorizations to actors that are legal wrt. the protection requirements of organizations is with the security administrators of these information systems. To make integrity of information stored in information systems independent from security administrators, we propose to design an authorization schema, which is a formal description of the protection requirements of an organization wrt. to the information maintained in an information system. An authorization schema is a central description of the protection requirements of an information system against which the authorization manager of that information system can test legality of authorizations.

This chapter is organized as follows. Section 6.1 gives a management definition of the fundamental modeling concepts presented in this chapter. Section 6.2 proposes a new architecture of information systems that considers an authorization schema as a basic component of each information system. Section 6.3 presents an object-oriented approach to the conceptual design of authorization schemas. An *authorization interface* collects a set of operations with similar protection requirements from the public interface of objects of a particular type. An *authorization constraint* specifies protection requirements of public operations that belong to a particular authorization interface. An *authorization contract* models an access right. Authorization contracts are first-class citizens, i.e., authorization contracts are objects with structure and behavior like ordinary business objects. Section 6.4 discusses basic principles of the administration of authorizations. Section 6.5 summarizes this chapter.

6.1 Fundamental Modeling Concepts

The core concept presented in this chapter is the *authorization schema*. In data modeling, the design of conceptual schemas is a widely accepted technique (e.g., [BCN92]). A conceptual schema is a semantically rich description of the entities in a particular business area. Information systems use a schema to exclude objects from their databases that do not represent legal entities from a business area. That is, a schema helps to ensure the integrity of information stored within an information system. An authorization schema, being a formal description of the protection requirements of information stored within an information system, could be used to exclude authorizations from the authorization base of information system that are not legal wrt. the protection requirements of an organization. That is, since it avoids that illegal authorizations are issued to actors, an authorization schema helps to avoid unauthorized modifications of information maintained within an information system. In other words, an authorization schema helps to ensure the integrity of information maintained within an information system[1]. Note: An *authorization* is the permit of its holder to invoke one or several operations on one or several objects. Frequently, the terms access right, access rule, authorization rule, permission, or privilege are used synonymously to the term authorization in the literature on security in information systems.

Conceptual schemas are easy to understand by humans but cannot be interpreted by computers. Therefore, conceptual schemas are mapped to logical schemas that can be

[1] Note: There are several definitions of "data integrity". For example, Jajodia [Jaj96] defines integrity as being "concerned with unauthorized modification of information".

interpreted by computers. Traditional information systems, however, do not know the concept of authorization schemas. In such systems the responsibility to determine whether authorizations are legal wrt. to the protection requirements of an organization is with the security administrator(s). We propose a new architecture for information systems considering authorization schemas as a major component of information systems (cf. Section 6.2).

Authorization contract: An *authorization contract* models an authorization (above). An authorization contract is a formal agreement between two or more parties. One or several parties, the suppliers, commit themselves to offer one or several services, i.e., operations from the public interface of the supplier(s), to the remaining parties, the clients. In addition, an authorization contract may state further terms of such an agreement, i.e., conditions that limit the situations in which clients can request the stated services from their suppliers. This means, authorization contracts can model both, unconditional as well as conditional authorizations.

Authorization interface: An *authorization interface* contains of a set of operations from the public interface of objects of a particular type. Operations that belong to the same authorization interface have similar protection requirements (below). The protection requirements of operations that are organized into authorization interfaces are specified using authorization constraints (below). Authorization interfaces extend the basic concept of public interfaces of objects (information-hiding) to the modeling of protection requirements of objects.

Authorization constraint: An *authorization constraint* defines the protection requirements of a set of operations that belong to the same authorization interface f in that it specifies an actor domain of these operations at f. Since authorization constraints are sentences of a deontic constraint language, that offers four deontic modalities, i.e., permission, obligation, discretion, and prohibition, we can give specific connotations to actor domains. For example, associating deontic modality P (permission) with an actor domain D defined at an authorization interface f says that a security administrator *may* authorize actors that are elements of domain D to invoke operations that belong to f. Associating deontic modality O (obligation) with an actor domain D defined at an authorization interface f says that a security administrator *must* authorize actors that are elements of domain D to invoke operations that belong to f.

Authorization manager: An *authorization manager* of an information system mediates all requests of actors and denies those requests for which either no authorization is stored in the authorization base of an information system or for which additional conditions associated with an authorization are not satisfied. In Section 6.2 we propose a new architecture of information systems in which the status of an authorization manager is enhanced in that it is established with an authorization schema (below), which is a formal description of the protection requirements of an organization. It allows an authorization manager to exclude illegal authorizations from an authorization base. Note: In traditional information systems this responsibility is with the security administrators who define authorizations.

Authorization schema: An *authorization schema* is a formal description of the protection requirements (below) of an organization. In other words, an authorization schema is a collection of constraints that act as a filter to exclude illegal authorizations from the authorization base of an information system. We use authorization interfaces (above) to group operations with similar protection requirements and authorization constraints (above) to specify the protection requirements of public operations that are organized into authorization interfaces. We use authorization contracts (above) to model authorizations, i.e., to state which actors actually hold a permit to invoke certain operations on certain objects. An authorization schema consists of a set of authorization interfaces, a set of authorization constraints, a set of authorization-contract types—they model individual structure and behavior of authorization contracts—, and a set of authorization-contract classes—they collect authorization contracts and model collective structure and behavior of authorization contracts.

Protection requirements: The *protection requirements* of an organization are statements that determine to which actors access to resources of the organization may be, must be, may not be, and must not be permitted. When designing an authorization schema (above) for an information system, statements on the protection requirements of information managed in that information system are formalized.

Security: *Security of information* is concerned with ensuring the secrecy, integrity, and availability of information maintained in an information system (e.g., [CFMS94]). In commercial environments ensuring the integrity of information is the primary concern (e.g., [CW87]). To ensure the integrity of information requires that information stored in an information system is correct and complete (e.g., [Jaj96]) and is protected from being modified by unauthorized users (e.g., [San93]).

Summing up, authorization schemas are formal description of the protection requirements of an organization. In fact, an authorization schema is a central description of how authorizations may be granted within an information system. The authorization manager of an information system can use an authorization schema to test the legality of authorizations. This means, an authorization schema acts as a filter to exclude illegal authorizations from the authorization base of an information system. An authorization schema is modeled by a set of authorization interfaces, a set of authorization constraints, a set of authorization-contract types, and a set of authorization-contract classes.

6.2 Information Security in Workflow Systems

Information security is concerned with ensuring the secrecy, integrity, and availability of information maintained in an information system (e.g., [CFMS94, Jaj96, Per94]). To ensure *secrecy* requires to protect information from improper disclosure. To ensure *integrity* requires to maintain only information that is correct and complete (*semantic integrity*) and to protect information from being modified by unauthorized actors (*protection integrity*). To ensure *availability* requires to protect information from improper withholding from authorized actors. As Clark and Wilson stated in their seminal paper [CW87], ensuring integrity of information is the primary concern of information security in commercial environments. Since information is a valuable property of commercial organizations, information that is incorrect, incomplete, or modified by unauthorized users could cause enormous financial losses and thus have severe impact on the competitiveness of organizations.

How to ensure integrity of information in information systems? An *information manager* (cf. Figure 6.1) is a crucial component of each information system. The responsibility of an information manager is to handle requests of actors, i.e., insertion, modification, or deletion of information, and to ensure that only correct and complete information is stored within an information base (cf. semantic integrity, above). An information manager uses an *information schema*, which is a collection of constraints and acts as a filter for incorrect and incomplete information, to test semantic integrity of information. To ensure semantic integrity despite system failures and simultaneous access of multiple users to information, an information manager relies on a recovery manager and a transaction manager, respectively. To ensure protection integrity, information systems must be extended (cf. Figure 6.2). An *authorization manager* mediates all requests of actors and denies those requests for which no appropriate authorization is stored in the authorization base of an information system. To issue and maintain authorizations is within the responsibility of a *security administrator* (SA). An *identification/authentication manager* ensures that log in is denied for unauthorized users. Although the architecture of information systems has been extended to include an authorization manager and an identification/authentication manager, information systems that fall into this category—to the best of our knowledge all existing DBMSs and WFMSs used in commercial environments do fall into this category of information systems—can guarantee protection integrity only partially. In the following, we will investigate the quality of “protection integrity” of information systems in more detail.

Protection integrity has been defined as the quality of an information system to avoid modification of information by unauthorized actors. We identify two distinct but related requirements information systems have to meet to achieve the quality of protection integrity. First, all requests of actors for which no appropriate authorizations are stored in the authorization base of an information system must be denied. We denote this aspect of protection integrity as *authorization-control integrity*. Second, no illegal authorizations are stored in the authorization base of an information system. We denote this aspect of protec-

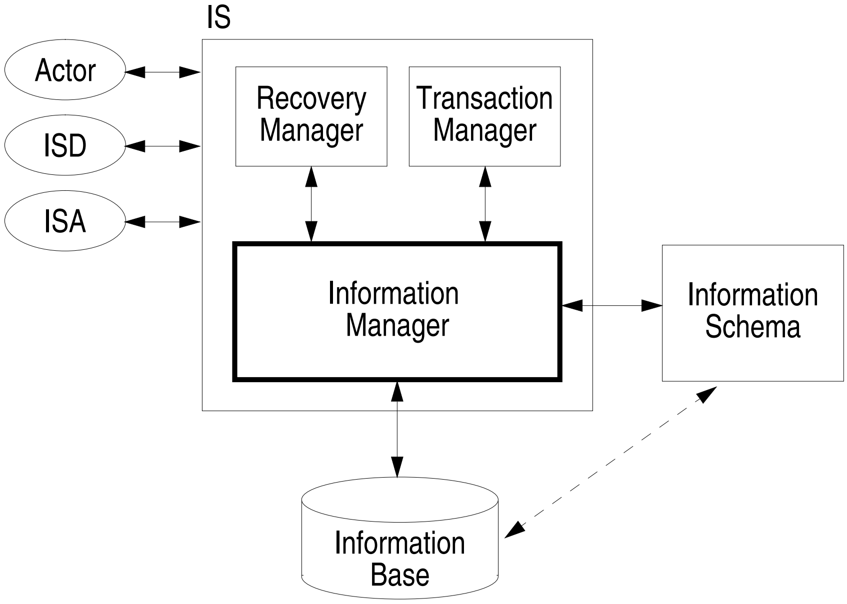


Figure 6.1: Basic architecture of information systems (schematically)

tion integrity as *authorization-state integrity*[2]. Existing DBMSs and WFMSs can ensure authorization-control integrity but they fail to ensure authorization-state integrity, since its the responsibility of a security administrator to determine whether the authorization-state of an information system meets the protection requirements of an organization. This means, unlike an information manager that is able to determine which information is incorrect and incomplete, *an authorization manager is unable to determine which authorizations are illegal wrt. to the protection requirements of an organization.* To guarantee authorization-state integrity, information systems must be extended in that a description of the protection requirements of an organization is made available to the authorization manager (cf. Figure 6.3).

We propose a new architecture of information systems that allows to guarantee authorization-control integrity *and* authorization-state integrity. In the new architecture of information systems (cf. Figure 6.3), an authorization manager uses an *authorization schema*, which is a collection of constraints and acts as a filter for illegal authorizations, to test authorization-state integrity. To design an authorization schema is within the responsibility of a *security designer* (SD).

The benefits of the improved architecture for information systems are: (a) *Division of labour*: Traditionally, security administrators are responsible to issue and revoke authorizations *and* to ensure that the authorization states of an information system are legal wrt. to the protection requirements of an organization. That is, security administrators play a central role in traditional information systems. In the improved setting (cf. Figure 6.3), it is the responsibility of a *security designer* to develop an authorization schema, i.e., to develop

[2]The *authorization state* of an information system is the set of all authorizations stored in its authorization base.

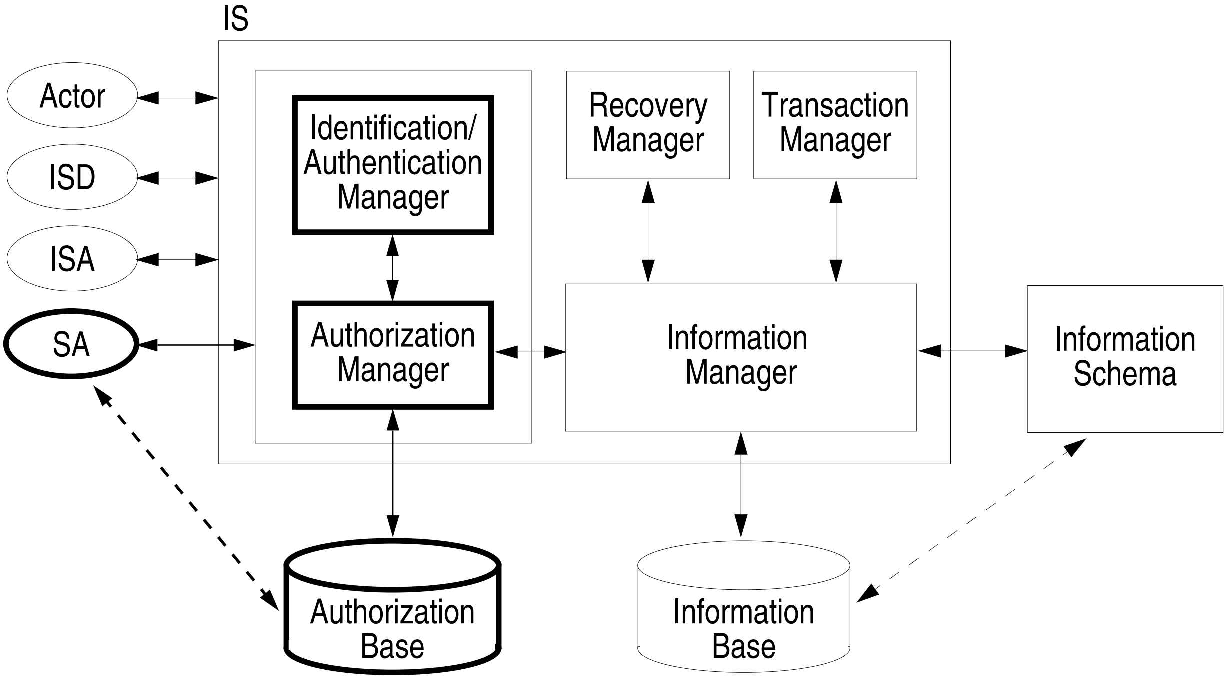


Figure 6.2: Extended architectures of information systems (schematically)

a formal description of the protection requirements of an organization considering selected security policies, e.g., least privilege, separation of duty, and it is the responsibility of an *authorization manager* (cf. Figure 6.3) to test that the authorization states of an information system are legal wrt. to the authorization schema. The responsibility of a *security administrator* is to administer authorizations, i.e., to issue, modify, and revoke authorizations. That is, in the improved setting security administrators require less technical skills. (b) *Assurance*: How well an information system meets the protection requirements of an organization can be tested by analyzing the quality of the authorization schema and does no longer depend on the technical skills of the security administrators of an information system. Note: The quality of an authorization schema depends on the quality of the data model used to define the authorization schema and on the skills and experience of a security designer responsible for the design of the authorization schema. (c) *Administration*: In the improved architecture of information systems (cf. Figure 6.3), administration is a less demanding task and requires less technical skills, i.e., security administrators need not be that expensive specialists. Furthermore, the existence of an authorization schema, which is a central definition of the protection requirements of an information system, against which the legitimacy of the authorization states of an information system can be tested, does encourage decentralization of security administration. (d) *Performance*: The authorization manager has to verify the legitimacy of requests of actors against the authorizations stored in the authorization base of an information system. This means, authorizations that are illegal wrt. to the current protection requirements of an organization downgrade the performance of an information system. The improved architecture of information sys-

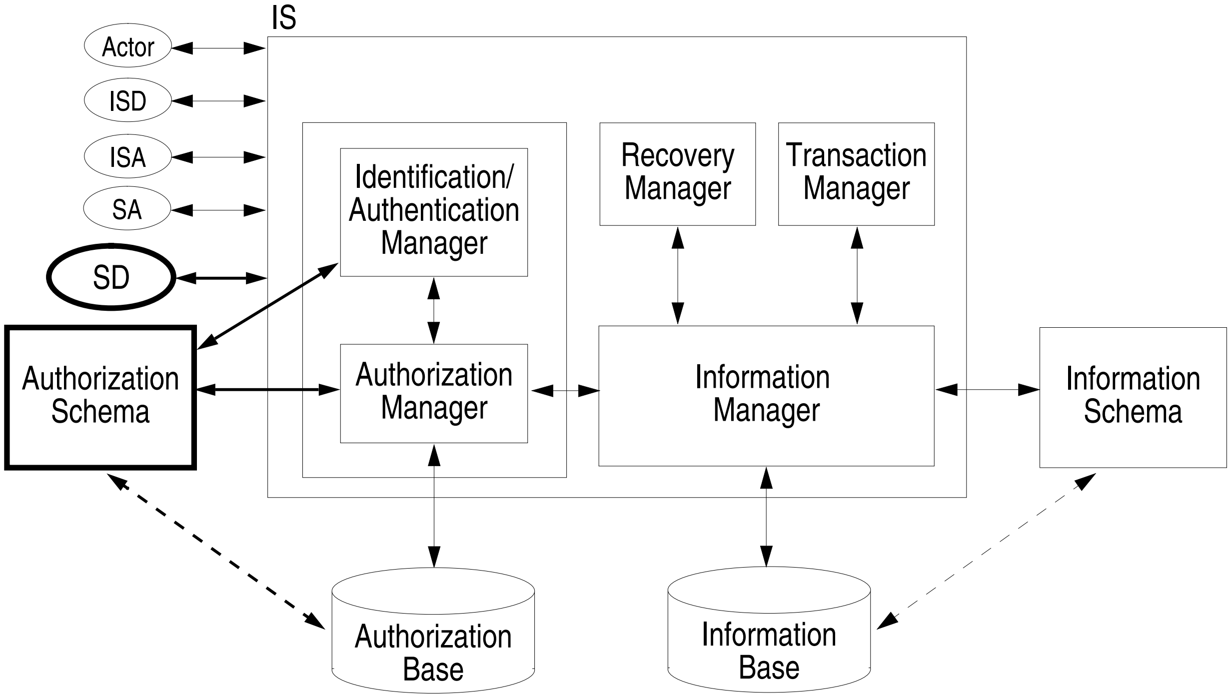


Figure 6.3: Improved architecture of information systems (schematically)

tems helps to avoid such downgrade of performance in that it ensures authorization-state integrity, i.e., no illegal authorizations are stored in an authorization base.

Workflow management systems (WFMSs) are information systems used in commercial and other civil organizations to define, enact, and maintain organizational processes. In such environments a security model is faced the following requirements: (a) The authority of an actor is usually determined via the position(s) the actor occupies within an organization (cf. Chapter 5). (b) Only those authorizations required for the tasks performed by an actor occupying a position are assigned to that position (least-privilege principle). (c) Some tasks might not be performed by the same actors (separation of duties principle). (d) Actors process business cases in that they perform individual steps of organizational processes, they do not manipulate the states of business cases directly (cf. Chapters 3 and 4). Recently one kind of authorization control has emerged that allows to naturally cope with these requirements, role-based access control (RBAC)[3] (e.g., [RBA95]). Note: The concept of positions offered by the basic organization model (cf. Chapter 5) corresponds to the concept of roles in RBAC models.

We aim at providing information security in workflow systems (WFSs). This means, we do apply the improved information-systems architecture to WFSs and present concepts for the conceptual design of semantically rich authorization schemas. Due to the requirements prevailing in commercial environments in which WFSs are used, we employ a role-base authorization control model.

[3]Note: We interpret the acronym RBAC as role-based authorization control.

6.3 Design of Authorization Schemas

An *authorization schema* is a formal description of the protection requirements of a workflow system (WFS). In general, a security analyst (SA) identifies the possible *threats* to which an information system is exposed, e.g., violation of secrecy, integrity, or availability, determines *information accessibility*, i.e., which kind of actors can be authorized to access which information for which purpose, and selects a set of *security policies* (cf. [CFMS94]), e.g., open-world vs. closed-world policy, minimum-privilege vs. maximum-privilege policy, that allow to best meet the protection requirements of the information system[4]. Many researchers agree that violations of integrity are the main threats to which information systems are exposed in commercial environments (e.g., [CW87]). Since information systems that follow a combination of closed-world policy, i.e., only authorized accesses are permitted, and minimum-privilege policy, i.e., actors are only established with authorizations they require to do their job (cf. least-privilege policy, above) best meet the protection requirements of information systems used in commercial environments, we assume such a setting for workflow systems, too. The result of the analysis of the protection requirements of a workflow system is thus a set of informal statements that express which kind of actors can be authorized to perform which worksteps on which kind of business cases (cf. information accessibility, above), e.g., "Only workflow administrators (WFAs) may archive business cases.". The responsibility of a security designer (SD) is to formalize such statements, i.e., to develop an authorization schema, considering the selected security policies.

In this section *we present a conceptual model for the design of authorization schemas for workflow systems*. This conceptual model is a high-level language security designers can use to formalize the protection requirements of a workflow system. It offers different concepts to represent different categories of protection requirements. In the following, we introduce a basic classification of possible protection requirements and show which concepts can be used to model which kind of protection requirements.

1. *Unconditional accessibility*: This kind of protection requirements express that only a certain kind of agents might be authorized to perform certain worksteps on certain kind of business cases, e.g., "Only workflow administrators may archive business cases." (above). Such protection requirements must be considered when authorizations are issued to ensure that no illegal authorizations are stored in an authorization base.

   We use *authorization interfaces* together with *authorization constraints* to model this kind of protection requirements (cf. Section 6.1).

2. *Conditional accessibility*: This kind of protection requirements express constraints that limit access of authorized actors, e.g., "Workflow administrators are not allowed to archive business cases during business hours.". Such constraints must be tested by

[4]Note: Castano *et al.* [CFMS94, pp. 278ff] give a set of criteria that assists security analysts in choosing security policies that best meet the protection requirements of an information system.

the authorization manager of a WFS at the time authorized actors invoke worksteps on business cases.

We use *authorization contracts* to model authorizations (cf. Section 6.1). Constraints that limit access of authorized actors are modeled as properties of authorization contracts. In fact, these constraints are specified as boolean queries using a powerful, graphical query-language presented by Lang *et al.* [LOS96a].

3. *Further protection requirements*: Further protection requirements might concern the administration of authorizations. For example, procedures that must be followed when administering authorizations can be modeled as the behavior of authorizations (cf. Subsection 6.3.3). Protection requirements that determine who is allowed to administer authorizations have impact on the organization schema of an organization (cf. Chapter 5). They could require certain positions to be established and thus have influence on the physical structure as well as on the control structure of organizations.

Summing up, when building an authorization schema, a security designer aims at developing a semantically rich description of the protection requirements identified for a workflow system during requirements analysis that can act as an effective filter for illegal authorizations.

An *authorization schema* consists of a set of authorization interfaces, a set of authorization constraints, a set of authorization-contract types, and a set of authorization-contract classes. An authorization interface groups public operations with similar protection requirements. The protection requirements of public operations that are organized into authorization interfaces are specified by authorization constraints. An authorization-contract type specifies the structure and behavior of its instances, called *authorization contracts.* An authorization-contract class collects a set of authorization contracts of a specific type, the member type of the class. Authorization interfaces restrict the sets of possible instances of authorization-contract types and thus the sets of members of authorization-contract classes to those authorization contracts which represent legal authorizations wrt. the protection requirements of an organization. An *instance* of an authorization schema consists of a set of authorization contracts that represent legal authorizations. Each authorization contract is instance of a particular authorization-contract type and is member of one or several authorization-contract classes. Authorization contracts may be related to and may communicate with other business objects.

Authorization contracts are business objects, too. This means, the concepts presented in Section 3.2 for the conceptual modeling of business objects can be applied in the same way to the modeling of authorization contracts. Below, the concepts presented for the conceptual modeling of business objects are reviewed in the realm of the conceptual modeling of authorization contracts.

The remainder of this section is organized as follows. Subsection 6.3.1 introduces authorization interfaces, Subsection 6.3.2 authorization constraints. Subsection 6.3.3 discusses the modeling of the structure and behavior of authorization contracts as well as the modeling of classes of authorization contracts in the realm of the object-oriented modeling concepts presented in Section 3.2.

6.3.1 Modeling of Authorization Interfaces

An *authorization interface* is a named set of operations from the public interface of objects of a particular type. Operations that belong to the same authorization interface have similar protection requirements (cf. Section 6.1). The protection requirements of a set of operations that belong to an authorization interface f are determined in that actor domains are defined for f. Actor domains of operations that are organized into authorization interfaces are specified using authorization constraints (cf. Subsection 6.3.2). Since authorization constraints are sentences of a deontic constraint language, we can associate different deontic modalities (i.e., permission, obligation, discretion, and prohibition) with actor domains to give actor domains a specific connotation. For example, associating deontic modality *permission* (P) with an actor domain D defined at an authorization interface f says that security administrators *may* authorize actors that are elements of domain D to invoke operations that belong to f. Authorization constraints allow for a flexible modeling of the protection requirements of public operations of objects organized in authorization interfaces.

Authorization interfaces extend the basic object-oriented principle of information-hiding (e.g., [KS96b]) to the modeling of protection requirements of objects. Information-hiding ensures that the states of objects can be manipulated only by invoking semantic operations from well-defined interfaces, called *public interfaces*, objects offer to their environments. Authorization interfaces determine the protection requirements of operations from the public interface of objects ensuring that the states of objects can be manipulated only by properly authorized actors (cf. authorization-state integrity, Section 6.2). In general, the public interface of objects of a particular type O consists of the set of public update activities and the set of public retrieval activities defined at O (cf. Subsection 3.2.2). In particular, *a public interface of business cases of a particular type O consists of a set of worksteps defined at a workflow process according to which business cases of type O could be processed* (cf. Subsection 4.4.4). In this thesis we show how to model the protection requirements of business cases processed in a WFMS that adheres to a two-schema architecture (cf. Section 4.2). Nevertheless, these concepts are general enough to be applied in the same way to the modeling of the protection requirements in object-oriented information systems in general. Note: The two-schema architecture of workflow systems (cf. Section 4.2) allows that several workflow processes are defined on the same business process. These different workflow processes may exist at the same time or at different times. This means, there could be defined a set of public interfaces for business cases of a particular type. For sake of simplicity of the subsequent discussion, however, we assume that for each type of business cases there is exactly one type of workflow process that defines the public interface of these business cases.

An *authorization interface* of business cases of a particular type is a named set of worksteps with similar protection requirements from the public interface of these business cases. Since not all worksteps from the public interface of business cases of some type will have similar protection requirements, several authorization interfaces will be defined for business cases of a particular type. These *authorization interfaces are organized in a specialization hierarchy.* A subordinate authorization interface contains a proper subset of the worksteps of its superordinate authorization interface. Authorization interfaces f_i that are subordinate to the same authorization interface f must be mutually exclusive. This means, if a workstep s belongs to two authorization interfaces f_1 and f_2, then either f_1 is subordinate to f_2 or vice versa. A subordinate authorization interface inherits actor domains from its superordinate authorization interface. Inherited actor domains may be specialized at subordinate authorization interfaces (cf. authorization constraints, Subsection 6.3.2).

To reduce the overall effort of modeling the protection requirements of business cases, we offer a predefined set of authorization interfaces for each public interface of business cases. A predefined authorization interface All is the root of each specialization hierarchy of authorization interfaces. Its name indicates that all worksteps of a public interface belong to this authorization interface. In addition, one authorization interface is defined for each individual workstep s that belongs to a public interface. These authorization interfaces are named after the individual worksteps to indicate that they consist of a single workstep. By default, the predefined authorization interface All defined for a public interface p is direct superordinate to all authorization interfaces defined for individual worksteps that belong to p. The task of security designers is to define authorization interfaces that are inner nodes in a specialization hierarchy of authorization interfaces, i.e., to repeatedly group worksteps with similar protection requirements into authorization interfaces. Since subordinate authorization interfaces inherit actor domains from their superordinate authorization interfaces, a well-designed specialization hierarchy allows for an efficient sharing of the specifications of the protection requirements of workflow operations and reduces the overall set of authorization constraints to be specified (cf. Subsection 6.3.2).

Graphically, an authorization interface is shown in a workflow-process diagram, which is slightly extended for this purpose: A specialization hierarchy of authorization interfaces defined on the worksteps of a workflow process is depicted in the upper-left corner of a workflow-process diagram showing that workflow process. If an authorization interface f is shown, all worksteps that belong to f are highlighted, i.e., the graphical symbols representing these worksteps are filled in light-grey color. Moreover, the name of f and the actor domains defined at f are shown in in the upper-right corner of a workflow-process diagram. The name of the depicted authorization interface and the appropriate type expression are marked with the keywords selected authorization interface and actor domains, respectively.

Example: Figure 6.4 shows an extended workflow-process diagram of a simple room-reservation workflow process. Besides the set of predefined authorization interfaces, authorization interfaces Archive and HandleReservations have been defined. Authorization interface Archive contains worksteps NegArchive and PosArchive. That is, authorization interface Archive is superordinate to the predefined authorization interfaces NegArchive and PosArchive. Authorization interface HandleReservations contains worksteps Issue, CheckIn, WaivePayment, and CheckOut_Charge_Bill&Pay.

Figure 6.4 depicts the actor domain defined at authorization interface Archive. A security administrator *may* issue authorizations to invoke worksteps NegArchive and PosArchive to workflow administrators. That is, deontic modality *permission* (P) has been associated with actor domain WFA.

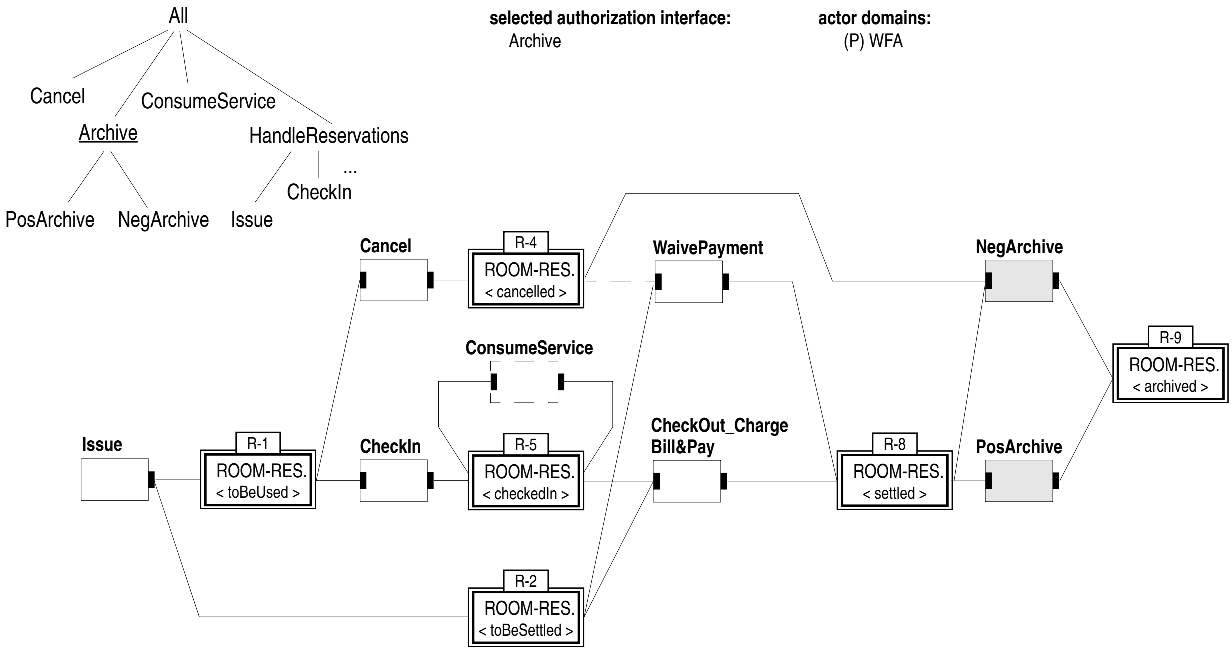


Figure 6.4: Authorization interface Archive of room reservations

We conclude this section on authorization interfaces with a short discussion on the potential of the modeling of authorization interfaces in a two-schema architecture. Remember that in a two-schema architecture the behavior of business cases, i.e., an organizational process, is modeled in two distinct but related schemas. The two schemas are a business-process schema and a workflow-process schema (cf. Section 4.2). The business-process schema of an organizational process abstracts from knowledge on that organizational process expressed in form of internal business rules. Since this knowledge is subject to frequent changes caused by the dynamics of businesses, business-process schemas may serve as a fundament for different versions of workflow-process schemas. This means, the two-schema architecture allows to share knowledge on organizational processes expressed in form of external business rules between different versions of workflow processes. This knowledge is shared between different versions of a workflow-process schema in that they are built on the same business-process schema.—Note: A business-process schema acts as a kind of correctness criterion for all workflow-process schemas that are built on that business-process schema, since organizational settings must not violate external business rules specified in business-process schemas.—We claim that the basic concept of the two-schema architecture, i.e., the sharing of knowledge on organizational processes into external knowledge and internal knowledge, could be extended to the modeling of protection requirements of organizational processes, too.

We could model authorization interfaces on the workflow-process interface *and* on the business-process interface of business cases of a particular type. This approach requires to separate knowledge on the protection requirements of business cases into external business rules, i.e., protection requirements based on natural facts or law, and internal business rules, i.e., organizational settings that further restrict external business rules. Separating knowledge on the protection requirements of business cases into external and internal knowledge offers the same advantages to the modeling of protection requirements as it offers to the modeling of organizational processes. First, authorization interfaces that have been defined on the business-process interface of business cases act as a consistency criteria for the authorization interfaces to be defined on a workflow-process interface of business cases. For example, assume and actor domain D together with the deontic modality *permission* (**P**) has been defined for some business-process activity a_i. Consequently, only actors that are elements of actor domain D may be authorized to perform worksteps that invoke business-process activity a_i. Second, actor domains that are defined for business-process activities restrict the grouping together of these activities into worksteps. For example, two business-process activities a_i and a_j may only be grouped together into a workstep A_k if the actor domains defined for a_i and a_j allow that the same kind of actors may be authorized to invoke both activities, a_i as well as a_j, via A_k.

Example: The actor domains of business-process activities a_1, a_2, and a_3 have been defined at authorization interfaces a_1, a_2, and a_3, respectively, by type expressions X, $Y \vee Z$, and Z, respectively (cf. Figure 6.5). We assume that the deontic modality *permission* has been associated with all actor domains defined at authorization interfaces a_1, a_2, and a_3. Since business-process activities may only be invoked via worksteps, the actor domains defined for business-process activities a_1, a_2, and a_3 say that only actors of type X may be authorized to perform worksteps that invoke activity a_1, only actors of type Y and actors of type Z to perform worksteps that invoke activity a_2, and only actors of type Z to perform worksteps that invoke activity a_3. The workflow process defined at type W_1 (cf. Figure 6.5(a)) violates the protection requirements of business-process activity a_1 in that it allows actors of type S to be authorized to perform workstep A_1 that invokes a_1. The workflow process defined at type W_2 (cf. Figure 6.5(b)) violates the protection requirements of business-process activity a_2 in that it allows actors of type X to be authorized to perform workstep $A_1\&A_2$ that invokes a_2. In fact, grouping business-process activities a_1 and a_2 into a single workstep causes in any case a violation of the protection requirements defined for a_1 or a_2, or both, since the actor domains of a_1 and a_2 are mutually exclusive.

Note: Violations of the protection requirements of activities of the underlying business process are indicated by an arrow in Figure 6.5. Furthermore, actor domains of activities and worksteps are shown in Figure 6.5 by directly annotating these activities and worksteps.

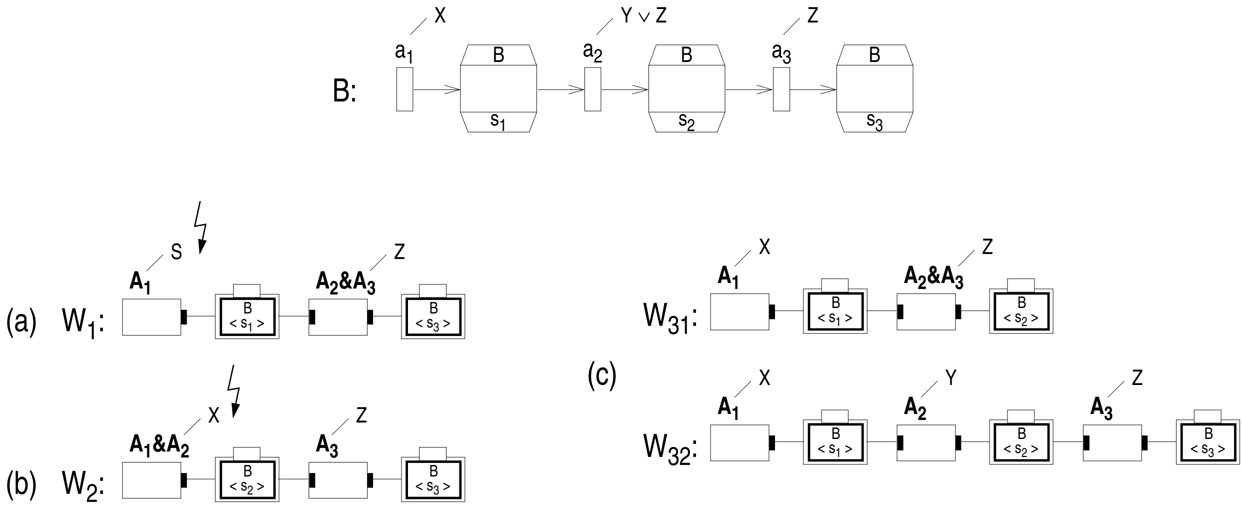


Figure 6.5: Authorization interfaces in a two-schema architecture (schematically)

Second, authorization interfaces that have been defined on the business-process interface of business cases allow to share protection requirements between different versions of workflow processes. As a consequence, the protection requirements of business cases are adhered to despite changes to workflow-process schemas of organizational processes.

Example: Authorization interfaces a_1 to a_3 defined on activities a_1 to a_3 at business-process type B (cf. Figure 6.5) ensure that the knowledge that actors of type Y may be authorized to perform worksteps that invoke business-process activity a_2 won't get lost between different versions of workflow processes defined on on the business process defined at B (cf. Figure 6.5(c)).

Summing up, authorization interfaces extend the basic concept of information-hiding to the modeling of protection requirements of objects. However, there is a clear separation between the public interface of an object and the authorization interfaces of an object. This clear separation allows for a separate modeling and maintenance of the semantic aspects and the security aspects of objects. This means, a change in the protection requirements of objects would not cause any changes to the public interfaces of objects. Finally, the potential of the modeling of authorization interfaces in a two-schema architecture has been discussed. However, we do not further elaborate this aspect of the design of authorization schemas in this thesis.

6.3.2 Modeling of Authorization Constraints

An *authorization constraint* (AC) specifies an actor domain of a set of public operations that belong to the same authorization interface (cf. authorization interfaces, Subsection 6.3.1). Authorization constraints allow for a flexible and precise modeling of the protection requirements of public operations of objects if these operations are organized into authorization interfaces. Authorization constraints are *deontic constraints* representing normative statements that apply to the authorization state in a business area. We use the language L_{auth} to specify authorization constraints. It is based on a deontic constraint language presented by Wieringa *et al.* [WWMD91]. L_{auth} offers four deontic modalities, i.e., permission, obligation, discretion, and prohibition, with which we can give specific connotations to actor domains specified at authorization interfaces. Since we assume a closed-system security policy, however, we are primarily interested in two deontic modalities: permissions ($\mathbf{P}$) and obligations ($\mathbf{O}$). For example, associating deontic modality $\mathbf{P}$ (permission) with an actor domain D defined at an authorization interface f says that a security administrator *may* authorize actors that are elements of domain D to invoke operations that belong to f. In this case, we denote D as a domain of potentially authorized actors (or potential actor-domain) of operations that belong to f. Associating deontic modality $\mathbf{O}$ (permission) with an actor domain D defined at an authorization interface f says that a security administrator *must* authorize actors that are elements of domain D to invoke operations that belong to f. In this case, we denote D as a domain of obligatorily authorized actors (or obligatory actor-domain) of operations that belong to f.

Example: AC 6.1 says that receptionists *may* hold an authorization to invoke worksteps that belong to authorization interface WaivePayment on room reservations processed in a reservation workflow-process. This means, AC 6.1 defines object class *Receptionists* as a domain of potentially authorized actors of workstep WaivePayment, which is the only workstep that belongs to authorization interface WaivePayment (cf. Figure 6.4). Note: $E_A(s,r,c)$ denotes a boolean predicate that evaluates to 'true' if an authorization exists for a supplier s, a service r, and a client c, and evaluates to 'false' otherwise (cf. definition of language L_{auth}, below). Consequently, the expression $P(E_A(s,r,c)) \Rightarrow c \in \mathit{Receptionists}$ says that the existence of an authorizations $< s,r,c >$ for given s and r is permitted if c is member of class *Receptionists*. Note: Public is a pseudo class. All position and agent objects are considered to be members of that pseudo class. The symbol $\in$ denotes the *memberOf* operator.

$$\forall s \in \mathit{Reservation\text{-}WFs}, r \in \mathit{WaivePayment}, c \in \mathit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \mathit{Receptionists}) \tag{6.1}$$

Example: AC 6.2 says that senior receptionists *must* hold an authorization to invoke worksteps that belong to authorization interface WaivePayment on room reservations processed in a reservation workflow-process. This means, AC 6.2 defines object class *SeniorReceptionists* as a domain of obligatorily authorized actors of workstep WaivePayment, which is the only workstep that belongs to authorization interface WaivePayment (cf. Figure 6.4).

$$\forall s \in \mathit{Reservation\text{-}WFs}, r \in \mathit{WaivePayment}, c \in \mathit{Public}$$
$$(\mathbf{O}(E_A(s,r,c)) \Rightarrow c \in \mathit{SeniorReceptionists}) \tag{6.2}$$

As mentioned before, authorization constraints are normative statements that apply to the authorization state in a business area. Since normative statements can be violated in a business area, an information system must allow violations of authorization constraints. This means, a violation of some authorization constraint does not cause an information system to be in an inconsistent state. Nevertheless, it is important to offer mechanisms that allow to detect violations of authorizations constraints and correct them. For example, active database capabilities (e.g., [Lan97, Obe98]) could be used to model proper reactions to violations of authorization constraints. If a violation is detected, an event will be raised. Event-condition-action rules can be defined to initiate proper corrective actions. This semantic of authorization constraints allows to naturally cope with violations of authorization-state integrity within an information system (cf. Section 6.2). For example, such violations could occur if new positions are established within an organization, e.g., obligatory authorizations may be missing for new positions. However, a thorough investigation of the detection and correction of violations of authorization constraints is beyond the scope of this thesis and is left for future work.

Authorization constraints allow to express very precisely the semantics of the protection requirements of worksteps that are organized into authorization interfaces. Remember that a domain D of potentially authorized actors defined at an authorization interface f does not say that all actors that belong to D are authorized to invoke worksteps belonging to f. Rather, it says that for all actors that belong to D there may exist an authorization to invoke worksteps belonging to f (cf. AC 6.1). In fact, the different deontic modalities offered by L_{auth} allow to distinguish actor domains into permitted domains, obligatory domains, discretionary domains, and forbidden domains. Consequently, the concept of authorization constraints is general enough to express protection requirements in open-world systems *and* in closed-world systems. As mentioned before, we assume a closed-world security policy and are thus mainly interested in the modeling of domains of potentially authorized actors and domains of obligatorily authorized actors (cf. ACs 6.1 and 6.2). In the sequel, we use the terms "potential actor-domain" and "domain of potentially authorized actors", as well as the terms "obligatory actor-domain" and "domain of obligatorily authorized actors" interchangeably.

Similar as we model type domains *and* class domains for relationships of objects (cf. Subsection 3.2.1), we model type domains and class domains of potentially authorized actors and obligatorily authorized actors of public operations that are organized into authorization interfaces, too. However, specifying type-level actor domains (class-level actor domains) at an authorization interface by single types (classes) would be too rigid to adequately model the protection requirements of public operations in many cases. That is, frequently different kinds of actors may (or must) hold authorizations to request the same services from the same kind(s) of objects. As a consequence, we extend the basic concepts of type domains and class domains for the modeling of actor domains of public operations. We model type-level actor domains and class-level actor domains of public operations by type expressions and class expressions, respectively. Note: In authorization constraints, type domains of services r are defined by types of worksteps, class domains by authorization interfaces. We assume that all worksteps are instances of a predefined type WORKSTEP.

Authorization interface All defined at some workflow-process type W corresponds to a class of worksteps that contains all worksteps defined at W.

Example: Only agents that are instances of type GUEST and positions that are instances of type RECEPTIONIST or type WFA (workflow administrators), are *permitted* to hold authorizations to invoke worksteps on room reservations processed in a reservation workflow-process (cf. Figure 6.4). Note: PUBLIC is a pseudo type. All position and agent objects are considered to be instances of PUBLIC. The symbol $\odot$ denotes the *instanceOf* operator.

$$\forall s \odot \mathit{RESERVATION\text{-}WF}, r \odot \mathit{WORKSTEP}, c \odot \mathit{PUBLIC}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \odot \mathit{GUEST} \vee \mathit{RECEPTIONIST} \vee \mathit{WFA}) \tag{6.3}$$

Example: Only guests that are members of class VIP-Guests and receptionists that are members of class Receptionists (the base class of position type RECEPTIONIST) are *permitted* to hold an authorization to invoke workstep Cancel on room reservations. Note: Workstep Cancel is the only workstep that belongs to authorization interface Cancel (cf. Figure 6.4).

$$\forall s \in \mathit{Reservation\text{-}WFs}, r \in \mathit{Cancel}, c \in \mathit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \mathit{VIP\text{-}Guests} \cup \mathit{Receptionists}) \tag{6.4}$$

A class-level actor domain defined at an authorization interface f can express the protection requirements of public operations that belong to f more precisely than a type-level actor domain could, since classes allow for a more fine-grained classification of actors than types do.

Example: AC 6.4 says that only VIP guests and receptionists are *permitted* to hold an authorization to invoke workstep Cancel on room reservations processed in a reservation workflow-process of a hotel.—Note: Actually, a conditional authorization will be issued to VIP guests such that each VIP guest may invoke workstep Cancel only on reservations issued for her or him.—This semantics could not be expressed by AC 6.3, since the basic model does not allow to define a type VIP-GUEST as subtype of type GUEST nor to define a type of worksteps CANCEL as subtype of predefined type WORKSTEP.

Authorization constraints allow to specify *necessary domains* as well as *sufficient domains* of actors of public operations organized in authorization interfaces. Specifying a necessary actor-domain D_{nec} at an authorization interface f says that actors requesting authorizations for operations that belong to f must belong to D_{nec}. Since necessary conditions inherit downwards conjunctively in a taxonomy (e.g., [WWMD91]), a necessary actor-domain D_{nec} defined at an authorization interface f can be strengthened at a subordinate authorization interface f' of f to domain D'_{nec}, where $D_{nec} \supset D'_{nec}$. Specifying a sufficient actor-domain D_{suff} at an authorization interface f says that all actors that belong to D_{suff} can obtain authorizations for operations that belong to f. Since sufficient conditions inherit downwards disjunctively in a taxonomy (e.g., [WWMD91]), a sufficient actor-domain D_{suff} defined at an authorization interface f can be weakened at a subordinate authorization interface f' of f to domain D'_{suff}, where $D_{suff} \subset D'_{suff}$. This means, necessary actor-domains defined at authorization interfaces become stronger as we go down in a specialization hierarchy of authorization interfaces, sufficient actor-domains become weaker. Note: Specifying a sufficient actor-domain D_{suff} at an authorization interface f allows to enforce actor domain D_{suff} at subordinate authorization interfaces f_i of f in a specialization hierarchy of authorization interfaces.

Example: AC 6.5 specifies class WfAdmins, the base class of type WFA, as a sufficient actor-domain for worksteps that belong to authorization interface Archive. Authorization interfaces NegArchive and PosArchive, which are subordinate to authorization interface Archive, inherit sufficient actor-domain WfAdmins. They may weaken the inherited actor domain but may not strengthen it. As a consequence, AC 6.6 is in conflict with AC 6.5 since it aims at restricting the sufficient actor-domain that authorization interface NegArchive inherits from authorization interface Archive to class SeniorWFAs, which is a subclass of class WfAdmins.

$$\begin{aligned}&\forall s \in \mathit{Reservation\text{-}WFs}, r \in \mathit{Archive}, c \in \mathit{Public}\\&\quad (c \in \mathit{WfAdmins} \Rightarrow \mathbf{P}(E_A(s,r,c)))\end{aligned} \tag{6.5}$$

$$\begin{aligned}&\forall s \in \mathit{Reservation\text{-}WFs}, r \in \mathit{NegArchive}, c \in \mathit{Public}\\&\quad (\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \mathit{SeniorWFAs})\end{aligned} \tag{6.6}$$

The remainder of this subsection is organized as follows. First, we define the syntax of L_{auth}, the deontic constraint language used to specify authorization constraints, and give a complete modeling example. Second, we discuss inheritance and specialization of authorization constraints. Third, we discuss the graphical specification of authorization constraints. We conclude this subsection with a short discussion on how to enforce authorization constraints.

$\mathbf{L}_{auth}$ – A Language to Specify Authorization Constraints

L_{auth} is a deontic constraint language used to specify authorization constraints. As mentioned before, L_{auth} is based on a language for the specification of deontic constraints presented in [WWMD91].

- Letters s, r, and c are object variables ranging over *type domains* (type level) or *class domains* (class level) of suppliers, services, and clients, respectively, of possible authorizations[5]. Letter p is used as a metavariable ranging over object variables s, r, and c. Note: Type domains of services r are defined by types of worksteps, class domains by authorization interfaces. As mentioned before, we assume that all worksteps are instances of a predefined type WORKSTEP and that authorization interface All defined at some workflow-process type W corresponds to a class of worksteps that contains all worksteps defined at W.

- Letters S, R, and C (possibly indexed) are domain variables ranging over the *type domains* or *class domains* of suppliers, services, and clients, respectively. Letter P (possibly indexed) is a metavariable ranging over domain variables S, R, and C.

- PUBLIC (Public) denotes a pseudo type (pseudo class). All agent and position objects are considered to be instances of PUBLIC (be members of Public).

- Symbol $\odot$ denotes the instanceOf operator, symbol $\in$ the memberOf operator.

- Type domains are modeled by type expressions, class domains by class expressions. Type expressions can be built from single types applying repeatedly type operator $\vee$ (logical "or"), class expressions can be built from single classes applying repeatedly class operator $\cup$ (union). Note: We write $p \odot P_1 \vee P_2$ ($p \in P_1 \cup P_2$) as a shorthand for $p \odot P_1 \vee p \odot P_2$ ($p \in P_1 \vee p \in P_2$).

- The set of deontic modalities $\Delta := \{\mathbf{P}, \mathbf{O}, \mathbf{D}, \mathbf{F}\}$, with deontic modalities $\mathbf{P}$ (permission), $\mathbf{O}$ (obligation), $\mathbf{D}$ (discretion), $\mathbf{F}$ (prohibition). The variable $\delta \in \Delta$ ranges over deontic modalities P, O, D, and F.

- The boolean predicate $E_A(s, r, c)$ evaluates to 'true' for a given supplier s, a service r, and a client c if there exists an entry for s, r, and c in access matrix A, it evaluates to 'false' otherwise. Access matrix A is a system maintained class of tuples $< s, r, c, a >$, where s identifies a supplier, r a service, and c a client of an authorization, and a identifies an authorization contract (cf. Subsection 6.3.3). When a security administrator issues (revokes) an authorization $< s, r, c >$ (cf. issuing (revoking) authorizations in the context of authorization contracts, Subsection 6.3.3) for a supplier s, a service r, and a client c, an entry $< s, r, c, a >$ is inserted into (removed from) access matrix A, where a is the object identifier of that authorization contract representing authorization $< s, r, c >$. Note: The authorization manager of an information system (cf. Section 6.2) can use the boolean predicate $E_A(s, r, c)$ when validating accesses of actors.

  The boolean predicate $E_A(s, r, c)$ abstracts from types of authorization contracts, making the modeling of authorization constraints independently from the modeling of authorization contracts.

[5]Note: To avoid confusion with the terminology of object-oriented modeling, we use the terms *supplier*, *service*, and *client* in place of the terms *object*, *access type*, and *subject*, respectively, which are commonly used in the database security area to denote the participants of an authorization.

Authorization constraints are formulas in L_{auth} built using '$\forall$', '$\vee$', '$\cup$', '$\odot$', '$\in$', '$\Rightarrow$', deontic modalities $\delta \in \Delta$, and the existence predicate $E_A(s, r, c)$. Typical sentences of L_{auth} are:

$$\forall s \odot S, r \odot R, c \odot C(\delta(E_A(s, r, c)) \Rightarrow c \odot C')$$

$$\forall s \odot S, r \odot R, c \odot C(c \odot C' \Rightarrow \delta(E_A(s, r, c)))$$

$$\forall s \in S, r \in R, c \in C(\delta(E_A(s, r, c)) \Rightarrow c \in C')$$

$$\forall s \in S, r \in R, c \in C(c \in C' \Rightarrow \delta(E_A(s, r, c)))$$

As mentioned before, we model type-level actor domains *and* class-level actor domains of public operations at authorization interfaces. A type-level actor domain defined at an authorization interface f determines the basic protection requirements of public operations that belong to f. A class-level actor domain defined at f may further specialize the protection requirements of public operations that belong to f, i.e., allows to specify the protection requirements of public operations in more detail, since classes allow for a more fine-grained classification of objects than types do. If class-level actor domains are not explicitly defined at an authorization interface f, class-level actor domains at f are implicitly defined by class expressions that consists of the base classes of the types of the corresponding type expressions. In fact, a class-level authorization constraint g_c is derived from a type-level authorization constraint g_t that defines a type-level actor domain of public operations that belong to an authorization interface f in that (a) the name of each type O in g_t is replaced by the name of the base class of O in g_c and (b) each occurrence of the instanceOf operator (symbol $\odot$) in g_t is replaced by an occurrence of the memberOf operator (symbol $\in$) in g_c. Note: In the following, we specify only class-level authorization constraints to keep examples small and understandable.

Example: The following set of authorization constraints determine the protection requirements of worksteps offered by a basic room-reservation workflow process (cf. Figure 6.4).

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{All}, c \in \textit{Public} \\ (\mathbf{P}(E_A(s, r, c)) \Rightarrow c \in \textit{Guests} \cup \textit{Receptionists} \cup \textit{WfAdmins}) \tag{6.7}$$

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{Archive}, c \in \textit{Public} \\ (\mathbf{P}(E_A(s, r, c)) \Rightarrow c \in \textit{WfAdmins}) \tag{6.8}$$

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{Archive}, c \in \textit{Public} \\ (c \in \textit{WfAdmins} \Rightarrow \mathbf{P}(E_A(s, r, c))) \tag{6.9}$$

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{Cancel}, c \in \textit{Public} \\ (\mathbf{P}(E_A(s, r, c)) \Rightarrow c \in \textit{VIP-Guests} \cup \textit{Receptionists}) \tag{6.10}$$

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{ConsumeServices}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{Guests}) \tag{6.11}$$

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{HandleReservations}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{Receptionists}) \tag{6.12}$$

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{WaivePayment}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{SeniorReceptionists}) \tag{6.13}$$

Inheritance of Authorization Constraints

Authorization constraints support *inheritance*. An authorization constraint specified on domains S, R, and C defines a constraint on all suppliers, services, and clients that are elements of S, R, and C, respectively[6]. This means, it specifies also a constraint on elements of domains S', R', and C' that specialize domains S, R, and C, respectively (cf. comment on specialization of type domains and class domains, below). This corresponds to the theorem of *basic inheritance* stated in [WWMD91]. It says that properties (including constraints) inherit downwards a taxonomy.

Example: AC 6.7 is inherited by all subdomains of supplier domain Reservation-WFs, is inherited by all subdomains of service domain All, and is inherited by all subdomains of client domain Public. AC 6.14 is an example for inheritance along the supplier domain, AC 6.15 for inheritance along the service domain, and AC 6.16 for inheritance along the client domain. Assume that VIP-Reservation-WFs is a subclass of Reservation-WFs. Note: AC 6.16 makes it obvious that—at least according to the current authorization policy—secretaries may not be authorized to invoke worksteps on room reservations in a reservation workflow-process that belong to class Reservation-WFs.

$$\forall s \in \textit{VIP-Reservation-WFs}, r \in \textit{All}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{Guests} \cup \textit{Receptionists} \cup \textit{WfAdmins}) \tag{6.14}$$

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{Archive}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{Guests} \cup \textit{Receptionists} \cup \textit{WfAdmins}) \tag{6.15}$$

$$\forall s \in \textit{Reservation-WFs}, r \in \textit{All}, c \in \textit{Secretaries}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{Guests} \cup \textit{Receptionists} \cup \textit{WfAdmins}) \tag{6.16}$$

Specialization of Authorization Constraints

Authorization constraints support *specialization*. Necessary domain restrictions defined on certain domains of suppliers, services, and clients may be strengthened for subdomains

[6]Note: An object is element of an object type o if it is (direct or indirect) *instance* of o, it is element of an object class c if it is *member* of c.

of that domains of suppliers, services, and clients, sufficient domain restrictions may be weakened. The interdependencies between necessary and sufficient domain restrictions have been discussed above (cf. ACs 6.5 and 6.6).

Given authorization constraints g_{nec} and g_{suff}, h_{nec} and h_{suff}, the set of operators $\Theta := \{\odot, \in\}$, the operator $\theta \in \Theta$. the set of deontic modalities $\Delta := \{\mathbf{P}, \mathbf{O}, \mathbf{D}, \mathbf{F}\}$, and the deontic modality $\delta \in \Delta$ are given, where θ and δ are fixed for pairs of authorization constraints g_{nec} (g_{suff}) and h_{nec} (h_{suff}).

$$\begin{array}{ll}
(g_{nec}) & \forall s\,\theta\,S_1, r\,\theta\,R_1, c\,\theta\,C_1\ (\delta(E_A(s,r,c)) \Rightarrow c\,\theta\,C_1') \\
(h_{nec}) & \forall s\,\theta\,S_2, r\,\theta\,R_2, c\,\theta\,C_2\ (\delta(E_A(s,r,c)) \Rightarrow c\,\theta\,C_2') \\
(g_{suff}) & \forall s\,\theta\,S_1, r\,\theta\,R_1, c\,\theta\,C_1\ (c\,\theta\,C_1' \Rightarrow \delta(E_A(s,r,c))) \\
(h_{suff}) & \forall s\,\theta\,S_2, r\,\theta\,R_2, c\,\theta\,C_2\ (c\,\theta\,C_2' \Rightarrow \delta(E_A(s,r,c)))
\end{array}$$

An authorization constraint h_{nec} (h_{suff}) defined on domains of suppliers, services, and clients S_2, R_2, and C_2, respectively, specializes an authorization constraint g_{nec} (g_{suff}) defined on domains S_1, R_1, and C_1, if and only if

1. S_2 is identical to S_1 or S_2 specializes S_1,
2. R_2 is identical to R_1 or R_2 specializes R_1,
3. C_2 is identical to C_1 or C_2 specializes C_1,
4. S_2 is different from S_1, or R_2 is different from R_1, or C_2 is different from C_1, and
5. C_2' specializes C_1' (necessary domain constraints) or C_1' specializes C_2' (sufficient domain constraints).

A type domain (class domain) consistently specializes another type domain (class domain) if the following consistency rules are adhered to: (a) *Specialization of type domains*: A type domain P_2 specializes another type domain P_1, written as $P_2 < P_1$, if the type expression that denotes P_2 consists of a proper subset of the types of that type expression that denotes P_1. For example, $P_2 := O_1 \vee O_2$ specializes $P_1 := O_1 \vee O_2 \vee O_3$, where O_1, O_2, and O_3 are types. (b) *Specialization of class domains*: A class domain P_2 specializes another class domain P_1, written as $P_2 \subset P_1$, if the class expression $\hat{c_2}$ that denotes P_2 consists of a proper subset of the classes of that class expression $\hat{c_1}$ that denotes P_1. In addition, $\hat{c_2}$ may replace any class c of $\hat{c_1}$ by a set of subclasses of c. For example, $P_2 := c_1 \cup c_{21} \cup c_{22}$ specializes $P_1 := c_1 \cup c_2$, where c_1 and c_2 are classes of objects of type O_1 and O_2, respectively, and classes c_{21} and c_{22} are direct subclasses of class c_2, where neither c_{21} is a subclass of c_{22} nor c_{22} a subclass of c_{21}. Note: A legal class expression must not contain two classes c_1 and c_2, where either c_2 is a direct or indirect subclass of c_1 or vice versa.

Note: Whereas type domains of services are defined by types of worksteps, *class domains of services are defined by authorization interfaces.* A class domain of services R_2 specializes another class domain of services R_1, if the authorization interface f_2 that denotes R_2 is subordinate to that authorization interface f_1 that denotes R_1. This means, authorization interfaces are treated like classes of worksteps and subclass-superclass relationships between these classes are defined by the hierarchical organization of authorization interfaces (cf. authorization interfaces, Subsection 6.3.1).

Example: ACs 6.8 to 6.12 specialize AC 6.7. For example, AC 6.8 specializes the inherited client domain $Guests \cup Receptionists \cup WfAdmins$ (cf. AC 6.16) to *WfAdmins*. AC 6.13 specializes AC 6.12.

Specifying authorization constraints in a taxonomic network has the advantage that constraints can be shared among elements in that network in a disciplined way. For example, following the rule of inheritance, a constraint defined on certain domains of suppliers, services, and clients is shared by all their subdomains, too. Note: In the basic model, the modeling of taxonomic networks of suppliers and clients is limited to the modeling of class hierarchies, since types may not be organized into type hierarchies in the basic conceptual model. The definitions given in this section, however, are general enough to cope with this future extension (cf. Chapter 7). Only the definition of the specializes relationship between type domains and between class domains must be re-defined in the advanced conceptual model supporting type hierarchies.

Graphical Specification of Authorization Constraints

Authorization constraints specify actor domains of public operations that are organized into authorization interfaces. In Subsection 6.3.1 the graphical representation of authorization interfaces has been discussed. Workflow-process diagrams have been extended for this purpose. In fact, the extended form of workflow-process diagrams presented in Subsection 6.3.1 does already allow for a graphical specification of authorization constraints. We annotate actor domains specified for a certain authorization interface with the corresponding deontic modalities, i.e., permission (**P**) and obligation (**O**), to distinguish domains of potentially authorized actors from domains of obligatorily authorized actors, respectively, and with keywords nec or suff to distinguish necessary from sufficient domains. To allow for a graphical specification of class-level authorization constraints, workflow-process diagrams must be further extended to show the names of that classes of workflow objects for which class-level actor domains are specified at the depicted authorization interfaces. In the case a class-level authorization constraint is depicted, we show the name of the respective class of workflow objects in the upper-right corner together with the name of the depicted authorization interface and the actor domain(s). The name of the selected class of workflow objects is marked with the keywords workflow-object class.

Example: Figure 6.6 shows class domains of potentially authorized actors defined at authorization interface Archive. Authorization Archive contains worksteps NegArchive and PosArchive that are defined for reservation workflow-processes that are members of class Reservation-WFs. Position class WfAdmins, the base class of position type WFA, has been specified as necessary *and* sufficient class-level actor domain at authorization interface Archive. Deontic modality **P** indicates that the necessary actor-domain and the sufficient actor-domain are domains of potentially authorized actors.

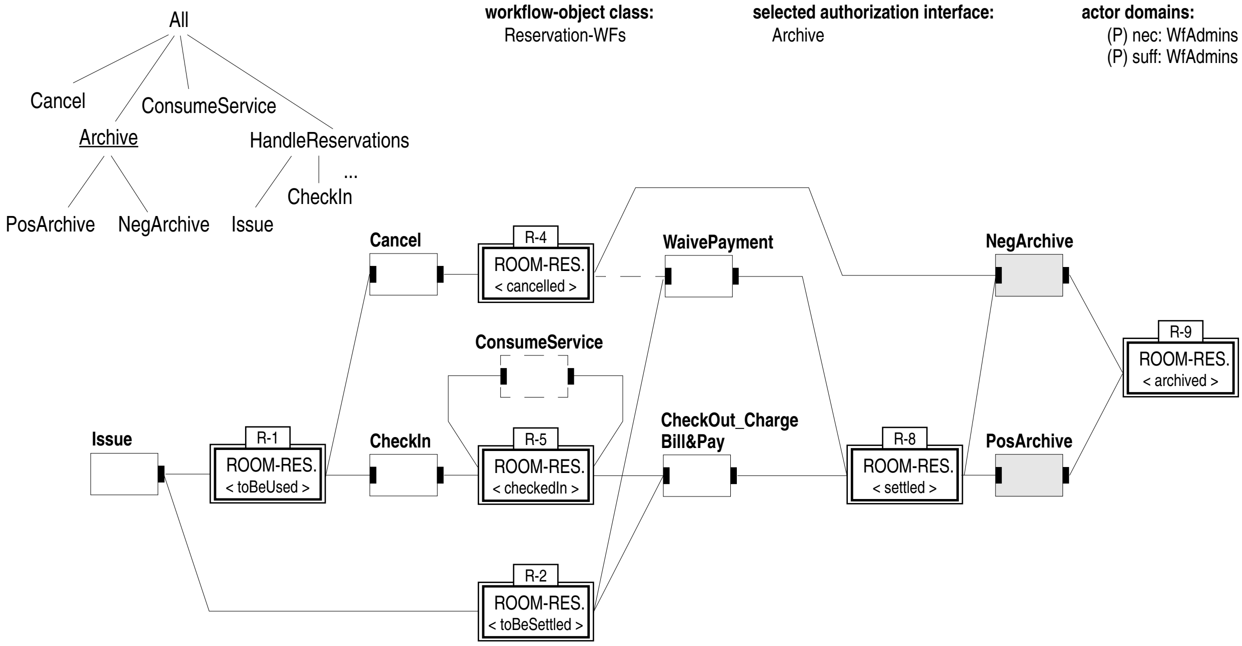


Figure 6.6: Authorization interface Archive of room reservations

Enforcement of Authorization Constraints

Authorization constraints model the protection requirements of public operations that are organized into authorization interfaces. To ensure that no improper authorizations exist and that no mandatory authorizations are missing in the authorization base of an information system, authorization constraints defined for that information system must be enforced. In fact, only class-level authorization constraints must be enforced to ensure authorization-state integrity, since for every type-level authorization constraint a corresponding class-level authorization constraint is specified (cf. definition of language L_{auth}, above). In the following, we discuss the basic problems of enforcing authorization constraints in the context of the conceptual model presented in this thesis. A thorough investigation of the enforcement of authorization constraints, however, is beyond the scope of this thesis and is left for future work.

To enforce authorization constraints, requires to test authorization constraints against the set of authorizations stored in the authorization base of an information system. The purpose of enforcing authorization constraints is to identify improper authorizations included in the authorization base of an information system as well as proper and mandatory authorizations that are missing. The enforcement of authorization constraints is a prerequisite to prevent that unauthorized accesses are accepted by the authorization manager of an information system due to illegal authorizations that are stored in the authorization base of that information system and to prevent that authorized accesses are denied due to missing authorizations. Of course, the proper reactions to violations of authorization constraints are to exclude improper authorizations, i.e., to revoke such authorizations, and to include missing authorizations, i.e., to issue such authorizations. The basic problem with authorization-constraint checking is to ensure that violations of authorization constraints are corrected in time. In the following, we discuss two different strategies to authorization-constraint checking: a lazy and an eager strategy. The lazy strategy corrects violations as late as possible, the eager strategy as soon as possible. Both approaches cope with dynamic modifications of the protection requirements captured within the authorization schema of an information system.

The *lazy strategy* to authorization-constraint checking tests authorization constraints each time an actor's access is validated by the authorization manager of an information system. This means, authorization constraints are tested at the latest possible point in time to ensure that proper accesses are allowed and improper accesses are denied. Obviously, this simple approach is very expensive and would degrade system performance drastically, since large sets of authorization constraints are to be checked each time an actor accesses an information system.

The *eager strategy* to authorization-constraint checking tests authorization constraints each time changes occur within an information system that might cause violations to the integrity of its authorization base. That is, we identify situations in which violations could occur and correct possible violations immediately. The eager strategy reduces the effort for authorization-constraint checking to a minimum. However, the eager strategy to authorization-constraint checking is more difficult to realize and its effectiveness depends on whether all situations in which violations of authorization constraints can be properly identified and corrected. Basically, violations of authorization constraints could occur in the following situations:

1. *Changes to the protection requirements of an information system.* Whenever the protection requirements of an information system change violations might occur. Very likely, existing authorizations are in conflict with changed protection requirements and some mandatory authorizations may be missing. Note: The protection requirements of an information system are changed in that existing authorization constraints are removed or modified and in that new authorization constraints are added.

2. *Issuing a new authorization* requires to check authorization constraints to ensure that the new authorization is not illegal.

3. *Revoking an existing authorization* requires to check authorization constraints to ensure that no obligatory authorization (i.e., an authorization of an obligatorily authorized actor) is removed.

4. *Instantiating new suppliers or clients* requires to check authorization constraints to ensure that no obligatory authorizations are missing for newly created clients or for existing clients on newly created suppliers. Destroying existing suppliers or clients does not require to check authorization constraints.

5. *Class memberships of suppliers or clients change.* Adding a supplier or client being member of some class c as new member to a subclass c' of c requires to check

authorization constraints to ensure that existing authorizations are still permitted and no obligatory authorizations are missing. Removing a supplier or client from a subclass c' does not require to check authorization constraints. Note: We assume a closed-world security policy, where only positive authorizations are stored in an authorization base.

The cost of enforcing authorization constraints increases with the frequency with which the above situations occur and depends on how well these situations can be identified and addressed. Situations of kind (1) to (4) can be clearly identified by operations that are invoked to create new authorization constraints and objects, and by operations that delete existing ones. Situations of kind (5) are problematic in some cases. Whereas changes of the membership of objects in basic classes can be clearly identified by operations that must be invoked to add or remove objects as members of basic classes, changes of the membership of objects in derived classes are hard to identify. This means, in most cases situations in which authorization constraints are to be checked can be clearly identified by executions of operations on objects. For example, active database capabilities, i.e., event-condition-action rules (e.g., [Lan97, Obe98]), could be used to initiate constraint checking whenever corresponding method events are raised. Initiating constraint checking whenever the membership of objects in derived classes changes requires to identify low-level system events and is beyond the scope of conceptual modeling. A simple approach would be to test authorization constraints that involve derived classes relying on an eager strategy (above). Nevertheless, additional research effort is necessary to guarantee a proper enforcement of authorization constraints (cf. future work, Chapter 11).

6.3.3 Modeling of Authorization Contracts

We use contracts as a metaphor to model authorizations in workflow systems. An *authorization contract* is a formal agreement between two or more parties, where one or several parties, the *suppliers*, commit themselves to offer one or several *services* to the remaining parties, the *clients*. Besides stating the parties of a contract and the services offered to the clients, an authorization contract may state further terms of such an agreement. That is, an authorization contract may specify conditions that limit the situations in which the clients are allowed to request the stated services from the suppliers (cf. conditional accessibility, above).

Authorization contracts are objects, too. This means, we offer a uniform approach to the modeling of all kinds of business objects i.e., actors, organizational units, business cases, workflow processes, *and* authorizations. As a result, authorization schemas offer the same qualities as other object-oriented schemas, i.e., understandability, reusability, and extendibility. Note: Since we model authorizations as first-class citizens of workflow systems, authorization contracts support fundamental object-oriented principles like object identity, encapsulation, and information-hiding. For example, encapsulation allows to model procedures to be followed when issuing or maintaining authorizations as behavior of authorizations and information-hiding ensures that the state of authorizations is only manipulated by invoking operations from their public interfaces.

Modeling the Structure of Authorization Contracts

The structure of authorization contracts is determined by a set of *attributes* and a set of *relationships* (cf. Section 3.2.1). In fact, authorization contracts are relationship objects that relate one or several suppliers to one or several clients, expressing that the client(s) might request one or several services from the supplier(s). Consequently, each authorization contract has three designated relationships supplier, service, and client[7]. If these relationships are multi-valued they are named suppliers, services, and clients, respectively. Note: We model type domains of relationships service (or services) of authorization contracts by the predefined type WORKSTEP and their class domains by classes of worksteps that correspond to authorization interfaces. This means, to limit allowed values of relationships service (or services) of authorization contracts to worksteps that belong to a particular authorization interface requires to specify appropriate class domains for these relationships (cf. modeling of classes of authorization contracts, below).

In a two-schema architecture the public interface of business cases is determined by a workflow process. Specifying an authorization contract between a business object s as supplier of a particular workstep r as service and a position c as client could be ambiguous for humans, since a business case could offer two or more public interfaces at a time, each one defining a workstep r[8]. To resolve this ambiguity for humans, we model workflow objects as suppliers in authorization contracts that allow some clients to request certain services from business cases. Furthermore, having workflow objects stand in for business cases reduces the overall number of authorizations to be issued. That is, a workflow object is a representative for the current set of business cases being full members of that workflow object. As a consequence, the domain of a relationship supplier (or suppliers) is given by a workflow-process type, its value by one or several workflow objects. Note: The modeling of business case specific authorizations is discussed below.

Example: Figure 6.7 shows a type diagram of authorization-contract type CONTRACT-CANCEL. An authorization contract of type CONTRACT-CANCEL is identified by the values of relationships supplier, service, and client from domains RESERVATION-WF, WORKSTEP, and RECEPTIONIST, respectively. That is, an authorization contract of type CONTRACT-CANCEL authorizes a particular receptionist, the client, to invoke a particular workstep, the service, from business cases that are full members of a particular room-reservation workflow object, the supplier. Note: A class-domain

[7]Note: In the literature on database security (e.g., [CFMS94]), an authorization is usually modeled as a triple $< o, t, s >$, where o is the *object*, t is the *access type*, and s is the *subject*. To avoid confusion with the terminology of object-oriented modeling, we use the terms supplier, service, and client in place of the terms object, access type, and subject, respectively.

[8]Note: Two specifications of a workstep with name r that belong to different workflow process do have different identity, but the object identifier is non-printable. Therefore, the names of worksteps are printed as "identifiers" of worksteps.

constraint given in Figure 6.13(a) specifies that only worksteps belonging to the authorization interface Cancel defined at workflow-process type RESERVATION-WF (cf. Figure 6.4) are allowed as values of relationship service of authorization contracts of type CONTRACT-CANCEL. In fact, the only workstep that belongs to authorization interface Cancel is workstep Cancel.

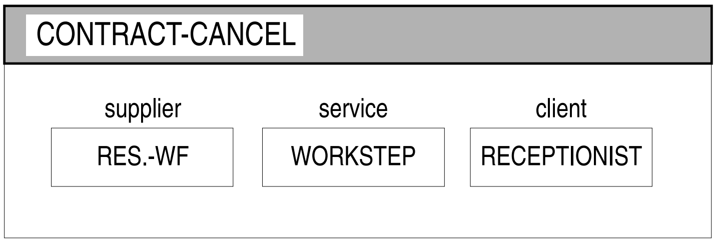


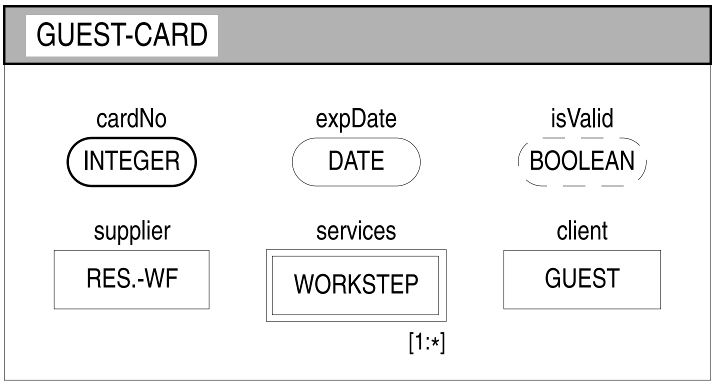
Figure 6.7: Authorization-contract type CONTRACT-CANCEL

Since workflow objects stand in as suppliers of authorizations for the current sets of business cases that are full members of these workflow objects, in general a client that is authorized to invoke some workstep s on one business case processed according to a workflow process w may invoke workstep s on all business cases processed according to w. Making authorizations dependent on individual business cases that are full members of a workflow object can be accomplished in two different ways: First, *conditional authorizations can be issued to clients*. This means, at the time an actor invokes a workstep on some business case the actor's authority is tested by searching for an appropriate authorization contract in the authorization base *and* by evaluating an additional condition on the business case on which the workstep has been invoked if an appropriate authorization has been found. The current business case on which a workstep has been invoked at the time the actors authorization is to be validated can be addressed by the system variable $\mathsf{current}_{\mathsf{bp}}$. Second, *besides one or several workflow objects one or several business cases can be modeled as suppliers of authorization contracts.* By convention, these supplier relationships are named supplier-wfp (suppliers-wfp) and supplier-bp (suppliers-bp), respectively, if they are single-valued (multi-valued).

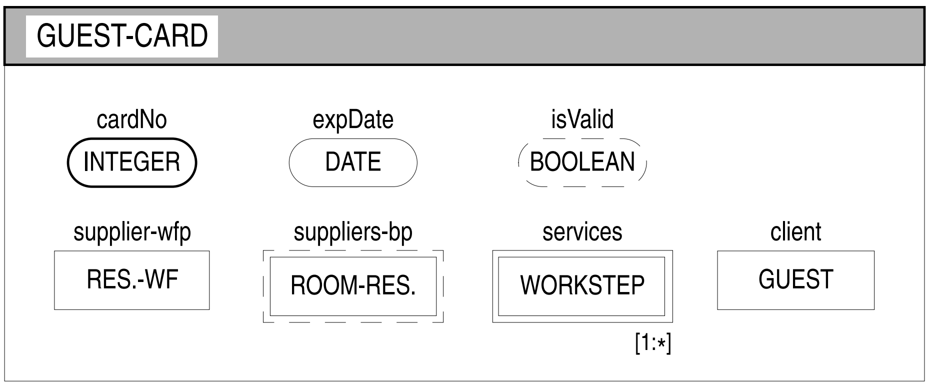
Example: Figure 6.8(a) shows a type diagram of authorization-contract type GUEST-CARD. A guest card is identified by the value of its key attribute cardNo (card number) and is described by the values of further attributes expDate (expiration date) and isValid. Furthermore, a guest card has relationships supplier, services, and client to values from domains RESERVATION-WF, WORKSTEP, and GUEST, respectively. That is, a guest card authorizes one guest to invoke *several* worksteps on that business cases that are full members of a particular room-reservation workflow object. In fact, a class-domain constraint given in Figure 6.13(b) specifies that only worksteps belonging to the authorization interface ConsumeService defined at workflow-process type RESERVATION-WF (cf. Figure 6.4) are allowed as values of relationship service of guest cards. The derivation expression of the boolean attribute isValid is expDate $\geq$ TODAY

$\wedge$ $\mathsf{current_{bp}.guest} = \mathsf{client}$. It ensures that guests can neither consume services after the expiration date of their guest cards nor on the expenses of other guests.

Figure 6.8(b) shows an alternative definition of authorization-contract type GUEST-CARD. Now a guest card has two supplier relationships. Relationship supplier-wfp denotes a room-reservation workflow process, relationship suppliers-bp a set of business cases that are full members of that room-reservation workflow process. Relationship suppliers-bp is a derived relationship, its derivation expression is $\{\ \mathsf{b} \mid \mathsf{b} \in \mathsf{supplier\text{-}wfp.reservations} \mid \mathsf{b.guest} = \mathsf{client}\ \}$. That is, a guest card authorizes one guest to invoke several worksteps that belong the authorization interface ConsumeService (above) from the given room reservations that are processed according to the given room-reservation workflow process. The derivation expression of the boolean attribute isValid is $\mathsf{expDate} \geq \mathsf{TODAY}$.



(a)



(b)

Figure 6.8: Authorization-contract type GUEST-CARD

We distinguish authorization contracts into individual contracts and collective contracts, depending on the multiplicity of the relationships denoting their suppliers, services, and clients. An *individual contract* authorizes one client to request one service from one supplier, i.e., an individual contract has three single-valued relationships supplier, service, and client. A *collective contract* authorizes one or several clients to request one or several services from one or several clients. In fact, one of the three designated relationships of a collective contract must be multi-valued, i.e., a collective contract has either a multi-valued relationship suppliers, services, or clients.

Example: Authorization contracts of type CONTRACT-CANCEL (cf. Figure 6.7) are individual contracts. That is, relationships supplier, service, and client defined at CONTRACT-CANCEL are single-valued.

Authorization contracts of type GUEST-CARD (cf. Figure 6.8(a)) are collective contracts. Relationship services is a multi-valued relationship.

To issue an authorization to (revoke an authorization from) a client, which authorizes that client to request a certain service from a particular supplier, could be accomplished in different ways. First, an individual authorization-contract could be instantiated (destroyed). Second, the client, the service, or the supplier could be added to (removed from) an existing collective authorization-contract. Note: In either case, an entry $< s, r, c, a >$ is inserted into (removed from) a system maintained access matrix A if an authorization is issued (revoked) for a supplier s, a service r, and a client c, where a is the object identifier of that authorization contract that represents the corresponding authorization $< s, r, c >$. Since authorization contracts are either individual or collective contracts, one authorization contract may represent several authorizations. As a consequence, the primary key of access matrix A is a compound key formed by columns s, r, and c, and column a is not a key of access matrix A. The system maintained access matrix A allows to efficiently determine the existence of an authorization using the boolean predicate $E_A(s, r, c)$ (cf. Subsection 6.3.2).

If the concept of derived relationships is used to model the designated relationships of collective authorization-contracts, the effort of issuing and revoking authorizations can be reduced. That is, the set of authorizations $< s, r, c >$ represented by such a collective authorization-contract is derived. Note: To allow to test the existence of authorizations via a system maintained access matrix A, requires to materialize the set of derived authorizations $< s, r, c >$ represented by a collective authorization-contract. View materialization techniques (e.g., [CW91]) could be used to ensure consistency of access matrix A.

Example: Figure 6.9 shows a type diagram of authorization-contract type CONTRACT-HANDLE-RESERVATIONS. An authorization contract of type CONTRACT-HANDLE-RESERVATIONS is a collective authorization-contract. It has three relationships supplier, services, and clients to values from domains RESERVATION-WF, WORKSTEP, and RECEPTIONIST, respectively. Relationships services and clients are derived, multi-valued relationships. The derivation expression of relationship services is { w | w ∈ HandleReservations } , that of relationship clients is { c | c ∈ Receptionists } . That is, all existing receptionists are authorized to invoke all worksteps that belong to authorization interface HandleReservations on all room reservations being full members of a particular room-reservation workflow process.

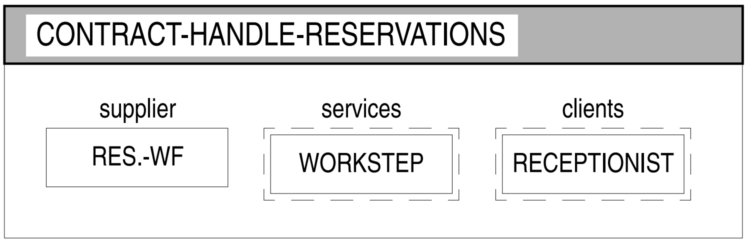


Figure 6.9: Authorization-contract type CONTRACT-HANDLE-RESERVATIONS

Modeling the Behavior of Authorization Contracts

The behavior of authorization contracts is determined by a set of *activities*, by *allowed execution orders* on these activities, and by their *interactions* with other objects. In other words, the behavior of authorization contracts is modeled in the same way as the behavior of ordinary business objects (cf. Subsection 3.2.2). This means, we consider authorizations as full-fledged objects, i.e., as authorization contracts, that have user-defined structure *and* behavior. A life-cycle models the public update behavior of authorization contracts using a set of states, a set of activities, and a set of arcs connecting states to activities and vice versa. Of course, the life-cycle of authorization contracts can be developed by stepwise refinements of states and activities. An activity specification models the interface of an activity, an activity realization the implementation of an activity. The public interface of authorization contracts of a particular type is determined by the life-cycle defined for them, i.e., the public interface of authorization contracts consists of all public update activities offered by their life-cycles.

The public interfaces of authorization contracts determine the set of operations a security administrator can perform to modify the authorization state of a workflow system. This means, we do not offer a fixed set of primitive operations to manipulate the authorization state of a security system[9], but allow security designers to model procedures to be followed when issuing, deleting, and maintaining authorizations as high-level semantic operations, called activities. This means, the same object-oriented language is used to manipulate ordinary business objects and authorization contracts. To allow security administrators to efficiently maintain large sets of authorization contracts, collective behavior

[9] Note: The access-matrix model offers six primitive operations to manipulate the authorization state of a security system. Operation *enter* $<o,t,s>$ creates a new entry in an access matrix, operation *delete* $<o,t,s>$ removes an entry from an access matrix, operations *create subject s* and *create object o* add a new row and a new column, respectively, to an access matrix, and operations *destroy subject s* and *destroy object o* remove a row and a column, respectively, from an access matrix.

could be defined for classes of authorization contracts (cf. modeling classes of authorization contracts, below).

Example: Figure 6.10 shows the life-cycle defined at authorization-contract type GUEST-CARD. After a guest card has been issued by a receptionist, a security administrator tests the trustworthiness of the guest. Depending on the result of this test, a security administrator either accepts or denies the guest card of that guest. If the guest card of a guest has been accepted, i.e., if it resides in state valid, the guest is authorized to consume services determined by the guest card. This means, the derivation expression of the boolean attribute isValid should be changed to $\mathsf{expDate} \geq \mathsf{TODAY} \wedge \{\mathsf{valid}\} \subseteq \mathsf{inStates}$[10]. If a guest's trustworthiness is violated, e.g., through misuse or manipulation of guest cards, the guest card of that guest will be invalidated by a security administrator. Invalid guest cards will be archived and destroyed later on. During its existence, a guest card can be manipulated in that new services are added to or existing services are removed from the guest card. Actually, this modifications of a guest card issue new authorizations to the holder of a guest card and revoke existing authorizations, respectively. Furthermore, the validity of a guest card can be prolonged if it expires.

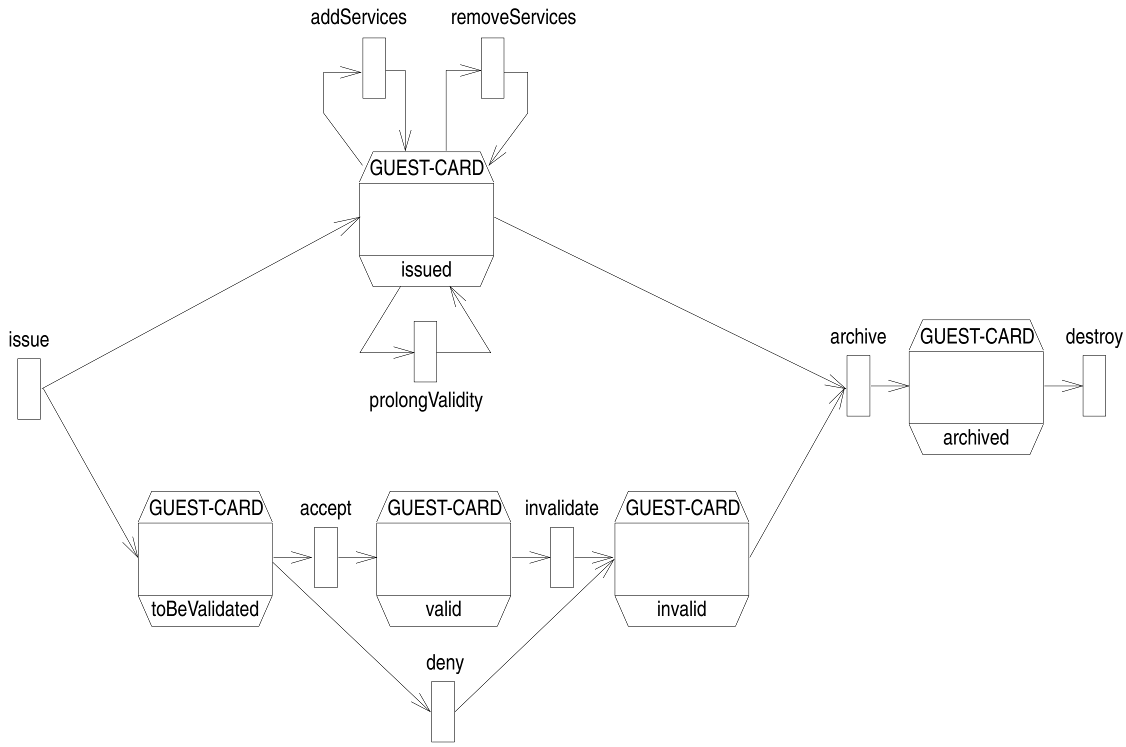


Figure 6.10: Life-cycle at GUEST-CARD

[10]Note: Each object has a multi-valued property inStates to record the current set of life-cycle states in which it resides (cf. Subsection 3.2.2).

Example: Figure 6.11 shows the specification of activity addServices of guest cards. Activity addServices creates new authorizations for the holder of a guest card (cf. relationship client in Figure 6.7) in that it adds one or several worksteps from authorization interface ConsumeService (cf. Figure 6.4) as values to relationship services of the corresponding guest card. Note: The precondition of activity addServices tests that only worksteps that belong to authorization interface ConsumeService are assigned to port r when activity addServices is invoked.

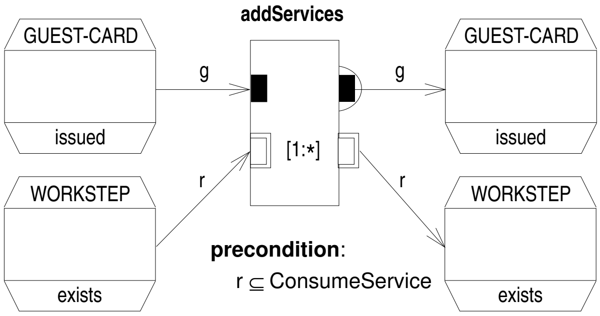


Figure 6.11: Specification of activity addServices

Modeling Classes of Authorization Contracts

An *authorization-contract class* collects a set of authorization contracts as its members. Like with an ordinary business-object class (cf. Subsection 3.2.3), individual structure and behavior of its members are specified at the member type, an authorization-contract type, of an authorization-contract class, and collective structure and behavior of its members are specified at the class type of the authorization-contract class.

Modeling of collective behavior for classes of authorization contracts can ease the maintenance of large sets of authorization contracts considerably. That is, invoking a single activity on a class of authorization contracts can manipulate the state of a set of members of that class at a time.

Example: Figure 6.12 shows an activity-specification diagram for class activity invalidate defined for classes of guest cards. Class activity invalidate can be invoked on an instance of class type GUEST-CARDS. Class activity invalidate invokes the individual activity invalidate (cf. Figure 6.10) on each guest card that is assigned to its secondary port. That is, with a single statement a set of guest cards can be invalidated. Note: Since allowed execution orders are not specified for class activities (cf. Subsection 3.2.3), no prestates and no poststates are defined for the primary ports of class activities.

Classes of authorization contracts function as object warehouses, i.e., each authorization contract is member of the base class of the type of which it is an instance. Organizing classes of authorization contracts into class hierarchies allows for a fine-grained classification of authorization contracts that could not be achieved with the concept of types alone.

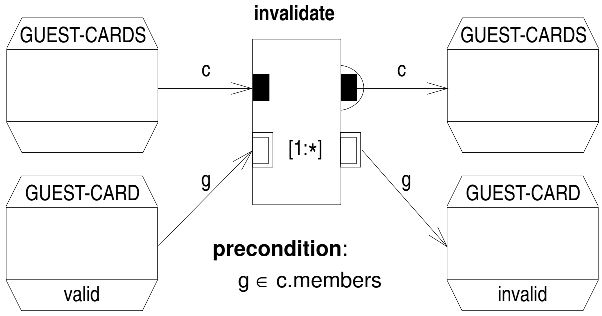


Figure 6.12: Class activity invalidate of guest cards

Thus the semantic expressiveness of authorization schemas is increased by the use of classes. Classes of authorization contracts are represented as named sets of authorization contracts in an authorization schema. Since classes allow to address sets of authorization contracts at a time, classes play an important role in administering authorizations (above). Furthermore, classes allow to incorporate further constraints on their members into authorization schemas (below).

Example: Figure 6.13(a) shows the specification of classes of guest cards. Class Guest-Cards is the base class of authorization-contract type GUEST-CARD. Class VIP-Guest-Cards is a basic subclass of class Guest-Cards, it collects guest cards that have been issued to VIP guests of the hotel. Note: For the moment, ignore the remaining class specifications.

A class of authorization contracts may specify other object classes as domains for relationships defined for its members. Class domains represent referential integrity constraints (cf. Subsection 3.2.3). Since each member of a class may be related via a relationship r only to objects that are members of the class domain of r, the specification of class domains for classes of authorization contracts constrains the extensions of these classes. Note: We model class domains of relationships service (or services) of authorization contracts by classes of worksteps that correspond to authorization interfaces. That is, for each authorization interface f defined at some workflow-process type W there is a class of worksteps c that contains all those worksteps defined at W that are contained in f. Since classes are global schema objects in the basic conceptual model, names of classes must be unique within a schema. As a consequence, the name of a class of worksteps c is derived by concatenating the name of authorization interface f that corresponds to c to the name of the workflow-process type W at which f has been defined. For example, the name of a class of worksteps c that corresponds to an authorization interface named Do that has been defined at a workflow-process type named W is W.Do. If no class domain is defined explicitly for a relationship service (or services) at a class of authorization contracts c_a, the class domain of relationship service (or services) at c_a is implicitly defined by the class of worksteps that corresponds to authorization interface All defined at that workflow-process type W that is defined as member type at the class domain of relationship supplier (or suppliers) defined at c_a.

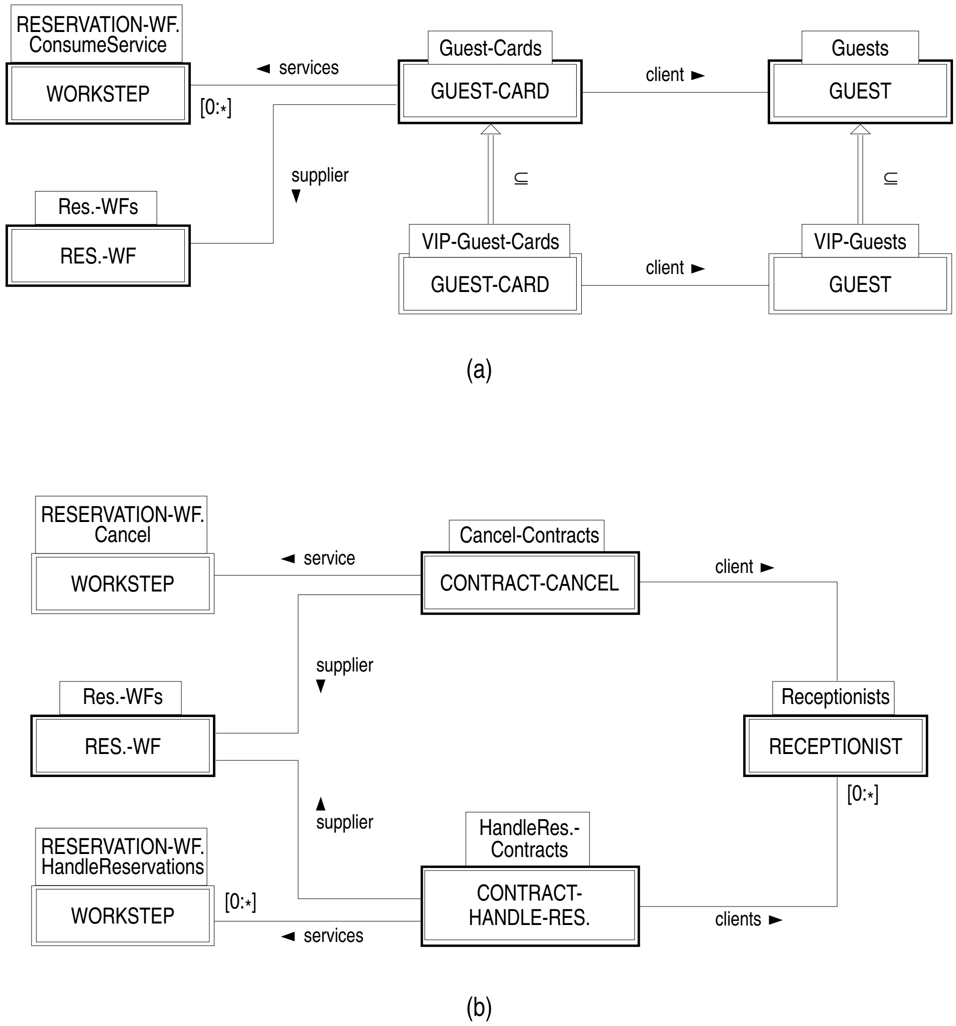


Figure 6.13: Class-domain diagrams of authorization contracts

Example: Figure 6.13(a) shows a class-domain diagram with object classes Guest-Cards, VIP-Guest-Cards, Guests, VIP-Guests, Reservation-WFs, and RESERVATION-WF.ConsumeService. Class Guest-Cards is the base class of authorization-contract type GUEST-CARD. It restricts the domain of relationship client to agent class Guests, the domain of relationship supplier to workflow-object class Reservation-WFs, and the domain of relationship services to workstep class RESERVATION-WF.ConsumeService. Class VIP-Guest-Cards is a direct subclass of base class Guest-Cards. It restricts the inherited class domain of relationship client to agent class VIP-Guests, a direct subclass of agent class Guests.

Figure 6.14 shows the specification of an alternative class-domain constraint on relationship client of class Guest-Cards. It restricts the domain of relationship client to agent class VIP-Guests. Since class Guest-Cards is the base class of authorization-contract type GUEST-CARD, this means that guest cards may only be issued for VIP guests of a hotel. However, notice that this class-domain constraint does not express the protection requirement that only VIP guests may be authorized to consume services in a hotel. There may exist other types of authorization contracts that could be issued to allow other kinds of clients, e.g., non-VIP guests, to consume services in a hotel. Note: Protection requirements of objects are expressed using the concepts of authorization interfaces and authorization constraints (cf. Subsections 6.3.1 and 6.3.2, respectively). For example, the following authorization constraint specifies that only VIP guests of a hotel might be authorized to consume services:

$$\forall s \in \mathit{Reservation\text{-}WFs}, r \in \mathit{ConsumeService}, c \in \mathit{Public} \quad (\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \mathit{VIP\text{-}Guests}) \tag{6.17}$$

Figure 6.13(b) shows a class-domain diagram with object classes Cancel-Contracts, HandleReservations-Contracts, Receptionists, Reservation-WFs, RESERVATION-WF.Cancel, and RESERVATION-WF.HandleReservations. Class Cancel-Contracts is the base class of authorization-contract type CONTRACT-CANCEL. It restricts the domain of relationship client to agent class Receptionists, the domain of relationship supplier to workflow-object class Reservation-WFs, and the domain of relationship services to workstep class RESERVATION-WF.Cancel. Class HandleReservations-Contracts is the base class of authorization-contract type CONTRACT-HANDLE-RESERVATIONS. It restricts the domain of relationship clients to agent class Receptionists, the domain of relationship supplier to workflow-object class Reservation-WFs, and the domain of relationship services to workstep class RESERVATION-WF.HandleReservations.

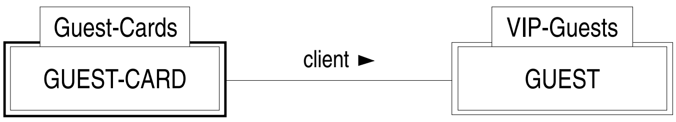


Figure 6.14: Class-domain diagram of guest cards

6.4 Administration of Authorizations

To administer authorizations means to create new authorization contracts, to destroy existing authorization contracts, and to invoke different public activities on authorization contracts[11]. This means, actors that administer authorizations within an organization have influence on the authority of other actors and are thus very powerful. Consequently, the administration of authorizations requires careful planning.

First, the set of actors "who" might grant and revoke authorizations on "what" objects must be determined. Basically, there are two different approaches to solve this problem:

[11] Note: The behavior of authorization contracts, i.e., the set of activities that may be invoked on authorization contracts as well as allowed execution orders on these activities (cf. Subsection 6.3.3), is defined at their life-cycles.

the ownership approach and the administrative approach. The *ownership approach* says that upon creation of an object the actor that did create an object becomes its owner (e.g., [CFMS94]). The owner has full control over an object. That is, the owner can request all services from an object and grant and revoke authorizations on that object to other actors. The *administrative approach* says that objects are owned by organizations (e.g., [FGS94]). The authority to administer authorizations on objects that belong to an organization is initially with the owner of an organization, but may be delegated to the management. Moffet and Sloman [MS88] denote the set of objects for which a manager has the authority to administer authorizations as the *resource domain* of that manager. Fundamental to the administrative approach is that managers do not administer authorizations on objects in their resource domains on themselves, but delegate this responsibility to designated security administrators (separation of duties principle). This delegation allows to properly separate administration of authorizations from use of authorizations. Nevertheless, security administrators are very powerful in such settings. Moffet and Sloman [MS88][p. 63] discuss the question of "who administers the security administrators". Although there is no general solution to the problem, they mention that the problem can be limited by "narrowing the scope for misbehavior, recording all significant actions, and concentrating personal supervision on the main point of vulnerability". Note: Since workflow management systems are used to enact organizational processes, we adopt the administrative approach.

Second, the set of actors to "whom" a manager may grant authorizations and from whom she or he may revoke authorizations must be determined. Moffet and Sloman [MS88] denote this set of actors as the *organization domain* of a manager. If a manager delegates a resource domain *and* an organization domain to a security administrator, the security administrator has the authority to issue (revoke) authorizations on objects in the resource domain to (from) actors in the organization domain. This means, the authority of a security administrator to issue (revoke) authorizations is limited by resource domains and organization domains delegated to that security administrator (cf. "narrowing the scope for misbehavior", above).

The knowledge on the control structure of an organization as well as the knowledge on the physical structure of an organization (cf. Chapter 5) can be used to determine a basic resource domain and a basic organization domain for each manager or supervisor of an organizational unit. The basic *resource domain* D_r of a manager m contains all objects that belong to the same organization unit as m. The basic *organization domain* D_o of a manager m contains m and all actors that are subordinates of m and belong to the same organizational unit as m. To allow a manager m to delegate her or his responsibility to administer authorizations to several security administrators, we allow to model subdomains of the initial organization domain and resource domain of m. Different subdomains may be delegated to different security administrators.

6.5 Summary

In this chapter we presented basic concepts for the design of *authorization schemas* for use in WFMSs. After we gave a management definition of the fundamental concepts presented in this chapter, we proposed a new architecture of information systems that considers an authorization schema as the central description of the protection requirements of an information system that allows an authorization manager to test the legality of authorizations stored in the authorization base of that information system. Then we presented basic concepts for the design of authorization schemas. An *authorization interface* contains a set of operations with similar protection requirements from the public interface of objects of a particular type. The protection requirements of public operations that are organized into authorization interfaces are specified using *authorization constraints*. An *authorization contract* models an authorization. Authorization contracts are first-class citizens, i.e., are objects with structure and behavior that is modeled at authorization-contract types and authorization-contract classes. Finally, basic principles of the administration of authorizations in an organization have been discussed.

Part III

Advanced Workflow Modeling

Chapter 7

Data Modeling

Contents

This chapter completes the presentation of our conceptual data model, which we started in Chapter 3. As mentioned before, the conceptual data model offers concepts that build on the following abstraction mechanisms: classification, association, aggregation, decomposition, and specialization. Modeling concepts that build on the abstraction mechanisms classification and association have already been presented in Chapter 3. This chapter presents modeling concepts that build on the abstraction mechanisms aggregation, decomposition, and specialization. *Semantic relationships* allow to capture the semantics of references between real-world objects (e.g., aggregation, composition). *Type hierarchies* organize object types into specialization relationships (type specialization). *Class hierarchies* organize object classes into specialization relationships[1] (class specialization). *Role hierarchies* organize objects into specialization relationships (object specialization). Specialization relationships between objects are modeled by a specific type of semantic relationships, called *role-of* relationship. Two objects related by a role-of relationship represent the same real-world entity. The two objects are called the player and the role, respectively. Since a player may abandon and acquire roles dynamically, role hierarchies allow to cope with the evolutionary nature of real-world entities.

[1]Note: The concept of class hierarchies has already been presented in Chapter 3. However, the semantic expressiveness of object classes and class hierarchies will be extended considerably in this chapter.

This chapter is organized as follows. Section 7.1 gives a management definition of the fundamental modeling concepts presented in this chapter. Section 7.2 presents advanced concepts of our conceptual data model. Section 7.3 summarizes this chapter.

7.1 Fundamental Modeling Concepts

The core concepts presented in this chapter are *abstraction principles* and *semantic relationships*. Abstraction is a mental process that helps us to understand, classify, and model reality (cf. [BCN92]). We use it when designing a conceptual schema of some business area. A conceptual schema is a semantically rich description of the entities in a particular business area. The language we use to build a conceptual schema is called *conceptual model*. We require that it offers a language construct for each abstraction principle we apply when trying to understand and model reality. The most prominent abstraction principle we use is *classification*. We use it when we describe a set of objects by common properties, and forget about their differences. Object types and object classes are the language constructs offered by the basic conceptual model (cf. Chapter 3) to model classification. Actually, we apply a number of further abstraction principles when designing a conceptual schema of some business area. Semantic relationships are the language constructs offered by the advanced conceptual model presented in this chapter to support a designer in building conceptual schemas. The advanced model offers semantic relationships for a rich set of abstraction principles.

Below the set of abstraction principles that may be used when building a conceptual schema is described. For each abstraction principle, the kind of semantic relationship offered by the advanced conceptual model to represent it is given.

Aggregation: *Aggregation* is an abstraction principle that defines a new object type from a set of other object types that represent the constituents of its instances. That is, by means of aggregation, related object types, the constituent types, are assembled into a more complex type, the aggregate type, in a bottom-up fashion. Aggregation is modeled using has-constituent relationships.

Decomposition: *Decomposition* is an abstraction principle that defines a set of new object types that represent components of the instances of another object type. That is, by means of decomposition, an object type, the composite type, is decomposed into a set of simpler types, the component types, in a top-down fashion. Decomposition is modeled using component-of relationships.

Specialization: *Specialization* is an abstraction principle that identifies one concept as more specific than the other. This means, specialization is used to build one concept based on an existing concept by suppressing commonalities of the two and by emphasizing differences between them. Traditionally, specialization is modeled by relationships between object classes or between object types (cf. class specialization and type specialization, below). However, specialization can also be modeled by relationships between objects (cf. object specialization, below).

Object Specialization: *Object specialization* is an abstraction principle that defines a specialization relationship between two objects, called the role and its player. Both objects represent the same real-world entity. Each player may occur in several roles that represent him in different contexts. Object specialization is modeled using role-of relationships.

Class Specialization: *Class specialization* is an abstraction principle that defines a specialization relationship between two object classes, called the subclass and its superclass. All members of the subclass must also be members of the superclass, i.e., there is a subset relationship between the sets of members of a subclass and its superclass. A class may have several subclasses. Specializations of classes may be *partial* and *overlapping*[2]. This means, a superclass c may have members that are not members of any subclass of c (partial) as well as members that are members of two or more subclasses of c (overlapping). Class specialization is modeled using subclass-superclass relationships (cf. Subsection 3.2.3).

Type Specialization: *Type specialization* is an abstraction principle that defines a specialization relationship between two object types, called the subtype and its supertype. A subtype must offer at least that properties for its instances that have been defined at its supertype, but it may also offer additional properties. This means, there is a superset relationship between the sets of properties of a subtype and its supertype. A type may have several subtypes. In general, we require that specializations of types must be *total* and *exclusive*[2]. This means, a supertype O may not have any instance that is not instance of at least one subtype of O, too (total). In fact, each instance of a supertype O must be instance of exactly one subtype of O (exclusive). As an exception to the above rule, we allow that specializations of authorization-contract types are *partial* and exclusive. The arguments for this decision are given in Chapter 10. Type specialization is modeled using subtype-supertype relationships.

Semantic Relationships: A *semantic relationship* models a particular type of reference between entities from a business area, between object types, or between object classes. A semantic relationship implicitly expresses the semantics of a reference, which otherwise would have to be specified explicitly in form of integrity constraints completing a conceptual schema or would be lost otherwise. The advanced conceptual model presented in this chapter offers a set of different types of semantic relationships, which allow a designer to model reality in the way she or he understands it. This means, there is one kind of semantic relationship for each of the abstraction principles discussed above.

[2]Note: Batini *et al.* [BCN92, Chapter 2] discuss different variants of specialization (generalization) relationships between types. Their discussion applies to specialization (generalization) relationships between classes in the same way.

Summing up, conceptual models that offer a rich set of semantic relationship support a designer in a natural way, since they offer language constructs allowing to describe reality in the way she or he understands it. Using semantic relationships improves understanding of conceptual schemas, since semantical overloading of language constructs is reduced. This means, whereas only one kind of relationship was available in the basic conceptual model (cf. Chapter 3) to model all kind of references between business objects, different kinds of relationships are offered by the advanced model, presented in this chapter, to model references between objects that have different semantics.

7.2 Design of Object-Oriented Schemas

This section presents an advanced model for the conceptual design of object-oriented database schemas. The advanced model relies on concepts known from semantic data modeling (e.g., [HK87, HM81]). It offers special kinds of relationships, so-called *semantic relationships*, that implicitly represent integrity constraints in a database schema, which otherwise must be stated explicitly through assertions that complement the schema. Thus, these semantic relationships increase the expressiveness of the basic conceptual model presented in Section 3.2.

The semantic relationship supported by most object-oriented systems is the subtype-supertype relationship that organizes object types into so-called *type hierarchies*[3]. Each subordinate object type (*subtype* for short) must specify at least all that structure and behavior for its instances, which its superordinate object type (*supertype* for short) specifies for his instances. In general, a subtype can specialize its supertype in that it redefines structure and behavior already defined at its supertype (*refinement*), and in that it specifies new structure and behavior for its instances (*extension*). Fundamental to object-oriented systems is that subtypes *inherit* the specifications of structure and behavior from their supertypes (e.g., [ABD$^+$89, BM91, Tai96]). That is, inheritance supports reuse of specifications of structure and behavior between object types organized in a type hierarchy[4]. Consequently, the effort to specify the structure and behavior of the instances of a subtype is reduced, since only redefined and new structure and behavior must be specified explicitly at a subtype.

For simplicity and consistency of the underlying object model, we require that every object type may inherit only from a single supertype (*Single Inheritance Rule for Object Types*), that specializations of types are exclusive, i.e., each object that is indirect instance of a supertype O may be direct instance of at most one subtype of O (*Single Instantiation Rule for Objects*), and that specializations of types are total, i.e., no object is direct instance of a supertype (*Abstract Supertype Rule for Object Types*, cf. [Hür94]).—Note: An

[3]The concept of object types corresponds to the concept termed "class" in most object-oriented programming languages corresponds.

[4]Usually, a subordinate type inherits also the realizations of activities from its supertype. At the conceptual level, however, we are mainly interested in specifications of structure and behavior not in specific realizations.

object is *direct instance* of the object type it is instantiated from and *indirect instance* of all predecessors of this object type in a type hierarchy.—Kappel and Schrefl [KS96b] state that the readability and changeability of the design of an object-oriented database schema is improved if objects are only instantiated from types at the leaf nodes of a type hierarchy, and that obeying to the *Abstract Supertype Rule for Object Types* does not mean a restriction of the expressiveness of a conceptual data model.

Since the object model of our conceptual model explicitly distinguishes between object types and object classes, it also supports a specialization relationship that organizes object classes in so-called *class hierarchies*. All members of a subordinate object class (*subclass* for short) must also be members of its superordinate object class (*superclass* for short).

The remaining semantic relationships model relationships between objects. In particular, they model relationships between component objects and their composite objects, between aggregate objects and their constituent objects, between player objects and their roles, or simply between independent objects (cf. Section 7.1). These semantic relationships focus on the structure of objects (cf. Subsection 7.2.1). However, their semantics has also influence on the synchronization of the behavior of related objects (cf. Subsection 7.2.2).

The advanced conceptual model presented in this section builds a common object-oriented fundament for the overall workflow model presented in this thesis. This means, all concepts presented for the conceptual modeling of business objects apply to the conceptual modeling of business-process objects, workflow-process objects, organizational objects, and authorization objects in the same way, unless stated otherwise in subsequent chapters. The model of business objects presented in this section builds on the model of objects of Object/Behavior Diagrams [KS91, KS96b], which has already been extended wrt. various aspects in Section 3.2). In this section, we extend the advanced object model of Object/Behavior Diagrams in that we explicitly distinguish local classes and has-component relationships as separate but related concepts.

The remainder of this section is organized as follows. Subsection 7.2.1 discusses the modeling of the structure of business objects, Subsection 7.2.2 the modeling of the behavior of business objects, and Subsection 7.2.3 the modeling of classes of business objects.

7.2.1 Modeling the Structure of Business Objects

The structure of business objects (or *objects* for short) is determined by a set of *attributes* and a set of *relationships*. Whereas attributes model descriptive features of objects, relationships model references between objects (cf. Subsection 3.2.1). Since references between objects may have different semantics, we discuss different kinds of so-called *semantic relationships*. We conclude this subsection with a discussion on the specialization of the structure of objects.

Semantic Relationships

Semantic relationships between objects can be distinguished into object-defining relationships and object-describing relationships. An *object-defining relationship* models existence dependence of a referencing object on the referenced object. In general, each object may have several object-defining relationships. Object-defining relationships are always single-valued, mandatory, and non-derived. This means, object-defining relationships must be initialized when creating objects and must not be changed throughout the lifetime of objects. An *object-describing relationship* models existence independence of a referencing object from one or several referenced objects. Each object may have several object-describing relationships. Object-describing relationships are either single-valued or multi-valued, are either optional or mandatory, and may be derived. Consequently, only mandatory object-describing relationships must be initialized when creating objects. However, all object-describing relationships might be changed during the lifetime of objects. Note: Single-valued object-describing relationships that have not been otherwise initialized refer to a special value nil[5], multi-valued relationship to an empty set.

Graphically, object-describing relationships and object-defining relationships can be distinguished in that the former are depicted by rectangles inside the body of a type diagram and in that the latter are depicted by bars stacked at the bottom of the body of a type diagram.

The discussion of the different kinds of semantic relationships starts with *general relationships*. General relationships correspond to the concept of relationships known from the basic object model (cf. Chapter 3). Consequently, all concepts for the modeling of relationships that were presented in Subsection 3.2.1 apply in the same way to the modeling of general relationships, as well as to the modeling of other kinds of semantic relationships, which extend general relationships. All exceptions to this rule will be discussed below in detail.

General Relationship. A *general relationship* models a relationship of an object to one or several other objects. Each object may have several general relationships. General relationships have no predefined semantics and are used whenever no other kind of semantic relationship is appropriate. General relationships are *object-describing*. A general relationship may be declared to be inverse to another general relationship. In a type diagram, a general relationship is depicted like relationships have been depicted in the basic model (cf. Subsection 3.2.1).

Example: Each reservation has a general relationship reservableObject to a value from domain RESERVABLE_OBJECT (cf. Figure 7.1). (For the moment, ignore the black triangle in the upper right corner of the graphical symbol representing relationship reservableObject.)

Component-Of Relationship. A *component-of relationship* models a relationship of a component object to a composite object. Each component object has exactly one

[5]The special value nil refers to an object used as the value of a single-valued relationship when no other object is appropriate, i.e., single-valued relationships that have not been otherwise initialized refer to nil (cf. [FE95]).

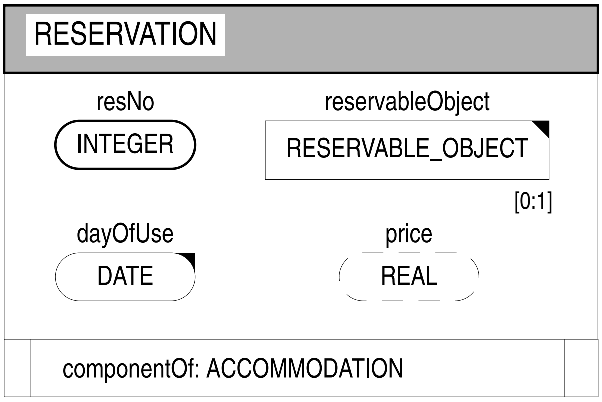


Figure 7.1: Object type RESERVATION

component-of relationship. The existence of a component object depends on the composite object to which it exclusively belongs. Consequently, component-of relationships are *object-defining*. Since each object type may define only one component-of relationship for its instances, naming of component-of relationships is optional. If a component-of relationship remains unnamed, the composite object can be referenced by means of the predefined name componentOf. Note: The notion of component objects we use in thesis corresponds to that of Kim *et al.* presented in [KBC+87][6].

Relationships inverse to component-of relationships are has-component relationships. A *has-component relationship* models a relationship of a composite object to several component objects of a specific type. Each composite object may have several has-component relationships. Of course, composite objects may also have attributes and other kinds of relationships. Has-component relationships are *object-describing*. There is an inverse has-component relationship for every component-of relationship and vice versa[7]. Since each object type may define only one component-of relationship for its instances, and since every relationship may have only one inverse relationship, each object type may be used as domain of only one has-component relationship. Another consequence is, that the knowledge about the inverse of a component-of relationship is implicitly available and does not have to be indicated. Note: The inverse of the component-of relationship of an object type O_1 with type domain O_2 corresponds to that has-component relationship of object type O_2 with type domain O_1[7]. Inverse declarations are optional for pairs of component-of and has-component relationships. If indicated, always the has-component relationship of a pair is annotated.

In a type diagram, component-of relationships are depicted by bars with vertical side

[6] In [KBG89], Kim *et al.* extended their notion of component objects to include also exclusive and independent component objects, shared and dependent component objects, as well as shared and independent component objects. In this thesis, however, we stick to the term's original semantics.

[7] Note: Two relationships may be declared to be inverses only in their original definition (cf. specialization of the structure of objects, below). Consequently, the statement on the implicit availability of the inverse of a component-of relationship considers only original definitions of component-of relationships and has-component relationships.

bars, has-component relationships by rectangles with vertical side bars.

Example: Each reservation has a component-of relationship to a value from domain ACCOMMODATION (cf. Figure 7.1), and each accommodation has a has-component relationship reservations to a value from domain RESERVATION (cf. Figure 7.2). These relationships are declared to be inverses.

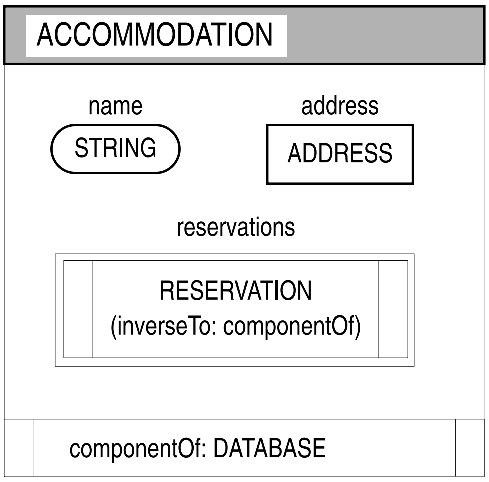


Figure 7.2: Object type ACCOMMODATION

A *has-indirect-component relationship* models a relationship of a composite object to several indirect component objects of a specific type. Note: An object o_3 is an indirect component of another object o_1 if there is an object o_2 such that o_3 is a direct component of o_2 and o_2 is either a direct or an indirect component of o_1. Each object may have several has-indirect-component relationships. Like has-component relationships, has-indirect-component relationships are *object-describing*. Has-indirect-component relationships are always *derived* relationships. The derivation expression of a has-indirect-component relationship corresponds to a cascade of has-component relationships starting at the object type at which the has-indirect-component relationship is defined and going down to the type of the indirect components. There is no explicit inverse relationship for 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 components and going up to the object type at which the has-indirect-component relationship is defined.

In a type diagram, a has-indirect-component relationship is depicted like a derived has-component relationship. However, the side bars of a rectangle that represents a has-indirect-component relationship are depicted with a dashed-line border, to distinguish between a has-indirect-component relationship and a derived has-component relationship.

Example: The database object has a has-indirect-component relationship reservations to values from domain RESERVATION (cf. Figure 7.3). Note: Reservations are direct components of accommodations (cf. Figure 7.2), which are direct components of a database.

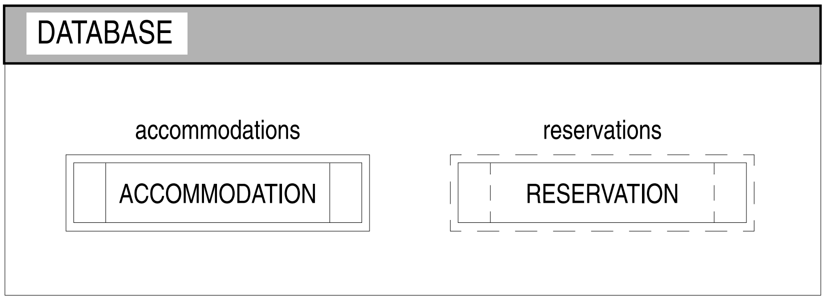


Figure 7.3: Object type DATABASE

Component-of relationships define a *composite-object hierarchy* that forms a tree in which the location of each component object is uniquely determined (*Unique Locality Rule for Component Objects*). Furthermore, since each object type may define only one component-of relationship for its instances, definitions of component-of relationships organize object types into a *composite-object-type hierarchy*, in which the usage of object types is restricted to a single context (*Single Context Rule for Object Types*). The *Single Context Rule for Object Types* makes it hard to represent situations in which real-world entities represented by objects of a particular type shall appear in different contexts. Has-indirect-component relationships as well as role-of relationships (below) are elegant means to cope with the limitation caused by the *Single Context Rule for Object Types*. They allow to represent the same real-world entities in different contexts obeying the *Single Context Rule for Object Types*. A has-indirect-component relationship represents an object in a more general context, as an indirect component. A role-of relationship allows to represent an object in a different and, usually, more specific context. That is, with role-of relationships, a single real-world entity is represented by a set of objects related via role-of relationships. Each object represents the same real-world entity, but in a different context.

Component-of relationships define a composite-object hierarchy, which forms a *directed acyclic graph*. That is, that no object may be a direct or indirect component of itself. For consistency of a composite-object hierarchy, we require further that no composite object may have direct or indirect components of the same type (cf. *Acyclicity Rule for Object-Defining Relationships*, below).

Kappel and Schrefl argued that maintaining objects belonging exclusively to one component of a system as members of global object classes would make a system hard to survey and hard to maintain (cf. [KS92, KS96b]). They have introduced *local object classes* to model such situations adequately. A local object class is a property of that composite object of which its members are components (cf. Subsection 7.2.3). To ensure that each object is maintained as member of some local class, we require that each object is component of another object. Previously independent objects are implicitly components of a special object, the *database object*. The database object is the root of the composite-object hierarchy, which contains all objects stored in a database. Consequently, there is a single composite-object hierarchy for a database, in which the location of each object stored in the database is uniquely determined (cf. *Unique Locality Rule for Component Objects*, above). Each object type must specify a component-of relationship for its instances, unless its instances are direct components of the database object.

Has-Constituent Relationship. A *has-constituent relationship* models a relationship of an aggregate object to one of its constituent objects. Each aggregate object may have several, usually two or more, has-constituent relationships. Of course, aggregate objects may also have attributes and other kinds of relationships. The subordinate objects participating in has-constituent relationships of an aggregate object are called its constituents, as they taken together "constitute" the aggregate object. This means, the existence of an aggregate object depends on the existence of each of its constituents. Consequently, has-constituent relationships are *object-defining*. For example, aggregate objects can be used to realize "relationship objects". Relationship objects represent general relationships (above) that are undirected (cf. Subsection 3.2.1) or have individual properties and behavior. Note: Relationship objects have the same semantics as aggregate objects, i.e., their existence depends on the existence of objects participating in the relationship and objects may participate in different relationships.

Relationships inverse to has-constituent relationships are constituent-of relationships. A *constituent-of relationship* models a relationship of a constituent object to several aggregate objects of a specific type. Each constituent object may have several constituent-of relationships. Consequently, constituent-of relationships are *object-describing*. There is an inverse constituent-of relationship for every has-constituent relationship and vice versa[8]. Inverse declarations are obligatory for pairs of has-constituent and constituent-of relationships. Inverses are always indicated by annotating the constituent-of relationship of a pair.

In a type diagram, a has-constituent relationship is depicted by a bar with a superimposed hexagon, a constituent-of relationship by a rectangle with a superimposed hexagon.

Example: Figure 7.4 shows the type diagrams of object types TARIFF, SEASON and CATEGORY. Each tariff object models a binary relationship between a season object and a room category object. More precisely, each tariff object has a has-constituent relationship season to a value from domain SEASON and a has-constituent relationship roomCategory to a value from domain CATEGORY. In addition, a tariff object is described by the value of attribute price. Both object types, SEASON as well as CATEGORY, define a constituent-of relationship tariffs that is inverse to the corresponding has-constituent relationship specified at object type TARIFF. The mandatory inverse declarations are shown by annotating the constituent-of relationships tariffs defined at object types SEASON and CATEGORY, respectively.

[8]Note: Two relationships may be declared to be inverses only in their original definition (cf. specialization of the structure of objects, below).

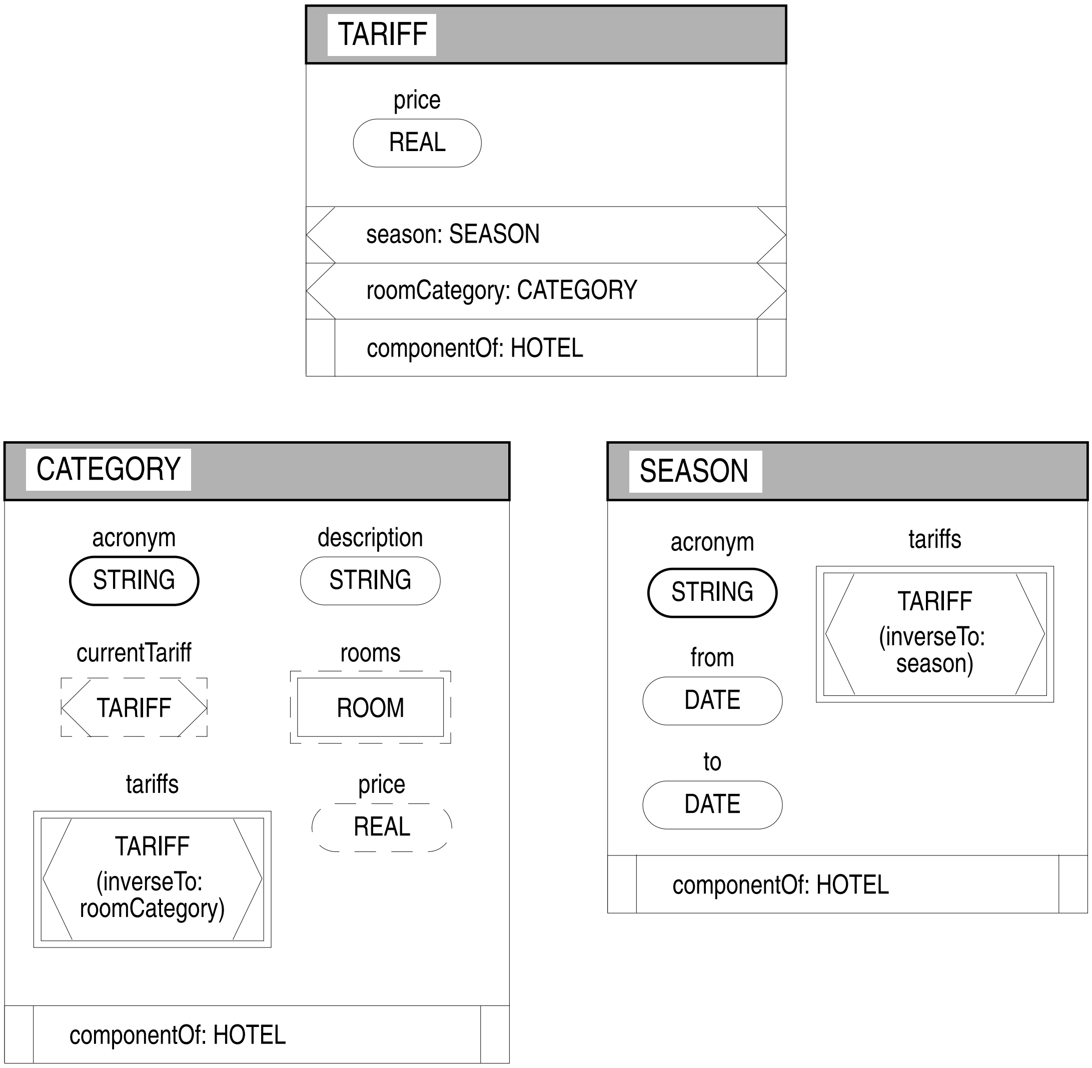


Figure 7.4: Object types TARIFF, SEASON, and CATEGORY

Has-constituent relationships define *aggregate-object hierarchies*, which form directed acyclic graphs. That is, that no object may be a direct or indirect constituent of itself. For consistency of aggregate-object hierarchies, we require further that no aggregate object may have direct or indirect constituents of the same type. Furthermore, an object must not be direct or indirect constituent and direct or indirect component of objects of the same type (cf. *Acyclicity Rule for Object-Defining Relationships*, below).

Role-Of Relationship. A *role-of relationship* models a relationship of a role object to another object, the player of the role, that represents the same real-world entity, but in a more general context. Each role object has exactly one role-of relationship. The existence of a role object depends on its player object. Consequently, role-of relationships are *object-defining*. Since an object type may define only one role-of relationship for its instances, naming of role-of relationships is optional. If a role-of relationship remains unnamed, the player object 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 relationship of a player object to several role objects of a specific type that represent the same real-world entity, but in a more specific context. If a has-role relationship is multi-valued, a player object may play several roles of the same type at a time. That is, role objects are a means to represent a single real-world entity several times in similar contexts, i.e., as components of different objects of the same type. Moreover, each player object may have several has-role relationships. That is, roles are also a means to represent a single real-world entity in different contexts, i.e., as components of objects of different types. Consequently, has-role relationships are *object-describing*.

There is an inverse has-role relationship for every role-of relationship and vice versa[9]. Since each object type may define only one role-of relationship for its instances, and since every relationship may have only one inverse relationship, each object type may be used as domain of only one has-role relationship. Another consequence is, that the knowledge about the inverse of a role-of relationship is implicitly available and does not have to be indicated. Note: The inverse of the role-of relationship of an object type O_1 with type domain O_2 corresponds to that has-role relationship of object type O_2 with type domain O_1[9]. Inverse declarations are optional for pairs of role-of and has-role relationships. If indicated, always the has-role relationship of a pair is annotated.

In a type diagram, a role-of relationship is depicted by a bar with a superimposed parallelogram, a has-role relationship by a rectangle with a superimposed parallelogram.

Example: Figure 7.5 shows the type diagrams of object types PERSON and GUEST. Each person object models a human. It is identified by the value of its key attribute SSN and is described by the values of attributes name and birthday. Furthermore, a person has a general relationship address to a value from domain ADDRESS. A person object represents a human in the general context of a database, i.e., as user of a database. However, a person may also occur in more specific contexts, i.e., it may play several roles. For example, a person may occur as guest of several hotels. In this case, a person object has a has-role relationship asGuest to one or several guest objects, which represent the person in the context of one or of different hotels. A person may also occur as employee of one hotel. In this case, a person object has a has-role relationship asEmployee to one employee object, which represents the person in the context of the hotel for which the person works. Note: Object type EMPLOYEE is not shown.

An instance of object type GUEST represents a person as guest of a particular hotel (cf. component-of relationship). Each guest object has a role-of relationship to a person object that represents the same human. A guest object is described by the

[9]Note: Two relationships may be declared to be inverses only in their original definition (cf. specialization of the structure of objects, below). Consequently, the statement on the implicit availability of the inverse of a role-of relationship considers only original definitions of role-of relationships and has-role relationships.

values of attributes creditworthiness and specialNeeds. Note: Attribute specialNeeds is a multi-valued property.

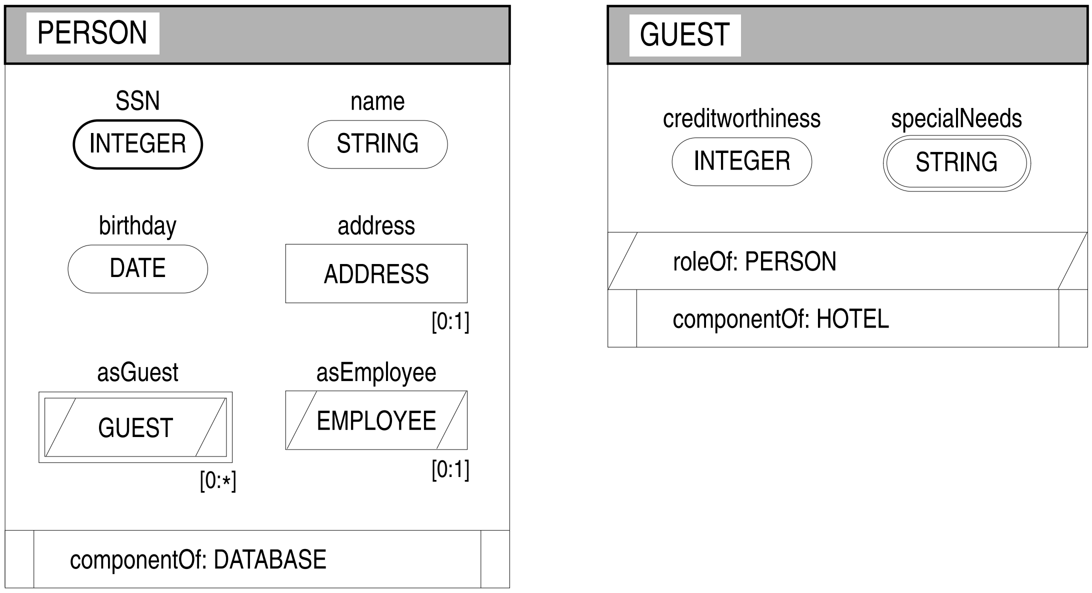


Figure 7.5: Object types PERSON and GUEST

Role-of relationships are a mechanism to realize the concept of *object specialization* (cf. [GSR96, Sch88, Sci89]). Role-of relationships organize different objects that represent the same real-world entity in a specialization hierarchy, called *role hierarchy*. The root object of a role hierarchy exists as long as the corresponding real-world entity is represented within a system. Subordinate objects, called *roles*, can be acquired and abandoned dynamically and independently of each other. Thus, role hierarchies are a very flexible means to deal with evolving entities, i.e., entities that change their type dynamically, within information systems. With type hierarchies, the object that represents a real-world entity must be replaced by an instance of another type, if the real-world entity evolves to another type. This replacement will cause very likely referential integrity problems.

Example: In 1997, the "Employee of the Year" award of the five star hotel "Five Wings" goes to Mrs. Green. The prize for this award is a free stay in the five star hotel "Seven Nuggets", a partner hotel of the "Five Wings". When Mrs. Green becomes guest of the "Seven Nuggets" she evolves from a person that is employee to a person that is employee and guest of a hotel. This can be easily achieved with role objects. The person object representing Mrs. Green, it is the root of the corresponding role hierarchy, acquires a role object of type GUEST, which represents her as guest of the "Seven Nuggets". With type hierarchies, we would have to replace the existing employee object by a new guest object, where object type GUEST would have to be a subtype (!) of object type EMPLOYEE.

With type hierarchies, an entity is usually represented only once as an instance of a particular type (cf. [BM91]). If the entity is represented by several instances, however, the information that they represent the same real-world entity gets lost. That means, type hierarchies cannot handle multiple occurrences of one entity in the same type in a consistent way. With role hierarchies, multiple occurrences of an entity in the same type can be easily achieved, i.e., each object can acquire several roles of a particular type, if they belong to different contexts. That is, *role hierarchies handle multiple occurrences of an entity in the same type in a natural and consistent way.* In the context of role hierarchies, the notion of object identity (cf. [KC86]) is extended to the notion of *entity identity* (e.g., [GSR96]). Two objects are said to be entity equivalent if they represent the same real-world entity.

Example: Modeling Mrs. Green as guest of other hotels can be achieved with role objects in the same way as before. That is, the person object representing Mrs. Green acquires further role objects of type GUEST, one for each hotel at which she stays. With type hierarchies, Mrs. Green would have to be represented by several instances of type GUEST. Since object type GUEST is a subtype (above) of object type EMPLOYEE, properties describing Mrs. Green as employee must be stored redundantly in each instance of type GUEST. Moreover, the information that several instances of type GUEST represent the same real-world entity, i.e., Mrs. Green, would be not available.

In a database schema, role hierarchies are specified by role-of relationships. Since each object type may specify only one role-of relationship for its instances, and since each object type may serve only as domain of one has-role relationship, role-of relationships define a hierarchy of object types, called *role-type hierarchy.* A role-type hierarchy defines the types of roles an instance of the root type of the hierarchy may acquire and abandon during its existence. Contrary to traditional type hierarchies (below), where a subordinate type inherits the specifications of structure and behavior defined at the superordinate type, a subordinate type in a role-type hierarchy does not inherit from its superordinate type. Instead, inheritance occurs at the instance level. That is, a subordinate object in a role hierarchy, i.e., a role, inherits property values and behavior of its superordinate object, called its player. That is, various roles representing the same real-world entity share common structure and behavior. Actually, a role object delegates a message it does not understand to its player object.

Example: The social security number, the name, the birthday, as well as the private address of Mrs. Green will be stored only once in the person object representing her. The employee object and all guest objects that represent her will share the same information stored in the person object representing her.

Role-of relationships define *role hierarchies*, which form directed acyclic graphs. That is, that no object may be a direct or indirect role of itself. For consistency of role hierarchies, we require further that no player object may have direct or indirect roles of the same type. Furthermore, an object must not be direct or indirect role and direct or indirect component of objects of the same type, and must not be a direct or indirect role and direct or indirect constituent of objects of the same type (*Acyclicity Rule for Object-Defining Relationships*).

Specialization of the Structure of Business Objects

Object types may be organized in *type hierarchies*, which are specified by subtype-supertype relationships between object types. We offer a set of predefined conceptual types, which designers specialize when developing a conceptual schema of a particular business area. Since predefined conceptual types incorporate special semantics of entities found frequently in a particular kind of business area, specializing predefined conceptual types results in uniform conceptual schemas and reduces the overall effort of designing conceptual schemas as their design need not start from scratch. The predefined conceptual types are organized in a type hierarchy (cf. Figure 7.6). The root of this type hierarchy is conceptual type B_OBJECT (Business OBJECT), which models the basic structure and behavior each object maintained in a workflow system offers to its environment. Each user-defined object type is a direct or indirect subtype of conceptual type B_OBJECT or of one of its predefined subtypes BP_OBJECT, WFP_OBJECT, O_OBJECT, and A_CONTRACT, which are explained below. Note: Figure 7.6 shows a type-hierarchy diagram depicting subtype-supertype relationships between the predefined conceptual types just introduced. Type-hierarchy diagrams are explained below.

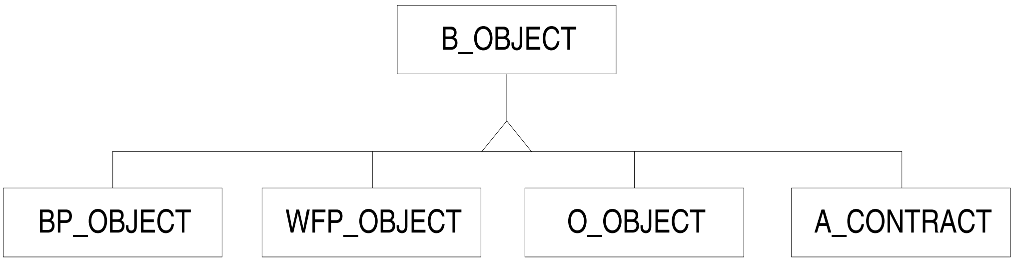


Figure 7.6: Conceptual-type hierarchy with root B_OBJECT

Conceptual type BP_OBJECT (Business-Process OBJECT) models the basic structure and behavior of all business-process objects (cf. Section 8.2). Conceptual type WFP_OBJECT (WorkFlow-Process OBJECT) basic structure and behavior of all workflow-process objects (cf. Section 8.3). Conceptual type O_OBJECT (Organizational OBJECT) basic structure and behavior of all organizational objects (cf. Chapter 9), and conceptual type A_CONTRACT (Authorization CONTRACT) basic structure and behavior of all authorization contracts, which model authorizations in workflow systems (cf. Chapter 10). In fact, we offer further conceptual types of organizational objects that incorporate special semantics of organizational entities that occur frequently in real-world organizations, e.g., actors and organizational units. These conceptual types, which are direct or indirect subtypes of conceptual type O_OBJECT, are discussed in Chapter 9. Note: *All predefined conceptual types, except the predefined class type* BO_CLASS *(cf. Subsection 7.2.3), are abstract types.*.

Type hierarchies form directed acyclic graphs. That is, no object type may be directly or indirectly a subtype (or supertype) of itself (*Acyclicity Rule for Subtype-Supertype Relationships*). Each subtype may specialize the structure it inherits from its supertype. To specialize inherited structure means either to *extend* it by specifying new properties or to *refine* it by modifying the specification of inherited ones. For consistency of conceptual schemas, we strictly separate redefinitions of a property from the property's original definition. A property redefinition must share the name with the property's original definition and may constrain the domain and the cardinality of the property, however, it must not change the property's original multiplicity. Furthermore, inverse declarations of relationships are only given in the relationships' original definitions and may not be redefined. This means, if defined, the inverse of a relationship r is always determined by referring to the original definition of r.

To enforce their redefinition, object types and properties can be explicitly declared to be *abstract*[10]. The redefinition of an abstract object-type or an abstract property may again be abstract. Unlike the redefinition of a concrete property, the redefinition of an abstract property must not share the name with its original definition. Graphically, a modeling element that has been explicitly declared to be abstract is depicted with a small black triangle in the upper right corner of the graphical symbol representing it.

Graphically, subtype-supertype relationships between object types are depicted in type diagrams and in type-hierarchy diagrams. Type-hierarchy diagrams are explained below. A type diagram showing a subtype O', represents the subtype-supertype relationship between O' and its direct supertype O by giving the name of supertype O in a bar labeled subtypeOf. Since a subtype-supertype relationship is not a property of the instances of an object type, it is a property of the type object itself, the bar is not located in the body of a type diagram, but in its foot. A type diagram showing a subtype may, but need not, depict inherited property definitions. Usually, an inherited property is depicted only if it is modified. A graphical symbol representing an inherited property that is not modified at a subtype is shown with a dotted-line border. Note: A type diagram showing a supertype O does not depict subtype-supertype relationships between O and direct subtypes of O, i.e., a subtype-supertype relationship between two object types is depicted only in type diagrams showing the subordinate type.

Example: Figure 7.7 shows type diagrams of object types ROOM-RESERVATION and TICKET-RESERVATION, which are direct subtypes of object type RESERVATION (cf. Figure 7.1).

A *type-hierarchy diagram* emphasizes the hierarchical nature of the subtype-supertype relationships. Type hierarchies are represented by type-hierarchy graphs. Nodes of the type-hierarchy graphs correspond to object types. Nodes are depicted by rectangles showing the name of the object type inside. Graphically, a subtype-supertype relationship is depicted by a solid line with a triangle symbol located in the middle of the line. The climax of the triangle is pointing to the node representing the supertype in a subtype-supertype relationship. The graphical presentation of different subtype-supertype relationships partially coincides if they share the same supertype. The graphical notation for subtype-supertype relationships is similar to that of OMT (cf. [RBPE91]).

[10]Note: The *Abstract Supertype Rule for Object Types* (above) *implicitly* declares supertypes to be abstract.

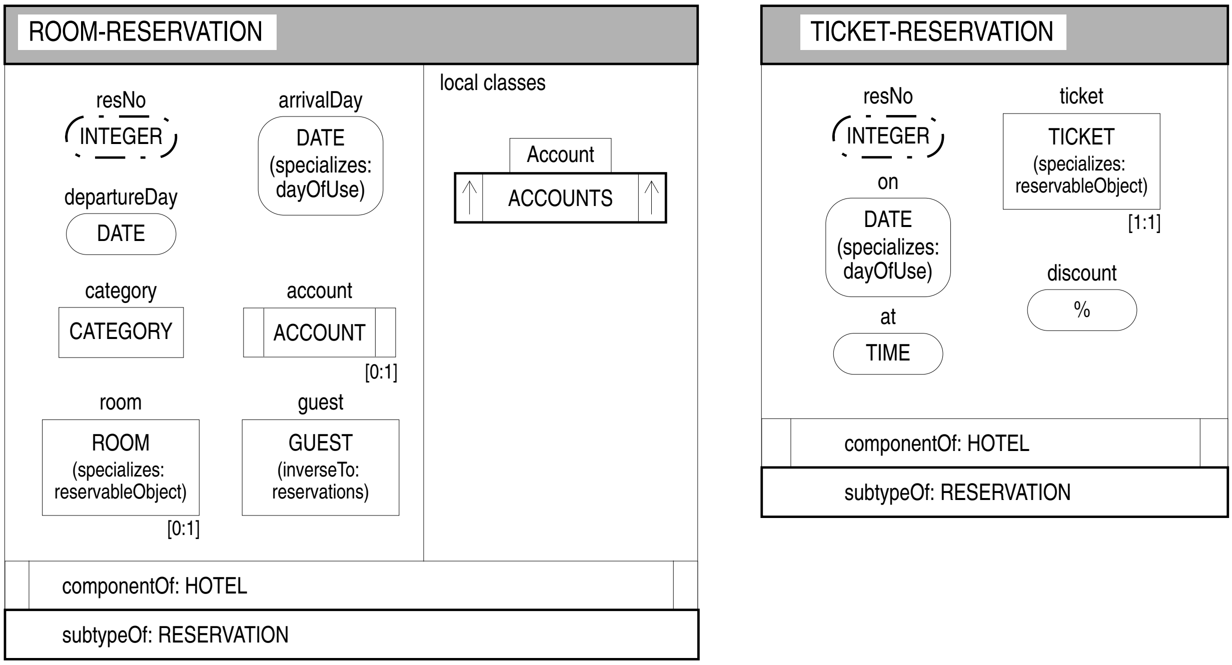


Figure 7.7: Object types ROOM-RESERVATION and TICKET-RESERVATION

Example: Figure 7.8 shows a type-hierarchy diagram with object types RESERVATION, ROOM-RESERVATION, and TICKET-RESERVATION.

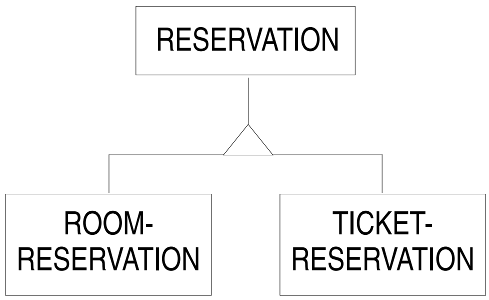


Figure 7.8: Type-hierarchy diagram

Below, we give a set of consistency rules that ensure that subtypes do not specialize inherited structure in an arbitrary way:

1. Every subtype inherits all properties from its supertype and may define new properties for its instances (*Rule of Extension of Properties*).

   Example: Object types ROOM-RESERVATION and TICKET-RESERVATION inherit all properties from their supertype RESERVATION (cf. Figures 7.1 and 7.7). That is, they inherit attributes resNo, dayOfUse, and price with domains INTEGER, DATE, and REAL, respectively, and relationships reservableObject and componentOf with domains RESERVABLE_OBJECT and ACCOMMODATION, respectively.

Object type ROOM-RESERVATION extends its supertype, in that it specifies a new attribute departureDay with domain DATE and new relationships guest and category with domains GUEST and CATEGORY, respectively, for its instances.

Object type TICKET-RESERVATION extends its supertype, in that it specifies new attributes on with domain DATE, at with domain TIME, and discount with domain % (percentage) for its instances.

2. Every property must be introduced at a single object type. If a property is defined at several object types, this property must also be defined at a common supertype of these object types (*Rule of Origin of Properties*).

Example: Attribute resNo is defined at object type ROOM-RESERVATION *and* at object type TICKET-RESERVATION (cf. Figure 7.7). It has been introduced at object type RESERVATION (cf. Figure 7.1), which is a common supertype of both object types.

Comment 7.1 (Single Inheritance vs. Multiple Inheritance)
The *Rule of Origin of Properties* might result in situations that require *multiple inheritance*. For example, consider object types O, O_1, O_2, and O_3, where object types O_1 to O_3 are direct subtypes of O. Assume there is a property m that is defined with O_1 and O_2 but is not defined with O_3. To obey to the *Rule of Origin of Properties* we must add an additional object type, say O_{12}, to the given type hierarchy, where O_{12} is direct subtype of O and O_{12} is direct supertype of O_1 and O_2, and introduce property m at O_{12}. Now, if there is another property n that is defined with, for example, O_2 and O_3 but is not defined with O_1, we would have to add an object type O_{23} in a way similar to O_{12}. This would result in a situation where an object type, i.e., O_2, has multiple direct supertypes, i.e., O_{12} and O_{23}.

For simplicity, we consider only *single inheritance* throughout the discussion of the underlying object model. The extension to multiple inheritance is straightforward. We claim that the same set of rules can be used in the context of multiple inheritance to test whether an object type specializes consistently its supertype(s) if we adhere to a special procedure. Assume there is a hierarchy of object types O_1, O_2, O_3, and O_4. Object types O_2 and O_3 are direct subtypes of object types O_1 and object type O_4 is a direct subtype of object types O_2 and O_3 (cf. Figure 7.9(a)). In short, object type O_4 is a consistent specialization of object types O_2 and O_3, if O_4 extended by features inherited from O_3 consistently specializes O_2 (cf. Figure 7.9(b)) and, if O_4 extended by features inherited from O_2 consistently specializes O_3 (cf. Figure 7.9(c)).

3. Every object-defining property[11] defined at an object type O must also be defined at the supertype of O, unless the supertype of O is a predefined conceptual type (cf. conceptual types, above). That is, object-defining properties may not be added at a subtype of a user-defined object type (*Rule of Immediate Definition of Object-Defining Properties*).

[11] Object-defining properties are either key properties or object-defining relationships.

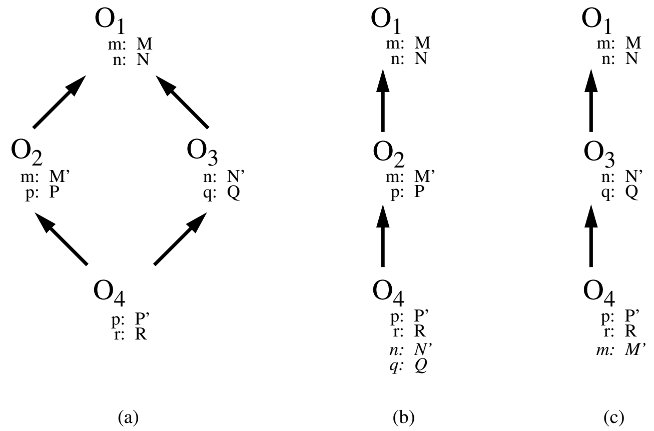


Figure 7.9: Testing consistency of multiple inheritance

Example: Key attribute resNo as well as the object-defining component-of relationship defined at object types ROOM-RESERVATION and TICKET-RESERVATION (cf. Figure 7.7) are also defined at their common supertype RESERVATION (cf. Figure 7.1).

4. A subtype may redefine the domain of an inherited property to a direct or indirect subtype of the inherited domain. The domain of attribute properties must not be redefined at subtypes (*Rule of Specialization of Domains of Properties*).

   Example: Object type ROOM-RESERVATION (cf. Figure 7.7) redefines the domain of the inherited component-of relationship to object type HOTEL, which is a direct subtype of object type ACCOMMODATION (cf. Figure 7.2).

5. A subtype may redefine an inherited abstract property. A concrete type, i.e., a type at a leaf node of a type hierarchy of which instances shall be created, *must* redefine inherited abstract properties. An abstract property is redefined by changing its name (optional), by specializing its domain to a direct or indirect subtype of the inherited domain, and by restricting its cardinality (below). Note: The domain of attribute properties must not be redefined at subtypes (*Rule of Specialization of Abstract Properties*).

   Example: Object type ROOM-RESERVATION (cf. Figure 7.7) redefines the name of the inherited abstract relationship reservableObject to room and its inherited domain to object type ROOM, which is a direct subtype of object type RESERVABLE_OBJECT (not shown). Moreover, it redefines the name of the inherited abstract attribute dayOfUse to arrivalDay.

   Object type TICKET-RESERVATION (cf. Figure 7.7) redefines the name of the inherited abstract relationship reservableObject to ticket, redefines its domain to object type TICKET, which is another direct subtype of RESERVABLE_OBJECT (not shown), and restricts its inherited cardinality, i.e., the previously optional relationship is declared to be mandatory. Moreover, it redefines the name of the inherited abstract attribute dayOfUse to on.

6. A subtype may restrict the cardinality[12] of an inherited property (*Rule of Specialization of Cardinalities of Properties*). Note: When redefining a property its original multiplicity must not be modified[13] (*Rule of Specialization of the Multiplicity of Properties*).

7. An object type O_1 must not specify an object-defining relationship r with domain O_2 for its instances, if O_2 is a direct or indirect subtype of O_1 or a direct or indirect supertype of O_1 (*Rule of Orthogonality of Subtype-Supertype Relationships and Object-Defining Relationships*).

7.2.2 Modeling the Behavior of Business Objects

The behavior of business objects (or *objects* for short) is determined by a set of *activities*, by *allowed execution orders* of these activities, and by their *interactions* with other objects (cf. Subsection 3.2.2). Considering synchronous interaction, an interaction between two or more objects can only take place if the participating objects are ready to interact, i.e., if they reside in an appropriate state. This means, the behavior of interrelated objects must be synchronized for successful interactions. After discussing means to synchronize the behavior of business objects, we discuss specialization of the behavior of business objects.

Synchronization of the Behavior of Business Objects

An object-oriented database usually consists of a set of interrelated objects, whose behavior must be synchronized. In Chapter 3, the only means to synchronize the behavior of objects we learned about were activity specifications. Specifying a set of life-cycle states that belong to the domain of a port p of an activity t as prestates or poststates of port p, synchronizes the behavior of objects on which activity t is invoked and the behavior of the objects assigned to port p as actual parameter.

Example: Figure 7.10 shows the specification of activity checkIn, redefined at object type ROOM-RESERVATION. The specification of activity checkIn synchronizes the behavior of room reservations and rooms, since activity checkIn can only be invoked on a room reservation that resides in state issued if there is a room that resides in state free. Note: Port r of activity checkIn has been declared mandatory.

[12] A cardinality v'_{min}:v'_{max} restricts a cardinality v_{min}:v_{max} if $v'_{min} \geq v_{min}$ and $v'_{max} \leq v_{max}$ and if either $v'_{min} > v_{min}$ or $v'_{max} < v_{max}$ (cf. Subsection 3.2.3).

[13] This means, a single-valued property must not become multi-valued when redefined and a multi-valued property must not become single-valued.

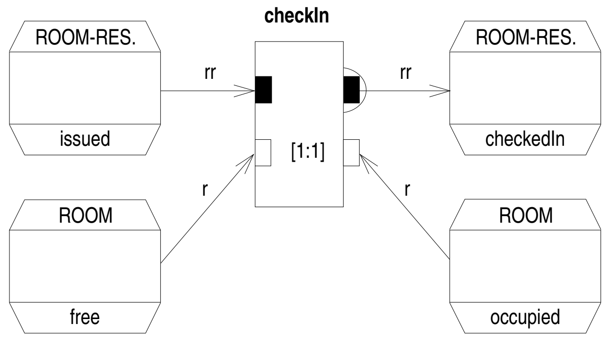


Figure 7.10: Specification of activity checkIn at ROOM-RESERVATION

Specifying life-cycle states as prestates or poststates of ports of an activity restricts possible execution sequences of activities defined for a single object (*intra-object synchronization*). Moreover, it restricts possible execution sequences of activities defined for different objects (*inter-object synchronization*). So far, however, synchronization of the behavior of different objects relies on the assignment of objects to ports of activities invoked on other objects. Subsequently, we will discuss further means to synchronize the behavior of different objects. Synchronization constraints are either specified with activities, e.g., prestates and poststates of ports, or with states. Similar to activities, synchronization constraints on states can be specified either as *precondition* or *postcondition*. A constraint specified as precondition must be fulfilled at the time the constrained state is entered, one specified as postcondition at the time the constrained state is left. Since states correspond to phases in the life-cycles of objects, a synchronization constraint on a state may also be specified as an *invariant*[14], which, by the way, is the most frequent case. A constraint specified as an invariant must be fulfilled at any time an object resides in the constrained state. To allow to define constraints on states either as precondition, invariant, or postcondition, state specifications (cf. Subsection 3.2.1) have been extended accordingly. In general, constraints on states can be specified graphically, e.g., domain-restriction constraints (below), or textually, e.g., using an OQL-like language (cf. [Atw93, CB97]). If specified graphically, the three categories of state constraints are separated by vertical solid lines. The individual sections are marked with the keywords pre, inv, and post, respectively. If no keyword is shown, the depicted constraints are invariants, which is the default. If specified textually, state constraints are shown as annotations below the graphical symbol representing a state. The keywords precondition, invariant, and postcondition, respectively, are used to distinguish the different categories of constraints. Note: State specifications in their extended form, i.e., state specifications considering also preconditions, invariants, as well as postconditions associated with states, are only shown in state-specification diagrams (cf. Subsection 3.2.2).

A *domain-restriction constraint* is specified between two states. It restricts the domain of a relationship r of objects of a type O that reside in a state s_1 to objects of the domain

[14] Of course, activities do also take time in our model, but they rather correspond to events in the life-cycles of objects then to phases. Thus, we do not support the specification of constraints on activities as invariants.

of r that reside in a state s_2. Graphically, a domain-restriction constraint is shown in a life-cycle diagram or an activity-specification diagram of type O. The domain-restriction constraint is depicted within the constrained life-cycle state s_1 showing state s_2 within a rectangle representing relationship r.

Example: Figure 7.11 shows the specification of a domain-restriction constraint on relationship room of objects of type ROOM-RESERVATION. It says that a room that is referenced by a room reservation in state checkedIn must reside in state occupied. Note: The domain-restriction constraint is implicitly specified as state invariant, since it is not declared to be a precondition or a postcondition.

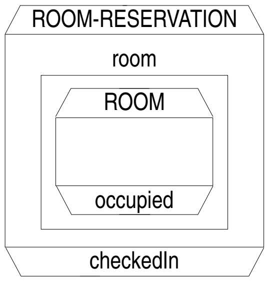


Figure 7.11: Domain-restriction constraint on state checkedIn of room reservations

A *visibility-restriction constraint* is specified between two activities. It expresses that an activity t defined at an object type O *may only be* invoked by a particular activity t' defined at another object type O'. The visibility of activity t is limited to activity t' by specifying an additional input port p for activity t. The domain of port p is object type O' at which activity t' is defined, the prestate of port p is the *activity state* of t'. Note: Restricting the visibility of an activity t to another activity t' does not express that t' must always invoke t when performed. There are two alternative ways to specify such a constraint. If the object on which the constrained activity t must be performed is assigned to a secondary port p' of activity t', prestates and poststates of activity t shall be specified as prestates and poststates, respectively, of secondary port p' of activity t'. Otherwise, if the object on which activity t must be performed is referenced via some relationship r within activity t', prestates and poststates of activity t shall be specified as domain-restriction constraints on a prestate and poststate, respectively, of activity t'.

Graphically, a visibility-restriction constraint of an activity is depicted in its specification diagram. In a life-cycle diagram, an activity on which a visibility-restriction constraint has been specified is shown in light-grey color to indicate that it is not public in general.

Example: Figure 7.12 shows the specification of a visibility-restriction constraint on activity occupy of object type ROOM. Activity state checkIn of object type ROOM-RESERVATION is specified as prestate of input port rr of activity occupy. Note: The visibility-restriction constraint does not ensure that activity checkIn of object type

ROOM-RESERVATION does always invoke activity occupy. A corresponding constraint must be specified with the calling activity (cf. prestate and poststate defined for activity checkIn in Figure 7.10).

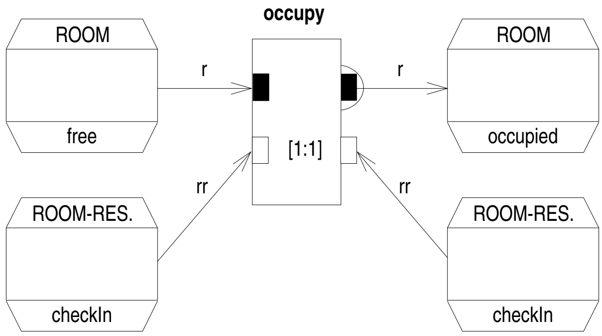


Figure 7.12: Visibility-restriction constraint on activity occupy defined for rooms

Specialization of the Behavior of Business Objects

If object types are organized in a type hierarchy, a subtype may specialize structure *and* behavior it inherits from its supertype. To specialize the behavior of objects means, to specialize *activities* and *life-cycles* defined for them. The predefined conceptual type B_OBJECT (Business OBJECT, cf. Figure 7.6) models the basic behavior each object offers to its environment. That is, each object offers abstract activities create and destroy as well as an abstract state exists organized in a basic life-cycle (cf. Figure 7.13). Each user-defined object type is a direct or indirect subtype of conceptual type B_OBJECT and may specialize this basic behavior.

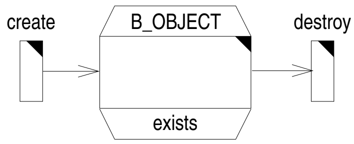


Figure 7.13: Life-cycle at B_OBJECT

Below, we give a set of consistency criteria that say in which way a subtype may specialize inherited activities and inherited life-cycles:

Specialization of activities corresponds to the concept of specialization of operations known from object-oriented programming languages (OOPLs). Different notions of consistent specialization of operations have been discussed (*conformance rules*): *Arbitrary inheritance* allows to change operation interfaces in an unrestricted way. *Strict inheritance* allows no changes to operation interfaces at all. In between, two alternative notions of consistent specialization exist: 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. Object/Behavior Diagrams (cf. [KS91, KS96b]) support a covariant specialization of activities.

Example: Figure 7.14 shows the activity-specification diagram of activity pay introduced at object type RESERVATION.

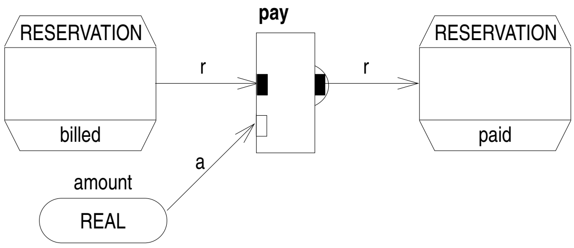


Figure 7.14: Specification of activity pay at RESERVATION

Example: Figure 7.15(a) shows the activity-specification diagram of activity pay as it has been redefined at object type ROOM-RESERVATION. A room reservation can only be paid for if the room, in which the guest lived, resides in life-cycle state occupied. After a room reservation has been paid for, the previously occupied room must reside in life-cycle state toBeServiced.—Note: These constraints that have been specified graphically in form of domain-restriction constraints are a precondition and a postcondition, respectively, of activity pay and must not be confused with a postcondition or an invariant on state billed and a precondition or an invariant on state paid, respectively, of room reservations. Actually, preconditions, invariants, and postconditions defined on life-cycle states are shown only in separate state-specification diagrams.—Such rooms get cleaned by the room service and will be vacant afterwards. Actually, activity pay has been refined at object type ROOM-RESERVATION into alternative activities payCash, payByCreditCard, payByMoneyTransfer, and byByCollectionAgency (cf. Chapter 3, Figure 3.3). Figure 7.15(b) shows the activity-specification diagram of activity payByCreditCard. Activity payByCreditCard requires an additional parameter—the credit card number with domain STRING—and specifies a textual precondition saying that it may only be invoked at the departure day of the guest.

Specialization of life-cycles has been discussed in [SS95, SS97]. Schrefl and Stumptner introduced a set of different notions of behavior consistency, which somehow correspond to the notions of covariance and contra-variance discussed in the context of specialization of operations. To specialize an inherited life-cycle means to *extend* it by adding new activities and states and to *refine* inherited activities and states into a life-cycle that defines the behavior of objects in more detail (cf. developing life-cycles by stepwise refinements, Subsection 3.2.2). First, different notions of behavior consistency for extensions of life-cycles at subtypes[15] are discussed, i.e., observation consistency, weak invocation

[15]A life-cycle may be extended by adding "parallel branches", i.e., there are inherited activities that produce added poststates or consume added prestates, or both, or "alternative branches", i.e., there are added activities that consume inherited prestates or produce inherited poststates, or both, at a subtype.

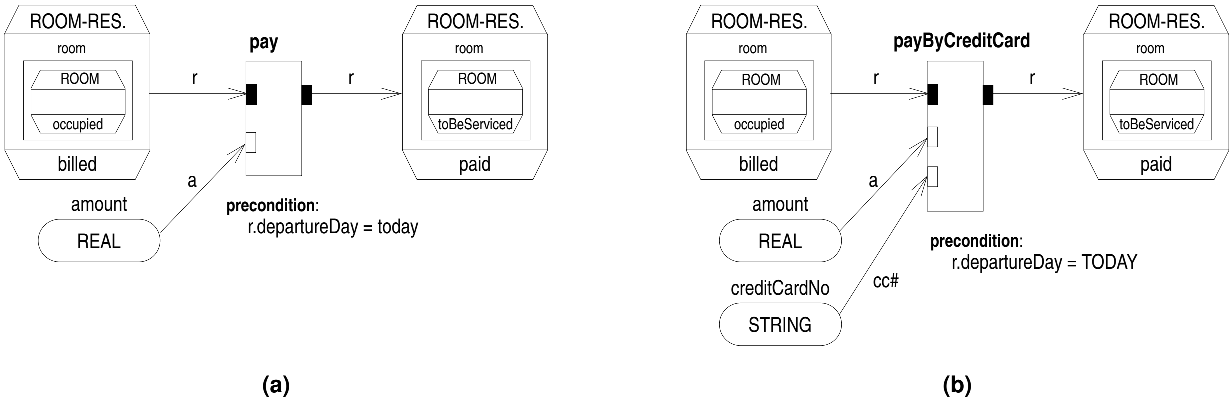


Figure 7.15: Specification of activities pay and payByCreditCard at ROOM-RESERVATION

consistency, and strong invocation consistency. Figure 7.16 schematically shows changes of inherited life-cycles allowed by the different notions of behavior consistency. Second, behavior consistency for refinements of activities and states of life-cycles at subtypes is discussed.

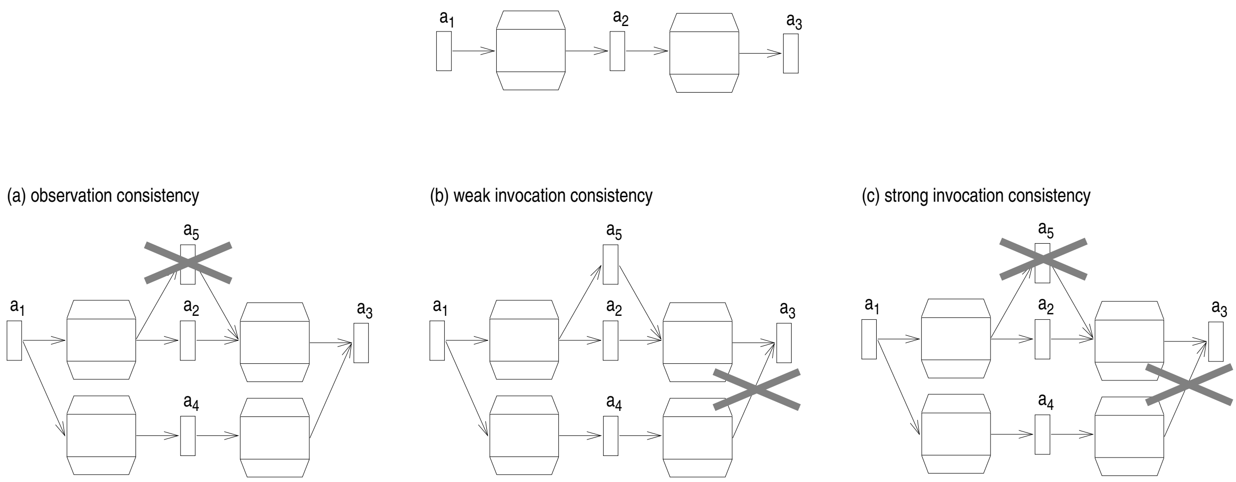


Figure 7.16: Notions of behavior consistency (extension)

Observation consistency requires that the behavior of instances of a subtype must always be observable according to the life-cycle specified at a supertype. As a consequence, a subtype may extend an inherited life-cycle by "parallel branches", i.e., inherited activities may produce additional poststates (cf. activity a_1 in Figure 7.16(a)) *and* may consume additional prestates (cf. activity a_3 in Figure 7.16(a)). But a subtype may not extend an inherited life-cycle by "alternative branches", i.e., activities added at a subtype may not consume from or produce into inherited states (cf. activity a_5 in Figure 7.16(a)), since the behavior of instances of a subtype may not be observed at the supertype if an added "alternative branch" is executed. This approach is in line with the concepts prevailing in semantic data models where each instance of a subtype is also considered an instance of the supertype. The principles of observation consistency correspond to those of covariant inheritance of operations.

Weak invocation consistency requires that any operation invocable on instances of a supertype must under the same precondition also be invocable on instances of any subtype. Moreover, executing the operation on instances of a subtype must meet the postcondition for the operation specified at the supertype. Weak invocation consistency ensures that if an object is extended with new features the object is usable the same way as without the extensions. That is, a subtype may extend the inherited life-cycle only by "open parallel branches", i.e., inherited activities may produce into states added at a subtype (cf. activity a_1 in Figure 7.16(b)) but no inherited activity may consume from a state added at a subtype[16](cf. activity a_3 in Figure 7.16(b)). In addition, however, a subtype may extend the inherited life-cycle by new "alternative branches" (cf. activity a_5 in Figure 7.16(b)), since instances of the subtype could be used in the same way as instances of a supertype if alternative activities added at the subtype are neglected. The principles of weak invocation consistency correspond to those of contra-variant inheritance of operations.

Strong invocation consistency goes beyond weak invocation consistency in that it requires additionally that activities added at a subtype do not interfere with activities inherited from the supertype. Strong invocation consistency ensures that if an object is extended with new features, the object is usable the same way as without the extensions despite new features being used. That is, a subtype may extend the inherited life-cycle only by "open parallel branches", i.e., inherited activities may produce into states added at a subtype (cf. activity a_1 in Figure 7.16(c)) but no inherited activity may consume from a state added at a subtype[16] (cf. activity a_3 in Figure 7.16(c)). This means, a subtype may not extend the inherited life-cycle by new "alternative branches" (cf. activity a_5 in Figure 7.16(c)), since instances of the subtype could no longer be used in the same way as instances of a supertype if alternative activities added at the subtype are used. Note: If a life-cycle has been specialized according to strong invocation consistency, both weak invocation consistency as well as observation consistency are satisfied, too.

Behavior consistency for refinement has been defined in the sense of *observation consistency* (cf. [SS97]). This means, the behavior of an instance of a subtype must always be observable according to the life-cycle specified at a supertype (above). Since objects reside in basic states *and* in abstract states, observation consistency is implicitly guaranteed for refinements of activities and states by the underlying object model. However, since inherited activities that have been refined at a subtype cannot be invoked on instances of the subtype, invocation consistency cannot be guaranteed if a life-cycle is developed by stepwise refinements of activities and states.

---

[16]If a life-cycle defined at some object type O offers a destroy activity (cf. Figure 7.13), subtypes of O may not specialize the inherited life-cycle according to weak or strong invocation consistency as defined by Schrefl and Stumptner (cf. [SS95, SS97]), since final states of "open parallel branches" must not be added as prestates of any inherited activity. Note: A destroy activity must remove an object from *all* life-cycle states in which it resides.

Example: Figure 7.17 shows a life-cycle diagram of object type RESERVATION. Once a reservation has been issued, it may be billed and paid, and in parallel it may be used by the person requesting the reservation. Finally, a reservation is to be archived.

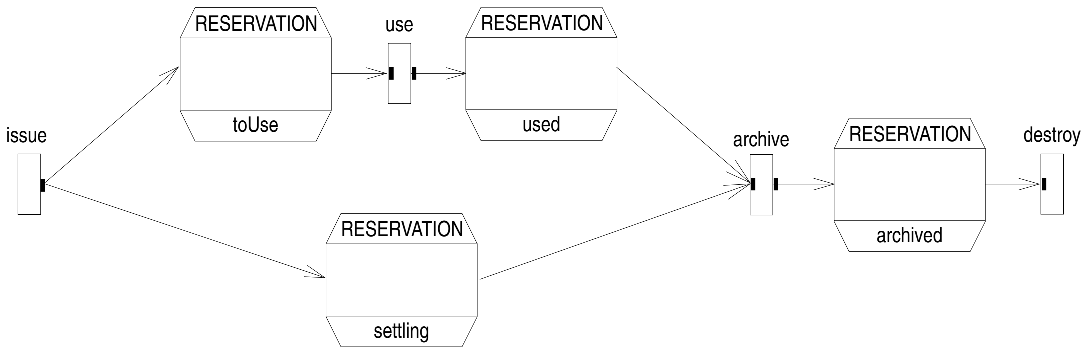


Figure 7.17: Life-cycle at RESERVATION

Example: Figure 7.18 shows a life-cycle diagram of object type ROOM-RESERVATION. Inherited sections of the life-cycle that are not redefined at object type ROOM-RESERVATION are shown in light-grey color. The inherited activity use is refined to a sequence of activities checkIn and checkOut. Inherited activity pay is refined to alternative activities payCash and payByCreditCard, respectively. Note: The life-cycle of object type ROOM-RESERVATION is an observation consistent refinement of the life-cycle of object type RESERVATION. That is, a manager may completely observe the behavior of room reservations via the life-cycle defined at object type RESERVATION.

7.2.3 Modeling Classes of Business Objects

Business-object classes (or *object classes* for short) have been introduced to increase the expressiveness of object-oriented database schemas. Object classes allow to model *class properties*, i.e., properties that describe the members of an object class considered as a unit, and allow to model referential integrity constraints in a natural way by specifying *class domains* for relationships defined for their members. Moreover, object classes can be organized into *class hierarchies*, which offer an additional means to structure real-world entities from a business area (cf. Subsection 3.2.3).

We propose to maintain component objects in classes that are *local to* composite objects on which the components depend. This means, object classes are no longer considered to be global schema objects, but are considered to be properties of composite objects. Consequently, there will no longer be a single object class that collects all components of a certain type. Instead, there will be a set of object classes each one collecting the components of a single composite object. Object classes that collect component objects are called *local object classes* to give emphasis to the fact that these classes are no longer global schema objects. Note: The *definitions* of local classes are still global.

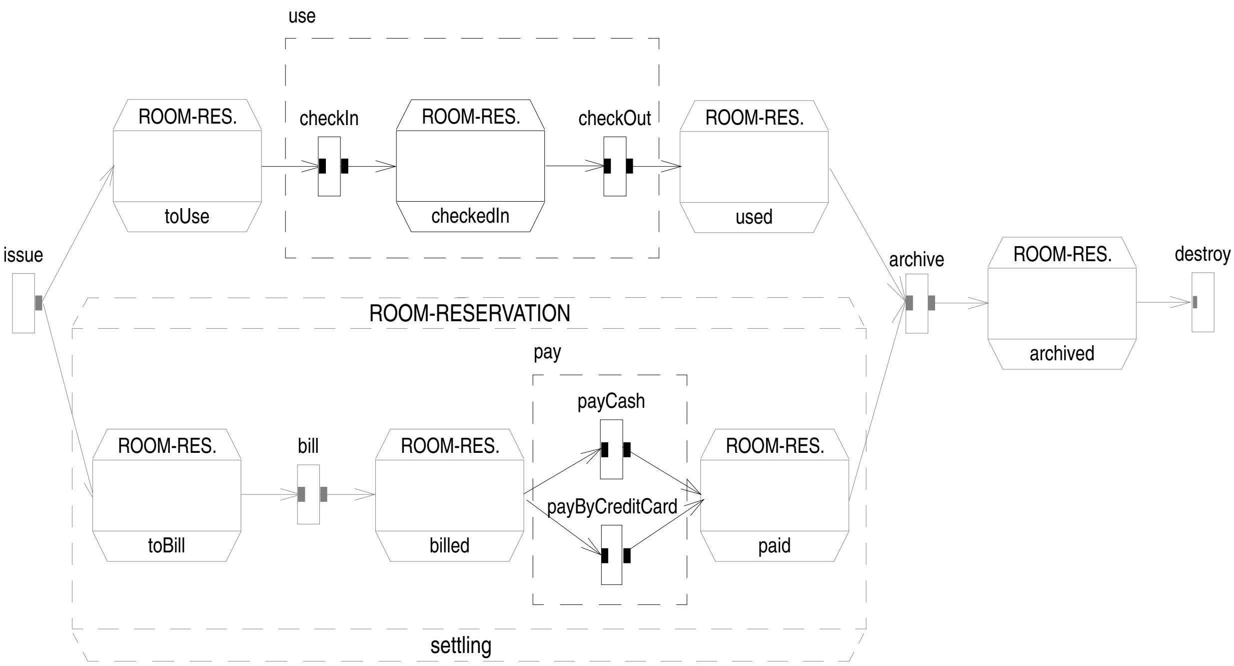


Figure 7.18: Life-cycle at ROOM-RESERVATION

In short, local object classes build on two fundamental concepts: *object classes* and *composite objects*. Both concepts are integrated conforming to object-oriented principles of encapsulation and information-hiding. The following list summarizes important prerequisites for the support of local object classes in an object-oriented data model:

1. **Object classes**: There is a clear distinction between *object types* and *object classes*.

2. **Composite objects**: There is a semantic relationship that allows to specify *composite-object hierarchies*.

3. **Classes are properties**: Object classes can be specified as properties of objects.

4. **Database object**: The database is considered an object itself, the *database object*. Its local object classes correspond to the schema of a "normal" object-oriented database, and members of these local object classes correspond to independent objects stored in this database. Consequently, *every object class is a local object class*.

The concept of *local object classes* is in line with object-oriented principles. Moreover, it further augments the expressiveness of object-oriented database schemas. For example, specifying local object classes as class domains for relationships allows to express *local referential integrity constraints* (cf. [KS92, KS96b]). In short, a local referential integrity constraint requires that two component objects taking part in a directed relationship must belong to the same composite object.

Class Types

A *class type* specifies a set of class properties, a set of class activities, and a specific object type, called member type, for its instances. In addition, individual properties and individual behavior of local-class objects can be specified at a class type (cf. Subsection 3.2.3). The existence of a local-class object depends solely on that composite object of which it is a property, i.e., local-class objects are created and destroyed with composite objects[17]. Consequently, local classes may have only general relationships.

Class types are organized in type hierarchies, which are specified by subtype-supertype relationships between class types. A subtype can specialize its supertype from which it inherits structure and behavior. Class structure inherited from a supertype is specialized in the same way as individual structure (cf. Subsection 7.2.1), class activities in the same way as individual activities (cf. Subsection 7.2.2). Each user-defined class type is a (direct or indirect) subtype of the predefined conceptual class-type BO_CLASS (Business-Object CLASS). This class type models basic collective structure and behavior of objects maintained as members of local classes in workflow systems. Note: Unlike the predefined conceptual object-types (cf. Subsection 7.2.1), the predefined conceptual class-type BO_CLASS is not an abstract type.

Example: Figure 7.19 shows the type diagrams of class types RESERVATIONS and ROOM-RESERVATIONS. Class type RESERVATIONS is a direct subtype of the conceptual class-type BO_CLASS. Local classes of type RESERVATIONS have members that are instances of object type RESERVATION (cf. Figure 7.1) and have a derived class attribute NoOfReservations. The derivation expression for this class attribute is COUNT(hasMembers), i.e., the value of class attribute NoOfReservations is determined by evaluating the function `COUNT` on the set of members of a local class. Class type ROOM-RESERVATIONS is a direct subtype of class type RESERVATIONS. It restricts the inherited member type to object type ROOM-RESERVATION (cf. Figure 7.7) and extends its supertype by introducing a new class attribute AvgLengthOfStay. This attribute is also derived. Its derivation expression is `AVG(to - from)`.

Specification of Local Object-Classes

A *local object-class* (or *local class* for short) is specified as a property of composite objects of a specific type. Consequently, the structure of (composite) business objects (cf. Subsection 3.2.1) is determined by a set of attributes, a set of relationships, *and* a set of local classes. Local classes are specified like global classes were specified (cf. Subsection 3.2.3). This means, each local class is specified by a *name*, a *domain*, a list of *superclasses* (possibly empty), a list of *subclasses* (possibly by empty), an indicator saying whether it is a *basic* or a *derived* class, a *derivation expression* if it is a derived class, an indicator saying

[17] Contrary to the object model of Object/Behavior Diagrams presented by Kappel and Schrefl [KS91, KS96b] and by Lang *et al.* [LOS96a, Lan97, Obe98], local classes are first-class citizens in the object model of Object/Behavior Diagrams presented in this thesis.

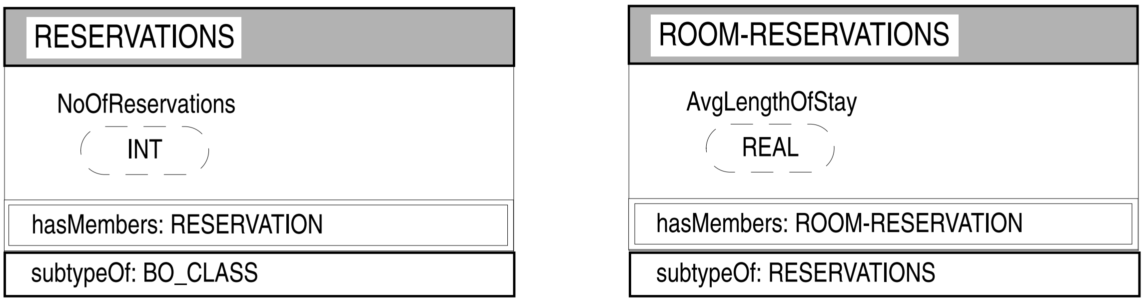


Figure 7.19: Class types RESERVATIONS and ROOM-RESERVATIONS

whether it is a *base class* or not, a multiplicity indicator saying whether it is *single-valued* or *multi-valued*, and a *cardinality*. The name of a local class must be unique within the set of properties defined for composite objects of a specific type. As a notational convention, we start the names of local classes with an uppercase letter. The domain of a local class is given by the predefined conceptual class-type BO_CLASS (cf. class types, above) or by any (direct or indirect) subtype thereof. That is, each local class is (direct or indirect) instance of the conceptual class-type BO_CLASS. Compared to global classes, the specification of local classes has been extended by an optional *member-type constraint*. A member-type constraint specifies an object type as member domain for a local class. This object type must be identical to the object type specified as member type at the class type that has been given as domain for the local class, or a direct or indirect subtype thereof. This means, a member-type constraint may be specified with a local class to restrict the member type specified at its class type.

Graphically, a local class is specified in the type diagram of the composite object of which the local class is a property. A special section of a type diagram is devoted to the specification of local classes. The local-class section of a type diagram is identified by the annotation local classes and is separated from the "general" section by a solid-line border. A member-type constraint on a local class c can be depicted in a type diagram (optionally) by giving the keyword memberType followed by the name of the member type within the graphical symbol representing c.

Example: Figure 7.20(a) shows partly the specification of object type HOTEL. Each hotel has three local classes named Reservations, Rooms, and Guests, from domains ROOM-RESERVATIONS, ROOMS, and GUESTS, respectively. Each of the local classes is specified to be a multi-valued base class. Since no cardinality is given explicitly, the cardinality of each of them is [0:$*$]. Ignore the remaining properties of hotels for the moment.

If class type ROOM-RESERVATIONS had not been defined, its supertype, i.e., class type RESERVATIONS (cf. Figure 7.19), would have to be used as domain of local class Reservations. As a consequence, a member-type constraint would have to be specified with local class Reservations, since the member domain of local class Reservations would have to be restricted to the object type given as domain of the corresponding has-component relationship reservations. Figure 7.20(b) shows the alternative specification of local class Reservations.

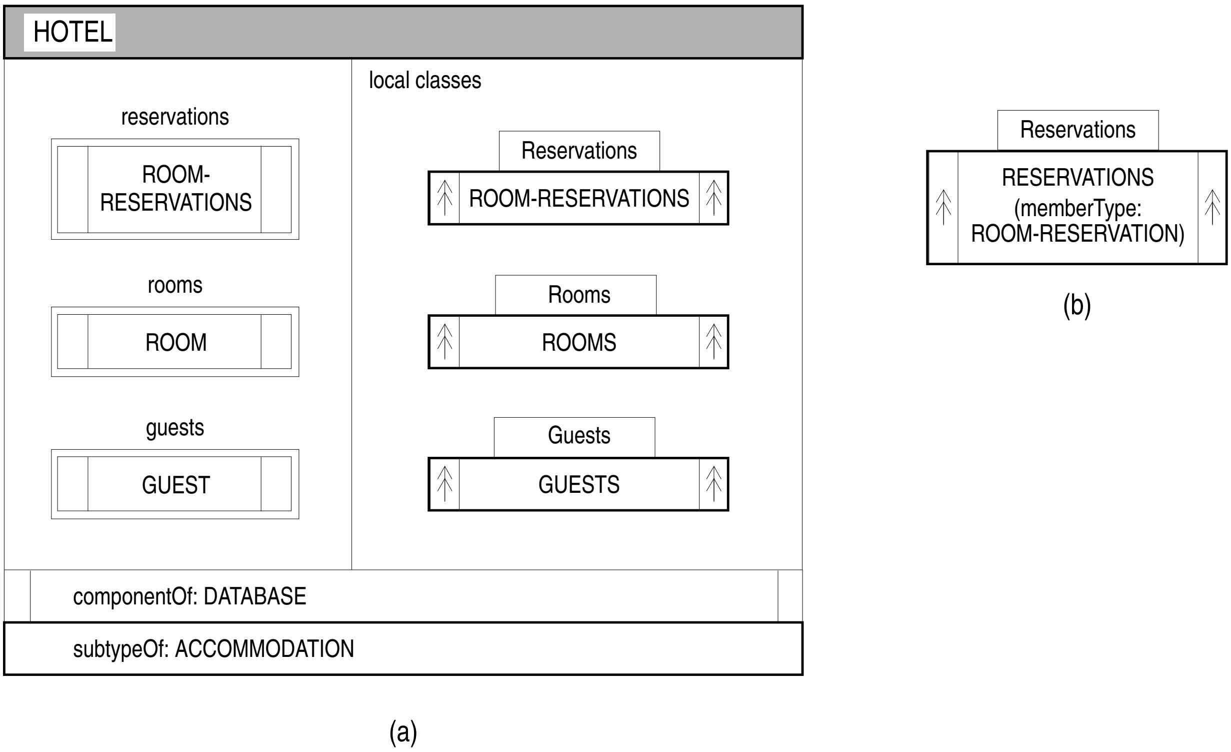


Figure 7.20: Object type HOTEL

Properties are abstract concepts, which are to be interpreted as public retrieval operations that return either one instance of the property's domain (single-valued property) or a set of instances of its domain (multi-valued property) (cf. Subsection 3.2.1). Since local classes are properties of composite objects, *local classes are to be interpreted as public retrieval operations*, too. Furthermore, since all classes are considered to be local classes, a class and its members can only be accessed by sending a message to the composite object of which it is a property. Maintaining objects as members of classes that are properties of those composite objects of which they are components is in-line with the fundamental object-oriented principles of encapsulation and information hiding. Since local-class properties play a dual role, there are two public retrieval operations associated with every local class property. The first is named after the local-class property. It is always single-valued and returns a local-class object. The second is also named after the local-class property, but its name starts with a lowercase letter. It is either single-valued or multi-valued and returns the current member(s) of a local class. In addition, hidden update operations are associated with local classes. Like hidden update operations that are associated with attribute properties and relationship properties, hidden update operations that are associated with local classes may only be invoked by public update operations specified in the life-cycle of an object type (cf. Subsection 3.2.1). In the case of a single-valued class, the hidden update operation takes one instance of the member domain of the local class. In the case of a multi-valued class, two hidden update operations are associated with the local-class property. One allows to add a set of instances of the member domain of the local class to its current set of members, the second allows to remove a set of members from the current set of members of a local class. The names of these hidden update operations are formed in the same way as the names of hidden update operations associated with multi-valued relationships (cf. Subsection 3.2.1). Note: Since local-class objects are automatically created and destroyed with composite objects, no hidden update operation is associated with a local-class property to assign a new local-class object as value of a local-class property of a composite object.

Example: The set of members of local class Reservations of a hotel h can be retrieved by sending message `reservations` to h. The local-class object itself is retrieved by sending message `Reservations` to h (cf. Figure 7.20).

Local classes can be organized into *class hierarchies*. Each local class may have several superclasses and may have several subclasses (cf. Subsection 3.2.3). Specializations of classes may be partial and overlapping (cf. Section 7.1). The set of members of a subclass must always be a subset of the set of members of its superclass(es). Usually, a subclass has exactly one superclass, which it specializes[18]. Specializing its superclass either means that (1) the domain of a subclass has been redefined to a direct or indirect subtype of the class type specified as domain of its superclass, (2) the member type of a subclass has been redefined to a direct or indirect subtype of the member type inherited from its superclass[19], (3) the cardinality of a subclass restricts the cardinality of its superclass, or (4) a subclass specializes the set of class-domain constraints it inherits from its superclass.

The semantic difference between class hierarchies defined between local classes and those defined between global object classes (cf. Subsection 3.2.3) is found in the locality of object classes, i.e., there is one hierarchy of local classes for every composite object for which the interrelated classes have been specified. Usually, all classes within a class hierarchy belong to the same composite object o and collect direct components of o. If a local classes collects indirect components of a composite object o of which it is a property, at least one of its subclasses does belong to a direct or indirect component of o.

Example: Figure 7.21 shows the specification of class hierarchies at object type HOTEL. Each hotel has three base classes Reservations, Rooms, and Guests. Base class Reservations has two subclasses Suite-Reservations and King's-Suite-Reservation. The class type of all three local classes is ROOM-RESERVATIONS. Base class Rooms has a subclass Suites. The class type of both local classes is ROOMS. Base class Guests has two subclasses VIP-Guests and King. The class type of all three local classes is GUESTS.

---

[18] If a class has more than one superclass, it must specialize all its superclasses.

[19] The *member type* of a class is specified at the class' type by the object type given as domain for the obligatory hasMembers relationship. Specifying a member-type constraint for a class can further restrict that member type of a class to a direct or indirect subtype thereof.

Local classes King's-Suite-Reservation and King are singleton classes. Local classes Suites, Suite-Reservations, and King's-Suite-Reservation are derived classes. Local class Suites collects all rooms of a hotel that belong to category `'Suite'`. Figure 7.22 shows a *query diagram* (cf. [LOS96b]) that determines the set of members of local class Suites. Local class Suite-Reservations collects all reservations of a hotel that have been issued for rooms that are members of local class Suites. Figure 7.23 shows a query diagram that determines the set of members of local class Suite-Reservations. Local class King's-Suite-Reservation has a single member that is the next reservation for the King's Suite or has no member if there is no such reservation. (Note: The query diagram that determines the extension of object class King's-Suite-Reservation is not shown!)

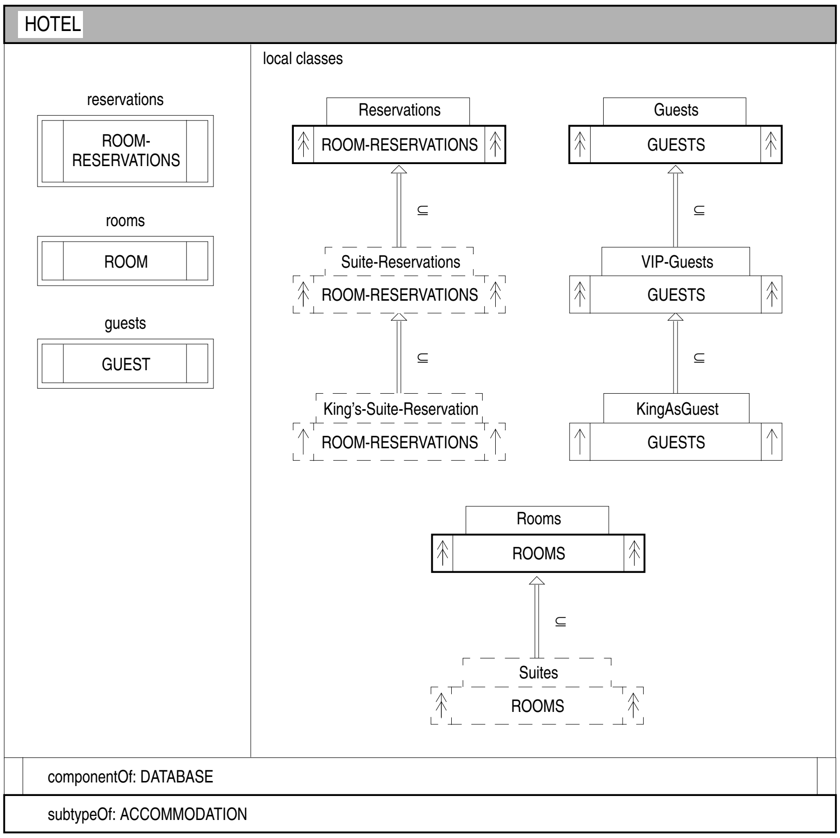


Figure 7.21: Object type HOTEL with local classes

Each local class collects a set of direct or indirect components of the composite object of which it is a property. This means, if a local class c is defined as a property of composite objects of type O_1, there will not be a single class c that collects all instances of the member domain O_2 of c, but there will be one local class c for every instance of O_1, each one collecting only that instances of O_2 that are (direct or indirect) components of a particular instance of O_1. Furthermore, there is a close correspondence between local

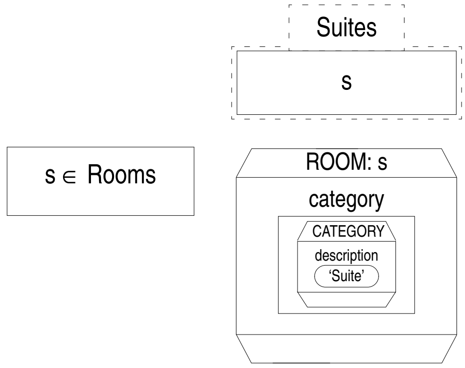


Figure 7.22: Query diagram for derived class Suites

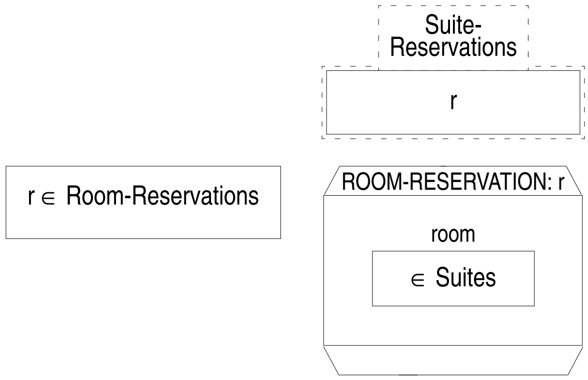


Figure 7.23: Query diagram for derived class Suite-Reservations

classes and has-component relationships and has-indirect-component relationships. The member domain of a local class[20] defined at an object type O_1, must either be identical to an object type O_2 specified as domain for a has-component relationship or a has-indirect-component relationship defined at O_1, or must be a direct or indirect subtype of O_2.

Example: The member domain of local class Reservations of a hotel is object type ROOM-RESERVATION. It is defined at class type ROOM-RESERVATIONS (cf. Figure 7.19) that has been specified as domain of local class Reservations. Thus, the member domain defined for local class Reservations is identical to the object type specified as domain for the has-component relationships reservations of hotels (cf. Figure 7.20). The member domain of local classes Rooms and Guests defined for objects of type HOTEL are object types ROOM and GUEST, respectively. These types are identical to the types specified as domains for has-component relationships rooms and guests, respectively, defined at object type HOTEL (cf. Figure 7.20). (Note: The type diagrams of class types ROOMS and GUESTS are not shown.)

[20]Note: The member domain of a local class is specified at its class type and may be restricted by a member domain constraint.

Local classes function as *object warehouses*. Like global classes could have been used to maintain the extensions of object types (cf. Subsection 3.2.3), local classes can be used to maintain the extensions of object types, too. To guarantee that each object is maintained as member of some local class, a special kind of local classes, called local base-classes, are associated with object types. Each instance of a component-object type O_2 is a member of some local base-class c associated with O_2 at the composite object of which it is a direct component.

A *local base-class* (or shortly *base class*) is a local class that collects all instances of the object type specified as its member domain that are *direct components* of the composite object of which the base class is a property. This means, there is a direct correspondence between has-component relationships and base classes. For every has-component relationship r with domain O_2 defined at some composite-object type O_1, there must also be defined a base class c with object type O_2 as member domain at O_1, and vice versa[21]. Every instance of component-object type O_2 will be automatically maintained as member of base class c of that composite object of type O_1 of which it is a component[22]. Note: To ensure that pairs of has-component relationships and base classes are defined (redefined) consistently, a tool for the design of conceptual schemas could automatically define a corresponding has-component relationship if a base class is defined and vice versa. Although the concepts of has-component relationships and base classes correspond to some degree, their differences justify that both has-component relationships and base classes are shown in type diagrams. To keep the presentation of type diagrams clear, a tool for the design and the browsing of conceptual schemas could offer different views. For example, one view might focus on local classes, the other on attributes and relationships defined for objects.

The correspondence between has-component relationships and base classes is limited to the *extensional point of view*. This means, the set of members of each base class c defined for an instance of some composite-object type will always be equivalent to the set of values of the corresponding has-component relationship. Actually, there will be only a single public retrieval operation to retrieve both sets of objects. As a notational convention, we require that each base class is named after the corresponding has-component relationship. Since we start local-class names with uppercase letters (above), has-component relationships and base classes can easily be distinguished. Note: The name of the public retrieval operation answering the set of members of a base class will then be identical to that of the public retrieval operation answering the current set of values of the corresponding has-component relationship.

From the *intensional point of view*, however, has-component relationships and local classes are different concepts. A has-component relationship models a directed relationship from a composite object to several dependent component objects. This means, the inverse of a has-component relationship, i.e., a component-of relationship is object-defining.

[21]Since, all component objects must fulfill the class constraints, too, restricting the member domain of a base class to a subtype of the object type specified as domain for the corresponding has-component relationship also restricts the domain of the has-component relationship.

[22]Note: The responsibility of maintaining objects as members of a base class is delegated from the component-object type (cf. Subsection 3.2.3) to the composite objects.

Consequently, *a component object must be component of exactly one composite object.* On the contrary, *a component object may be member of several local classes.* That is, member-of relationships, which are the inverses of has-member relationships defined for local classes, are object-describing[23]. Moreover, a local class holds values of class properties, may be related to other classes via subclass-superclass relationships, and can specify other local classes as domains for relationships defined for its members[24].

Due to their correspondence with has-component relationships, base classes usually are the roots of class hierarchies. However, a base class c_2 may also be defined as a direct subclass of another base class c_1, if the member domain of c_1 is a supertype O_1. The member domain of c_2 must then be a direct or indirect subtype of O_1. However, there need not be such a subclass c_2 of c_1, although the member domain of c_1 is a supertype. Note: There will be no corresponding has-component relationship for base classes that are subclasses of other base classes.

Each object will be created as component of some composite object (cf. composite-object hierarchies, Subsection 7.2.1). Upon creation, an object o' automatically becomes member of every base class collecting instances of the type of o' as its members at that composite object o of which o' is a component.

Example: Objects of type HOTEL (cf. Figure 7.20) have three base classes named Reservations, Rooms, and Guests. The corresponding has-component relationships are named reservations, rooms, and guests, respectively.

A local class that collects indirect components of a composite object must be a *union class*. In general, a union class is a derived class that is introduced as a direct superclass of several other classes. The derivation expression of a union class is a set expression determining the union of the sets of members of the subclasses of the union class. In particular, the definition of a union class that corresponds to a has-indirect-component relationship lists only a single local-class definition as subclass. However, since the local classes defined as subclasses of such union classes are not defined for the same objects for which the union classes are defined, but for direct or indirect components thereof, each of these union classes may have several subclasses. Each of these subclasses has the same name and belongs to a different object of the same type. Each subclass of a union class that corresponds to a has-indirect-component relationship must be a property of a direct or indirect component of the object of which the union class is a property. As a notational convention, union classes that collect indirect components of a composite object and their subclasses share the *same* name[25]. Note: For every has-indirect-component relationship r with domain O_2 defined at some composite-object type O_1, there must also be defined a union class c with member domain O_2.

[23]Note: Member-of relationships are not modeled explicitly.

[24]Specifying local classes as domain for relationships expresses a special kind of referential integrity constraints, so-called local referential integrity constraints (discussed below).

[25]Note: No name clash might occur in this situation since no composite-object type will be a direct or indirect subtype of the composite-object type in question and vice versa. This requirement is expressed by the *Coherent Usage Rule of Subtypes* in [KS92].

In a type diagram, union superclasses are emphasized by marking the subclass-superclass relationship between a union superclass and its subclass with the mathematical symbol for set union ($\cup$). For readability reasons, the specifications of the subclasses of a union superclass collecting indirect components of a composite object are shown, too, although they are not properties of the composite objects. Instead of showing the name of the subclasses, the navigation expression that yields the subclasses is shown at the top of the graphical symbol representing the subclasses.

Example: Object type DATABASE (cf. Figure 7.24) specifies two local classes that collect indirect components of the database object[26]. Local class Suites collects all suites of all hotels that are members of local class Hotels. That means, local class Suites is specified as a union class on local classes Suites defined for members of local class Hotels (cf. Figure 7.20). The corresponding navigation expression, i.e., Hotels.Suites, is shown in place of the name of the subclass. The derivation expression is $\bigcup \mathit{Hotels.Suites}$.

Local class Guests collects all guests registered at any hotel that are members of local class Hotels. It is specified as a union class on local class Guests, a base class, defined form members of local class Hotels (cf. Figure 7.20). The corresponding navigation expression, i.e., Hotels.Guests, is shown in place of the name of the subclass. The derivation expression is $\bigcup \mathit{Hotels.Guests}$. Ignore the subclass-superclass relationship between local class Guests and local class Persons for the moment.

Allowing to organize classes into class hierarchies, offers an additional means to structure real-world entities from a business area, beside type hierarchies. In general, each entity that is represented as member of some subclass c must also be represented as member of all direct and indirect superclasses of c. That is, *the members of a subclass represent a subset of the real-world entities represented by the members of its superclass* (extensional view). In traditional object-oriented systems, the subset-superset relationship between the set of entities represented by the members of the subclass and its superclass is based on the object identity of the members of both classes. This means, each subclass contains a subset of the members of its direct superclass. For traditional object-oriented systems, which model each real-world entity by a single object (cf. [BM91]), this semantics of subclass-superclass relationships is sufficient. For object-oriented systems, however, which allow to represent each real-world entity by several objects at a time, i.e., by a player object and several role objects (cf. role hierarchies, Subsection 7.2.1), the semantics of subclass-superclass relationships must be extended to cope with situations in which a class c_2 collects a set of members that represent a subset of the real-world entities represented by the set of members collected by another class c_1, but there is no subset relationship between the sets of members of c_2 and c_1, since c_2 and c_1 collect objects that represent the same real-world entities in different roles. Consequently, the subset-superset relationship between a subclass and its superclass must, in general, be based on the entity identity of the members. To emphasize

[26] Object type DATABASE has exactly one instance, the database object. The database object is the root of the composite-object hierarchy into which all objects stored in the database are organized.

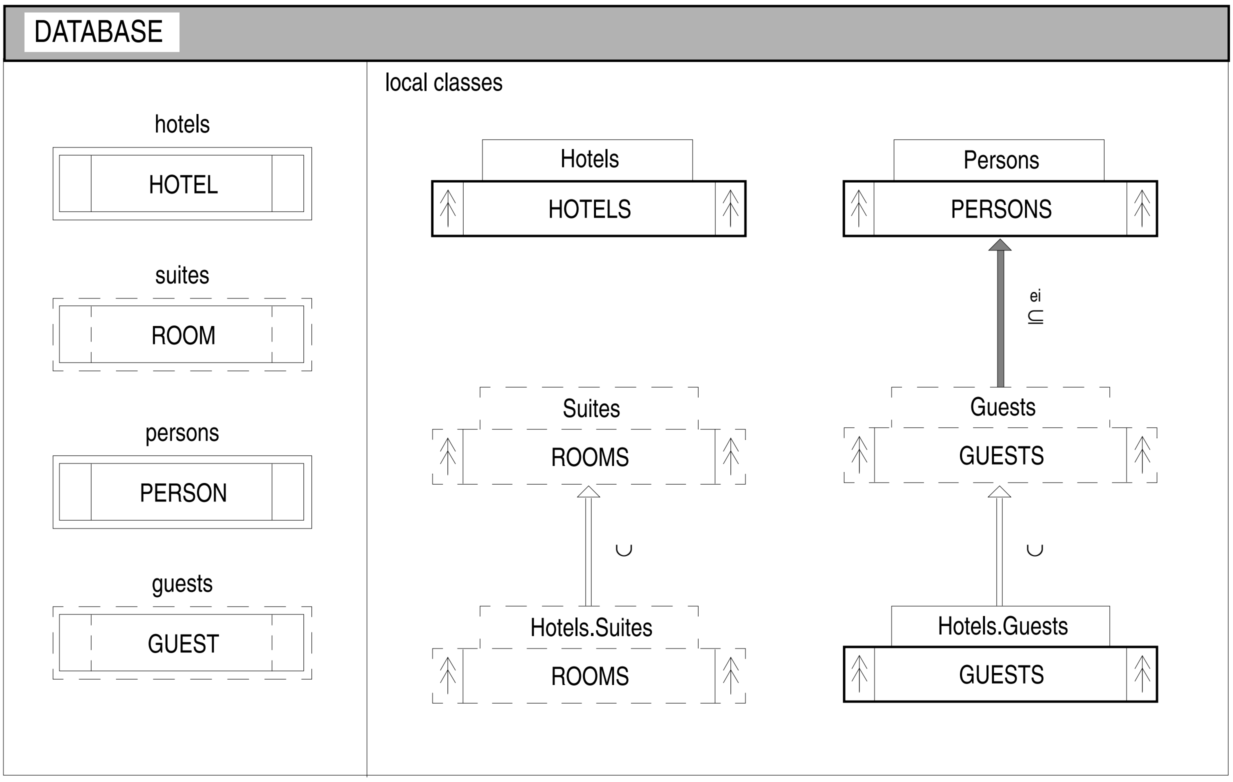


Figure 7.24: Object type DATABASE with local classes

the special semantics of the relationships between classes collecting objects that represent the same real-world entities in different roles, we introduce a specific subclass-superclass relationship, called role subclass-superclass relationship.

A *role subclass-superclass relationship* models a specialization relationship between two classes that collect objects that represent the same real-world entities in different roles. A role subclass specializes its role superclass in that it collects objects that represent a subset of the real-world entities represented by the members of its superclass (extensional view), and in that an object type O_1 specified as domain of the role-of relationship defined for members of a role subclass must either be identical to an object type O_2 given as member domain of the corresponding role superclass, or a direct or indirect subtype of O_2, and the cardinality of a role subclass restricts the cardinality of its role superclass (intensional view). This means, each member of a role subclass must be related to a member of the role subclass via a role-of relationship. Two classes related via a role subclass-superclass relationship are independent otherwise. This means, a role subclass does neither inherit class-domain constraints from its role superclass nor may it be derived from its role superclass. Moreover, a role subclass and the corresponding role superclass will usually be properties of different composite objects.

Actually, a role subclass-superclass relationship is the specification of a class domain for the role-of relationship defined for members of the role subclass and of a class domain for the has-role relationship defined for members of the role superclass. However, the role subclass-superclass relationship emphasizes the special semantics of the role-of relationship (has-role relationship) between the interrelated classes.

Graphically, role subclass-superclass relationships are distinguished from traditional subclass-superclass relationships by showing the arc representing a role subclass-superclass relationship in light-grey color. Moreover, the keyword ei is placed at the top of the mathematical symbol representing the subset-superset relationship ($\overset{\text{ei}}{\subseteq}$), to indicate that the subset-superset relationship between the sets of members of a role subclass and a role superclass is based on entity identity, not on object identity. If a role subclass and its role superclass are specified as properties at different object types, the corresponding role subclass-superclass relationship may be depicted twice. First, it may be depicted at the object type of which the role superclass is a property. In this case, the navigation expression that yields the subclasses is shown

Example: Object type DATABASE (cf. Figure 7.24) specifies a role subclass-superclass relationship between role subclass Guests (a union class that collects indirect components of the database object, above) and role superclass Persons. That means, all persons that are members of local class Guests of any hotel in local class Hotels, must (as persons) be members of local class Persons. Note: Both classes, the role subclass as well as the role superclass, are specified as properties at the same object type in this case.

Figure 7.25 presents both views of a role subclass-superclass relationship specified between role superclass Persons and role subclass Guests. First, it is depicted at object type DATABASE at which the role superclass is specified as a property. The navigation expression Hotels.Guests is shown instead of the name of the role subclass in this case. Second, it is depicted at object type HOTEL at which the role subclass is specified as a property. The navigation expression componentOf.Persons is shown instead of the name of the role superclass in this case.

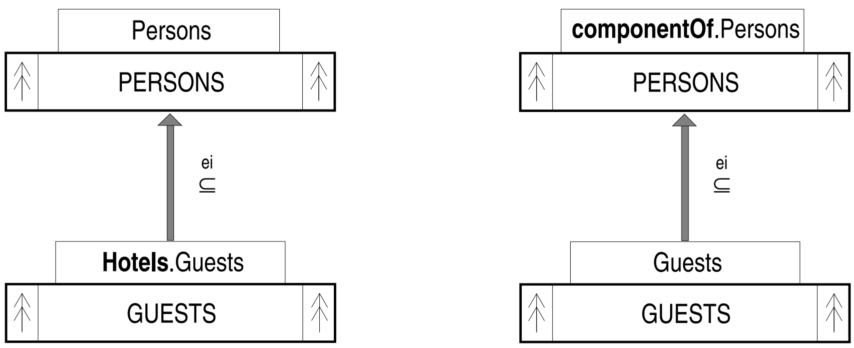


Figure 7.25: Role subclass-superclass relationships

Class-Domain Constraints

A local class may specify other local classes as domains for relationships defined for its members. Specifying local classes as domains for relationships is a mechanism to represent the concept of *local referential integrity* (cf. [KS92, KS96b]) in an object-oriented database schema. Local referential integrity extends the concept of referential integrity (cf. Subsection 3.2.3) for composite objects with dependent and exclusive components. A local referential integrity constraint requires that two objects taking part in a directed relationship must be direct or indirect components of the same composite object. Local referential integrity constraints are modeled by class-domain constraints. Since local classes do not exist at schema-definition time, however, class-domain constraints cannot be specified at local classes. Instead, each class-domain constraint C is specified at a type of composite objects of which all objects taking part in the relationship constrained by C are direct or indirect components.

A class-domain constraint consists of a source class c_{source}, a relationship r, a target class c_{target}, and a cardinality constraint v (cf. Subsection 3.2.3). If a class-domain constraint is specified at some composite-object type O, the source class c_{source} identifies a set of direct or indirect components of instances of O, for which the domain of relationship r is constrained. Relationship r may be a general relationship, a has-role relationship, a role-of relationship, a has-constituent relationship, or a constituent-of relationship (cf. Subsection 7.2.1). The target class c_{target} identifies a set of direct or indirect components of instances of O that serve as domain for relationship r of objects identified by the source class c_{source}. The source class c_{source} and the target class c_{target} of a class-domain constraint are identified by locality paths. Each locality path is a list of local classes starting at the object type O at which a class-domain constraint is specified. The first element of a locality path must be the name of a local class defined at O. In general, the (i+1)-th element, $i \geq 1$, of a locality path must be the name of a local class defined at an object type O_2 that is identical to the member type O_1 of the local class identified by the i-th element of that locality path or must be a direct or indirect subtype of O_1. For each element e of a locality path, an object type O' may be specified as a member-type constraint. O' must be identical to the object type O specified as member type for e or must be a direct or indirect subtype of O. Remember that cardinality constraint v must be identical to or restrict the cardinality constraint v_M specified for relationship r at the member type M of the type of the source class c_{source} and that cardinality constraint v is implicitly defined by v_M if v is not defined explicitly for a cardinality constraint (cf. Subsection 3.2.3). A class-domain constraint defined at an object type O is applied to every instance o_i of O. $c_{source}(o_i)$ identifies the direct or indirect components of o_i that are contained in c_{source}, $c_{target}(o_i)$ identifies the direct or indirect components of o_i that are contained in c_{target}. For every instance o_i of O, the domain of relationship r of objects in $c_{source}(o_i)$ is restricted to objects in $c_{target}(o_i)$.

Example: Class-domain constraint C_1 (cf. running example, Figures 7.21 and 7.24)

```
DATABASE:  Hotels().Suite-Reservations():room:Hotels().Suites(): [0:1]
```

says that each suite reservation may reference any suite that exists in the database. Note: This class-domain constraint has the same semantics as that specified on relationship room for members of global class Suite-Reservations (cf. Subsection 3.2.3, Figure 3.8). For example, it does allow a hotel to issue reservations for suites of other hotels.

The restrictiveness of a class-domain constraint C is determined by the object type at which C has been defined. In general, a relationship r between objects of types O_s and O_t can be constrained by a class-domain constraint C defined at a composite-object type O if instances of O_s and instances of O_t are *direct or indirect* components of instances of O. That is, there may be different types of composite objects at which class-domain constraints on certain relationships can be defined, where two class-domain constraints specified on the same relationship r at different types of composite objects have different restrictiveness. The *least specific type* at which a class-domain constraint on some relationship r can be specified is the predefined object type DATABASE, since all objects are (direct or indirect) components of the database object[27]. The *most specific type* at which a class-domain constraint on some relationship r may be specified is that composite-object type O at which a class-domain constraint on r can be specified (i.e., instances of O_s and instances of O_t are (direct or indirect) components of instances of O), such that there is no other composite-object type O' at which a class-domain constraint on r can be specified, and instances of O' are direct or indirect components of instances of O.

The *local restrictiveness* of a class-domain constraint C on some relationship r is a numeric measure that determines whether C has been specified at the most specific type at which a class-domain constraint on r could have been specified. Based on the observation that the locality paths that identify the source class and the target class, respectively, of a class-domain constraint C on some relationship r share a common prefix if C is not specified at the most specific type at which a class-domain constraint on r could have been specified, we compute the local restrictiveness of a class-domain constraint C from the length of the common prefix of the locality paths that identify the source class and the target class of C. A local restrictiveness of zero of a class-domain constraint C on some relationship r, i.e., $\triangle(C) = 0$, says that C has been specified at the most specific type at which a class-domain constraint on r could have been specified.

Example: Class-domain constraint C_2 (cf. running example, Figures 7.21)

```
HOTEL:  Suite-Reservations():room:Suites(): [0:1]
```

says that each suite reservation that belongs to a particular hotel may reference via relationship room only a suite that belongs to the same hotel. Unlike that given in

[27]Note: The database object is the only instance of the predefined object type DATABASE and is the root of the composite-object hierarchy into which all objects stored in a database are organized according to their component-of relationships (cf. Subsection 7.2.1).

the previous example, the class-domain constraint given in this example does not allow a hotel to issue reservations for suites of other hotels.

Obviously, the class-domain constraint given in this example is more restrictive than that given in the previous example, since the locality paths that identify its source class and its target class, respectively, do not share a common prefix. More formally, its local restrictiveness is zero, i.e., $\triangle(C_2) = 0$. This means, there is no object type O such that suite reservations and suites are direct or indirect components of instances of O that are direct or indirect components of instances of HOTEL. The local restrictiveness of class-domain constraint C_1 given in the previous example is one, i.e., $\triangle(C_1) = 1$. This means, suite reservations as well as suites are components of the same type of objects, i.e., hotels, which are themselves direct components of instances of that type at which the class-domain constraint is specified. As a consequence, the locality paths that identify the source class and the target class, respectively, of C_1 share a common prefix of length one, i.e., local class Hotels defined at object type DATABASE.

Graphically, class-domain constraints are specified for local classes in the same way as they are specified for global classes (cf. Chapter 3.2.3). If a locality path consists of a list of more than one local class, the locality path is depicted by nesting local classes. If a local class c' is depicted within a local class c, then the type diagram of object type O that has been specified as member type of c is partially shown within the graphical symbol representing c. Actually, only local classes specified at O may be shown. If several locality paths navigate along partially the same local classes, the common parts are depicted only once.

Example: Figure 7.26 shows part of the class-domain diagram of object type DATABASE (cf. Figure 7.24), representing the graphical specification of the first class-domain constraint given above. The domain of relationship room of suite reservations that are members of local classes Suite-Reservations of members of a local class Hotels of a database object is restricted to members of local classes Suites of members of local class Hotels of the database object.

Figure 7.27 shows the class-domain diagram of object type HOTEL (cf. Figure 7.20). For every hotel the following class-domain constraints are defined: For members of local class Reservations the domain of relationship room is restricted to members of local class Rooms of the same hotel, for relationship guest to members of local class Guests. For members of local class Suite-Reservations the domain of relationship room is restricted to members of local class Suites of the same hotel, for relationship guest to members of local class VIP-Guests. For members of local class King's-Suite-Reservation the domain of relationship guest is restricted to the member of local class King of the same hotel, i.e., the hotel may not let the King's Suite to anyone else than the King. Moreover, the cardinality of relationship room is restricted to [1:1].

For members of local class Guests the domain of relationship reservations is restricted to members of local class Reservations of the same hotel. No subclass of local class Guests specializes the class-domain constraint on relationship reservations.

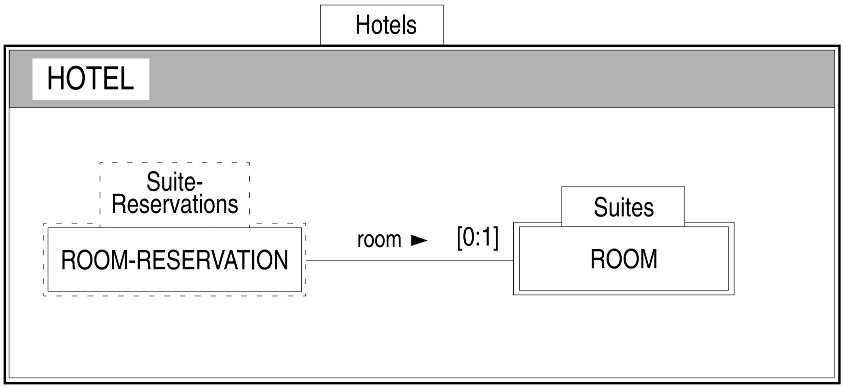


Figure 7.26: Class-domain diagram of object type DATABASE

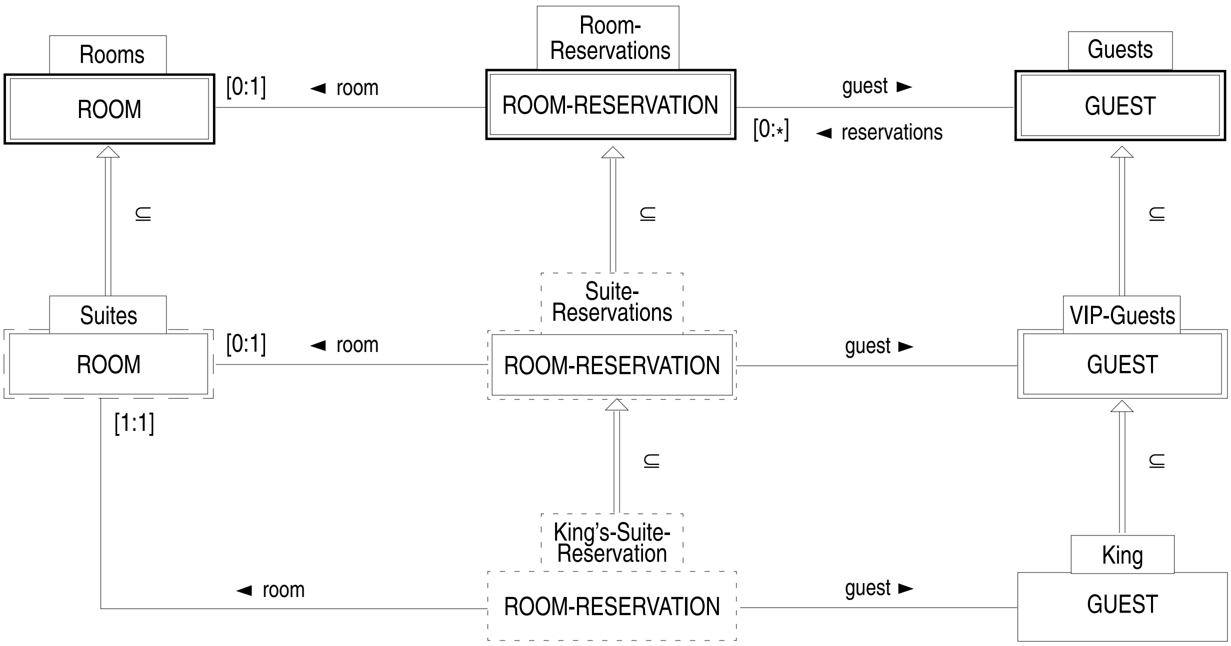


Figure 7.27: Class-domain diagram of object type HOTEL

Example: In Figure 7.27, a set of class-domain constraints have been specified graphically. Below, a textual specification of these class domain constraints is given. (Note: The textual specification of the class-domain constraint depicted in Figure 7.26 is not repeated below. It has already been presented above.)

```
HOTEL:  Reservations():guest:Guests()
HOTEL:  Reservations():room:Rooms(): [0:1]
```

```
HOTEL:  Guests():reservations:Reservations(): [0:*]
HOTEL:  Suite-Reservations():guest:VIP-Guests()
HOTEL:  Suite-Reservations():room:Suites(): [0:1]
HOTEL:  King's-Suite-Reservation():guest:King()
HOTEL:  King's-Suite-Reservation():room:Suites(): [1:1]
```

All class-domain constraints are specified at object type HOTEL.

Specialization of Local-Class Properties and Class-Domain Constraints

Each subtype may specialize the structure it inherits from its supertype. To specialize inherited structure means either to extend it by specifying new properties or to refine it by modifying the specifications of inherited ones (cf. Subsection 7.2.1). Since in the advanced conceptual model the structure of business objects is defined by a set of attributes, a set of relationships, *and* a set of local classes, a subtype may specialize its supertype also in that it specifies new local classes or redefines inherited ones. In addition, a subtype inherits class-domain constraints defined at its supertype. The set of consistency rules to ensure that subtypes do not arbitrarily modify inherited structure (cf. Subsection 7.2.1) apply to inherited local-class properties in a similar way. Below, we discuss additional consistency rules that focus on the specialization of inherited local-class properties and on the specialization of inherited class-domain constraints. Note: The specialization of local classes and class-domain constraints in class hierarchies has already been discussed in Subsection 3.2.3.

1. A subtype may redefine the domain of an inherited local class to a direct or indirect subtype of the inherited domain (cf. *Rule of Specialization of Domains of Properties*, Subsection 7.2.1).

2. A subtype may restrict the cardinality of an inherited local class (cf. *Rule of Specialization of Cardinalities of Properties*, Subsection 7.2.1).

3. A subtype may redefine the member type of an inherited local class to a direct or indirect subtype of the inherited domain (*Rule of Specialization of Member Types of Local Classes*). Note: The member type of an inherited local class can be redefined either by redefining the domain of the local class or by specifying a member-type constraint.

   Example: The member type of local class Reservations defined for objects of type ACCOMMODATION (cf. Figure 7.2) is object type RESERVATION (cf. Figure 7.1). It is specified at class type RESERVATIONS (cf. Figure 7.19), which has been specified as domain of local class Reservations at object type ACCOMMODATION. (Note: The specification of local class Reservations is not shown in Figure 7.2.).

   Object type HOTEL, a direct subtype of object type ACCOMMODATION, redefines the member type of local class Reservations to object type ROOM-

RESERVATION (cf. Figure 7.7). In Figure 7.20(a) the domain of local class Reservations has been redefined to class type ROOM-RESERVATIONS (cf. Figure 7.19) that redefines the member type of its instances to object type ROOM-RESERVATION. In Figure 7.20(b) the member type of local class Reservations has been redefined by specifying an appropriate member-type constraint.

4. A subtype that redefines the domain or the cardinality of an inherited has-component relationship or a has-indirect-component relationship, must also redefine the member domain or the cardinality, respectively, of the corresponding local class in the same way (*Rule of Symmetrical Specialization of Has-Component Relationships and Local Classes*).

5. A subtype may restrict the target class of an inherited class-domain constraint $c_{source} : r : c_{target} : v$ to a target class c'_{target}, where the set of members of c'_{target} is a subset of the set of members of c_{target}. That is, either c'_{target} is a subclass of c_{target} or the member-type constraints in c'_{target} are more restrictive than those in c_{target} (*Rule of Specialization of Target Classes in Class-Domain Constraints for Object Types*).

6. A subtype may restrict the cardinality v of an inherited class-domain constraint (*Rule of Specialization of Cardinalities of Class-Domain Constraints for Object Types*) (cf. definition of relation *restricts* between cardinalities, Subsection 3.2.3).

Redefining the member type or the cardinality of an inherited local class c, might cause redefinitions of member types or cardinalities, respectively, of subclasses of c. Below, we outline a situation in which the redefinition of the inherited domain of a has-component relationship, and in consequence the redefinition of the inherited member type of the corresponding base class, results in a conflicting situation. Semantically, the main problem within that example is that two local classes have been introduced at a supertype O although not all (indirect) instances of O offer these local classes.

Assume that at object type O_1 a has-component relationship r with domain O_2 has been specified. Object type O_1 specifies a base class c collecting components related to objects of type O_1 via relationship r and specifies two subclasses c_1 and c_2 of c that collect components of type O_{21} and O_{22}, respectively, which are direct subtypes of O_2. Restricting the domain of has-component relationship r at some subtype O_{11} of O_1 to a subtype of O_2, e.g., O_{21}, leads to a conflicting situation. First, restricting the member type of base class c to object type O_{21} is not allowed, since there is no common subtype of O_{21} and O_{22} to which the member type of local class c_2, a direct subclass of base class c, could be refined. Second, local class c_2 would become obsolete as objects of type O_{11} will never have components of type O_{22}.

Conflicts of the kind described above are semantic conflicts that should prompt a designer to re-think his design. As an alternative, a special object type NIL that is subtype of all object types defined in an object-base schema may be introduced. A conflicting situation like that described above could be resolved by restricting the inherited member type of local class c_2 to object type NIL, which is a subtype of both object types, of O_{21} and of O_{22}. Since this solution does not resolve the semantic problems that may be inherent to a schema, we do not support this alternative.

7.3 Summary

In this chapter we presented advanced concepts for the conceptual design of object-oriented schemas. First, we gave a management definition of the fundamental concepts presented in this chapter, i.e., abstraction principles and semantic relationships, to build a common terminology. Then, a set of advanced modeling concepts together with their graphic notation were presented. *Semantic relationships* allow to capture the semantics of references between real-world objects (e.g., aggregation, decomposition). *Type hierarchies* organize object types into specialization hierarchies (type specialization) and allow for the incremental design of object types. *Class hierarchies* organize object classes into specialization hierarchies and allow for the incremental design of object classes (class specialization). *Role hierarchies* organize objects into specialization hierarchies (object specialization). Role hierarchies are modeled by a specific type of semantic relationships, called role-of relationships. Two objects related by a role-of relationship represent the same real-world entity. *Local classes* extend the basic concept of object classes (cf. Chapter 3). Local classes are properties of composite objects, not global schema objects. A local class that is a property of a particular composite object o collects only objects as its members that are (direct or indirect) components of o. *Class-domain constraints* specify local classes as domains of relationships defined for members of other local classes. Specifying local classes as domains of relationships of objects models so-called local-referential-integrity constraints, which require that two objects taking part in some relationship are components of the same composite object. Summing up, the conceptual data model offers a rich set of modeling concepts that allow to build semantically rich descriptions of reality. It integrates concepts known from semantic data modeling (e.g., [HM81, HK87]) into an object-oriented framework.

Chapter 8

Process Modeling

Contents

Every organization carries out a number of organizational processes to achieve its business objectives (cf. [Oul95]. These organizational processes are not isolated, but are interwoven in a network of core processes, support processes, and management processes (cf. Section 4.3). That is, organizational processes are related to other organizational processes. Since relationships between organizational processes may have different semantics, for example, one might be a subprocess of another, we model relationships between organizational processes by semantic relationships (cf. Chapter 7). Whereas business processes can be modeled relying solely on the advanced object-oriented concepts presented in Chapter 7, we introduce further concepts for the modeling of workflow processes. For example, workflow processes may consist of a set of nested workflow processes in the advanced process model presented in this chapter. A nested workflow process is embedded into another workflow process, called its superordinate workflow process, from which it takes over part of the responsibility to process business cases. The relationship between nested workflow processes and their superordinate workflow processes is best described by a supplier-client relationship. That is, nested workflow processes are suppliers of the superordinate workflow processes. As a consequence, workflow processes are no longer monolithic models of organizational processes. In fact, a client may change its suppliers if requirements change. Furthermore, we discuss the specialization of business processes and workflow processes in this chapter. Organizing business-process types and workflow-process types into type hierarchies allows for an incremental design of organizational processes and for a disciplined sharing and modification of knowledge on organizational processes between different business-process types and workflow-process types, respectively.

This chapter is organized as follows. Section 8.1 gives a management definition of the fundamental modeling concepts presented in this chapter. Section 8.2 shows how to use the advanced object-oriented modeling concepts presented in Chapter 7 for the modeling of business processes. Section 8.3 shows how to use the advanced object-oriented modeling concepts presented in Chapter 7 for the modeling of workflow-process objects and introduces further concepts for the modeling of workflow processes, e.g., nested workflow processes. Section 8.4 summarizes this chapter.

8.1 Fundamental Modeling Concepts

Whereas business processes can be modeled relying solely on the advanced object-oriented modeling concepts presented in Chapter 7, additional concepts must be introduced to allow for a flexible modeling of workflow processes. The core concepts presented in this chapter are *nested workflow processes* and the *specialization of workflow processes*. The use of nested workflow processes considerably improves the design and maintenance of workflow processes. Instead of developing a workflow process as a monolithic model of an organizational process, a workflow process could model an organizational process using a set of nested workflow processes to which it delegates part of its duty. The philosophy to model an organizational process as a set of nested workflow processes has considerable impact on the design and maintenance of workflow-process models of organizational processes. Nested workflow processes can be designed and maintained independently[1]. Specialization of workflow processes allows to develop workflow processes incrementally. Note: Related concepts such as *organizational process*, *business process*, and *workflow process* have already been defined in Section 4.1.

Nested workflow process: A *nested workflow process* is a workflow process (cf. Section 4.1) that models only a part of the organizational process that the workflow process into which it is embedded models. The superordinate workflow process into

[1] To decompose large systems into smaller parts that may be designed and maintained independently is a modern design philosophy that limits the effects of changes to small parts of an overall system that could be exchanged independently from other parts. Nested workflow processes help to decompose the model of an organizational processes into a set of workflow processes. Each nested workflow-process models a small part of an organizational process that may be replaced by another nested workflow processes fulfilling the specification of the nested workflow process (*plug-compatible workflow processes*).

which a nested workflow process is embedded delegates part of its duty to the nested workflow process. SYNONYM: subworkflow.

Specialization of workflow processes: A workflow process w' may specialize another workflow process w in that it *extends* w with new worksteps, boxes, and nested workflow processes, and in that it *refines* some worksteps and boxes defined at w into a workflow process consisting of other worksteps, boxes, and nested workflow processes.

Summing up, we emphasize that the concepts introduced in this chapter, i.e., nested workflow processes and the specialization of workflow processes, definitely improve the flexibility of the model of workflow processes introduced in Chapter 4. The use of nested workflow processes when modeling organizational processes avoids large monolithic workflow processes that are hard to understand, difficult to maintain, hard to reuse, and impede decentralized design. Specialization of workflow processes allows for an incremental design of workflow processes. In general, the quality of the process model introduced in Chapter 4 has been improved by integrating advanced object-oriented modeling concepts into the process model. This means, modeling concepts such as semantic relationships, object-specialization, type specialization, and class specialization are available for the modeling of business processes and workflow processes, too.

8.2 Design of Business-Process Schemas

A *business-process schema* consists of a set of business-process types and a set of business-process classes. An *instance* of a business-process schema consists of a set of business cases. Each business case is instance of one business-process type and is member of one or several business-process classes (cf. Section 4.3).

In this section, we discuss the design of business-process schemas in the realm of the advanced conceptual model (cf. Section 7.2). Actually, we concentrate on the usage of semantic relationships when designing business-process schemas, in that we give a comprehensive modeling example. For details on semantic relationships refer to Subsection 7.2.1, for basics on the design of business-process schemas to Section 4.3.

The remainder of this section is organized as follows. Subsection 8.2.1 discusses the modeling of the structure of business-process objects, Subsection 8.2.2 the modeling of the behavior of business-process objects, i.e., the modeling of business processes, and Subsection 8.2.3 the modeling of classes of business-process objects.

8.2.1 Modeling the Structure of Business-Process Objects

The structure of business-process objects (or *business cases*) is determined by a set of *attributes* and a set of *relationships* (cf. Subsection 4.3.1). In Subsection 7.2.1 advanced concepts of structure modeling, i.e., semantic relationships and type specialization, have been discussed. These concepts of structure modeling apply to the modeling of the structure of business-process objects in the same way. Below, we shortly review these concepts in the context of business-process objects.

Semantic Relationships

Example: Figure 7.7 shows the type diagram of business-process type ROOM-RESERVATION. Each room reservation is a model of an occurrence of a room-reservation business process, which is a core process of hotels. A room reservation is identified by the value of its key attribute resNo and is described by the values of attributes arrivalDay and departureDay. Furthermore, a room reservations has general relationships guest, room, and category to values from domain GUEST, ROOM, and CATEGORY, respectively. All financial matters to be considered when dealing with room reservations (cf. Figure 3.1) are delegated to instances of business-process type ACCOUNT (below). That is, each room reservation has an optional has-component relationship account to a value from domain ACCOUNT. An account object models an occurrence of a business process that is subordinate to the occurrence of the business process modeled by the room-reservation object of which it is a component. Note: General relationship room models a reference of an occurrence of a room-reservation business process to an independent occurrence of a room business-process (cf. Figure 4.3 and Figure 8.4), which is actually a support process of hotels.

Example: Figure 8.1 shows the type diagram of business-process type ACCOUNT. Each account has a component-of relationship to a room reservation (above). Moreover, each account is described by the values of attributes balance, amountDue, amountPaid, discount, settlementDay, and noOfReminders. The value of general relationship room of an account is derived. The corresponding derivation expression is componentOf.room, i.e., the value of general relationship room of an account is identical to the value of general relationship room of the room-reservation of which the account is a component. Furthermore, an account has a has-component relationship services to values from domain SERVICE. Instances of type SERVICE are maintained as members of base classes Services and correspond to services consumed by a guest during her or his stay. Local class Services has two subclasses FreeServices and ChargedServices. Note: Local class FreeServices is a basic class. This means, services consumed by a customer must be explicitly declared to be free of charge in that they are added as members of local class FreeServices. Derived class ChargedServices contains all services that are members of its superclass and that are not members of its sibling class FreeServices.

Example: Figure 4.3 shows the type diagram of business-process type ROOM. Each room is a model of an occurrence of a room business-process, which is a support process of hotels. A room is identified by the value of its key attribute resNo. Furthermore, a room has general relationships category and reservations to values from domains CATEGORY and ROOM-RESERVATION, respectively. Note: An occurrence of general

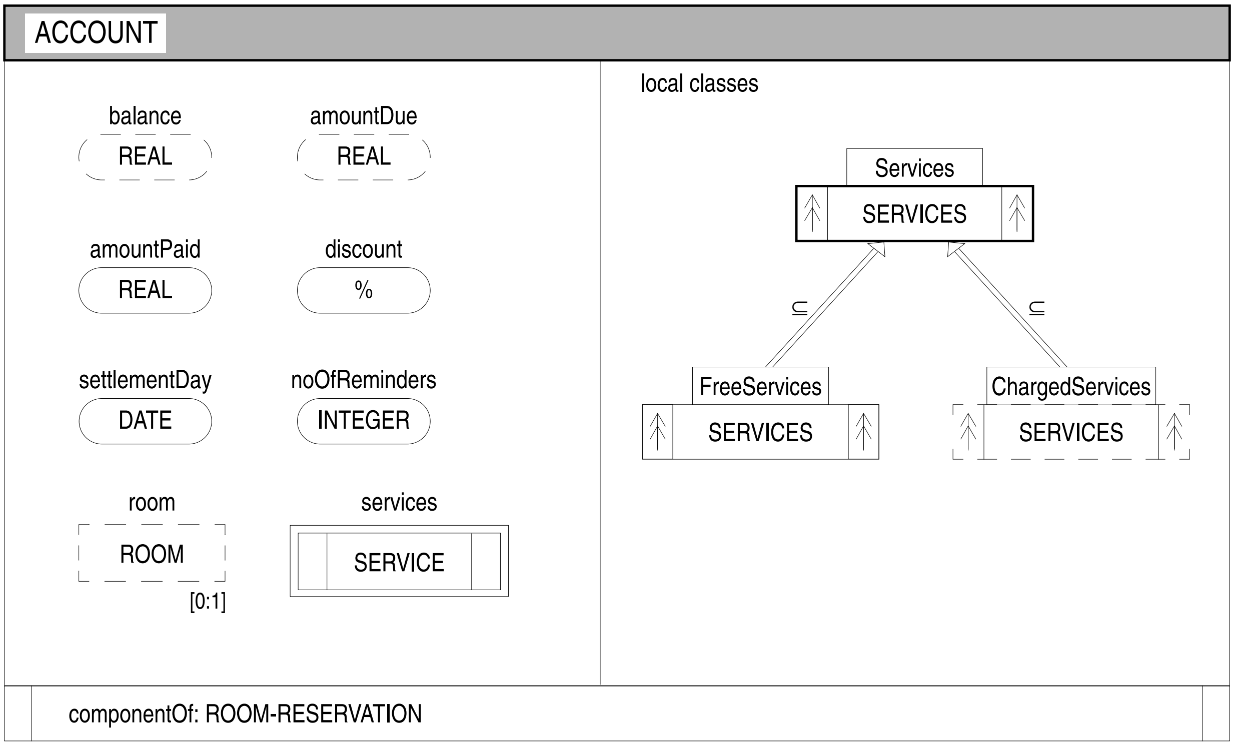


Figure 8.1: Business-process type ACCOUNT

relationship reservations models a reference of an occurrence of a room business-process to an independent occurrence of a room-reservation business process (above).

Specialization of the Structure of Business-Process Objects

To specialize the structure of business-process objects means, to specialize the set of *attributes*, the set of *relationships*, and the set of *local classes* workflow-process types inherit from their supertypes. Each user-defined business-process type is a (direct or indirect) subtype of the predefined conceptual type BP_OBJECT (Business-Process OBJECT, cf. Figure 7.6). Since neither conceptual type BP_OBJECT nor its supertype B_OBJECT (cf. Figure 7.6) do specify structure for their instances, business-process objects offer no predefined structure to their environment. Subtypes of BP_OBJECT may specialize their supertype. Note: Criteria for a consistent specialization of the structure of objects have been discussed in Subsection 7.2.1.

8.2.2 Modeling the Behavior of Business-Process Objects

The behavior of business-process objects is determined by a set of *activities*, by *allowed execution orders* on these activities, and by their *interactions* with other objects (cf. Subsection 4.3.2). In Subsection 7.2.2, advanced concepts of behavior modeling, i.e., synchronization and specialization of the behavior of objects, have been discussed. These concepts of behavior modeling apply to the modeling of the behavior of business-process objects in the same way. Below, we shortly review these concepts in the context of business-process objects.

Life-cycle

A *life-cycle* identifies business-process activities and their allowed execution orders. This means, a life-cycle presents a high-level view of a business process. Subsequently, we model a set of related business processes identified for our running example.

Example: Figure 8.2 shows the life-cycle defined at business-process type ROOM-RESERVATION. An occurrence of the room-reservation business process is always initiated by a customer requesting a reservation for a particular category of room and for a specific period. The request will be checked and, provided that there are still rooms available in the desired category and for the desired period and that the customer's creditworthiness is okay, it will be accepted. To accept a customer's request means to make an offer for the customer, which a customer may either confirm or reject. If a customer confirms, she or he can use a reservation in that she or he checks in at the hotel. Once checked in, a customer may consume services, which will be charged to his account (cf. Figures 8.1 and 8.3). On departure day a customer checks out. Note: A customer may prolong his stay in the occupied room if the room is available. However, a customer that confirmed an offer may also cancel a reservation. Irrespective of whether a reservation has not been accepted, or has been rejected, cancelled, or used, a customer may be invited to stay with the hotel next time being in town. (cf. activities apologize, saySorry, sayThanks&Sorry, and sayThanks in Figure 8.2).

Example: The financial matters of room reservations, e.g., charging, billing, and paying, are handled by a separate business process, an account business-process. Actually, an account business-process is a component business-process of a room-reservation business process. This means, for each room reservation an individual account will be created and processed. Business-process type ACCOUNT (cf. Figure 8.3) allows for a flexible handling of charging, billing, and paying of services associated with room reservations. For example, one might agree with a customer on a pre payment. In this case, a specified amount will be charged to the appropriate account, the customer will receive a bill that she or he has to pay before checking in at the hotel. If a customer does not settle the pre payment on which she or he agreed, she or he will be sent one or several reminders. If a customer that did make a pre payment cancels, she or he will get a refund. If no pre payment is agreed upon with a customer, the pre-settlement phase is simply waived.

If a customer checks in at a hotel, she or he may use several services. All services a customer consumes will be charged to a customer's account individually. Note: A visibility-restriction constraint is defined on activities startCharging, chargeService, and

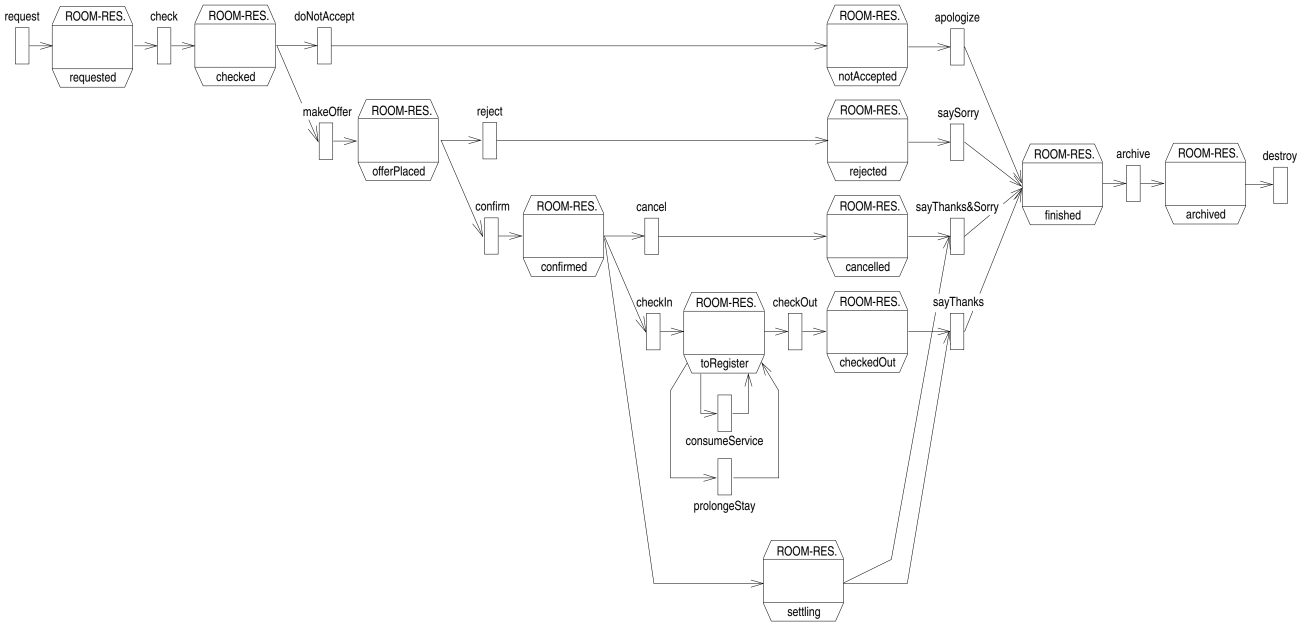


Figure 8.2: Business process at ROOM-RESERVATION

stopCharging. These activities may only be invoked by activities checkIn, useService, and checkOut, respectively, defined for room reservations. Authorized persons may also uncharge services consumed by a customer, for what reason ever. If a customer cancels a reservation, she or he may be charged with a cancellation fee. If no cancellation fee is charged, the settlement phase is simply waived. The payment phase of an account starts by billing the customer and finishes when the amount due has been cleared, i.e., the balance of the account is zero, or the hotel waives payment, for what reason ever, e.g., there is no chance to get money from a customer. One might also agree with a customer on paying the amount due in parts. In case of an overpayment, a customer will get a refund. If a customer does not pay in time, she or he will get several reminders. Finally, an account gets archived and may be destroyed sometime.

Example: Figure 8.4 shows the life-cycle defined at business-process type ROOM. A room object models an occurrence of a room business-process, which is a support process of a hotel. Before a room can be rented out, it must be furnished and explicitly made available to occurrences of a room-reservation business process of a hotel (cf. activity startRentingOut). If a room is vacant, it may be occupied for a certain room reservation. After a room becomes free, it must be serviced. Then a room is available for another reservation. If a hotel decides to re-furnish a room, the room is made unavailable to occurrences of the room-reservation business process of the hotel, in that activity stopRentingOut is performed. If the object representing a room is going to be destroyed, the furniture will be removed before.

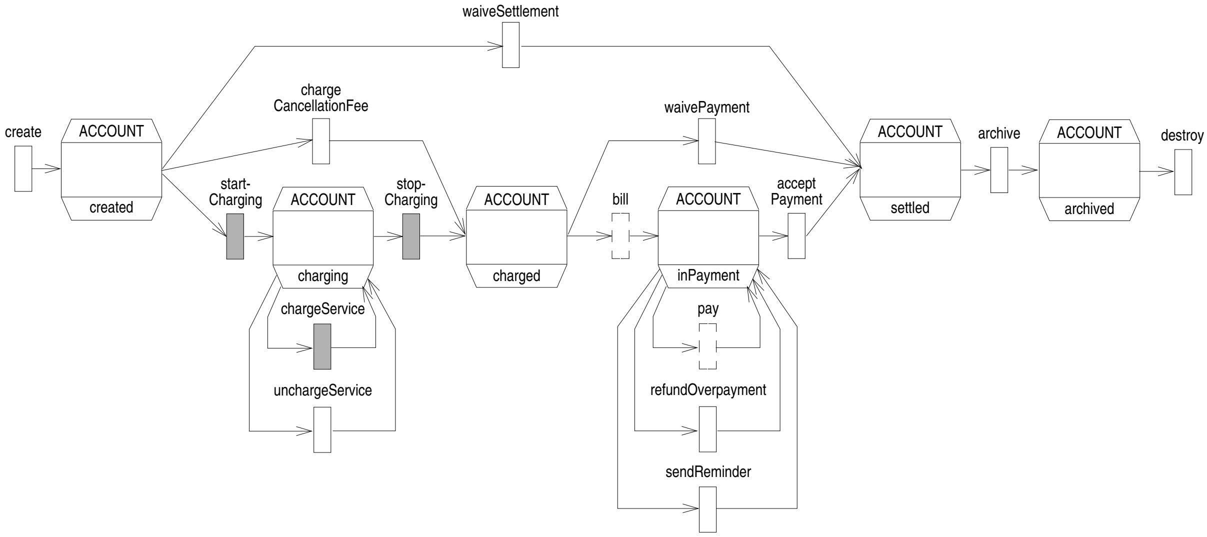


Figure 8.3: Business process at ACCOUNT

Synchronization of the Behavior of Business-Process Objects

There are three means to synchronize the behavior of business-process objects (cf. Subsection 7.2.2): (a) specifying life-cycle states as prestates or poststates of ports of activities, (b) domain-restriction constraints, and (c) visibility-restriction constraints. A visibility-restriction constraint is the most restrictive constraint to synchronize the behavior of business-process objects, since it requires that the constrained activity may only be invoked by a specific activity. Frequently, however, such strong constraints on the synchronization of the behavior of business-process objects enforce internal business rules, not external business rules, which are considered in a business-process schema. Most frequently, less restrictive means to synchronize the behavior of business-process objects are used.

Example: Figure 8.5 shows a domain-restriction constraint, which says that room-reservations may only leave state used if they reference an account that resides in state settled via has-component relationship account.

Example: Figure 8.6 shows a visibility-restriction constraint, which says that activity chargeService may only be invoked by activity useService defined for room reservations. This means, services may only be charged to a customer's account directly when she or he consumes them. This way charging of services is transparent to customers. Note: The precondition guarantees that services are charged to the "right" accounts.

Specialization of the Behavior of Business-Process Objects

To specialize the behavior of business-process objects means, to specialize *activities* and *life-cycles* business-process types inherit from their supertypes. In fact, each user-defined

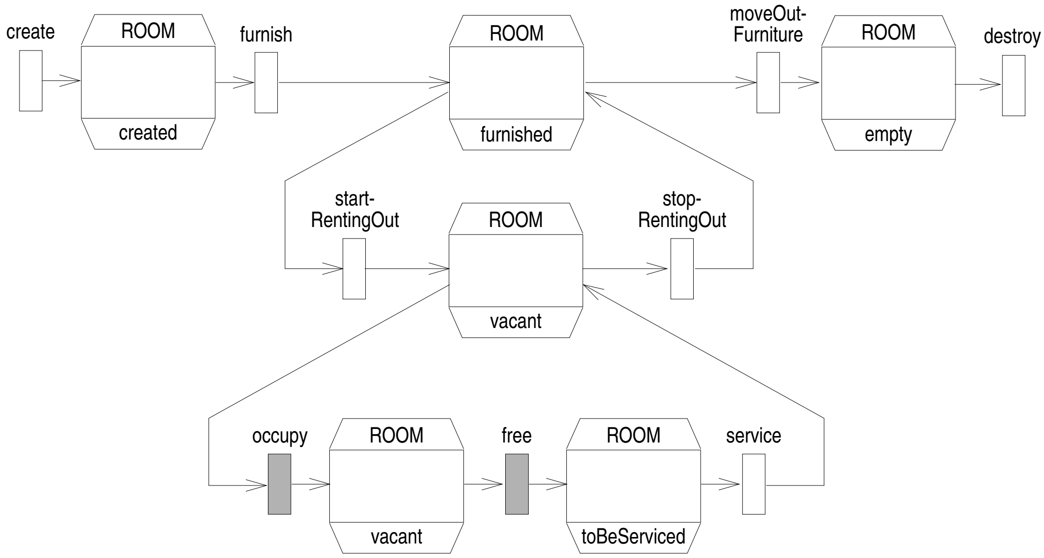


Figure 8.4: Business process at ROOM

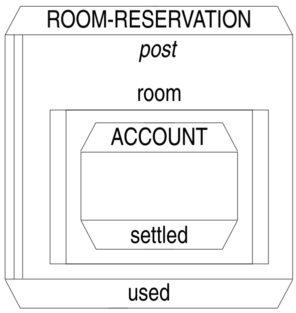


Figure 8.5: Domain-restriction constraint on state used of room reservations

business-process type is a (direct or indirect) subtype of the predefined conceptual type BP_OBJECT (Business-Process OBJECT, cf. Figure 7.6), which models the basic behavior each business-process object offers to its environment. In fact, conceptual type BP_OBJECT does not specialize the behavior it inherits from its supertype B_OBJECT (cf. Figure 7.13). Subtypes of conceptual type BP_OBJECT may specialize inherited behavior. Specializing the behavior of business-process objects, builds a specialization hierarchy of behavioral business rules (cf. Section 4.2) and allows their disciplined sharing between types organized in type hierarchies. That is, each subtype of an object type O inherits all behavioral business rules specified at O and may specialize these behavioral business rules. Note: Criteria for a consistent specialization of the behavior of objects have been discussed in Subsection 7.2.2.

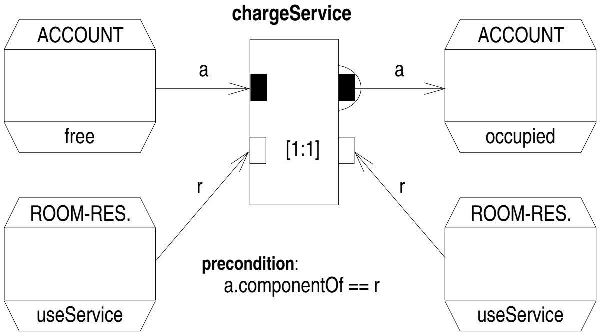


Figure 8.6: Visibility-restriction constraint on activity chargeService defined for accounts

8.2.3 Modeling Classes of Business-Process Objects

A *business-process class* collects a set of business-process objects (or business cases) as its members. Classes of business cases are modeled in the same way as classes of business objects (cf. Subsection 7.2.3). Note: To guarantee the quality of workflow transparency (cf. Section 4.2), a business case must not have a user-defined relationship to a workflow object o if it is processed according to the workflow process of o (cf. Subsection 4.3.1). That is, business cases must not be maintained as components of such workflow objects.

Example: Each room reservation is maintained as member of a base class Reservations that is a property of the hotel to which both, the room for which the reservation has been issued as well as the reservation itself, belong (cf. Figure 7.21).

For each room reservation confirmed by a customer, an account will be opened. This account will be maintained as member of a singleton base-class Account that is a property of the corresponding room-reservation object (cf. Figure 7.7). Accounts are also maintained as members of a local class Accounts that is a property of the hotel of which the reservation to which it belongs is a direct component (cf. Figure 7.21). This means, a local class Accounts of a hotel collects accounts that are indirect components of it.

During her or his stay, a guest may consume several services offered by a hotel. Each service consumed by a guest is maintained as member of a base class Services that is a property of the reservation object for his room (cf. Figure 8.1). Each base class Services has two subclasses. Local class FreeServices collects services for which guests are not charged. Since local class FreeServices is a basic class, in general, authorized persons may decide whether services consumed by a guest shall be charged to the guest's account or not (cf. activity uncharge in business process of accounts, Figure 8.3). Local class ChargedServices collects all services consumed by a guest that are not free. This means, the derivation expression for the derived class ChargedServices is services - freeServices.

For details on class types, on the specification of object classes, and on the modeling of class-domain constraints on business-process classes see Subsection 7.2.3.

8.3 Design of Workflow-Process Schemas

A *workflow-process schema* consists of a set of workflow-process types and a set of workflow-process classes. An *instance* of a workflow-process schema consists of a set of workflow objects. Each workflow object is instance of one workflow-process type and is member of one or several workflow-process classes.

Workflow objects are business objects, too. This means, the concepts presented in Section 7.2 for the conceptual modeling of business objects can be applied in the same way to the modeling of workflow objects. In this section, we review these concepts in the realm of the conceptual modeling of workflow objects. In addition, we present advanced concepts for the conceptual modeling of workflow processes.

This section is organized as follows. Subsection 8.3.1 reviews the modeling of the structure of workflow objects and introduces the concept of "behavioral" semantic relationships between workflow objects. A behavioral semantic relationship r between two workflow objects indicates that one of the workflow objects participating in r delegates part of its duty to the second one. Subsection 8.3.2 reviews the modeling of the behavior of workflow objects and Subsection 8.3.3 the modeling of classes of workflow objects in the realm of. Subsection 8.3.4 presents advanced concepts for the modeling of workflow processes. A workflow process w may delegate part of its duty to nested workflow processes, which may be processed in different organizational contexts than w.

8.3.1 Modeling the Structure of Workflow Objects

The structure of workflow objects is determined by a set of *attributes* and a set of *relationships* (cf. Subsection 4.4.1). In Subsection 7.2.1 advanced concepts of structure modeling, i.e., semantic relationships and type specialization, have been discussed. These concepts of structure modeling apply to the modeling of the structure of workflow objects in the same way. Below, we review these concepts in the context of workflow objects.

Semantic Relationships

Semantic relationships implicitly express specific semantics of references between business objects. Their semantics is not chosen arbitrary, but corresponds to abstraction principles designers frequently use to understand reality (cf. Section 7.1). If a conceptual model offers semantic relationships, designers can model reality in the way they understand it. In Subsection 7.2.1 we presented several kinds of semantic relationships that can be used to model the structure of any kind of real-world entities in an arbitrary business area. In Subsection 4.4.1 we introduced two semantic relationships that are specific to workflow objects in two-schema architectures (cf. Section 4.2). Below, we discuss criteria for the consistent use of semantic relationships to model the structure of workflow objects.

Has-full-member relationships and has-affiliate-member relationships are specific to workflow objects in two-schema architectures (cf. Subsection 4.4.1 and Section 4.2). A workflow object has one has-full-member relationship to each type of business cases that are handled according to its workflow process. This means, a workflow object has at least one has-full-member relationship. Moreover, a workflow object has one has-affiliate-member relationship to each type of business cases that are indirectly handled by its workflow process. *The domains of has-full-member relationships as well as of has-affiliate-member relationships must be business-process types.* Has-full-member relationships must not be derived. Specifying a set of life-cycle constraints for each has-full-member relationship and for each has-affiliate-member relationship increases the effort to specify them and to specialize them (below). The additional effort, however, definitely pays off because of the increased expressiveness of workflow-process schemas. Specifying life-cycle constraints on has-full-member relationships and has-affiliate-member relationships is like specifying a contract between workflow processes and business cases in which the services a workflow process may (directly or indirectly) invoke to achieve its objectives are explicitly mentioned. For example, the availability of such a "contract" supports decentralized design of workflow processes and makes it easy to test whether it is legal to assign a business case as full member to a workflow process or not[2].

Example: Figure 8.7 shows a type diagram of workflow-process type ROOM-RESERVATION-WF. Each room-reservation workflow object has two has-full-member relationships. That is, relationship reservations to values from domain ROOM-RESERVATION and relationship accounts to values from domain ACCOUNT. Moreover, a room-reservation workflow object has a derived has-affiliate-member relationship rooms to values from domain ROOM. The life-cycle specified as constraint on relationship rooms has already been shown in Figure 4.7. The life-cycle specified as constraint on relationship reservations corresponds to the life-cycle defined at type ROOM-RESERVATION (cf. Figure 8.2), that specified as constraint on relationship accounts to the life-cycle defined at type ACCOUNT (cf. Figure 8.3).

Semantic relationships presented in Subsection 7.2.1 are general in that they can be used to model the structure of any kind of real-world entity in an arbitrary business area. These semantic relationships allow to express that a business object is a component, a constituent, or a role of another business object, and that related business objects are independent. That is, these semantic relationships correspond to the abstraction principles of decomposition, aggregation, object specialization, and association, respectively (cf. Section 7.1). From the semantics of the abstraction principles, we infer the following rules on the consistent use of semantic relationships for the modeling of the structure of workflow objects in the context of overall workflow model (cf. Chapters 7 to 10):

1. A workflow object may have *general relationships* to any kind of business objects.

2. A workflow object may have *has-component relationships* only to other workflow objects. Each workflow object has a *component-of relationship* either to another

[2]A business case may be full member of two workflow processes at a time either if one workflow process is a nested workflow process of the other or if the life-cycles defined as constraint on the has-full-member relationships of the two workflow processes do not overlap in any business-process activity.

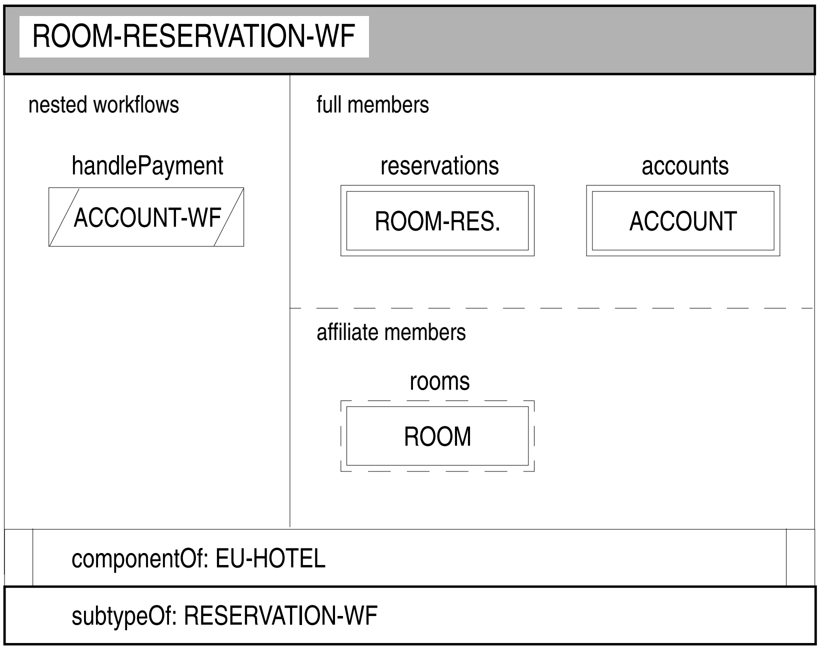


Figure 8.7: Workflow-process type ROOM-RESERVATION-WF

workflow object (cf. nested workflow processes, below) or to an organizational unit (cf. Chapter 9).

3. A workflow object may have *has-constituent relationships* only to other workflow objects and may have *constituent-of relationships* to other workflow objects and to organizational units (cf. Chapter 9).

4. A workflow object may have both *has-role relationships* as well as *role-of relationships* only to other workflow objects.

Relationships between workflow objects model a network of workflow processes an organization carries out to achieve its business objectives (cf. Subsection 4.4.2). Semantic relationships impose a specific semantics on these relationships. Because semantic relationships defined in Subsection 7.2.1 have their origin in semantic data modeling (cf. [HM81, HK87]), their semantics is defined with a focus on structural aspects of business objects. Semantic relationships between processes have their origin in process modeling (e.g., [CKO92, KB95]). As a consequence, their semantics is defined with a focus on behavioral aspects of processes.

We distinguish two basic kinds of "behavioral" semantic relationships between workflow processes: First, a workflow process can be *nested into* another workflow process. A workflow process into which one or several other workflow processes are nested is called complex workflow process. The development of a nested workflow process is very tightly related to the development of the complex workflow process into which it is nested. Assume a workflow process is developed by stepwise refinements of its elements, i.e., boxes and worksteps, (cf. Subsection 4.4.4). The refinement of an element of a workflow process constitutes itself a workflow process. The refinement of an element of a workflow process w typically is specified at the same workflow-process type as w. If a workflow process w' that represents the refinement of an element of a workflow process w is defined at a different workflow-process type as w, w' is said to be a nested workflow process of w. Considering the processing of a single business case there is a synchronous interaction between complex workflow processes and respective nested workflow processes. We model references from complex workflow processes to nested workflow processes by *has-nested-workflow relationships* and their inverses by *nested-workflow-of relationships*. Modeling of inverses is mandatory. Second, two workflow processes may be related but "*behaviorally independent*". Two related but "behaviorally independent" workflow processes represent different organizational processes. Behaviorally independent workflow processes can be developed independently and typically exchange resources relying on buffers. This means, two related but "behaviorally independent" workflow processes interact asynchronously considering the processing of single business cases. We model references between related but "behaviorally independent" workflow processes by *has-independent-workflow relationships*. Relationships inverse to has-independent-workflow relationships must be has-independent-workflow relationships, too. Modeling of inverses is mandatory.

These "behavioral" semantic relationships have a behavioral aspect as well as a structural aspect. The behavioral part of a has-nested-workflow relationship and its inverse nested-workflow-of relationship is modeled by the specification of a nested workflow process within a superordinate workflow process (cf. Subsection 8.3.4). The behavioral part of a has-independent-workflow relationship is modeled by a set of boxes that collect business cases the supplier offers to the client (mandatory) and by a set of boxes that collect business cases the client passes to the supplier for processing (optional). The specification of the behavioral part of "behavioral" semantic relationships is discussed in more detail in Subsection 8.3.4. The structural aspect of "behavioral" semantic relationships is modeled relying on "structural" semantic relationships (cf. Subsection 7.2.1).

The structural aspect of a has-nested-workflow relationship can be modeled by any kind of object-describing relationship. That is, the structural aspect of a has-nested-workflow relationship can be modeled using either a general relationship, a has-component relationship, a constituent-of relationship, or a has-role relationship (cf. Subsection 7.2.1). Has-nested-workflow relationships are either single-valued or multi-valued, and may be optional. The structural aspect of a nested-workflow-of relationship can be modeled using either a mandatory general relationship or any kind of object-defining relationship, i.e., a component-of relationship, a has-constituent relationship, or a role-of relationship. This means, nested-workflow-of relationships are always mandatory. Object-defining relationships express that the existence of a nested workflow process depends on the existence of the superordinate workflow process into which it is embedded. General relationships express that the superordinate workflow process into which a workflow process is nested may change during the existence of the nested workflow process. However, since general relationships must be mandatory, they express that a nested workflow process cannot exist without a superordinate workflow process into which it is embedded. Below, we give a

existence		superordinates		org. contexts		*kind of semantic relationship(s)*
independent	dependent	one	several	same	different	
x		x	x[†]	x	x	general
	x		x[‡]	x	x	has-constituent
	x	x		x		component-of
	x	x			x	role-of

[†]A nested workflow process that exists independently from several workflow processes may have one or several general relationships to its superordinate workflow processes.

[‡]If the existence of a nested workflow process depends on two or more superordinate workflow processes, each of a different type, the nested workflow process has several has-constituent relationships, one to each of its superordinate workflow processes.

Table 8.1: Semantics of nested-workflow-of relationships

guideline saying in which situation to use what kind of semantic relationship to model the structural aspect of a nested-workflow-of relationship. Inverses of nested-workflow-of relationships are modeled by allowed inverses of the selected "structural" semantic relationships. Note: Modeling of inverses is mandatory for nested-workflow-of relationships.

Table 8.1 shows several criteria that influence the decision on which kind of "structural" semantic relationship to use to model the structural aspect of a nested-workflow-of relationship. The *existence* of a nested workflow process *need not* but *may* depend on the existence of its superordinate workflow process(es). In fact, a nested workflow process cannot exist without a superordinate workflow process into which it is nested, but it may or may not be nested into different superordinate workflow processes during its lifetime. If a workflow process can be nested into different superordinate workflow processes during its lifetime, use a general relationship to model a nested-workflow-of relationship. If the existence of a workflow process depends on the existence of the superordinate workflow process(es) into which it is nested, use an object-defining relationship to model a nested-workflow-of relationship. Role-of relationships and component-of relationships express that the existence of the nested workflow process depends on a single superordinate workflow. Use a role-of relationship if a nested workflow process that depends on a single superordinate workflow process performs in a different organizational context as its superordinate workflow process, use a component-of relationship if it performs in the same organizational context. The *organizational context* within which a workflow process performs is determined by the value of its component-of relationship[3]. This means, if a nested workflow process performs in a different organizational context than its superordinate workflow process, it must not be a component of its superordinate workflow process. Has-constituent relationships allow to model situations in which a workflow process is nested into several superordinate workflow processes, maybe each of a different type, on which its existence depends. That

[3]We assume that each workflow process is either component of an organizational unit, i.e., workflow processes are resources of organizational units to which they exclusively belong and on which their existence depends (cf. Chapter 4), or component of a workflow process into which it is nested.

is, a workflow process that depends on two or more superordinate workflow processes has several has-constituent relationships, one to each of them. In situations not mentioned above, use general relationships to model nested-workflow-of relationships.

A has-nested-workflow relationship is modeled by some kind of object-describing relationship depending on the kind of semantic relationship used to model its inverse nested-workflow-of relationship. Consequently, the *existence* of a superordinate workflow process is always independent from the existence of its nested workflow process(es).

In a type diagram, nested-workflow-of relationships are emphasized in that their name must start with the prefix nestedWorkflowOf. Has-nested-workflow relationships are emphasized in that a special section of the type diagram is devoted to them. The section is separated from other sections of a type diagram by a solid-line border and is identified by the keywords nested workflows.

Example: Each room-reservation workflow process has a has-nested-workflow relationship handlePayment to an account workflow-process that depends on it but exclusively belongs to the organizational unit in which it occurs. That is, the has-nested-workflow relationship handlePayment is modeled by a has-role relationship (cf. Figure 8.7). The inverse nested-workflow-of relationship of account workflow-processes is modeled by a role-of relationship (cf. Figure 8.8).

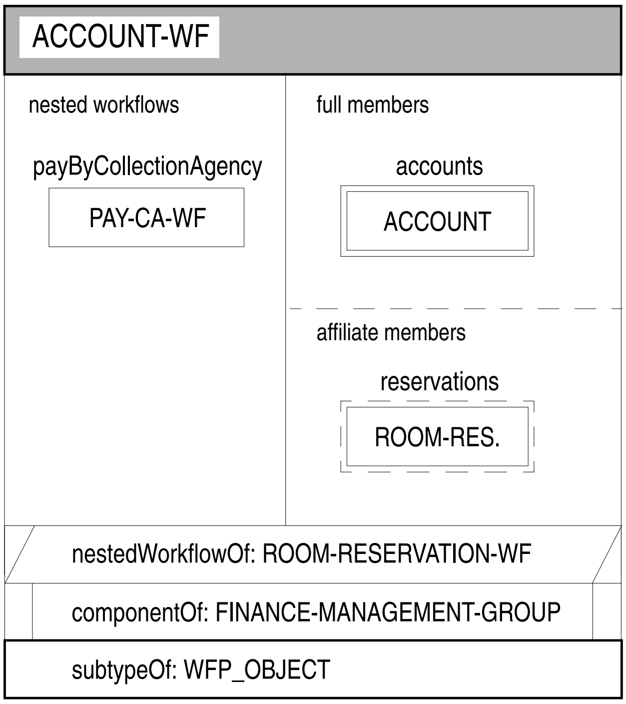


Figure 8.8: Workflow-process type ACCOUNT-WF

A *has-independent-workflow relationship* models a relationship between related but "behaviorally independent" workflow processes (cf. definition of semantics of has-nested-workflow relationships and has-independent-workflow relationships, above). The structural aspect of a has-independent-workflow relationship is always modeled using a general relationship. In a type diagram, has-independent-workflow relationships are shown in the "general" section of the type diagram.

Example: Each room-reservation workflow process has a has-independent-workflow relationship room-wf to a room workflow-process, which offers vacant rooms to a room-reservation workflow process. Note: Full members of this related room workflow-process are affiliate members of a room-reservation workflow process, i.e., rooms assigned to reservations of a hotel (cf. Figure 8.7).

Specialization of the Structure of Workflow Objects

To specialize the structure of workflow objects means, to specialize the set of *attributes*, the set of *relationships*, and the set of *local classes* workflow-process types inherit from their supertypes. In fact, each user-defined workflow-process type is a (direct or indirect) subtype of the predefined conceptual type WFP_OBJECT (WorkFlow-Process OBJECT, cf. Figure 8.9), which models the basic structure each workflow-process object offers to its environment. That is, each workflow object has a has-full-member relationship to a set of instances of some business-process type. Conceptual type WFP_OBJECT is a direct subtype of conceptual type B_OBJECT (cf. Figure 7.6). Subtypes of WFP_OBJECT may specialize inherited structure. Note: Criteria for a consistent specialization of the structure of objects have been discussed in Subsection 7.2.1.

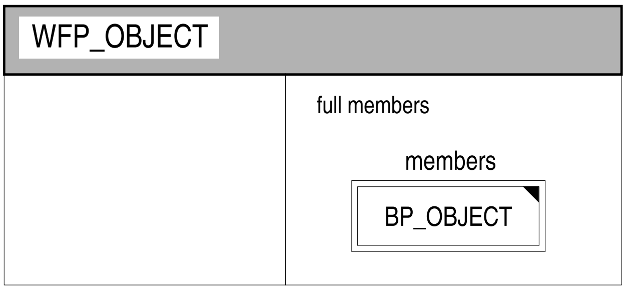


Figure 8.9: Conceptual type WFP_OBJECT

The criteria for a consistent specialization of the structure of business objects (cf. Subsection 7.2.1) must be extended to ensure a consistent specialization of has-full-member relationships and has-affiliate-member relationships of workflow objects.

1. A subtype may specialize a life-cycle[4] specified as constraint on an inherited has-full-member relationship or on an inherited has-affiliate-member relationship (*Rule of Specialization of Life-Cycle Constraints on Has-Full-Member Relationships and Has-Affiliate-Member Relationships*).

[4]Rules for the consistent specialization of life-cycles are given in Subsection 7.2.2.

Example: The life-cycle specified as constraint on has-full-member relationship reservations at workflow-process type ROOM-RESERVATION-WF, it corresponds to the life-cycle defined at the type domain of relationship reservations (cf. Figure 8.2), is an invocation consistent specialization of the inherited life-cycle constraint (cf. Figure 8.10). Activities issue, use, and finish are abstract. This means, they need not be invocable on indirect instances of type RESERVATION. Note: Invocation consistency requires only that activities that are *invocable* on instances of a supertype (if there would be any) are under the same conditions invocable on instances of a subtype. Abstract activities are not invocable.

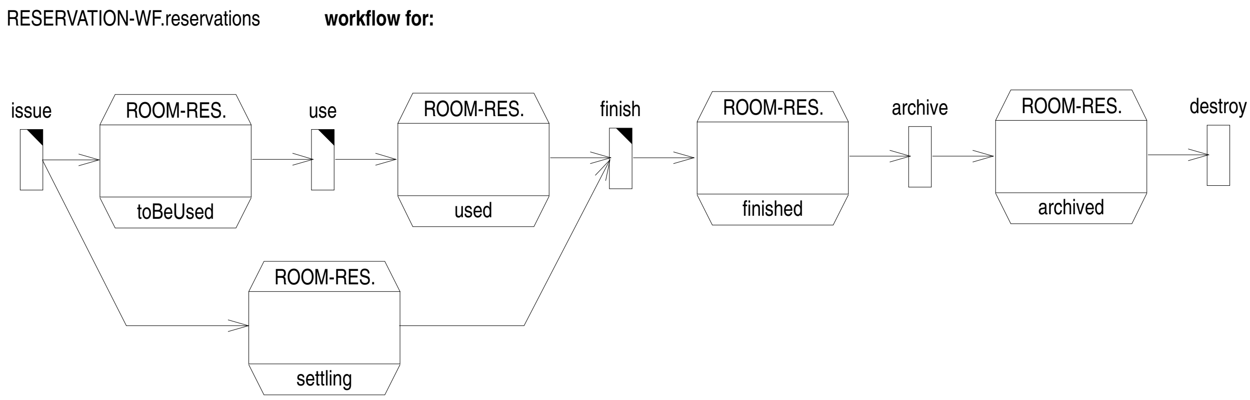


Figure 8.10: Life-cycle constraint at RESERVATION-WF

8.3.2 Modeling the Behavior of Workflow Objects

The behavior of workflow objects is determined by a set of *activities*, by *allowed execution orders* on these activities, and by their *interactions* with other objects (cf. Subsection 4.4.2). In Subsection 7.2.2 advanced concepts of behavior modeling, i.e., synchronization and specialization of the behavior of objects, have been discussed. These concepts of behavior modeling apply to the modeling of the behavior of workflow objects in the same way. Below, we discussion a special aspect of the synchronization of the behavior of workflow objects not discussed in Subsection 8.2.2, i.e., the synchronization of the behavior of workflow objects with the behavior of the business cases that are processes according to the workflow processes of these workflow objects. Afterwards, we shortly review the specialization of the behavior of workflow objects.

Synchronization of the Behavior of Workflow Objects

An organization carries out a number of organizational processes to meet its objectives (cf. Subsection 4.3.1). Usually, these organizational processes form a network of processes, where individual processes cannot be executed in isolation but depend on other processes for support. This means, workflow processes may only be executed on business cases in a consistent way if relationships to other workflow processes are properly defined and are not changed arbitrarily during the executions of workflow processes. Below, we will discuss a set of requirements that ensure a consistent execution of workflow processes if they are satisfied. First, however, we discuss the life-cycle defined at the predefined conceptual type WFP_OBJECT (WorkFlow-Process OBJECT, cf. Figure 7.6), which models the basic behavior each workflow-process object offers to its environment.

Example: Figure 8.11 shows the life-cycle of workflow objects defined at type WFP_OBJECT. After a workflow object has been created, it resides in state suspended. For example, relationships to other workflow objects may be determined and modified in this phase of the life-cycle of a workflow object. Workflow objects must be activated explicitly. If a workflow object resides in state active, business cases may be added to and removed from it. Invoking activity suspend transfers a workflow object again into state suspended. Before a room-reservation workflow object may be archived, it must be deactivated explicitly. Finally, a workflow object may be destroyed.

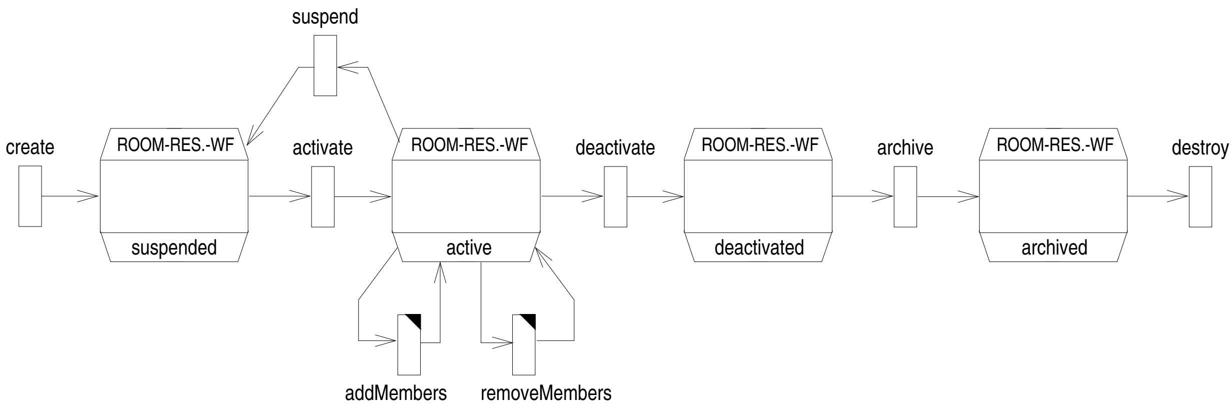


Figure 8.11: Life-cycle at WFP_OBJECT

As mentioned before, in general a workflow process depends on related workflow processes to meet its objectives. For example, a workflow process w may have a nested workflow processes n to which it delegates part of its duty of processing certain business cases. For w to meet its objectives requires that the availability of n is guaranteed for w whenever w wants to delegate business cases to n, where workflow process n is available to w if w has a relationship to n and n is ready to process business cases it receives from w. To ensure a consistent execution of workflow processes, we give a rather rigid model that requires (a) that relationships between workflow objects are properly defined when workflow processes start to process business cases and (b) that relationships between workflow objects are not modified while participating workflow processes do process business cases. Comment 8.1 discusses in which way these requirements may be relaxed and what must be considered in such situations.

The behavior of the predefined conceptual type WFP_OBJECT (WorkFlow Process OBJECT) (cf. Figure 8.11) defines separate states that allow to properly separate matters.

That is, a workflow process may process business cases only if the underlying workflow object resides in state active, and relationships between workflow objects may be modified only if the participating workflow objects reside in state suspended. Note: Since objects reside in basic *and* in abstract states, subtypes of WFP_OBJECT may refine states suspended and active without causing any problems to consistency of execution of workflow processes.

To guarantee that workflow processes may only process business cases if the underlying workflow object resides in state active, the precondition this_{wf}.inStates: {active} is implicitly defined for every workstep and nested workflow process of workflow processes, where this_{wf} refers to the underlying workflow object. Moreover, if workflow processes shall accept business cases as new members and shall allow to remove current members only when the underlying workflow objects reside in state active, public update operations for has-full-member relationships and has-affiliate-member relationships of workflow objects may be defined as refinements of the abstract activities addMembers and removeMembers inherited from type WFP_OBJECT (cf. Figure 8.11).

To guarantee that relationships between workflow objects may be modified only if the participating objects reside in state suspended, designers have to ensure that life-cycle state suspended is given as prestate *and* as poststate of public update activities on relationships between workflow objects. If a subtype of WFP_OBJECT refines state suspended, states in its refinements can be given as prestates and poststates of public update activities on relationships between workflow objects.

To guarantee that relationships between workflow objects are properly determined when a workflow process starts to process business cases, preconditions of activities that transfer a workflow object into state active (or a refinement thereof) has to test that relationships to other workflow objects are properly defined (cf. activity activate in Figure 8.11). Note: Mandatory relationships of workflow objects must already be defined at the time workflow objects are created.

In short, we require that a workflow process w may only processes business cases if relationships to other workflow processes are properly determined such that w can meet its objectives.

Comment 8.1 (Instantiation of Nested Workflow Processes)
The given requirements are rather rigid. For example, they do not allow that nested workflow processes are instantiated while their superordinate workflow processes do process business cases. If this is desired, e.g., an individual instance of a nested workflow process shall be created for every business case on which the nested workflow process is started, the given requirements must be relaxed accordingly. Below, we will discuss several points to be considered in such a situation:

1. *The instantiation of an object must be synchronized with the behavior of that composite object of which the instantiated object is a component.* If a nested workflow process is a *component* of its superordinate workflow process, instantiation of the nested workflow process must be synchronized only with the behavior of the workflow object underlying the superordinate workflow process. If the workflow object underlying a nested workflow process has a general relationship, a has-constituent relationship, or a role-of relationship to the workflow object underlying the superordinate workflow process, instantiation of the nested workflow process must be synchronized with the behavior of the workflow object underlying the superordinate workflow process and at least with the behavior of

the composite object to which the workflow object underlying the nested workflow process belongs, i.e., of which it is a component.

In short, if the effort to synchronize the behaviors of objects at the creation of a nested workflow process is limited to the workflow objects that underlie the nested workflow process and the superordinate workflow process, nested workflow processes might be created and deleted while the superordinate workflow process does process business cases. We assume that one nested workflow process will be created for each business case on which the nested workflow process is started. Further, we assume that a designer may specify *declaratively* whether a nested workflow process shall be instantiated when started on a business case or not. Note: If a nested workflow process is instantiated when started on a business case, proper authorizations must be determined for agents on the new workflow-process instance.

2. *Dependencies between workflow processes could be minimized by the use of management processes.* We assume that a management process could be used as a *proxy* for a related workflow process thereby decoupling related workflow processes. Assume each workflow process w_s of type W_s has a nested workflow process w_n of type W_n that occurs in a different organizational context then w_s, e.g., w_s is processed in the reception of a hotel and w_n in the finance-management group of a hotel. Whenever a new nested workflow process of type W_n is to be introduced for a particular superordinate workflow process of type W_s, two organizational units have to synchronize their matters. If, however, workflow processes of types W_s and W_n are decoupled in that each workflow process of type W_s refers to a management process of type W_m that refers to a nested workflow process of type W_n, a new instance of W_n could be introduced for a particular superordinate workflow process w_s of type W_s without requiring w_s to transfer to state suspended. The introduction of a new instance of W_n must only be synchronized with that management process w_m of type W_m to which w_s refers. If related workflow processes occur in different organizational contexts, management processes may occur in either of these organizational contexts. Assume a management process of type W_m occurs in the context of a nested workflow process W_n. In this case, the organizational context in which a superordinate workflow process of type W_s occurs may not be effected at all by a change of a nested workflow process.

Specialization of the Behavior of Workflow Objects

To specialize the behavior of workflow objects means, to specialize *activities* and *life-cycles* workflow-process types inherit from their supertypes. In fact, each user-defined workflow-process type is a (direct or indirect) subtype of the predefined conceptual type WFP_OBJECT (WorkFlow-Process OBJECT, cf. Figure 7.6), which models the basic behavior each workflow-process object offers to its environment. Conceptual type WFP_OBJECT specializes the basic behavior it inherits from its supertype B_OBJECT (cf. Figure 7.13). The life-cycle defined at conceptual type WFP_OBJECT is shown in Figure 8.11. Subtypes of conceptual type WFP_OBJECT may specialize inherited behavior. Note: Criteria for a consistent specialization of the behavior of objects have been discussed in Subsection 7.2.2.

Example: Figure 8.12(a) shows the specification of activity create at RESERVATION-WF. Activity create may only be invoked by activity setReservation-wf invoked on an instance of type ACCOMMODATION. (Note: setReservation-wf is a public update operation of the single-valued relationship reservation-wf of accommodations.) Activity create instantiates a new object of type RESERVATION-WF that resides initially in state suspended.

Figure 8.12(b) shows the redefinition of activity create at ROOM-RESERVATION-WF. The type domain of port c has been refined to HOTEL, a direct subtype of ACCOMMODATION (cf. Figure 7.20). That is, activity create may only be invoked by activity setReservation-wf invoked on an instance of type HOTEL.

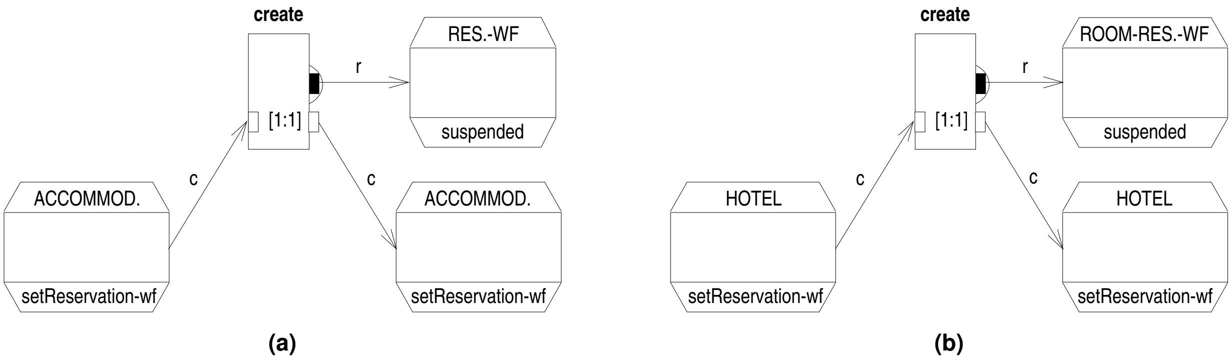


Figure 8.12: Specifications of activity create

Example: Figure 8.13 shows the life-cycle defined at ROOM-RESERVATION-WF. Workflow-process type ROOM-RESERVATION-WF is a direct subtype of RESERVATION-WF, which is again a direct subtype of WFP_OBJECT. Workflow-process type RESERVATION-WF specializes the life-cycle defined at WFP_OBJECT (cf. Figure 8.11) in that it specializes abstract activities addMembers and removeMembers to activities addReservations and removeReservations, respectively. (Note: The life-cycle defined at workflow-process type RESERVATION-WF is not shown.) Workflow-process type ROOM-RESERVATION-WF specializes the life-cycle defined at RESERVATION-WF in that it extends it with additional activities setHandlePayment and setRoom-wf, which allow to modify relationships to related workflow objects if a room-reservation workflow object resides in state suspended.

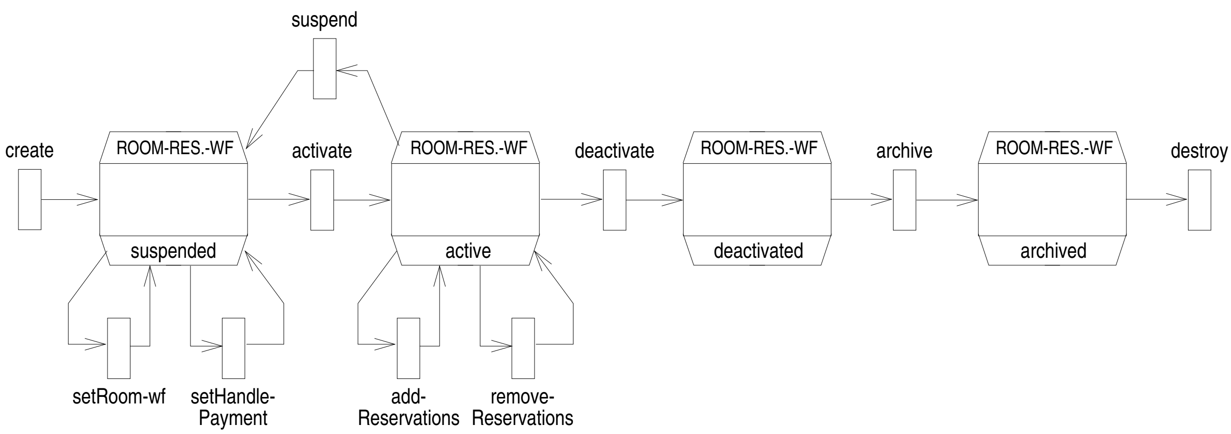


Figure 8.13: Life-cycle at ROOM-RESERVATION-WF

8.3.3 Modeling Classes of Workflow Objects

A *workflow-object class* collects a set of workflow objects as its members. Classes of workflow objects are modeled in the same way as classes of business objects (cf. Subsection 7.2.3).

8.3.4 Modeling of Workflow Processes

A *workflow process* is a model of an organizational process as it is currently performed in a particular organization. The objective of a workflow process is to handle business cases in a way that is consistent with current organizational settings. Since a workflow process modeled according to the concepts offered by the basic conceptual model does only know worksteps to handle business cases (cf. Subsection 4.4.4) but does not know how to delegate part of his duty to other workflow processes, resulting workflow process models of organizational processes are monolithic, consisting of a set of basic worksteps and a set of basic boxes.

From other disciplines like semantic data modeling (e.g., [HM81, HK87]) or software engineering (e.g., [PB93]), we know that large monolithic models of reality are hard to understand, difficult to maintain, hard to reuse, and impede decentralized design. To motivate a more flexible model of workflow processes, we give some examples that highlight the weaknesses of monolithic workflow-process models wrt. their decentralized design, their reuse, and their maintenance. Assume there is an organizational process that crosses several organizational units of an organization. Decentralized design of such an organizational process, e.g., each organizational unit designs that part of the organizational process for which it is responsible, is hindered by a monolithic workflow-process model, since it is designed at a single workflow-process type making coordination of the design activities difficult simply because of several organizational units must have access to that single workflow-process type. Reuse of design results of an individual organizational unit is also hindered, since they are tied into a monolithic workflow-process model of the organizational process.

Example: Assume the room-reservation process of a hotel crosses two organizational units of a hotel. A *reception* that is the main responsible of the process, and a *finance-management group* that is responsible for charging of cancelled reservations as well as for charging of reservations of customers that did check out without paying. This means, the finance-management group is responsible for worksteps ChargeCancellationFee&Bill, ChargeServices&Bill, AcceptPayment (cf. Figure 4.9), and worksteps in the refinement of box R-7 (cf. Figures 4.9 and 4.11). Neither decentralized design nor reuse of the design results of the finance-management group is supported.

Maintenance of workflow processes is concerned with adapting workflow processes to changes in the environment of an organization. The instrument of organizations to react to such changes are organizational commitments, which are modeled by internal business rules. This means, changes to workflow processes are most frequently caused by changes in the set of internal business rules of an organization. The two-schema architecture for workflow systems (cf. Section 4.2) builds on this fundamental observation. A workflow system that relies on the two-schema architecture has two models of an organizational process it enacts. A business-process model, which considers only external business rules, and a workflow-process model, which considers only internal business rules explicitly. As a consequence, many changes require only to adapt the workflow-process model of an organizational process. Although the two-schema architecture does considerably improve the quality of workflow-process schemas wrt. maintenance, further improvements are still possible. Assume again there is an organizational process that crosses several organizational units of an organization. Assume further that a part of the overall organizational process that falls into the responsibility of a single organizational unit changes considerably requiring a redesign of the workflow-process model. With a monolithic workflow-process model of an organizational process, such a change influences all organizational units participating in the organizational process, since a new workflow process object must replace the old one to make the changes effective in the organization. Such a change requires to migrate existing business cases from the old workflow process to the new one and requires to grant appropriate authorizations for agents on worksteps of the new workflow process, even for those parts of an organizational process not affected by the changes.

Example: Assume the finance-management group of a hotel decides to considerably change payment for room reservations in its responsibility (cf. example, above). Despite changes are limited to a narrow scope of the room-reservation workflow process, a new room-reservation workflow-process type must be designed considering the changes[5] and an instance of the new room-reservation workflow-process type must replace the existing workflow-process instance making the changes effective. Making changes effective requires also that room-reservations have not been paid for yet, are migrated from the old workflow-process instance to the new workflow-process instance. For example, room reservations that have just been issued, i.e., room reservations that qualify for boxes R-1 and R-2, must be migrated to the new workflow-process instance, to guarantee that they are processed according to the changes in the payment if the respective customer cancels or checks out without paying[6]. This means, changes in a workflow process that is within the responsibility of the finance group of a hotel causes changes to a workflow process that is within the responsibility of the reception of a hotel. Besides having to migrate business cases to a new workflow process, the reception of a hotel must care for creating appropriate authorizations for its agents on the new workflow-process instance.

The advanced conceptual model will considerably improve the quality of workflow-

[5] If versioning is supported, the 'new' workflow-process type may be a version of the changed one.

[6] Ellis and Keddara [EK95] coined the term *dynamic change* for the problem of migrating existing business cases to changed workflow processes. In [BPS97a] we did present a flexible approach to "dynamic change" for the workflow model presented in this thesis. For example, business cases need not migrate at once to a new workflow process.

process models in that it allows to model an organizational process by a set of nested workflow processes. A *nested workflow process* models part of the organizational process the workflow process into which it is embedded models. This means, in the advanced conceptual model a workflow process may rely on worksteps *and* nested workflow processes to meet its objectives. Nested workflow processes support decentralized design of organizational processes, maybe changed independently from the workflow process into which they are embedded thereby limiting the effect of changes to a narrow scope, and improve readability of workflow-process schemas. Nesting workflow processes also has impact on the structure of workflow objects, i.e., a workflow process has relationships to its nested workflow processes and vice versa.

In the remainder of this subsection we will discuss the extended model of workflow processes, the specification and realization of nested workflow processes, and the specialization of workflow processes in general.

Workflow Process

A *workflow process* models an organizational process as it is currently performed in a particular organization by a set of *boxes*, a set of *worksteps*, a set of *nested workflow processes*, and a set of directed *arcs* connecting worksteps as well as nested workflow processes to boxes and vice versa.

In a workflow-process diagram, a nested workflow process is depicted by a rectangle showing its name above, like worksteps are depicted. To distinguish nested workflow processes from ordinary worksteps, however, a shaded graphical symbol indicating the semantics of the underlying has-nested-workflow relationship is superimposed on the graphical symbol representing a nested workflow process. For example, a shaded parallelogram is superimposed in the case of an underlying has-role relationship (cf. Subsection 8.3.1). Furthermore, the workflow-process type specified as domain of the corresponding has-nested-workflow relationship is shown behind the name of a nested workflow process. As a notational convention, a nested workflow process and the underlying has-nested-workflow relationship share the same name. The former starts with a capital letter, the latter with a small one.

Example: Figure 8.14 shows a workflow-process that models a room-reservation process of a hotel, including payment for reservations. It is specified at workflow-process type ROOM-RESERVATION-WF. The workflow process should be self-explanatory. It models the room-reservation process by a set of boxes, a set of worksteps, and a nested workflow process HandlePayment. The nested workflow process can be started on cancelled as well as on used reservations (cf. disjunctive boxes R-6 and R-9 connected to the nested workflow process). The structural aspect the corresponding has-nested-workflow relationship handlePayment is modeled by a has-role relationship (cf. Figure 8.7) with domain ACCOUNT-WF.

Nested workflow processes are smoothly integrated into the basic model of workflow processes (cf. Subsection 4.4.4). *A nested workflow process is modeled like an abstract*

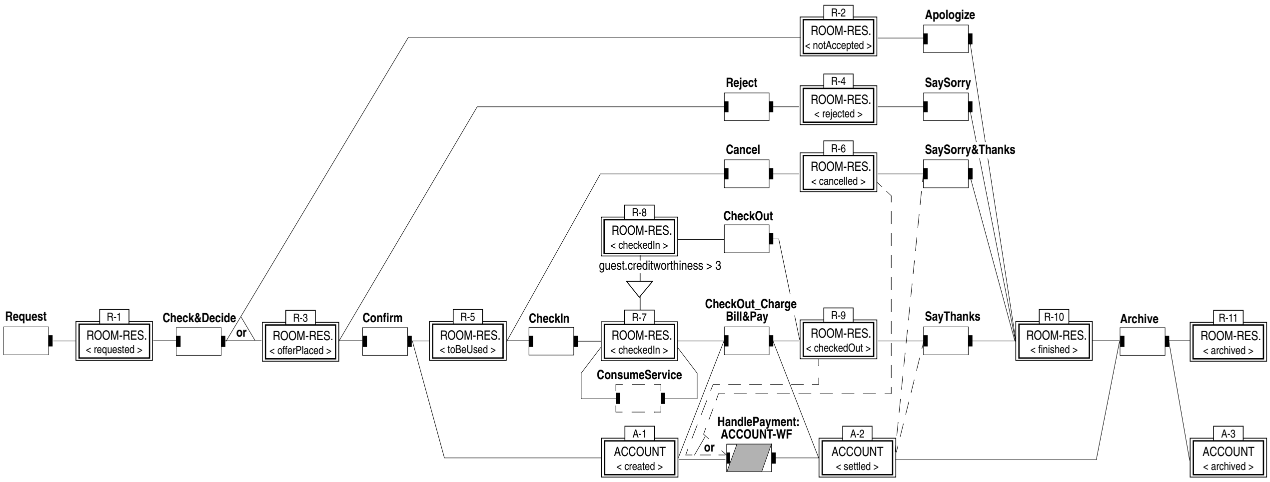


Figure 8.14: Workflow process at ROOM-RESERVATION-WF

workstep. This means, a nested workflow process may have alternative boxes connected to it by incoming arcs as well as by outgoing arcs, since a refinement of a nested workflow process will be given. However, the refinement of a nested workflow process will belong to a different workflow-process type. *A nested workflow process is executed like a basic workstep.* This means, a nested workflow process can be started if its input port assignment rule is satisfied, and may commit if its output port assignment rule is satisfied. Since a nested workflow process may have a disjunctive postcondition *and* a disjunctive precondition, the input port assignment rule of nested workflow processes extends the input port assignment rule of worksteps. The execution semantics of nested workflow processes is discussed subsequently (cf. specification of nested workflow processes, below).

To draw again an analogy to Petri nets, nested workflow processes correspond to transitions of Petri nets. Different to transitions, however, executions of nested workflow processes take time. This means, from the point of view of a superordinate workflow process a nested workflow process looks like a basic workstep (cf. Subsection 4.4.4). The duty to meet the objectives of a nested workflow process, however, is contracted out to a separate supplier.

Specification of Nested Workflow Processes

The specification of a nested workflow-process models the behavioral part of a has-nested-workflow relationship and its inverse nested-workflow-of relationship between a superordinate workflow process and one or several nested workflow processes of the same type. A workflow process interacts synchronously with its nested workflow processes. The structural part of a has-nested-workflow relationship and its inverse nested-workflow-of relationship is modeled by “structural” semantic relationships (cf. Subsection 8.3.1).

A *specification of a nested workflow process* models the interface of a nested workflow process by a *name*, a set of *input ports*, a set of *output ports*, a *precondition*, a *postcondi-*

tion, and a set of *life-cycles* that determine the set of business-process activities a nested workflow process may invoke on business cases passed to it for processing to meet its objectives. That is, the specification of a nested workflow process is similar to the specification of a workstep. Different from worksteps, however, postconditions *and* preconditions of a nested workflow process may be disjunctive. Furthermore, we assume that a nested workflow process has only formal ports (cf. Subsection 4.4.4). The names of worksteps and nested workflow processes must be unique within a workflow process.

Graphically, the specification of a nested workflow process is shown in a workstep-specification diagram. A nested workflow process is depicted as in a workflow-process diagram (above). The specification of the life-cycles indicating the business-process activities a nested workflow process may invoke on business cases passed to it for processing is marked with the keyword nested workflow for.

Example: Figure 8.15 shows the specification of a nested workflow process HandlePayment of a superordinate room-reservation workflow process. A nested workflow process HandlePayment depends on its superordinate workflow process but occurs in a different organizational context. We assume that the nested workflow process occurs in the context of the finance-management group of a hotel, the superordinate workflow process in the context of the reception of a hotel. Consequently, the nested workflow process is modeled as a role of its superordinate workflow process (cf. the shaded parallelogram that is superimposed on the rectangle representing the nested workflow process).

Nested workflow process HandlePayment may be started on accounts of either cancelled or used room reservations (cf. alternative boxes R-6 and R-9 connected by read arcs to the secondary input port of workflow process HandlePayment). After nested workflow process HandlePayment has been performed on an account it must qualify for box A-2, i.e., the account must be settled. If an account belongs to a cancelled reservation, business-process activity billCancellationFee must be invoked to enter state inPayment. If it belongs to a used reservation, business-process activity billServices must be invoked. If a customer did clear the balance of his account, payment will be accepted and the execution of the nested workflow process may complete on the appropriate account. Alternatively, payment of a reservation can be waived by the finance-management group handling a reservation.

The execution semantics of nested workflow processes is determined by two port assignment rules, which are defined on the basis of the port assignment rules for worksteps (cf. Subsection 4.4.4). The input port assignment rule determines under which conditions a nested workflow process can be started. Since alternative boxes may not be connected to input ports of worksteps, the input port assignment rule for nested workflow processes must be adapted accordingly. The *input port assignment rule* of a nested workflow process n is satisfied if: (1) Every input port p of n is assigned the correct number of values, which is specified by the cardinality of p. (2) Every value assigned to an input port p must belong to the type domain of p. (3) Every object assigned to an input port p must belong to the

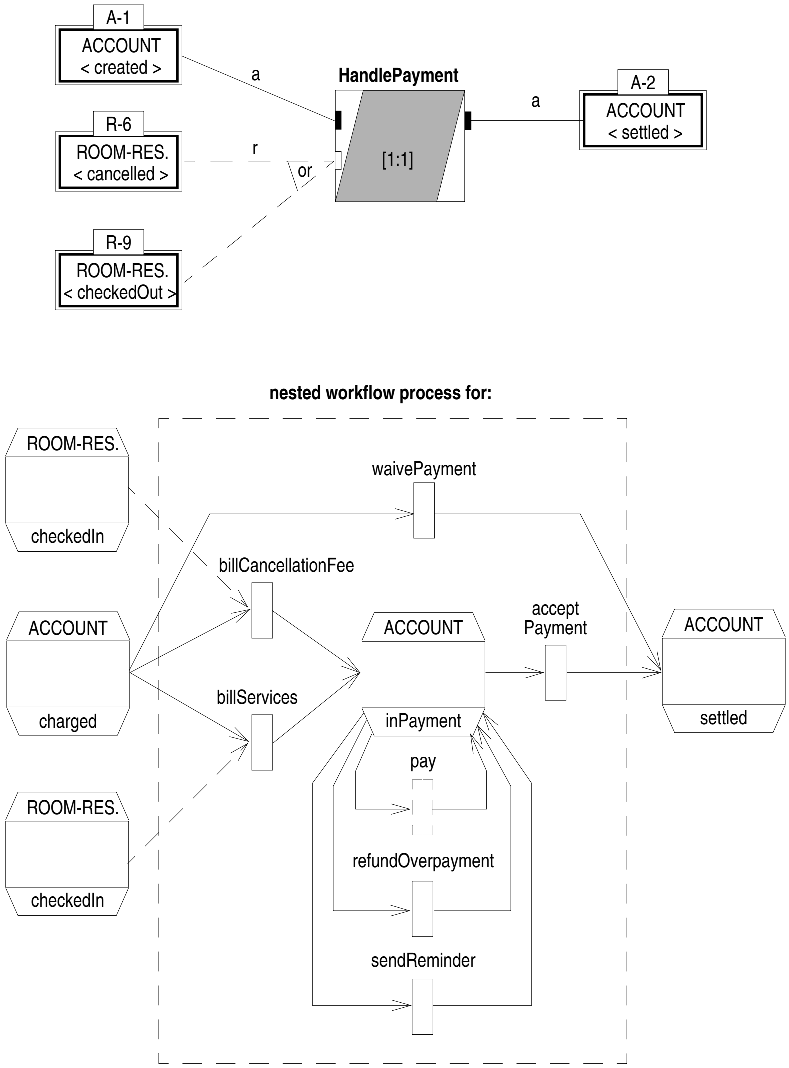


Figure 8.15: Specification of nested workflow process HandlePayment

active domain of p, i.e., it must be member of every box connected to p by incoming arcs. However, if two or more boxes connected to p are disjunctive, a business case must be member of only one of these boxes. (4) The precondition of n is satisfied. (5) Neither nested workflow process n, a nested workflow process n' alternative to n, nor a workstep s alternative to n, is currently active on the business case assigned to the primary port of n[7]. Note: The above input port assignment rule incorporates an optimistic strategy for the consistent execution of nested workflow processes. This means, it aims at achieving a high degree of parallel execution of nested workflow processes and worksteps at the risk that conflicts might arise during the execution of nested workflow processes. More precisely, the above input port assignment rule allows that two nested workflow processes, or a nested workflow process and a workstep may be started although a conflict might arise during their execution with respect to an object or a set of objects assigned to a secondary port of one or both of the two nested workflow processes or of the workstep, respectively. A pessimistic strategy aims at avoiding conflicts as far as possible. To avoid the above kind of conflicts the input port assignment rule could be extended by the following condition (optional): (6) No object assigned to a secondary port of n is also assigned to a port of a workstep s or a nested workflow process n' that is currently active[7], where n' may denote either the same or a different nested workflow process as n. The *output port assignment rule* determines under which conditions a nested workflow process n can complete successfully. The output port assignment rule for nested workflow processes is similar to that for worksteps, which has been defined in Subsection 4.4.4.

To ensure consistency of the execution of nested workflow processes and worksteps of a workflow process, information on the status of their processing is written to a trace associated with that workflow process. If a nested workflow process n is performed on some business case c, several entries are written to the trace that indicate the *start* of n on c, the *assignment of values to secondary ports* of n, and the *abort* or *completion* of n on c. These entries are sufficient to decide whether a particular nested workflow process is active on some business case or not.

As mentioned before, nesting a workflow process into another workflow process models a synchronous interaction between the superordinate workflow process and the nested workflow process. This synchronous interaction is always driven by the superordinate workflow process. It passes business cases for processing to the nested workflow process and accepts processed business cases. The superordinate workflow process may also abort processing of a nested workflow process. Actually, a nested workflow process is implicitly refined into three basic worksteps within a superordinate workflow process into which it is nested. These worksteps allow to start, to commit, and to abort a nested workflow process on a particular business case. As a notational convention, the names of these implicitly defined worksteps are formed by adding the prefixes Start, Commit, and Abort to the name of a nested workflow process. A workstep starting a nested workflow process may be invoked if the input port assignment rule of the nested workflow process is satisfied. A workstep committing a nested workflow process may be invoked if the output port assignment rule of the nested workflow process is satisfied. A workstep aborting a nested workflow process may be invoked if conditions (1) to (3) of the input port assignment rule of a nested workflow process are satisfied and the nested workflow process is still active on the corresponding business case. Note: In this simple execution model of nested workflow processes we require that a nested workflow process either fulfills its obligation or returns business cases passed to it for processing in the same condition in which it received them.

Realization of Nested Workflow Processes

A nested workflow process is realized by another *workflow process* that is defined at a separate workflow-process type. The workflow process defined at such a workflow-process

[7] Whether a nested workflow process is active on some business case or not, can be determined based on information on the execution of nested workflow processes and worksteps that is written to traces associated with workflow processes (below).

type can be developed by stepwise refinements of an abstract workstep that corresponds to the specification of the nested workflow process in its superordinate workflow process. This means, the abstract workstep from which the nested workflow process is developed "inherits" all boxes connected to the specification of the nested workflow process in the superordinate workflow process. These boxes may be specialized in the nested workflow process, too. Note: Since preconditions of worksteps may not be disjunctive, a nested workflow process must offer alternative source worksteps if disjunctive boxes have been connected to an input port of the nested workflow process in the superordinate workflow process.

Example: Figure 8.16 shows the initial workstep from which the nested account workflow process will be developed. Workstep HandlePayment is declared to be abstract. Since boxes R-6 and R-9 connected to the secondary port of workstep HandlePayment are disjunctive boxes (cf. specification of nested workflow process HandlePayment in Figure 8.15), there must be two alternative source worksteps in the refinement of workstep HandlePayment. These alternative worksteps are workstep BillCancellationFee and workstep BillServices shown in Figure 8.17.

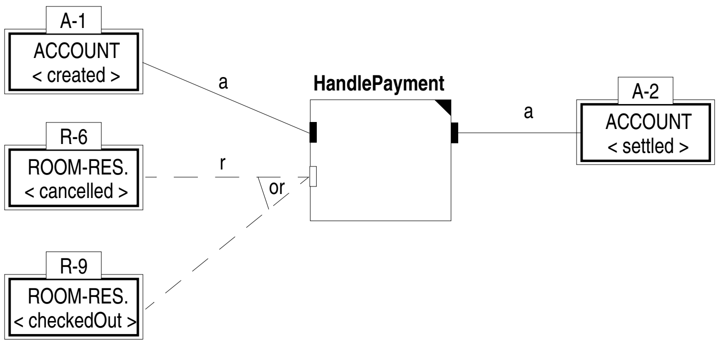


Figure 8.16: Workflow process at ACCOUNT-WF

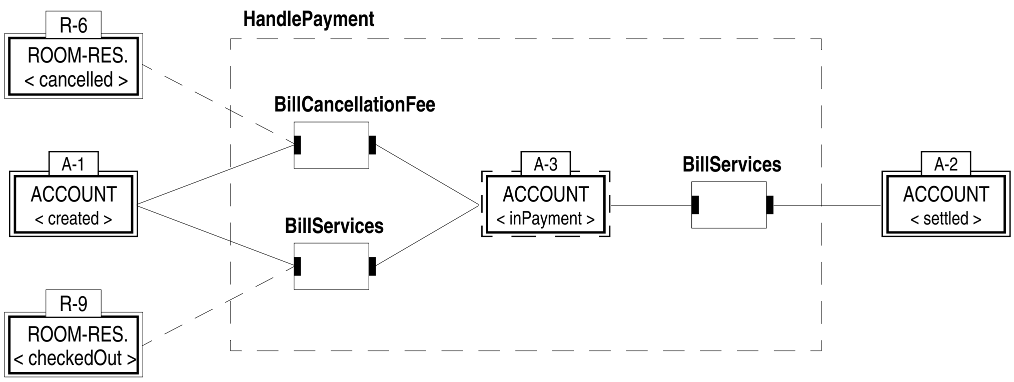


Figure 8.17: Refinement of workstep HandlePayment at ACCOUNT-WF

The specification of a nested workflow process given in a superordinate workflow process has influence on the structure of workflow objects that model a nested workflow process satisfying that specification. A workflow object that models a nested workflow process must have one has-full-member relationship for each type of business objects on which it directly invokes business-process activities, and one has-affiliate-member relationship for each other type of business objects passed to it. The life-cycle constraints specified on these has-full-member relationships and has-affiliate-member relationships must be sub life-cycles of the life-cycle(s) given in the specification of the nested workflow process. Moreover, a workflow object that models a nested workflow process must have a nested-workflow-of relationship of the "right" semantic to the workflow object that models the superordinate workflow process.

Example: Instances of workflow-process type ACCOUNT-WF (cf. Figure 8.8) model workflow processes that are nested into room-reservation workflow processes as nested workflow processes called HandlePayment (cf. Figure 8.14). Each account workflow-process has a has-full-member relationship accounts to accounts passed to it for processing, a has-affiliate-member relationship reservations, and a role-of relationship nestedWorkflowOf to its superordinate room-reservations workflow process.

Specialization of Workflow Processes

If workflow-process types are organized in type hierarchies, a subtype may not only specialize the structure and behavior it inherits from its supertype (cf. Subsections 8.3.1 and 8.3.2, respectively), but it may also specialize the workflow process, worksteps, and nested workflow processes it inherits from its supertype. In fact, each user-defined workflow-process type is a (direct or indirect) subtype of the predefined conceptual type WFP_OBJECT (WorkFlow-Process OBJECT, cf. Figure 7.6), which models a basic workflow process that each workflow object offers to its environment. The workflow process defined at conceptual type WFP_OBJECT is shown in Figure 8.18(a). It consists of one abstract box that collects business process-objects in state exists[8]. Subtypes of conceptual type WFP_OBJECT may specialize the inherited workflow process. If a workflow process creates and/or destroys business cases, a subtype of WFP_OBJECT must extend the inherited workflow process accordingly (cf. Figures 8.18(b) and (c)).

Specialization of worksteps and nested workflow processes corresponds to the specialization of activities (cf. Subsection 7.2.2). Similar to specifications of activities, specifications of worksteps and nested workflow processes may be specialized in a *covariant* way. This means, the domains of their input and output ports may be restricted, their preconditions and their postconditions may be strengthened, and the life-cycles given as constraints to indicate which activities they may invoke may be specialized (cf. Subsection 7.2.2) when worksteps or nested workflow processes are redefined at a subtype. The type domain of a

[8]Note: We assume that business processes are developed by stepwise refinements and extensions from a life-cycle that consists of an abstract create activity, an abstract state—named exists—, and an abstract destroy activity (cf. Figure 7.13).

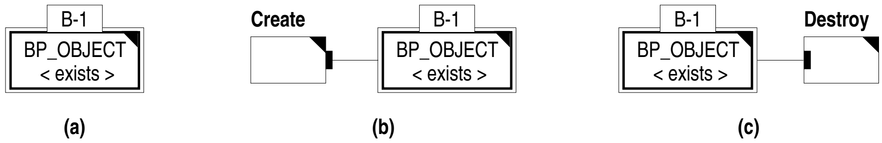


Figure 8.18: Workflow process at WFP_OBJECT

port may be restricted to a subtype of the inherited one, the active domain of a port to a box that specializes[9] the inherited one. Furthermore, the workflow-process type given as domain of an inherited nested workflow process may be restricted to a subtype.

Example: Figure 8.19(a) shows the specification of workstep Archive at RESERVATION-WF. Workstep Archive may be invoked on reservations that qualify for box R-10, i.e., reservations that reside in state finished. After workstep Archive did complete, the reservation it processed must qualify for box R-11, i.e., the reservation must reside in state archived.

Figure 8.19(b) shows the redefinition of workstep Archive at ROOM-RESERVATION-WF. Room reservations for which an account has been created, i.e., an account will be created when a customer confirms her or his reservation (cf. Figure 8.2), must be archived together with that account. Note: Port a added at type ROOM-RESERVATION-WF is an optional, single-valued port.

Specialization of workflow processes corresponds somehow to the specialization of life-cycles (cf. Subsection 7.2.2). This means, a workflow process may be specialized by *extension*, i.e., boxes, worksteps, and nested workflow processes added at a subtype extend the inherited workflow process, or by *refinement*, i.e., boxes, worksteps, and nested workflow processes added at a subtype refine inherited boxes and worksteps. In Subsection 7.2.2, we discussed different notions of behavior consistency for specializations of life-cycles, i.e., observation consistency, weak invocation consistency, and strong invocation consistency. Analogous to their definitions for specializations of life-cycles, these notions of behavior consistency could also be defined for specializations of workflow processes. However, we will not elaborate on these notions of behavior consistency for specializations of workflow processes, since, opposite to specializations of life-cycles, substitutability is not *the* issue for specializations of workflow processes. Instead, we will focus on the correspondence between a specialization of a workflow process and the underlying business process when testing the consistency of a specialization of a workflow process. Since an appropriate notion of consistency of workflow processes has already been given by the *Rule of Business-Process Consistency of Workflow Processes* in Subsection 4.4.4, subsequently, we will discuss different situations in which a workflow process may or must be specialized.

[9] A box b_2 specializes another box b_1, if the member type, the member state, and the boolean condition of b_2 are either identical to or restrict that of b_1, and either the member type, the member state, or the boolean condition of b_2 restricts that of b_1.

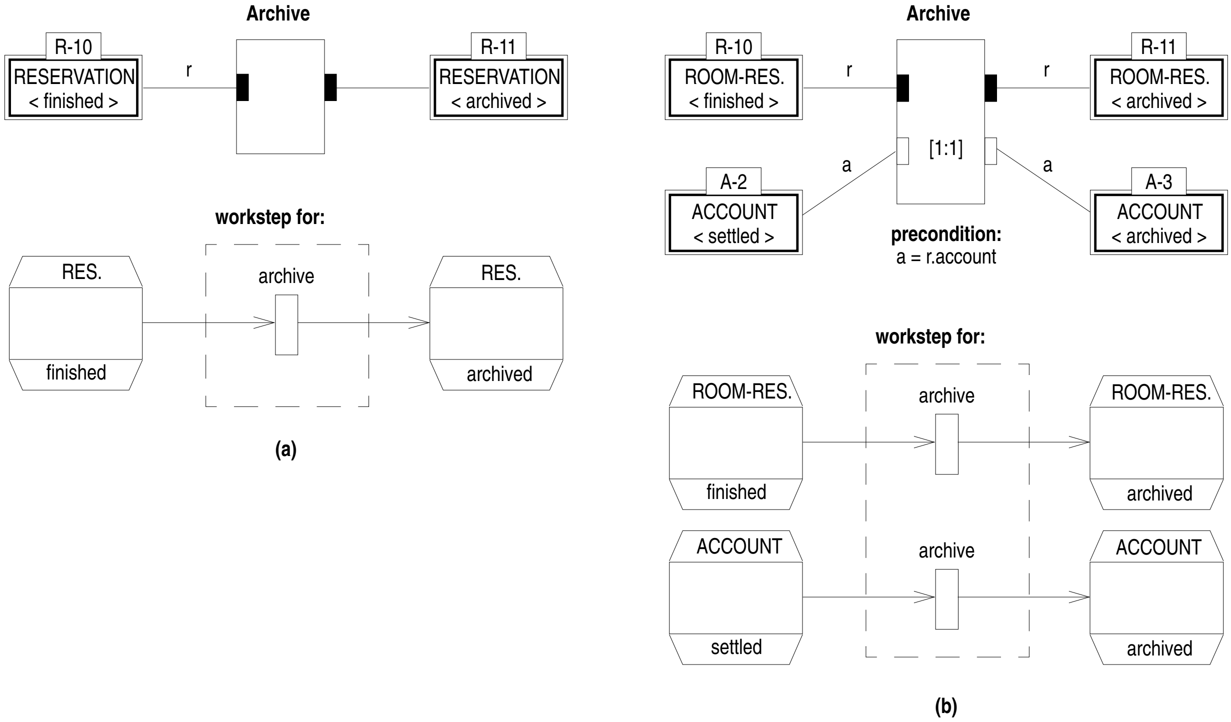


Figure 8.19: Specifications of workstep Archive

For the sake of simplicity of the following discussion, we assume that (a) a workflow process has only one type of members, i.e., only one has-full-member relationship is defined for a workflow object, (b) a separate workstep is defined for each business-process activity considered in a workflow process, (c) the "underlying business process" is defined by the life-cycle given as constraint on the has-full-member relationship of a workflow object, and (d) only workflow process that satisfy the *Rule of Business-Process Consistency of Workflow Processes* (cf. Subsection 4.4.4) will be specialized.

First, we will discuss how a workflow process w' may specialize a workflow process w if both w and w' build on the *same* business process b. Since w' inherits from w, it is already consistent wrt. the underlying business process b. This means, w' *need not* extend w and *must not* refine w. However, w' may extend w in that it considers "alternatives" specified at b that have not been considered at w. This means, besides offering a richer set of alternatives to meet its objectives, workflow process w' allows to handle business cases in the same way as w. Note: If assumption (b) (above) would be relaxed, i.e., if worksteps may be defined for a set of business-process activities, workflow process w' could refine inherited workstep that have been defined for more than one business-process activity.

Second, we will discuss how a workflow process w' may or has to specialize a workflow process w if w' builds on a business process b' that is a specialization of a business process b on which w builds. Business process b' may extends and refine b. If b' extends b according to *observation consistency* (cf. Subsection 7.2.2 and Figure 7.16(a)), w' must extend w in that it has to add worksteps and boxes for "parallel branches" that have been added at b' (cf. workstep A_4 and associated boxes in Figure 8.20(a)). This means, some of the inherited worksteps must be connected to added boxes by outgoing arcs (cf. workstep A_1 in Figure 8.20(a)) *and* some by incoming arcs (cf. workstep A_3 in Figure 8.20(a)). If b' extends b according to *weak invocation consistency* (cf. Subsection 7.2.2 and Figure 7.16(b)), w' must extend w in that it has to add worksteps and boxes for "open parallel branches" that have been added at b' (cf. workstep A_4 and associated boxes in Figure 8.20(b)). This means, some inherited worksteps must be connected to added boxes by outgoing arcs (cf. workstep A_1 in Figure 8.20(b)), but no inherited workstep must be connected to added boxes by incoming arcs, since an invocation consistent specialization of the underlying business process ensures that no additional prestates are defined for inherited activities at a subtype. In addition, new "alternative branches" added at b' may be supported by w' in that appropriate worksteps and boxes may be added at w'. This means, added worksteps may be connected to inherited boxes by incoming arcs and by outgoing arcs (cf. workstep A_5 in Figure 8.20(a)). If b' extends b according to *strong invocation consistency* (cf. Subsection 7.2.2 and Figure 7.16(c)), w' must extend w in that it has to add worksteps and boxes for "open parallel branches" that have been added at b'. This means, inherited worksteps must be added in the same way as if b' would specialize b according to weak invocation consistency (above). Since b' may not extend b by new "alternative branches", b' does not cause further extensions to w (cf. Figure 8.20(c)). In all cases however, w' may extend w in that it adds support for "alternative branches" that have already been defined at b, but have not been supported by w.

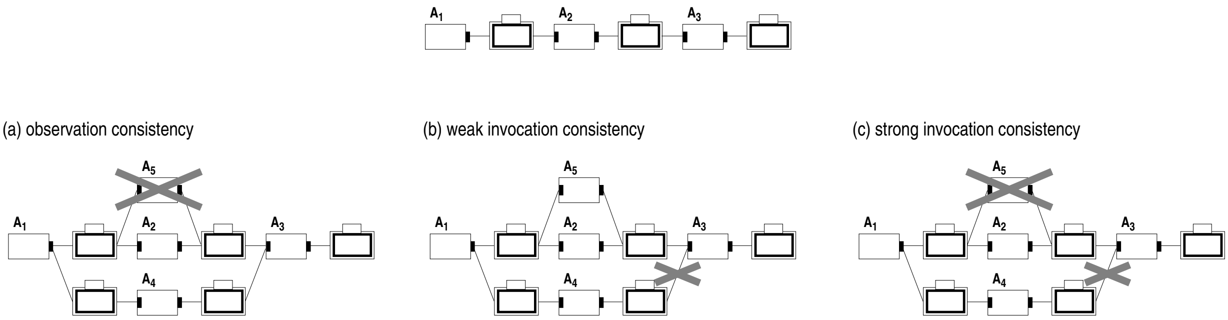


Figure 8.20: Consistent extensions of workflow processes

If b' refines b, w' must also refine inherited worksteps and boxes that have been defined for activities and states of b that have been refined. The refinements of worksteps and boxes must be consistent wrt. the refinements of the underlying activities and states. This means, the consistency of refinements of worksteps and boxes can be tested without considering the overall workflow process and the overall business process.

Third, we will discuss under which conditions a workflow process could handle business cases of different types sharing a common supertype. Assume a workflow process w has been defined for a business process b that has been defined at a business-process type being a supertype. Since supertypes are abstract (cf. *Abstract Supertype Rule for Object Types*, Subsection 7.2.1), all business cases handled by an instance of w will be indirect instances of that type B at which b has been defined and b could be specialized at each subtype of B. If B', a subtype of B, extends b to b' according to observation consistency, some worksteps defined at w may fail although features added at b' are ignored, since b' might define additional prestates for inherited activities (cf. Subsection 7.2.2). If B', a subtype of B, extends b to b' according to invocation consistency, all worksteps defined at w can handle instances of B' without problems if features added at b' are ignored, since b' cannot define additional prestates for inherited activities (cf. Subsection 7.2.2). However, workflow process w could not simply ignore features added at a business process that specializes b. An important criterion for workflow processes is the consistency with the underlying business process (cf. *Rule of Business-Process Consistency of Workflow Processes*, Subsection 4.4.4). If instances of workflow process w should handle business cases of different subtypes of B, it must be consistent wrt. all business processes defined at those different subtypes of B. In short, if a business process b' extends b by "parallel branches", workflow process w is inconsistent wrt. to b', since no workstep in workflow process w defined on business process b will be connected to a box that collects business cases in some life-cycle state that may be added at some business process b' extending b (cf. condition (2) of the *Rule of Business-Process Consistency of Workflow Processes*, Subsection 4.4.4). Extensions to workflow process w are required in such situations.

If B', a subtype of B, defines business process b' that refines activities and states it inherits from b, all those worksteps of w that have been defined for activities refined at B' will fail if invoked on instances of B', since no realizations are given for abstract activities. In short, if a business process b' refines activities and states it inherits from b, workflow process w is inconsistent wrt. to b', since it would have to offer worksteps and boxes for new activities and new states, respectively, in the refinements of inherited activities and states to allow to handle business cases passed to it for processing such that they finally reside in final states of the underlying business process (cf. *Rule of Business-Process Consistency of Workflow Processes*, Subsection 4.4.4). Refinements of worksteps and boxes of workflow process w are required in such situations.

Since different subtypes may specialize a business process in different ways, workflow processes must support means to offer different specifications and different realizations of worksteps and nested workflow processes if different types of business cases should be handled by a workflow process. This means, worksteps and nested workflow processes must be polymorph with respect to workflow-process types *and* business-process types. Without extension to the existing model, we could offer different specifications and different realizations of worksteps and nested workflow process only if we would simulate *ad-hoc polymorphism* (e.g., [FE95]). This means, we would bind different specifications and realizations of logically the same workstep or nested workflow process to *different* names and would specialize boxes, which are connected by incoming arcs to worksteps that offer different specifications and realizations, according to different subtypes if individual specifications and realizations are offered for these subtypes. However, as Firesmith and Eykholt state [FE95, p. 333]: "This form of polymorphism is neither maintainable, extensible, nor object oriented." More desirable would be a system-maintained polymorphism, called *inherent polymorphism*, which automatically selects the right specification and realization of a workstep or nested workflow process invoked on a certain business case o in a certain workflow process w depending on the type of w *and* the type of o (*double dispatching*). That is, different specifications and realizations could be bound to the *same* name, one specification and one realization for each type of business case handled by workflow processes of a particular type. To properly identify specifications and realizations of worksteps and nested workflow processes defined at a workflow process of type W for a particular type of business cases B processed by instances of W would require to add the name of type B to the names of appropriate worksteps and nested workflow processes. Nevertheless, an appropriate extension of the workflow-process model goes beyond the scope of this thesis and is left for future work (cf. Chapter 11).

8.4 Summary

In this chapter we presented advanced concepts for the modeling of organizational processes. We started with a management definition of the fundamental concepts that were presented in this chapter. Whereas business processes can be modeled relying solely on advanced concepts presented in Chapter 7 (e.g., semantic relationships, specialization, behavior synchronization), additional concepts were presented for the modeling of workflow processes, i.e., *nested workflow processes* and *specialization of workflow processes*. The advanced modeling concepts presented in this chapter allow designers to follow a modern philosophy when modeling organizational processes. This design philosophy requires that large systems are decomposed into a set of smaller parts that are easier to understand, maintain, and reuse. For example, a workflow process can consists of a set of nested workflow processes. To each of its nested workflow processes, a workflow process delegates part of its responsibility to handle business processes. That is, nested workflow processes are suppliers of workflow processes into which they are embedded. Specialization of workflow processes and specialization of business processes allows for an incremental modeling of organizational processes and for a disciplined sharing and modification of knowledge on organizational processes. To ensure consistency between workflow processes and corresponding business processes that are designed incrementally, we presented criteria for a consistent specialization of workflow processes wrt. the underlying business process.

Chapter 9

Organization Modeling

Contents

Traditionally, organizations are hierarchical systems [Daf95, GE96], i.e., each actor has a supervisor and each organizational units has a superordinate organizational unit. Recently, with the shift to process-oriented thinking in business modeling (e.g., [HC93]), there is a trend organize the structures of organizations around organizational processes. This kind of organization structure is called *network* or *horizontal organization* (e.g., [Daf95]). Consequently, workflow management systems (WFMSs) that claim to increase the productivity of organizations in that they enact their processes should support the modeling of network-like organizations. In this chapter we discuss the modeling of networks of actors and organizational units of an organization. Moreover, we discuss how *roles* (cf. Subsection 7.2.1) can be used to model actors and positions adequately.

This chapter is organized as follows. Section 9.1 gives a management definition of the fundamental modeling concepts presented in this chapter. Section 9.2 presents concepts for the conceptual modeling of organization schemas. The focus is on the modeling of organizational units with a network control structure and on the use of roles to model actors and positions. Section 9.3 summarizes this chapter.

9.1 Fundamental Modeling Concepts

The core concepts discussed in this chapter are *roles* and *network*. Roles (cf. Subsection 7.2.1) allow to represent one real-world entity by a set of objects that may occur in different contexts. We will employ roles to model actors and positions. Network is a form of a control structure of organizations that is alternative to hierarchy (cf. Chapter 5). If an organization installs a hierarchical control structure in a particular organizational unit, all managerial tasks, e.g., coordination of the activities performed by actors of that organizational unit, are within the responsibility of a single actor called the *supervisor* of that organizational unit. If an organization installs a network control structure in a particular organizational unit, managerial tasks are within the responsibility of all actors of that organizational unit in common.

The fundamental organization modeling concepts can be distinguished into two categories: *organizational objects* and *organizational relationships*. Basic definitions of fundamental organization modeling concepts have already been given in Section 5.1.

Organizational objects. In Chapter 5, organizational objects have been distinguished into *agents*, *positions*, *organizational units*, and *resources* (cf. Section 5.1). In this section, we add a new category of organizational objects, roles. *Roles* are used to model actors and positions. Furthermore, we re-define the concepts agent and position.

Role: A *role* is an object that models the *same* real-world entity as another object, called its *player*, but in a different context. All objects that represent the same real-world entity are organized in a hierarchy, called role hierarchy. We focus on two kinds of roles in this chapter: *individual roles* and *organizational roles*. Sets of individual roles model an actor, sets of organizational roles model positions. The root of a role hierarchy containing individual roles is an agent object, that of a role hierarchy containing organizational roles a position object. Note: Esswein (cf. [Ess93]) discussed the use of roles for the modeling of positions.

Agent: An *agent* is a model of an actor (cf. Section 5.1). In general, each actor is represented by one agent object and by several individual-role objects (cf. roles, above), which are organized into a role hierarchy. The agent object is the root of such a role hierarchy.

Position: A *position* is a model of a kind of "abstract actor" (cf. Section 5.1). An abstract actor is represented by one position object and by several organizational-role objects (cf. roles, below), which are organized into a role hierarchy. The position object is the root of such a role hierarchy.

Organizational relationships. Organizational relationships determine the structure of an organization. In this chapter, we will discuss the modeling of the *physical structure* of organizations in the realm of semantic relationships (cf. Subsection 7.2.1). Furthermore, we will discuss *network* as a structural means to coordinate activities performed by different actors of an organization. The network control structure is alternative to the hierarchical control structure presented in Chapter 5.

Physical structure: The *physical structure* of an organization determines how *positions are grouped* together into organizational units and how *organizational units are grouped* together forming the overall organization (cf. Section 5.1). Positions (organizational units) that belong to an organizational unit are either its components or its constituents. The advanced conceptual model knows different types of semantic relationships to model such situations adequately (cf. Subsection 7.2.1).

Control structure: Organizations install a control structure among its actors (organizational units) where selected actors (organizational units) are designated to coordinate the activities of others. *Network* is such a control structure. If a network control structure is used to coordinate the activities performed by different actors (organizational units) within an organizational unit, managerial tasks, e.g., coordination, are within the responsibility of all of the actors (organizational units) of that organizational unit in common.

Organization chart: *Organization charts* are a tool that may be used to visualize *different* dimensions of the structure of an organization. We distinguish organization charts into *control-structure diagrams*, which depict hierarchical and network control structures of organizational units, and *part-whole diagrams*, which show the physical structure of organizations.

Substitution: A *substitution relationship* between two actors determines that the substitute actor stands in for the regular actor in cases where the regular actor is absent. To produce the services and goods an organization offers to its customers, an organization depends on the *availability* of its actors. Basically, two problems must be solved to guarantee the availability of actors: First, an appropriate number of *regular actors* must be provided. Second, a *substitution plan* must be established to guarantee availability in situations in which regular actors are not available, e.g., human actors might be on vacation.

Summing up, we emphasize that roles increase the flexibility of organization-schema design tools. The concept of roles allows to adequately model situations in which actors occur in different contexts within an organization, having different data requirements, different rank, and different authorizations in different contexts. Although an actor may be represented by several objects, the information that these objects represent the same real-world entity is not lost in a model of an organization if the concept of roles is used.

The control structure of an organization models a chain of command within an organization. The shift to process-oriented thinking in business modeling (e.g., [HC93]) caused a shift from hierarchical to network control structures within organizational units. Typical for network control structures is that the managerial tasks of an organizational unit are within the responsibility of all its actors (organizational units) in common and that they tend to flatten the control structures of organizations. The control structure of an organization is a logical dimension of the structure of an organization. *Substitution* is another logical dimension of the structure of an organization. Substitution relationships among actors guarantee their availability.

9.2 Design of Organization Schemas

An *organization schema* consists of a set of organizational-object types and a set of organizational-object classes. Each organizational-object type defines the structure and behavior of its instances, called *organizational objects*, e.g., agents, positions, roles, organizational units, or resources. An organizational-object class collects a set of organizational objects of a specific type, the member type of the class.

Organizational objects are business objects, too. This means, the concepts presented in Section 7.2 for the conceptual modeling of business objects can be applied in the same way to the modeling of organizational objects. In this section, these concepts are reviewed in the realm of conceptual modeling of organizational objects, e.g., actors and positions, which are both evolving entities, are modeled using the concept of roles, and new concepts specific to the modeling of organizational objects are introduced, e.g., the network control structure of organizational units.

The remainder of this section is organized as follows. Subsection 9.2.1 discusses the modeling of the structure of organizational objects. The emphasize is on modeling the control structure of self-managed organizational units and on the flexible modeling of actors and positions of an organization. Subsection 9.2.2 reviews the modeling of the behavior of organizational objects and Subsection 9.2.3 the modeling of classes of organizational objects.

9.2.1 Modeling the Structure of Organizational Objects

The structure of organizational objects is determined by a set of *attributes* and a set of *relationships* (cf. Subsection 5.2.1). In Subsection 7.2.1 advanced concepts of structure modeling, i.e., semantic relationships and type specialization, have been discussed. These concepts of structure modeling apply to the modeling of the structure of organizational objects in the same way. Below, we shortly review these concepts in the context of organization modeling.

Organizational Objects

We distinguish organizational objects into *agents*, *positions*, *roles*, *organizational units*, and *resources* (cf. Section 5.1). Although the concept of roles is a general modeling concept to represent evolving entities from a business area, we will focus on the use of roles to model actors and positions within an organization in this chapter. We offer several predefined conceptual types of organizational objects (cf. Figure 9.1), which a designer specializes when developing a schema of a particular organization (cf. specialization of the structure of organizational objects, below). Note: Figure 9.1 shows meta information incorporated in a conceptual-type hierarchy. To avoid confusion with regular diagrams depicted using the graphical notation of Object/Behavior Diagrams, we borrowed the graphical notation from OMT (cf. [RBPE91]) to depict the conceptual-type hierarchy and start names of relationships with a capital letter.

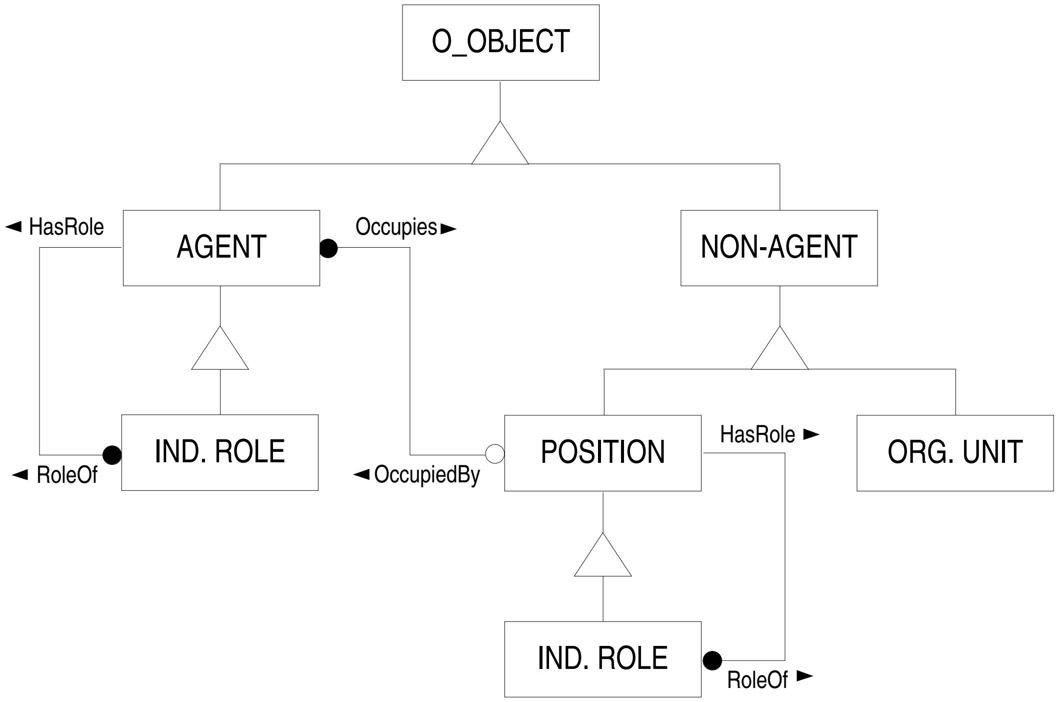


Figure 9.1: Conceptual types of organizational objects

Agents. An *agent* is a model of an actor (cf. Section 5.1). To cope with the evolving nature of actors[1], we allow to model each actor by a set of objects that are organized in a role hierarchy (cf. Subsection 7.2.1). We refer to the root of such a role hierarchy as agent. It may acquire and abandon roles. That is, the set of objects that represents an actor changes during the lifetime of the actor. We refer to roles that represent an actor as *individual roles* (cf. roles, below).

Example: Figure 9.2(a) shows a type diagram of object type PERSON. A person is identified by the values of its key attributes name and birthday. Furthermore, a person has a general relationship address to a value from domain ADDRESS. Persons are stored in a "global" class, i.e., the domain of the component-of relationship defined for persons is object type DATABASE, and are simple information objects, not agents, i.e., object type PERSON is a direct subtype of the predefined type B_OBJECT (cf. Subsection 7.2.1). However, a person may occur in several roles that are agents. That is, a person may occur as employee of a hotel (cf. has-role relationship asEmployee) and as guest of several hotels (cf. has-role relationship asGuests).

Figure 9.2(b) shows a type diagram of agent type EMPLOYEE. An employee is identified by the value of key attribute ssn, i.e., the employee's social security number,

[1]Note: Training humans extends their skills and allows to enrich their positions with new authority and responsibility. Research in artificial intelligence aims at making software agents "intelligent", having software agents to extend their skills over time, too.

and is further described by the value of attribute hireDate. Employees are agents (cf. subtypeOf relationship), are direct components of hotels (cf. component-of relationship), and are roles of persons (cf. role-of relationship). Moreover, an employee may occur in different roles within a hotel. An employee may occur several times as manager of some organizational unit of a hotel (cf. has-role relationship asManagers), as secretary (cf. has-role relationship asSecretary), as assistant (cf. has-role relationship asAssistant), as receptionist (cf. has-role relationship asReceptionist), or as workflow administrator (cf. has-role relationship asWfAdmin).

Figure 9.2(c) shows a type diagram of agent type GUEST. A guest is described by the values of attributes creditworthiness and specialNeeds. Guests are agents (cf. subtypeOf relationship), are direct components of hotels (cf. component-of relationship), and are roles of persons (cf. role-of relationship). Note: Attribute specialNeeds is multi-valued.

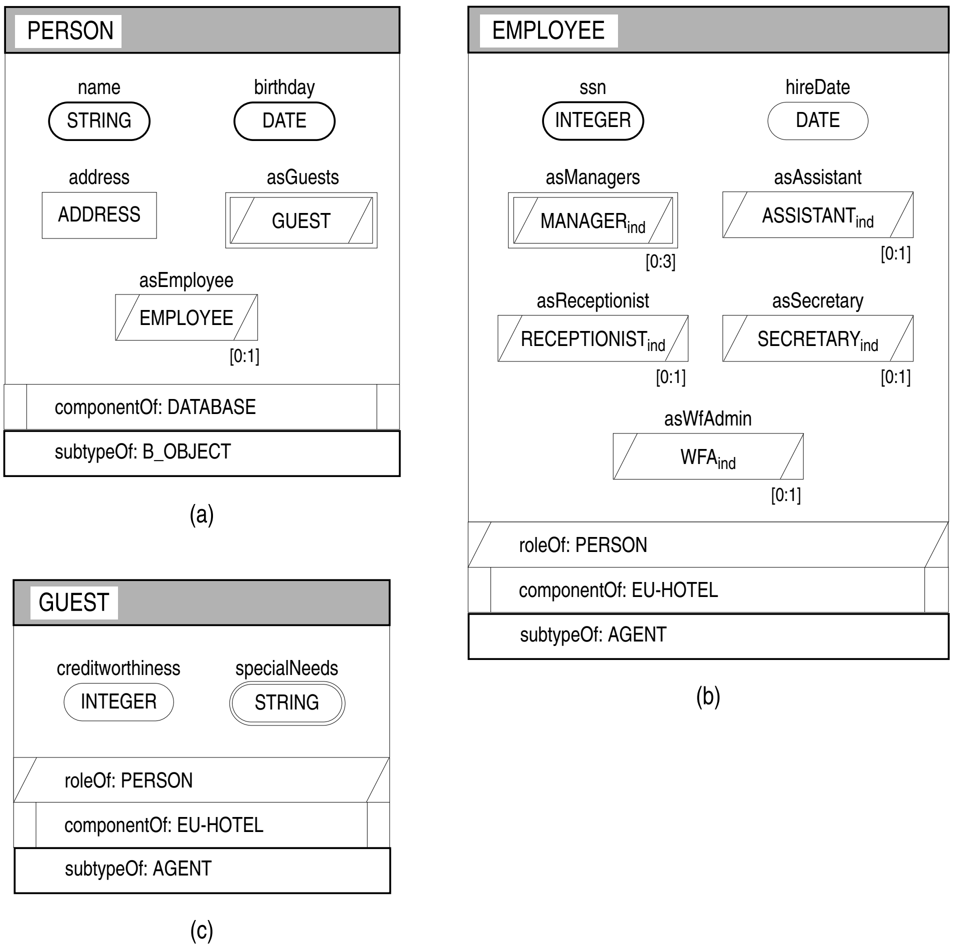


Figure 9.2: Object type PERSON as well as agent types EMPLOYEE and GUEST

Positions. A *position* models a kind of "abstract actor" of an organization (cf. Section 5.1). A position determines the authority and responsibility of an actor within an organization. The authority and responsibility of a specific actor is determined in that the actor is assigned to a particular position. In addition, a position determines the organizational unit within which an actor has to fulfill its responsibilities, determines the rank of the actor within that organizational unit, and holds values of properties that are shared by all individual actors occupying that position at different times. We allow to model each position by a set of objects that are organized in a role hierarchy (cf. Subsection 7.2.1). We refer to the root of such a role hierarchy as position. It may acquire and abandon roles. That is, the set of objects that constitute a position may change during the lifetime of the position. We refer to roles that represent a position as *organizational roles* (cf. roles, below). Note: In [Ess93], Esswein did present an organization model that also uses roles to represent positions.

Using roles to model positions increases the flexibility of organization models both at design-time as well as at run-time. A designer can model a position as a set of easy to understand objects, i.e., roles that capture a specific aspect of a position, thereby reducing the overall complexity of modeling positions. Furthermore, the definitions of roles can be reused at different types of positions if these types are organized into a type hierarchy. Having positions acquire new roles or abandon existing ones allows to cope with evolution of positions. "Job enrichment" and "job enlargement" are buzzwords frequently found in the literature to describe such changes of positions (cf. [Daf95, LARB^{+}91]).

Example: Figure 9.3 shows a type diagram of position type $\mathsf{MANAGER}_{\mathsf{org}}$[2]. Each manager position is described by the values of attributes phoneNo, i.e., the number of a manager's office phone, and email. This means, there are electronic-mail addresses associated with manager positions that are independent from individual agents occupying that position (cf. organizational roles, below). A manager position belongs to a hierarchical group of a hotel (cf. component-of relationship) and may be occupied by an employee playing an individual role of type $\mathsf{MANAGER}_{\mathsf{ind}}$[3] (cf. relationship occupiedBy).

Roles. A *role* is an object that represents another object, called its *player*, in a more specific context. Both objects, the role as well as the player, represent the same real-world entity but represent it in different contexts (cf. Subsection 7.2.1). We focus on two kinds of roles in this section: organizational roles and individual roles. Sets of organizational roles form positions, sets of individual roles form agents.

An *organizational role* represents specific aspects of a position. In fact, *organizational roles are non-agents, too*, i.e., the predefined conceptual type ORGANIZATIONAL_ROLE is a subtype of the conceptual type POSITION (cf. Figure 9.1). In fact, an organizational role determines part of the authority and responsibility of an agent occupying a position, the organizational unit within which an agent occupying that organizational role has to fulfill its responsibility, and the rank of the agent within that organizational unit. If

[2]Note: The suffix *org* indicates that instances of type $\mathsf{MANAGER}_{\mathsf{org}}$ are organizational roles or positions.
[3]Note: The suffix *ind* indicates that instances of type $\mathsf{MANAGER}_{\mathsf{ind}}$ are individual roles or agents.

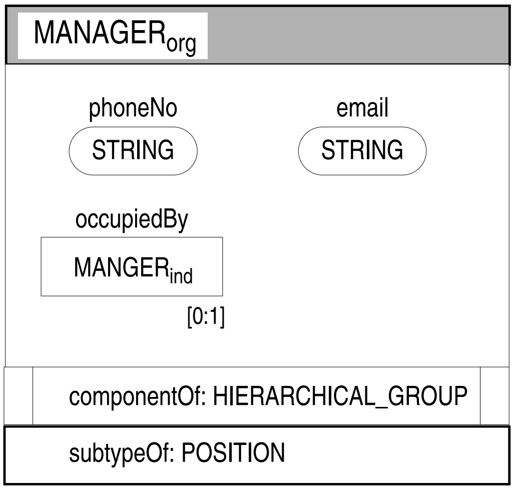


Figure 9.3: Position type MANAGER$_{org}$

an organizational role occurs in the same organizational unit as its player, i.e., another organizational role or position, the rank of the agent occupying that organizational role is fully determined by its player. However, if an organizational role occurs in a different organizational unit as its player, the organizational role must determine the rank of the agent occupying that organizational role within that organizational unit. Furthermore, if all agents that occupy a certain organizational role at different times share the same property and do share the same value for that property, the property is associated with the organizational role.

Example: The head of the finance group of the hotel "Five Wings" is reachable via the electronic-mail address head@finance.fiveWings.com—so were her or his predecessors. This means, the given electronic-mail address is a property of the organizational role or position that represents the head of the finance group of a hotel (cf. position type MANAGER$_{org}$ in Figure 9.3).

The player of an organizational role is either a position or another organizational role. In fact, a real-world position is represented by one position object and by several organizational-role objects organized in a role hierarchy—the position object is the root of such a role hierarchy—, where each organizational-role object represents the same real-world position as the position object or the organizational-role object that plays it. As with positions, each organizational role may be occupied by different individual roles over time. The individual role occupying an organizational role at a particular time must represent the same real-world actor as that individual role or agent occupying the player of the organizational role. Note: When no individual role (agent) occupies an organization role (a position), that organizational role (position) cannot be acted.

As mentioned before, a role occurs either in the same or in a more specific context then its player. This means, organizational roles that represent the same position might belong to different organizational units. In that organizational roles cross organizational boundaries they help to increase the *horizontal linkage* within an organization. In organization theory (e.g., [Daf95]), organizational roles occupied by an agent from a different organizational unit are called *liaison roles* and are primarily introduced to improve communication and coordination across boundaries of organizational units. Other mechanisms to increase the horizontal linkage between organizational units, like *task forces* or *non-permanent teams*, are indeed groups of agents from different organizational units. Again organizational roles can be used to represent agents in different organizational contexts.

An *individual role* represents individual aspects of an agent. In fact, *individual roles are agents, too*, i.e., the predefined conceptual type INDIVIDUAL_ROLE is a subtype of the conceptual type AGENT (cf. Figure 9.1). More precisely, an individual role represents individual aspects of an agent that are specific for the organizational unit in which the agent plays that individual role. Note: In general, individual roles are associated with organizational roles, i.e., an individual role *occupies* an organizational role, where each individual role must belong to the same organizational unit as the organizational role it occupies. However, individual roles may exist independently from organizational roles, too.

Example: The heads of organizational units of the hotel "Five Wings" have personal electronic-mail addresses. For example, the head of the finance group, i.e., Mrs. Joan Miller, is reachable via the personal electronic-mail address joan.miller@finance.five-Wings.com—of course, her predecessors were reachable via different personal electronic-mail addresses. That means, the personal electronic-mail address is a property of an actor occupying a position of the head of an organizational unit (cf. individual-role type $\mathsf{MANAGER}_{\mathsf{ind}}$ in Figure 9.4).

The player of an individual role is either an agent or another individual role. In fact, a real-world actor is represented by an agent object and by a set of individual-role objects organized in a role hierarchy—the agent object is the root of such a role hierarchy—, where each individual-role object represents the same real-world actor as the agent object or the individual-role object that plays it. That is, opposite to the basic organization model where each real-world actor was represented by exactly one agent object (cf. Chapter 5), the advanced organization model represents a real-world actor by an extra agent object and an extra individual-role object for every position and every organizational role, respectively, the actor occupies. As a consequence, the advanced organization model allows to adequately model situations in which properties and authorizations of individual actors depend on both on the positions or organizational roles these actors occupy *and* on the individual actors themselves.

Example: Figure 9.4 shows a type diagram of individual-role type $\mathsf{MANAGER}_{\mathsf{ind}}$[4]. Each individual manager role is described by the values of attributes phoneNo, i.e., the number of the portable phone of a manager, and email. This means, agents that occupy a manager position have a personal electronic-mail address (cf. example,

[4]Note: The suffix *ind* indicates that instances of type $\mathsf{MANAGER}_{\mathsf{ind}}$ are individual roles or agents.

above). An individual manager role belongs to a hierarchical group of a hotel (cf. component-of relationship), occupies a manager position of type $\mathsf{MANAGER_{org}}$ (cf. relationship occupies), and is played by an agent of type EMPLOYEE (cf. Figure 9.2).

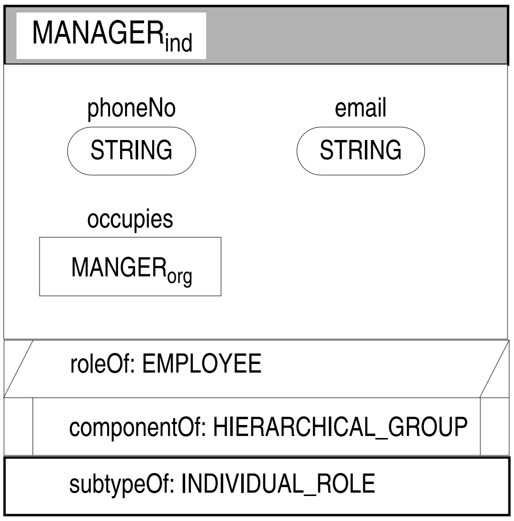


Figure 9.4: Individual-role type $\mathsf{MANAGER_{ind}}$

Roles are organized in role hierarchies (cf. Subsection 7.2.1). All organizational roles that belong to the same role hierarchy represent the same position and are thus occupied by the same actor[5]. More precisely, each actor occupying a certain position is modeled by one individual role object for every organizational role of the position she or he occupies. This means, for each organizational role there exist an individual role that represents the actor that occupies the position to which the organizational role belongs. This partial duality in the hierarchy of organizational roles and individual roles allows to adequately model situations in which we have to separate properties and authorizations of actors that depend on the organizational context in which the actor occupies a position but are independent from the individual actor occupying that position at a particular time, i.e., such properties and authorizations are associated with position objects or organizational-role objects, from properties and authorizations of actors that depend on the organizational context in which the actor occupies a position *and* depend on the individual actor occupying the position, i.e., such properties and authorizations are associated with agent objects or individual-role objects.

Note: To reduce the effort of specifying dual role hierarchies that model positions and actors, we extend type diagrams such that they allow for a joint specification of types of organizational roles (positions) and types of corresponding individual roles (agents). This means, corresponding types are shown in the *same* type diagram. As a notational convention, names of types of organizational roles (positions) and names of types of corresponding

[5]Note: We assume that each position is occupied by exactly one actor at a time. As an exception to this rule, coordinator positions (cf. network control structure, below) may be occupied by several actors at a time.

individual roles (agents) share a user-defined prefix. To distinguish the two kinds of types, names of types of organizational roles and positions share the system-defined suffix org and the names of types of individual roles and agents share the system-defined suffix ind. Tools used to jointly specify types of organizational roles (positions) and corresponding types of individual roles (agents) could reduce the effort of their specification in that they (a) prompt the user for the common prefix of their names, (b) prompt the user for the common domain of their component-of relationships, and (c) propose a default specification of the inverse relationships occupiedBy and occupies.

Example: The type diagram in figure 9.5 shows the joint specification of position type $\mathsf{MANAGER}_{\mathsf{org}}$ (cf. Figure 9.3) and individual-role type $\mathsf{MANAGER}_{\mathsf{ind}}$ (cf. Figure 9.4).

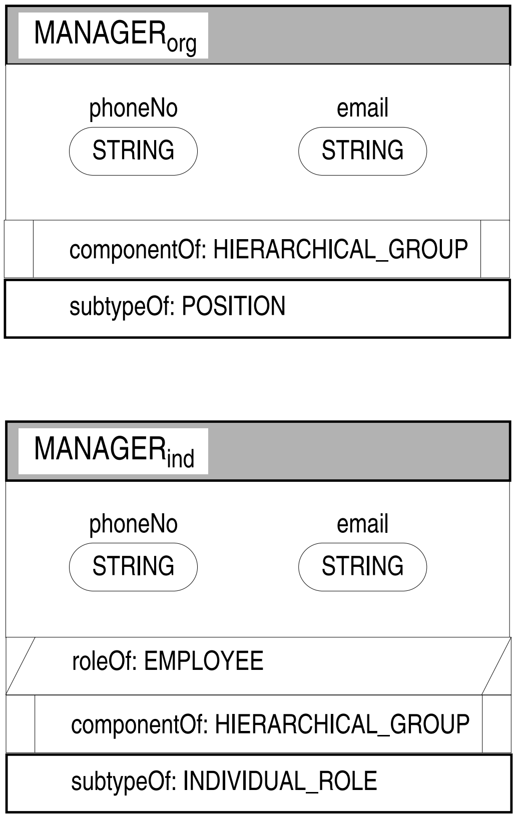


Figure 9.5: Joint specification of $\mathsf{MANAGER}_{\mathsf{org}}$ and $\mathsf{MANAGER}_{\mathsf{ind}}$

Traditionally, organizational roles (positions) and corresponding individual roles (agents) must be specified as independent properties of organizational units, i.e., separate graphical symbols are used to display their specifications. Thus type diagrams that show organizational-unit types tend to be graphically overloaded. However, at the same time they fail to represent the duality of organizational roles (positions) and corresponding individual roles (agents). We aim at keeping type diagrams showing organizational-unit types small and at making them at the same time more informative in that we *jointly* specify organizational-role (position) properties and corresponding individual-role (agent) properties. However, a joint specification of such properties is only possible in cases, where organizational roles (positions) and corresponding individual roles (agents) are either both components of or both constituents of the organizational unit to which they belong. Moreover, we require that the notational convention introduced for names of types of organizational roles (positions) and corresponding types of individual roles (agents) is adhered to when specifying names of organizational-role (position) properties and corresponding individual-role (agent) properties, too. That is, pairs of corresponding properties share the same user-defined prefix and are distinguished by system-defined suffix org and suffix ind. Figure 9.6 shows examples for joint specifications of organizational-role (position) properties and corresponding individual-role (agent) properties and compares them to traditional specifications.

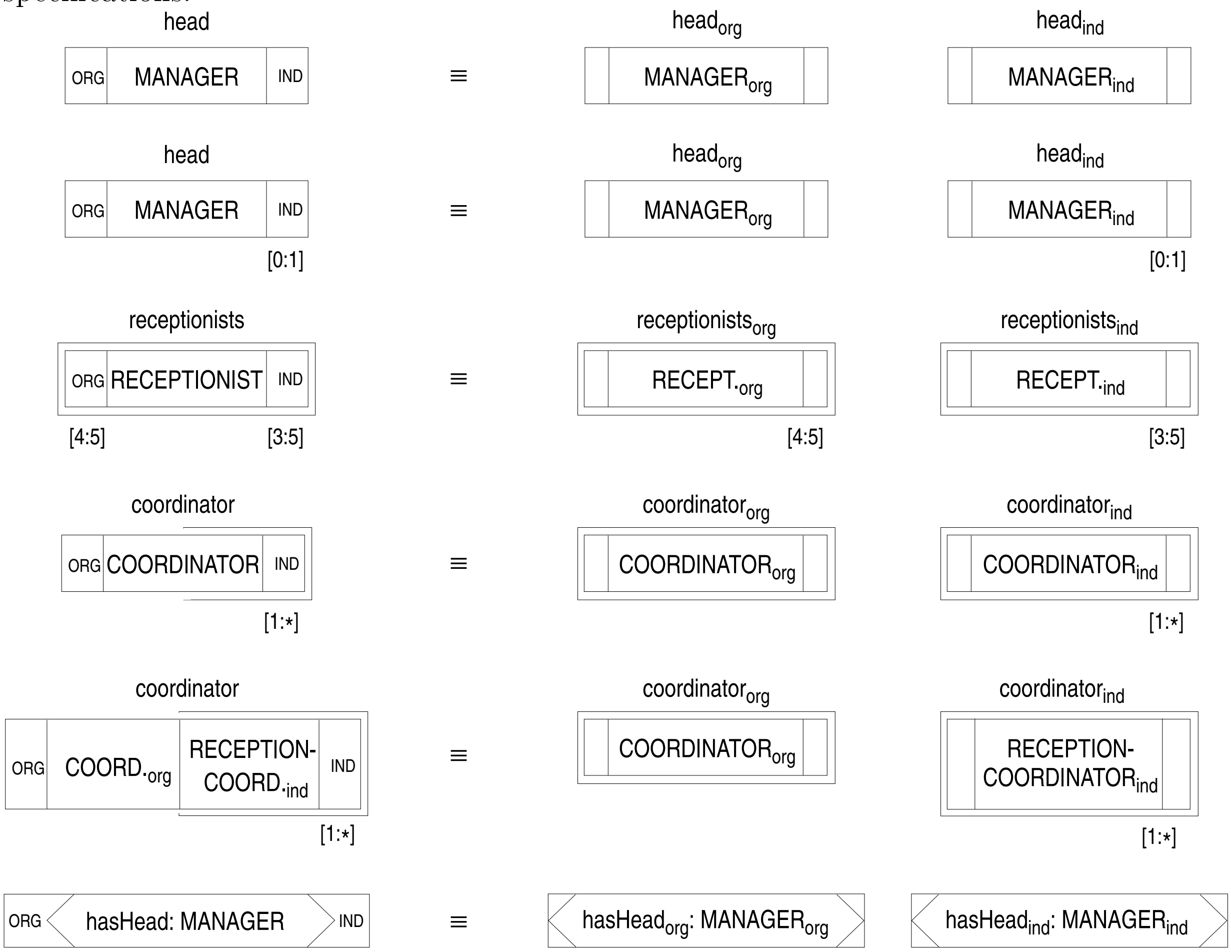


Figure 9.6: Joint specification of organizational roles (positions) and individual roles (agents)

We assume that to each organizational role an individual worklist (cf. Subsection 5.2.1) is assigned. Thus each agent has to care for a whole set of worklists. This architectural decision to associate with each organizational role an individual worklist has several advantages. First, tools an agent uses to maintain his worklists may organize his duties according to different organizational roles. Second, several organizational roles an agent occupies may have special security requirements, e.g., access to entries in the worklist of a particular organizational role may be secured by additional authentication mechanisms (cf. Chapter 6). The decision to associate with each organizational role an individual worklist naturally copes with these requirements.

Role Constraints. Role-of relationships and has-role relationships organize role types into a role-type hierarchy (cf. Subsection 7.2.1). In general, an instance of a superordinate type, called player, can acquire and abandon instances of subordinate types, called roles, dynamically and independent of each other. Below, we will discuss different kinds of so-called *role constraints* that allow to model restrictions on role playing. Note: Although the constraints presented below are discussed in the context of organization modeling, they can be applied to business objects in general.

Usually, agents and positions (cf. agents and positions, above) are constrained in role playing in that they may acquire only roles of specific types, may not play roles of certain types simultaneously, may acquire roles of different types only in a predefined order, or may play a certain type of role only a limited number of times, just to mention a few kinds of constraints. Below, we discuss different kinds of constraints in more detail and show how these constraints can be modeled in an organization schema. Note: Since a position and associated organizational roles determine the authority of an actor occupying that position, modeling of role constraints is relevant in the security context, too. In Chapter 10 (cf. Subsection 10.2.4), we discuss the modeling of role constraints that restrict the activation of roles during sessions.

Type of roles : A player of type O_1 may acquire a role of type O_2 only if O_1 or a supertype of O_1 is the domain of the role-of relationship defined at O_2.

Example: Persons may play only employee roles and guest roles (cf. Figure 9.2(a)). Employee roles may play only manager roles, secretary roles, assistant roles, receptionist roles, and workflow-administrator roles (cf. Figure 9.2(b)).

Cardinality : In general, a player may exist in the same type of role several times in different contexts. Specifying a *cardinality constraint* for a has-role relationship r at the player type P limits the number of roles of type O, where O is the domain of r, in which an instance of P can exist simultaneously.

Example: The cardinality constraint [0:3] associated with has-role relationship asManagers specified at role type EMPLOYEE (cf. Figure 9.2(b)) says that an employee need not exist in a manager role and that it may exist in at most three manager roles at the same time.

Mutual exclusiveness of roles : In general, a player may exist in roles of different types simultaneously (cf. types of roles, above). Specifying a *mutual-exclusiveness constraint* says that a player must not exist in two types of roles at the same time. Graphically, a mutual-exclusiveness constraint can be specified in a role-type hierarchy diagram. A role-type hierarchy diagram resembles much a type-hierarchy diagram (cf. Subsection 7.2.1). To distinguish role-type hierarchies from ordinary type hierarchies the graphical symbol depicting a subtype-supertype relationship, the triangle, is replaced by a semi-circle. An unfilled semi-circle denotes that subroles are mutual exclusive, a filled semi-circle that there is no restriction on simultaneous playing of subroles.

Example: A person may exist in an employee role and in a guest role at the same time. At any time, however, an employee may only exist either in a manager role, in a secretary role, in an assistant role, in a receptionist role, or in a workflow-administrator role (cf. Figure 9.7).

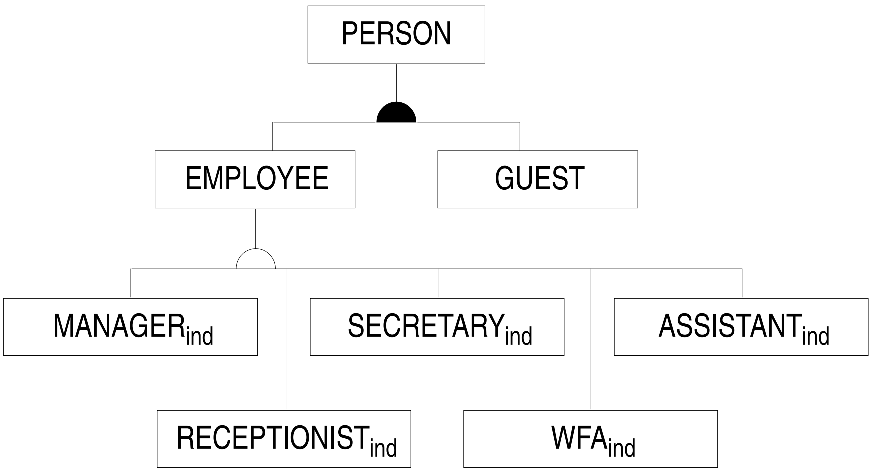


Figure 9.7: Role-type hierarchy with root PERSON

Order of role acquiring and abandoning: In general, a player may acquire and abandon roles in any order. Constraints on the order of role acquiring and abandoning are specified by role-type hierarchies and by the life-cycles that define the behavior of players. A *role-type hierarchy* specifies constraints on the order of role acquiring and abandoning in that a player p must acquire a role r_1 first, before it may acquire a role r_2 of which r_1 is the player. Moreover, p may not abandon r_1 before r_2.

Example: A person must acquire an employee role first, before it may acquire a manager role (cf. Figure 9.7). Moreover, if a person exists in an employee role and a manager role it may not abandon the employee role before the manager role.

Note: A life-cycle defined for players of a particular type can define further constraints on the order in which the players may acquire and abandon *direct* roles (cf. Subsection 9.2.2 for an example).

Organizational units. An *organizational unit* may consist of a set of positions, a set of organizational roles, a set of agents, a set of individual roles, a set of resources, and a set of other organizational units. We distinguish two different kinds of organizational units depending on the basic mechanism according to which they coordinate their subsystems. Organizational units that are instances of object type HIERARCHICAL_GROUP (cf. Figure 9.8) rely on the concept of *hierarchy* (cf. hierarchical control structure, Subsection 5.2.1) to coordinate their subsystems. Organizational units that are instances of object type NETWORK_GROUP (cf. Figure 9.8) rely on the concept of *self-management* (cf. network control structure, below) to coordinate their activities.

An overall organization may consist of different kinds of organizational units, organizational units with a hierarchical control structure and ones with a network control structure. No restriction is posed on the way in which these different kinds of organizational units are composed to form an overall organization. Actually, several subtypes of the predefined ones will be defined in the conceptual schema of a real-world organization to represent special requirements and constraints.

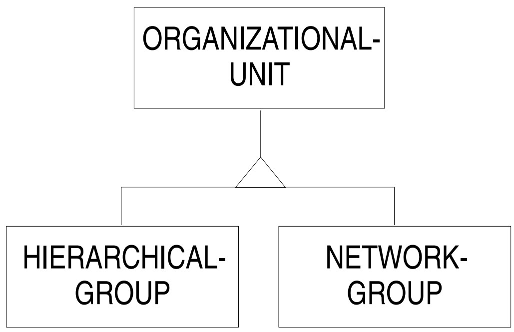


Figure 9.8: Basic types of organizational units

Organizational Relationships

The structure of an organization is determined by a set of organizational relationships. In this subsection, we will investigate the use of semantic relationships (cf. Subsection 7.2.1) for the modeling of the *physical structure* of an organization and will investigate a second form of *control structure* an organization can install to coordinate its actors, the *network*. The network is a control structure that is alternative to the *hierarchy*. This means, an organization installs in a particular organizational unit either a hierarchical or a network control structure. However, it may install different forms of control structures in different organizational units. Furthermore, we will investigate *substitution relationships* as another logical dimension of the structure of organizations.

Physical structure of organizations. The physical structure of an organization is determined in that agents, individual roles, positions, organizational roles, resources, and organizational units are grouped together forming an overall organization. In the advanced conceptual model, the part-whole tree depicting the physical structure of an organization has to consider further types of organizational objects. Again the root of such a tree represents an overall organization and intermediate levels represent organizational units. Leafs represent resources, positions, organizational roles, agents, and individual roles. Again resources are children of nodes representing organizational units, positions are children of nodes representing organizational units, and agents are children of nodes representing overall organizations. Organizational roles as well as individual roles are children of nodes representing organizational units (cf. Figure 9.9). In a physical-structure diagram, organizational roles are depicted like positions, individual roles like agents. To distinguish organizational roles (individual roles) from positions (agents), graphical symbols depicting organizational roles (individual roles) are of smaller size than those depicting positions (agents). Role-of relationships are indicated by dotted lines. To keep the presentation clear, a graphical symbol representing an individual role or agent that occupies an organizational role or position is stacked to the graphical symbol representing the corresponding organizational role or position. Moreover, a solid line indicating that a certain organizational role belongs to an organizational unit is not shown if the organizational role belongs to the same organizational unit as its player.

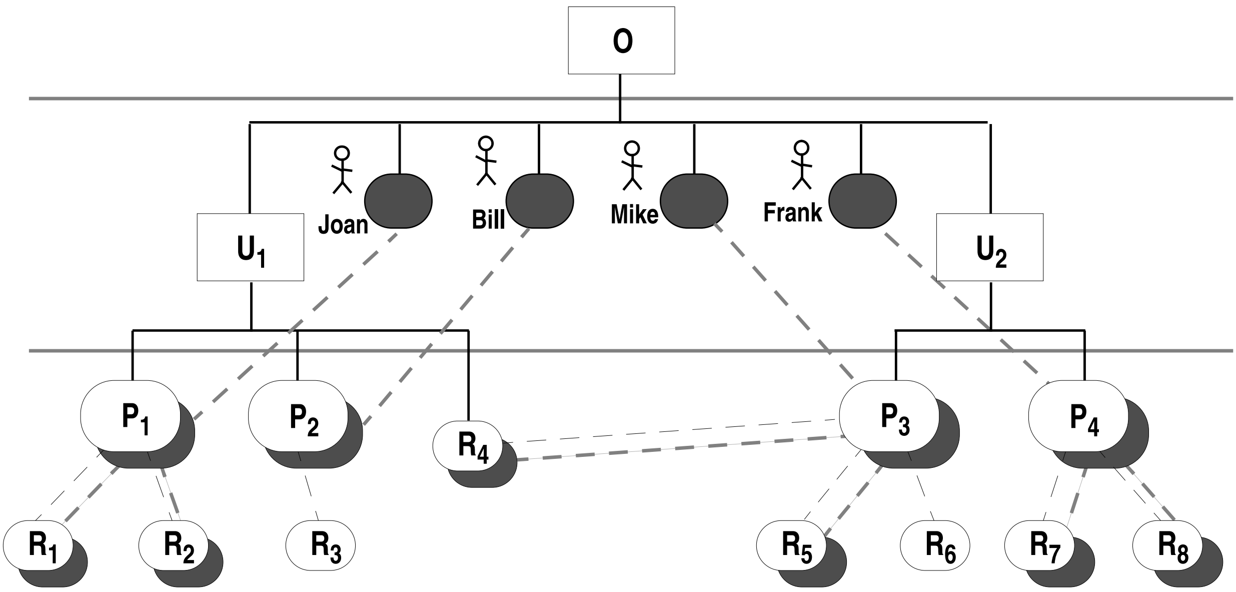


Figure 9.9: Part-whole tree of organizational objects (schematically)

Whereas the basic conceptual model in Part II (cf. Subsection 5.2.1) did not offer special concepts to model the physical structure of organizations, the advanced conceptual model offers specific types of semantic relationships to model the physical structure of organizations and objects, in general. If the existence of organizational objects depends on the organizational unit to which they exclusively belong, we use component-of relationships to model references of these organizational objects to the organizational unit and has-component relationships to model their inverses (cf. Subsection 7.2.1). On the contrary, if the existence of an organizational unit depends on organizational objects that belong to it, we use has-constituent relationships to model references of the organizational unit to these organizational objects and constituent-of relationships to model their inverses (cf. Subsection 7.2.1).

Example: Figure 9.10 shows a type diagram of organizational-unit type EU-HOTEL[6] using semantic relationships to model the physical structure of hotels (cf. Figure 5.4 and the associated example text for further explanation). Hotels of type EU-HOTEL belong to a hotel chain of type EU-STAY[6].

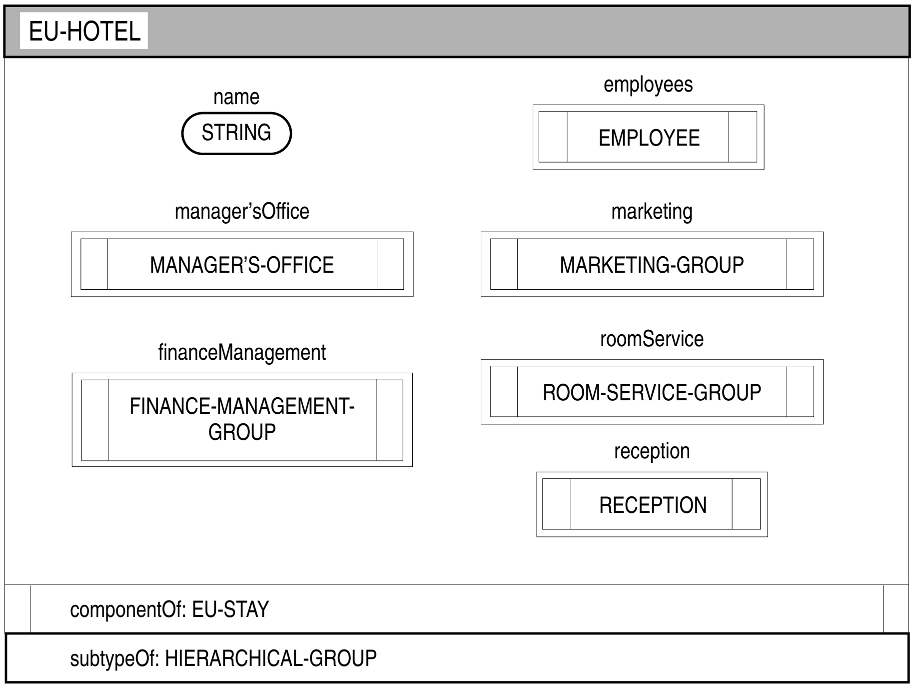


Figure 9.10: Organizational-unit type EU-HOTEL

Figure 9.11 shows a type diagram of organizational-unit type EU-STAY[6]. A hotel chain of type EU-STAY is one division of an overall organization of type EU-CORPORATION[6] (cf. component-of relationship). It is identified by the value of key attribute name and has several organizational units as components: a vice-president's office, a marketing department, a finance-management department, and several hotels (cf. has-component relationships vicePresident'sOffice, marketing, financeManagement, and hotels with domains VICE-PRESIDENT'S-OFFICE, MARKETING-DEPARTMENT, FINANCE-MANAGEMENT-DEPARTMENT, and EU-HOTEL, respectively). Furthermore, a hotel

[6] Note: The acronym EU in type names EU-HOTEL, EU-STAY, and EU-CORPORATION stands for European Union.

chain of type EU-STAY may install several temporary task forces to solve diverse problems (cf. has-component relationship taskForces with domain EU-TASK-FORCE).

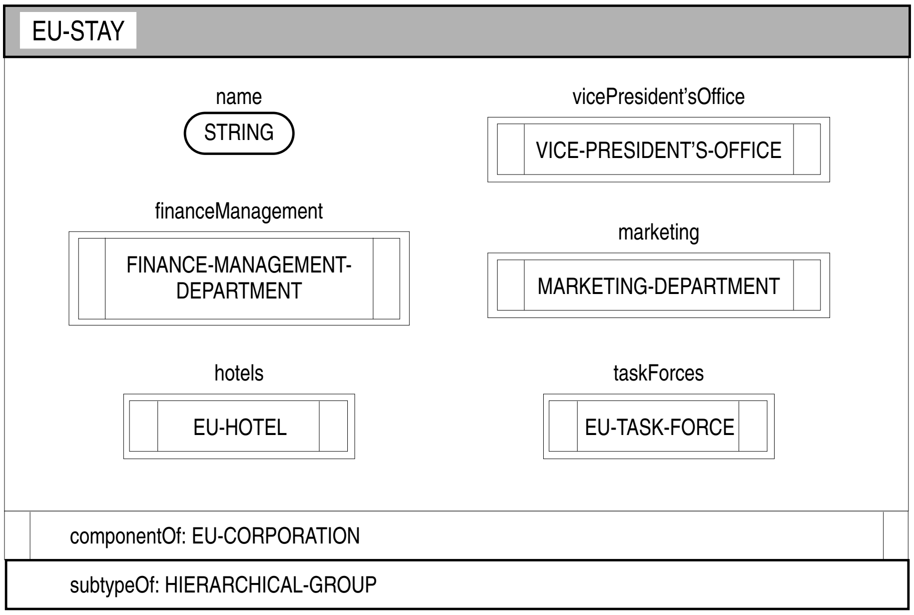


Figure 9.11: Organizational-unit type EU-STAY

Example: Figure 9.12(a) shows a part-whole tree of hotel chains of type EU-STAY. A hotel chain of type EU-STAY consists of a vice-president's office, a marketing department, a finance-management department, several task forces, and several hotels (cf. Figure 9.11). A hotel chain of type EU-STAY belongs to an overall organization of type EU-CORPORATION.

Figure 9.12(b) shows a part-whole tree of organizations of type EU-CORPORATION. An organization of type EU-CORPORATION consists of a board of directors (non-executive directors), a president's office, and three divisions EU-stay (a hotel chain), EU-fly (an airline), EU-rent (a car-rental chain).

Figure 9.12(c) shows a part-whole tree of hotels of type EU-HOTEL. A hotel of type EU-HOTEL consists of a manager's office, a marketing group, a finance-management group, a room-service group, and a reception. A hotel of type EU-HOTEL belongs to a hotel chain of type EU-STAY (cf. Figure 9.12(a)).

Control structure of organizations. The control structure of an organization serves as communication channels, called *chain of command*, along which authority and responsibility may be delegated to and removed from individual actors within an organization. In Chapter 5 (cf. Subsection 5.2.1) we presented *hierarchy* as a particular form of the control structure of organizations. In this subsection, we will present the *network* as an alternative

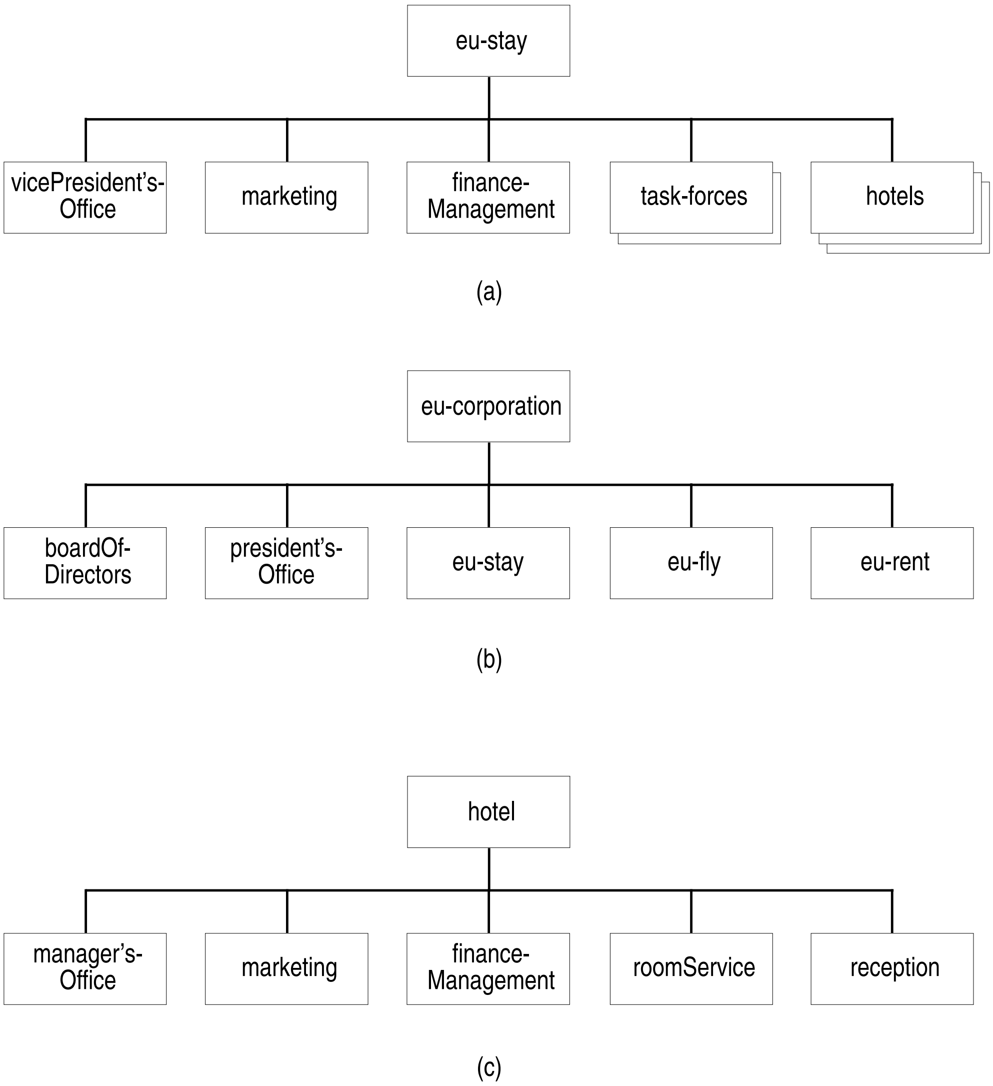


Figure 9.12: Selected part-whole trees of EU-CORPORATION

form of the control structure of organizations. First, however, we will give a short summary of the characteristics of hierarchical control structures.

If *hierarchy* is used as a structural means to coordinate activities performed by different actors (organizational units) within an organizational unit, all managerial tasks are within the responsibility of a single actor (organizational unit), called supervisor. All actors (organizational units) that belong to the same organizational unit are subordinate to the same supervisor (cf. Subsection 5.2.1).

Network is another form of the control structure of an organization or of several organizational units thereof. Network control structures are alternative to hierarchical control structures in that a single type of organizational units has either a network or a hierarchical control structure. Different organizational units of the same organization, however, may have different forms of control structures. This means, *network control structures and hierarchical control structures can coexist within organizations.*

If *network* is used as a structural means to coordinate activities performed by different actors (organizational units) within an organizational unit, managerial tasks are within the responsibility of all of the actors (organizational units) of that organizational unit in common. Since networks of actors (organizational units) rely on the principle of *self-management* to coordinate their activities, the size of such networks is typically limited. In the literature, self-management is usually discussed in the context of teams or groups (e.g., Daft [Daf95] discusses *self-managed teams*). The degree to which actors (organizational units) of a self-managed organizational unit participate in decisions is high. Basically, two settings can be distinguished: First, decisions are made by the actors (organizational units) of a self-managed organizational unit in common. Second, decisions are made by a designated actor (organizational unit) selected from the overall set of actors (organizational units) of a self-managed organizational unit. In each setting, we identify a *coordinator* whose job is either to guide a self-managed organizational unit in making decisions[7], to make decisions on behalf of a self-managed organizational unit on its own, or a mix of both kinds of tasks.

We model *a coordinator of a network of actors* as a multi-agent position (or a multi-agent organizational role) that can be occupied by all or at least by most of the agents (individual roles) within a network[8]. In general, actors of self-managed organizational units have a wide range of skills and are cross-trained to perform jobs of other team members, too. As a consequence, role hierarchies representing actors in a self-managed organizational unit are rather flat compared to role hierarchies representing actors in a hierarchical organizational unit. Typically, the assignment of an agent (individual role) to a coordinator position (organizational role) is temporary and is decided upon by the actors of a self-managed organizational unit, which may follow different strategies. For example, depending on the actual problem(s) the most experienced actor is selected as coordinator or each actor is selected as coordinator periodically. In any case, however, the formal authority is with the self-managed organizational unit as such, i.e., no actor of a self-managed organizational unit is privileged *per se*.

We model *a coordinator of a network of organizational units* as a self-managed organizational unit. The actors of such a self-managed organizational unit are actors of the organizational units belonging to the network. More precisely, the actors of such a self-managed organizational unit coordinating a network of organizational units must be actors occupying a coordinator position or organizational role within a coordinated organizational unit u if u has a network control structure, it must be actors occupying a supervisor position or organization role within u if u has a hierarchical control structure. The control structure of the coordinating organizational unit is modeled as a network of actors (above).

As mentioned before, network control structures and hierarchical control structures

[7] Groupware systems could support self-managed organizational units in decision-making processes.

[8] Note: Each multi-agent position (organizational role) has a multi-valued relationship occupiedBy, i.e., it can be occupied by several agents (individual roles) at a time (cf. positions, Section 5.1 and Subsection 5.2.1).

can coexist in organizations. Katzenbach and Smith put it this way [KS95]: "Those who see [self-managed] teams as a replacement for hierarchy are missing the true potential of teams." Actually, self-managed organizational units are organized around processes and *complete* the hierarchical form of organization reducing the number of hierarchical levels within an organization. This means, independent of the number of hierarchical levels and they way in which authority and responsibility is passed to members of a group, either by a supervisor or by a coordinator, each actor of an organization must be linked to the owner of an organization or its representatives within that organization. As mentioned above, this is necessary to verify the legitimacy of authority and responsibilities of the actors of an organization.

In an organizational unit with a hierarchical control structure, communication channels represent *vertical links*, i.e., supervisors are connected with their subordinates (cf. Figure 9.13). In an organizational unit with a network control structure, communication channels represent *horizontal links*, i.e., all actors or organizational units are connected to each other.—Note: The circle in the graphical presentation of a network control structure in Figure 9.13 is intended to represent a communication channel that links all actors or organizational units connected to that channel.—To allow for a smooth integration of self-managed organizational units in horizontal organizations, all managerial task of a self-managed team are concentrated within a single position, organizational role, or organizational unit, i.e., the *coordinator*, introducing some kind of virtual hierarchy within that organizational unit. This virtual hierarchy provides communication channels to delegate authority and responsibilities. Inside a network of actors this virtual hierarchy disappears, since several or all of the actors of a network occupy the coordinator position or organizational role within that network simultaneously. That is, the coordinator of a network of actors is modeled by a multi-agent position (organizational role) (cf. modeling of a coordinator of a network of actors, above). Inside a network of organizational units this virtual hierarchy disappears, since the coordinator of a network of organizational units is modeled by another self-managed organizational unit and since positions and organizational roles in that self-managed organizational unit are occupied by actors belonging to organizational units within the network (cf. modeling of a coordinator of a network of organizational units, above).

A network control structure between positions, organizational roles, or organizational units of a self-managed organizational unit can be fully specified by a so-called coordination graph. A *coordination graph* $G_U = (V, E)$ is a graph defined at an organizational-unit type U. It defines a network of organizational roles, positions, or organizational units that belong to organizational units of type U. V, the set of vertices of G_U, is the set of *names* of organizational roles, positions, or organizational units that belong to organizational units of type U. $E \subseteq V \times V$, the set of directed edges of G_U, is the set of coordination relationships between names of organizational roles, positions, or organizational units contained in V. An edge $(v_1, v_2) \in E$ between organizational roles and positions (between organizational units) v_1 and v_2 says that v_1 is the coordinator of v_2. The name of a coordinator is identified as that node of a coordination graph to which all other nodes are directly linked, i.e., in a network all positions, organizational roles, or organizational units are connected to the coordinator

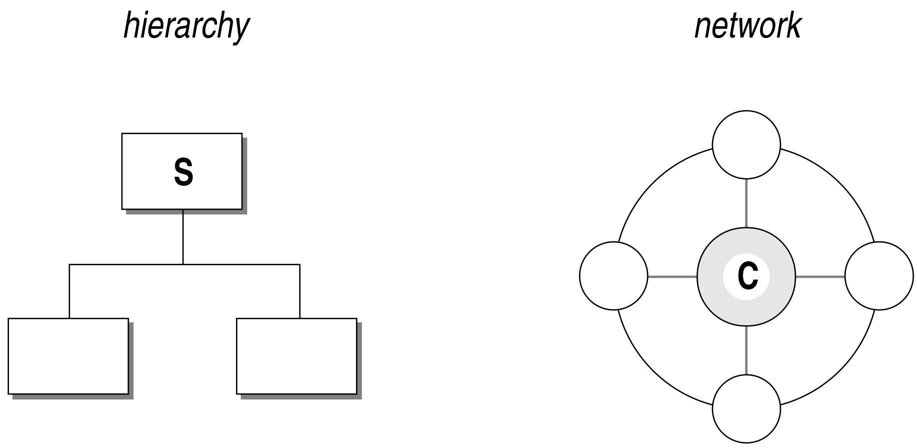


Figure 9.13: Hierarchical and network control structure (schematically)

of the network by a path of length 1. As a consequence, designing a coordination graph G_U of self-managed organizational units of a particular type U requires only two steps: (1) Identify the names of all positions and organizational roles, or organizational units that belong to instances of U, i.e., determine the set of vertices V of G_U, and (2) identify the name of the coordinators of the positions and organizational roles, or organizational units that belong to instances of U. In fact, a coordination graph G_U can be derived automatically if the set V and the name of the coordinators of self-managed organizational units of type U are specified.

As with subordination graphs (cf. Subsection 5.2.1), *a coordination graph G_U need not be instantiated for organizational units of type U.* To refer to positions, organizational roles, or organizational units that coordinate or are coordinated by a position, an organizational role, or an organizational unit, predefined operations hasCoordinator and coordinatorOf can be invoked on positions, organizational roles, and organizational units. Invoking these operations on a position, an organizational role, or an organizational unit o returns a set of *names* of positions, organizational roles, or organizational units that coordinate o or that are coordinated by o. These names are properties of the organizational unit to which o belongs. Consequently, these names can be applied as retrieval operations to o. This means, the knowledge specified in a coordination graph G_U fully determines the control structure among the positions, the organizational roles, or the organizational units that belong to organizational units of type U.

A self-managed organizational unit u may itself be embedded into an organizational unit with a hierarchical or a network control structure (cf. coexistence of hierarchical and network control structures within the same organization, above). In the former case, a coordinator of u is itself subordinate to the supervisor of a superordinate organizational unit. In the latter case, a coordinator of u has itself a coordinator in a related organizational unit. If u has no superordinate and no coordinating organizational unit, then the coordinator of u is the top position or the top organizational-unit in the control structure of the organization to which u belongs.

Graphically, a coordination graph is shown in a *control-structure diagram.* A coordination graph is depicted by a set of nodes, which correspond to names of positions, organizational roles, or organizational units. An edge is depicted by a solid line. Coordinators are shown in the center of a control-structure diagram and coordinated organizational roles, positions, or organizational units are aligned in a circle around the coordinators. As mentioned before, the circle in a graphical representation of a network control structure is intended to represent a communication channel that links all positions and organizational roles, or organizational units connected to that channel (cf. Figure 9.13 for a schematic representation of a network control structure). Control-structure diagrams are one form of the well-known organization charts, showing coordination relationships among the positions and the organizational roles, and among the organizational units of an organization. Note: A coordination graph may also be depicted in a type diagram showing the organizational-unit type at which the coordination graph has been defined. In this case a separate section of the type diagram is devoted to the presentation of a coordination graph, which is marked by the keywords control structure.

Example: Figure 9.14 shows a type diagram of organizational-unit type RECEPTION. A reception is a process team (cf. relationship subtypeOf) that belongs to a hotel of type EU-HOTEL(cf. has-component relationship). A reception has a multi-agent position $\mathsf{coordinator}_{\mathsf{org}}$, between five and ten single-agent positions $\mathsf{receptionists}_{\mathsf{org}}$, and one or two single-agent positions $\mathsf{wfAdmins}_{\mathsf{org}}$. Relationships $\mathsf{receptionists}_{\mathsf{org}}$ and $\mathsf{receptionists}_{\mathsf{ind}}$ specialize the inherited abstract relationships $\mathsf{members}_{\mathsf{org}}$ and $\mathsf{members}_{\mathsf{ind}}$, respectively. Furthermore, the domain of the inherited relationship $\mathsf{coordinator}_{\mathsf{ind}}$ is specialized to individual-role type $\mathsf{RECEPTION\text{-}COORDINATOR}_{\mathsf{ind}}$.

Note: Relationships $\mathsf{coordinator}_{\mathsf{org}}$ and $\mathsf{coordinator}_{\mathsf{ind}}$, relationships $\mathsf{receptionists}_{\mathsf{org}}$ and $\mathsf{receptionists}_{\mathsf{ind}}$, as well as relationships $\mathsf{wfAdmins}_{\mathsf{org}}$ and $\mathsf{wfAdmins}_{\mathsf{ind}}$ are specified in condensed form (cf. Figure 9.6).

The control structure depicted in a separated section of the type diagram in Figure 9.14 says that the coordinator position coordinates the receptionist positions and the workflow-administrator positions. Actually, the coordinator position is occupied by several receptionists simultaneously, i.e., the coordinator position is a multi-agent position.

Substitution. Each organization offers services or goods to its customers. To produce these services and goods, an organization depends on the *availability* of its actors. Basically, two problems must be solved to guarantee the availability of actors: First, an appropriate number of *regular actors* must be provided. Second, a *substitution plan* must be established to guarantee availability in situations in which regular actors are not available, e.g., human actors might be on vacation.

Planning substitution incorporates several dimensions: *granularity*, *prerequisites*, *activation*, and *validity.* We allow to model substitution at different granularity levels, i.e.,

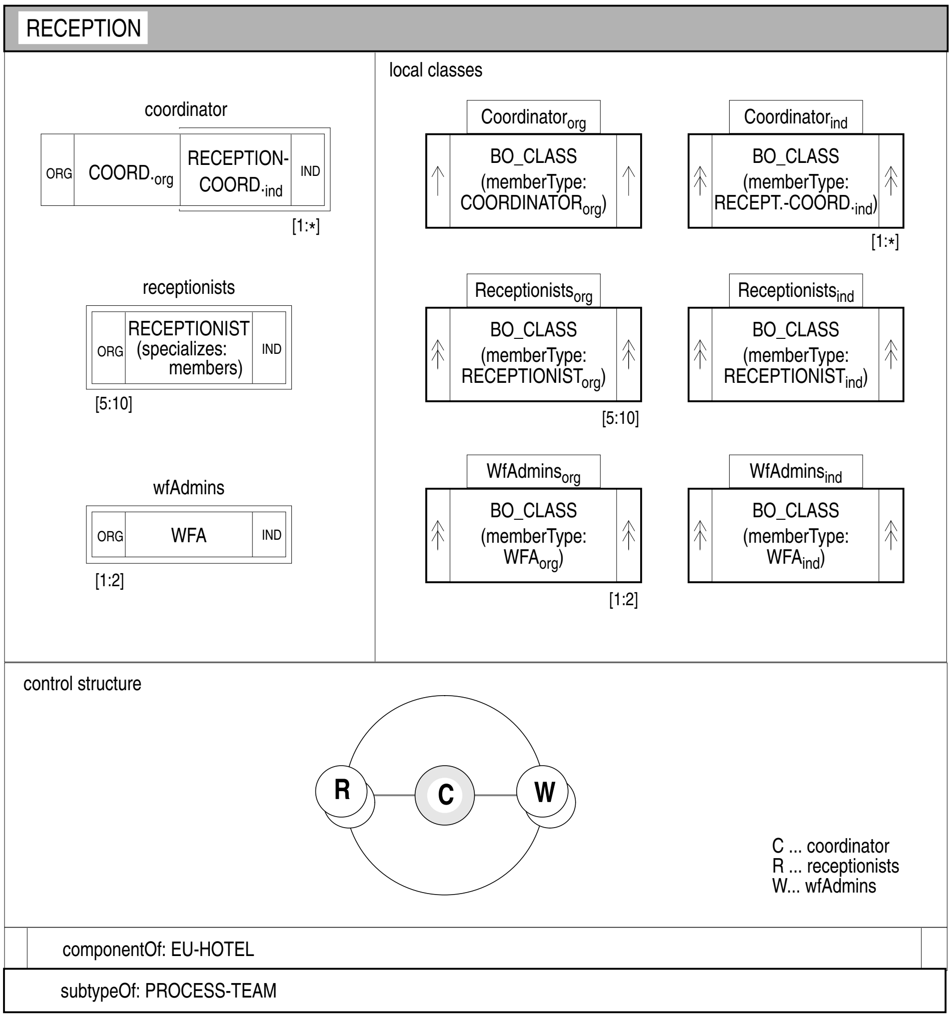


Figure 9.14: Organizational-unit type RECEPTION

position, organizational role, and activity. Substitutions specified at lower levels of granularity have priority over substitutions specified at higher ones. The weakest prerequisite an actor must fulfill to substitute for another actor is that the actor has appropriate skills. Stronger prerequisites are, for example, that the substitute must exist in appropriate individual roles that would allow him to occupy the organizational roles of the actor he substitutes. Different events can be defined activating a substitution. For example, a substitution may be active if the regular actor is on vacation, is off the office, or is logged off. Finally, a user may specify a temporal condition that determines the validity of a substitution. For example, a time interval may be specified to determine a period of time during which the substitution is valid. Note: We model substitutions as relationship objects. In fact, a substitution object must have at least the following constituents: a position or an organizational role representing the regular actor as well as a position or an organizational role representing the substitute actor. Further constituents could be a particular business case, a particular workstep specification, etc.

Example: Figure 9.15 shows a type diagram of relationship type SUBSTITUTION. It models substitutions between positions or organizational roles[9]. That is, each substitution has two has-constituent relationships subject and substitute to values from domains POSITION. Moreover, such substitutions are only valid within a user-defined interval (cf. attributes from and to with domain DATE). The derived boolean attribute isValid says whether a substitution is valid or not. Its derivation expression is: from $\leq$ TODAY $\leq$ to, where TODAY is a system variable answering the actual date.

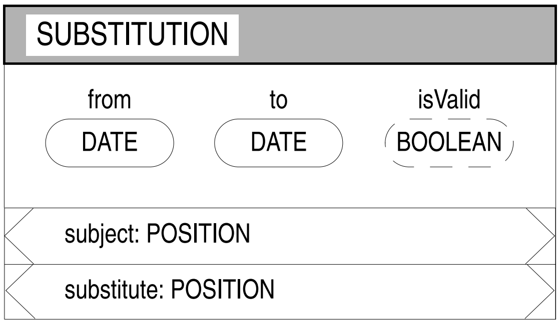


Figure 9.15: Relationship type SUBSTITUTION

Specialization of the Structure of Organizational Objects

To specialize the structure of organizational objects means, to specialize the set of *attributes*, the set of *relationships*, and the set of *local classes* organizational-object types inherit from their supertypes. Each user-defined organizational-object type is a (direct or indirect) subtype of predefined conceptual type O_OBJECT (Organizational OBJECT, cf. Figure 7.6) or of one of its subtypes (cf. Figure 9.1). Each subtype of one of these predefined conceptual types may specialize its supertype. The semantics of the predefined conceptual types of organizational objects has already been discussed in Section 9.1 and in more detail in this section. Below, we limit ourselves to a short summary. Note: Criteria for a consistent specialization of the structure of objects have been discussed in Subsection 7.2.1.

Each organizational object is an instance of the conceptual type O_OBJECT, which is a direct subtype of conceptual type B_OBJECT (cf. Figure 7.6). Organizational objects are distinguished into non-agents and agents. That is, conceptual types NON-AGENT and AGENT are direct subtypes of O_OBJECT (cf. Figure 9.1). Non-agents are either positions, which model a kind of virtual actors, or organizational units, which model groups of actors. That is, conceptual types POSITION and ORGANIZATIONAL_UNIT are direct subtypes of NON-AGENT (cf. Figure 9.1). Each position is modeled by one position object and several organizational-role objects. Position objects and organizational-role objects are organized

[9]Note: The predefined type ORGANIZATIONAL_ROLE is a subtype of the predefined type POSITION (cf. Figure 9.1).

in a so-called role hierarchies, where position objects are the roots of such role hierarchies. Conceptual type ORGANIZATIONAL_ROLE is a direct subtype of POSITION (cf. Figure 9.1). Organizational units are distinguished into hierarchical groups or network groups depending on the type of control structure they utilize to coordinate their activities, i.e., hierarchy or network. Conceptual types HIERARCHICAL_GROUP and NETWORK_GROUP are direct subtypes of ORGANIZATIONAL_UNIT (cf. Figure 9.8). Each actor is modeled by one agent object and several individual-role objects. Agent objects and individual-role objects are organized in a role hierarchies, where agent objects are the roots of such role hierarchies. Conceptual type INDIVIDUAL_ROLE is a direct subtype of AGENT (cf. Figure 9.1).

9.2.2 Modeling the Behavior of Organizational Objects

The behavior of organizational objects is determined by a set of *activities*, by *allowed execution orders* on these activities, and by their *interactions* with other objects (cf. Subsection 5.2.2). In Subsection 7.2.2, advanced concepts of behavior modeling, i.e., synchronization and specialization of the behavior of objects, have been discussed. These concepts of behavior modeling apply to the modeling of the behavior of organizational objects in the same way. Below, we discuss a special aspect of the synchronization of the behavior of objects not discussed in Subsection 7.2.2, i.e., synchronizing the acquiring and abandoning of roles for players. Afterwards, we shortly review the specialization of the behavior of organizational objects.

Synchronization of the Behavior of Organizational Objects

Whenever a non-derived has-role relationship r is defined at a player type P, a public update-operation must be invoked on an instance of P if it wants to acquire or abandon a role from the domain of r. Since a life-cycle defined at P restricts the allowed execution orders of the public update-operations defined at P, a life-cycle can be used to model constraints one role acquiring and role abandoning.

Example: Figure 9.16 shows part of the life-cycle defined at role type EMPLOYEE (cf. Figure 9.2). Employees have no chance to enhance their career. For example, an employee that starts as receptionist at a hotel may never be promoted a manager of the same hotel. The same is true for employees starting as secretaries or as assistants.

Specialization of the Behavior of Organizational Objects

To specialize the behavior of organizational objects means, to specialize *activities* and *life-cycles* organizational-object types inherit from their supertypes. In fact, each user-defined organizational-object type is a (direct or indirect) subtype of the predefined conceptual type O_OBJECT (Organizational OBJECT, cf. Figure 7.6), which models the basic behavior each organizational object offers to its environment. In fact, conceptual type O_OBJECT does not specialize the behavior it inherits from its supertype B_OBJECT (cf. Figure 7.13). Subtypes

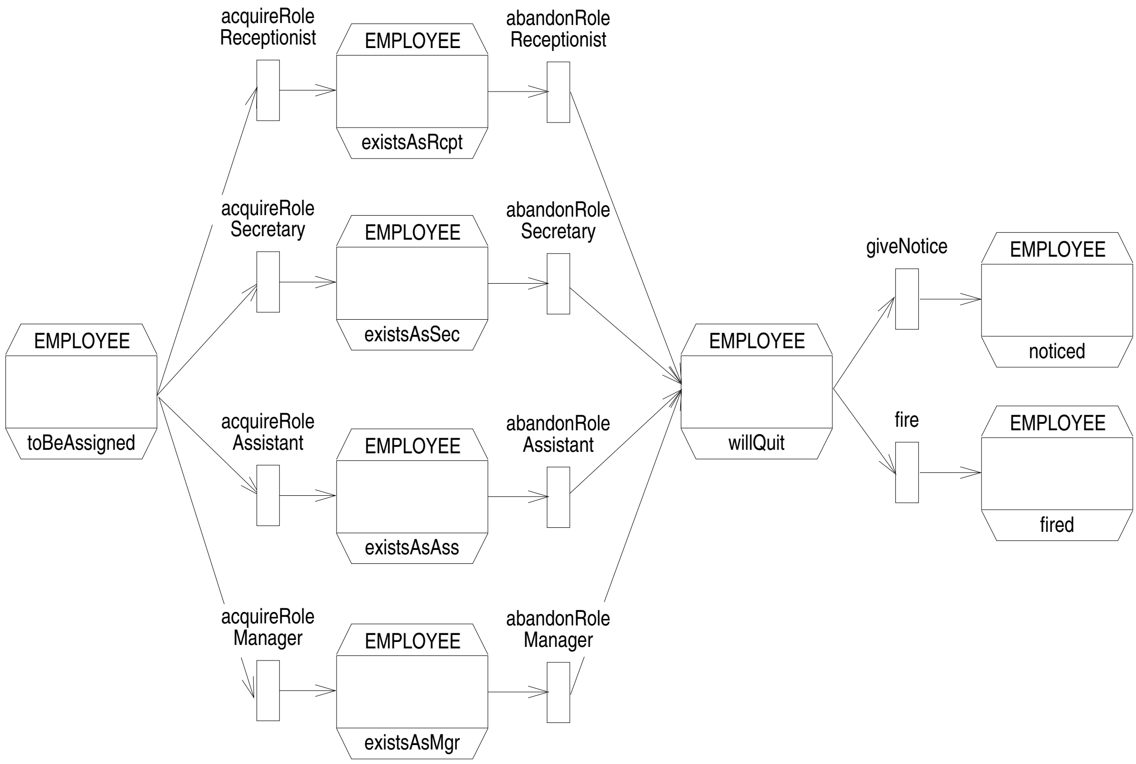


Figure 9.16: Life-cycle of role type EMPLOYEE (partially)

of conceptual type O_OBJECT may specialize inherited behavior. Note: Criteria for a consistent specialization of the behavior of objects have been discussed in Subsection 7.2.2.

9.2.3 Modeling Classes of Organizational Objects

An *organizational-object class* collects a set of organizational objects as its members. Classes of organizational objects are modeled in the same way as classes of business objects (cf. Subsection 7.2.3). Specifying local classes as domains of relationships guarantees local referential integrity. In general, local referential integrity requires that two objects taking part in a directed relationship must be direct or indirect components of the same composite object (cf. Subsection 7.2.3). Due to the architecture of our workflow model (cf. physical structure of organizations, Subsections 5.2.1 and 9.2.1), local referential integrity requires that two objects taking part in a directed relationship must be direct or indirect components of the same organization or organizational unit. For example, local referential integrity constraints might require that, as employee of an organization o, a person may only occupy positions within o, may only be authorized to use resources of o, and may only process business cases of o.

Example: Organizational-unit type RECEPTION (cf. Figure 9.14(a)) specifies six local classes that collect positions, organizational roles, and individual roles. Local class

$\mathsf{Coordinator}_{\mathsf{org}}$ collects a position of type $\mathsf{COORDINATOR}_{\mathsf{org}}$, local class $\mathsf{Coordinator}_{\mathsf{ind}}$ collects individual roles of type $\mathsf{COORDINATOR}_{\mathsf{ind}}$, local class $\mathsf{Receptionists}_{\mathsf{org}}$ collects organizational roles of type $\mathsf{RECEPTIONIST}_{\mathsf{org}}$, local class $\mathsf{Receptionists}_{\mathsf{ind}}$ collects individual roles of type $\mathsf{RECEPTIONIST}_{\mathsf{ind}}$, local class $\mathsf{WfAdmins}_{\mathsf{org}}$ collects positions of type $\mathsf{WFA}_{\mathsf{org}}$, and local class $\mathsf{WfAdmins}_{\mathsf{ind}}$ collects individual roles of type $\mathsf{WFA}_{\mathsf{ind}}$.

Example: Figure 9.17 shows part of a class-domain diagram of organizational-unit type RECEPTION (cf. Figure 9.14(a)). The domain of relationship occupiedBy of coordinator positions that are members of local class $\mathsf{Coordinator}_{\mathsf{org}}$ of a reception is restricted to members of local class $\mathsf{Coordinator}_{\mathsf{ind}}$ of that reception. Since members of local class $\mathsf{Coordinator}_{\mathsf{ind}}$ of a reception are individual roles that may be played only by receptionists of the same reception (cf. role subclass-superclass relationship between local class $\mathsf{Receptionists}_{\mathsf{ind}}$ and local class $\mathsf{Coordinator}_{\mathsf{ind}}$), this class-domain constraint guarantees self-management of receptionists of a reception. Relationship occupies is inverse to relationship occupiedBy. The domain of relationship occupies of individual coordinator positions that are members of local class $\mathsf{Coordinator}_{\mathsf{ind}}$ of a reception is restricted to members of local class $\mathsf{Coordinator}_{\mathsf{org}}$ of that reception.

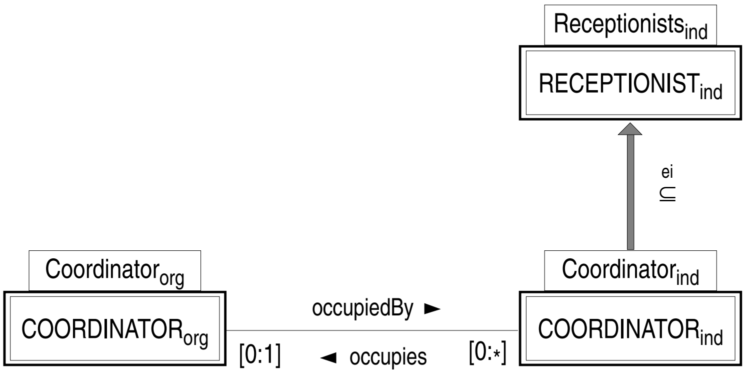


Figure 9.17: Class-domain diagram at RECEPTION

9.3 Summary

The shift to process-orientation in business modeling (e.g., [HC93]) has severe impact on the structure of organizations. Structure is created around organizational processes, i.e., actors with *different* skills working on the *same* processes are grouped together into so-called process teams. Typically, process teams, which are empowered with decision-making authority, are self-managed organizational units. This means, all actors of a team are to the same extent responsible to achieve the objectives of that team. In line with concepts from organization theory (e.g., [Daf95, LARB+91]), we model self-managed organizational units as organizational units with a network control structure. Katzenbach and Smith [KS95] emphasize that process teams do *not* replace hierarchical control structures within organizations but complete them. Thus, network control structures and hierarchical control structures must coexist within organizations to guarantee that the control structure of an organization models an unbroken connection between the owner of an organization and each employee of that organization, which is the fundament for a formal authorization system. The advanced organization model naturally copes with this requirement.

Furthermore, the advanced model naturally copes with the evolutionary nature of actors. Actors of an organization are modeled by sets of objects that are organized in a role hierarchy. Positions, which represent a kind of "virtual actors" of an organization that stand in for real actors, are also modeled by sets of objects organized in a role hierarchy. If an actor occupies a position, that actor is represented by an extra role object that models individual properties of the actor occupying that position. If all actors that occupy a certain position share the same value for a property, this property is associated with the position not with the role object representing an individual actor occupying that position. The advanced model of actors and positions allows also to properly distinguish between authorizations that are granted to all actors occupying a certain position at different times, authorizations that are granted to an individual actor while occupying a certain position, and authorizations that are granted to an individual actor independent of the position it occupies within an organization.

Chapter 10

Authorization Modeling

Contents

An *authorization schema* is a formal description of the protection requirements of information maintained and processed within a workflow system. An authorization schema could be used by the authorization manager of a workflow system (cf. Section 6.2) to exclude illegal authorizations from the authorization base of that workflow system. Since they avoid that illegal authorizations are issued to actors, authorization schemas help to avoid unauthorized modification of information maintained within workflow systems, i.e., authorization schemas help to ensure integrity of information. We design conceptual authorization schemas using a set of authorization interfaces, a set of authorization constraints, a set of authorization-contract types, and a set of authorization-contract classes. Authorization contracts model authorizations in a workflow system and are treated as first-class citizens, i.e., authorizations are modeled as full-fledged objects. Authorization interfaces group together public operations with similar protection requirements that suppliers of a particular type offer to clients. Authorization constraints specify the protection requirements of public operations that are organized into authorization interface.

This chapter is organized as follows. Section 10.1 gives a management definition of the fundamental modeling concepts presented in this chapter. Section 10.2 discusses the design of authorization schemas in the realm of advanced concepts presented in this part of the thesis for the modeling of business objects in general (cf. Chapter 7) and for the modeling of business-process objects (cf. Chapter 7), workflow-process objects (cf. Chapter 8), and organizational objects (cf. Chapter 9) in particular. Section 10.3 discusses the basic principles of the administration of authorizations in organizations with mixed hierarchical and network control structures. Section 10.4 summarizes this chapter.

10.1 Fundamental Modeling Concepts

The core concept discussed in this chapter is the *authorization schema*. Actually, the design of authorization schemas, i.e., the modeling of authorization interfaces, authorization constraints, and authorization-contracts, has already been discussed in Chapter 6. However, the advanced conceptual model presented in this part of the thesis requires adaptions of some concepts. For example, authorization interfaces and authorization constraints must be adapted to guarantee the proper enforcement of protection requirements over workflow-process boundaries, since the advanced conceptual model allows workflow processes to delegate part of their duties to so-called nested workflow processes (cf. Chapter 8).

Advanced modeling concepts increase the semantic expressiveness of authorization schemas. For example, specifying local classes as actor domains of public operations being organized into authorization interfaces expresses so-called local-authorization integrity constraints. Basically, a local-authorization integrity constraint requires that actors may only be authorized to use resources of that organization or organizational unit to which they belong. Role constraints restrict the set of roles that actors may play within organizations. Since positions and organizational roles determine the authority and responsibility of actors (cf. Chapter 9), role constraints allow to limit the authority and responsibility of individual actors within workflow systems. Furthermore, since authorization contracts are full-fledged objects, advanced concepts for the modeling of business objects in general can be applied to the modeling of authorizations, too. That is, authorization schemas can be semantically rich descriptions of the protection requirements of workflow systems, or information systems in general.

Note: Basic concepts such as *authorization*, *authorization contract*, *authorization interface*, and *authorization schema* have already been defined in Section 6.1.

Local-authorization constraint: A *local-authorization constraint* is an authorization constraint (cf. Section 6.1) that specifies local classes of positions and/or agents as actor domain of public operations that are organized into a particular authorization interface. Informally, a local-authorization constraint requires (depending on the deontic modality it associates with the specified actor domain) that *a client may be, must be, may not be, or must not be authorized to request a certain service from a supplier if both the client as well as the supplier (directly or indirectly) belong to the same organizational unit or organization.* This means, local-authorization constraints take the physical structure of organizations into account when modeling the protection requirements of workflow systems (cf. physical structure of organizations, Subsection 9.2.1).

Role constraint: A *role constraint* either determines a legal configuration of roles that represent a particular actor within a workflow system or restricts the activation of roles representing an actor during sessions. Since the positions an actor occupies and the roles an actor plays within an organization determine the authority and responsibility of that actor, modeling of role constraints is important to avoid unwanted accumulations of authority for individual actors. The modeling of role constraints that determine legal configurations of roles representing particular actors has already been discussed in Chapter 9 (cf. Subsection 9.2.1). In this chapter (cf. Subsection 10.2.4), we discuss the modeling of role constraints that restrict the activation of roles during sessions.

Summing up, the semantic expressiveness of authorization schemas is increased in the advanced conceptual model. Existing concepts, which have already been introduced in Chapter 6, are adapted to advanced modeling concepts. In addition, role constraints are introduced to limit the authority and responsibility that individual actors may acquire within a workflow system, since the authority and responsibility of actors is determined by the positions they occupy and the roles they play within an organization.

10.2 Design of Authorization Schemas

An *authorization schema* is a formal description of the protection requirements of a workflow system (WFS). It consists of a set of authorization interfaces, a set of authorization constraints, a set of authorization-contract types, and a set of authorization-contract classes (cf. Section 6.3). In this section, we review the modeling of authorization schemas in the realm of the advanced modeling concepts presented in this part of the thesis (cf. Chapters 7 to 9), e.g., semantic relationships, local classes, and type hierarchies. In addition, we discuss the modeling of a particular category of *role constraints* that restrict the activation of roles during sessions and determine procedures to be followed to authenticate the identity of an actor when she or he activates roles. Note: The modeling of further categories of role constraints has been discussed in Subsection 9.2.1.

The remainder of this section is organized as follows. Subsection 10.2.1 discusses the modeling of authorization interfaces to enforce protection requirements over workflow-process boundaries, i.e., from a workflow process to the nested workflow processes to which it delegates parts of its duty, and to enforce protection requirements when workflow processes get specialized. Subsection 10.2.2 extends authorization constraints to cope with advanced modeling concepts presented for the modeling of suppliers, services, and constraints in Chapters 7 to 9. Subsection 10.2.3 reviews the modeling of structure and behavior of authorization contracts as well as the modeling of classes of authorization contracts in the realm of the advanced modeling concepts for business objects presented in Chapter 7. Subsection 10.2.4 introduces the modeling of role constraints that restrict the activation of roles during sessions.

10.2.1 Modeling of Authorization Interfaces

An *authorization interface* of business cases of a particular type is a named set of worksteps with similar protection requirements from the public interface of these business cases (cf. Subsection 6.3.1). A public interface of business cases of a particular type is modeled by a workflow process (cf. Subsection 4.4.4). In Subsection 8.3.4, advanced concepts for the modeling of workflow processes have been discussed, i.e., nested workflow processes and specialization of workflow processes. Below, we review the concept of authorization interfaces in the context of the advanced concepts for the modeling of workflow processes.

Nested Workflow Processes

In the advanced conceptual model, a workflow process models an organizational process by a set of boxes, a set of worksteps, *and* a set of nested workflow processes (cf. Subsection 8.3.4). The use of nested workflow process has several consequences for the modeling of authorization interfaces: First, the specification of a nested workflow process n causes the implicit specification of three basic workstep Start<n>, Commit<n>, and Abort<n>. The protection requirements of these worksteps must be determined, too. Usually, the protection requirements of these worksteps are independent from the protection requirements of a nested workflow process itself. We introduce four additional authorization interfaces for each specification of a nested workflow process n. Authorization interface All<n> is subordinate to authorization interface All defined at the superordinate workflow process. It contains the three basic worksteps Start<n>, Commit<n>, and Abort<n>. In addition, one authorization interface for each of these basic worksteps is defined. These authorization interfaces are subordinate to the authorization interface All<n>.

Second, the use of nested workflow processes when modeling organizational processes requires to enforce protection requirements over workflow-process boundaries. Remember that a nested workflow process is realized by a separate workflow process, which is specified at a different workflow-process type as its superordinate workflow process. Since protection requirements of business cases must be enforced independently of the concepts used to model an organizational process, i.e., worksteps or nested workflow processes, authorization interfaces must be extended to cope with workflow-process boundaries. In fact, the predefined authorization interface All of a workflow process w' that realizes the specification of a nested workflow process n in a superordinate workflow process w is modeled as a subordinate authorization interface of that authorization interface f that determines the protection requirements of n in w. That is, authorization interface All defined at w' inherits actor domains defined for n at f in w.

Example: Figure 10.1 shows authorization interface HandlePayment defined at workflow-process type ROOM-RESERVATION-WF. It contains the nested workflow process HandlePayment. Financial clerks and secretaries may be authorized to invoke worksteps of the nested workflow process HandlePayment. That is, some authorization constraint specifies the type domain of potentially authorized actors at authorization interface HandlePayment by the type expression $\mathsf{FINANCE\_CLERK}_{\mathsf{org}} \vee \mathsf{SECRETARY}_{\mathsf{org}}$.

Note: The specification of the room-reservation workflow process of workflow-process type ROOM-RESERVATION-WF is shown only partially in Figure 10.1. For a complete specification refer to Figure 10.5.

Figure 10.2 shows authorization interface All defined at workflow-process type ACCOUNT-WF. It inherits actor domains from authorization-interface HandlePayment that is defined at workflow-process type ROOM-RESERVATION-WF, the type of the superordinate room-reservation workflow process (cf. Figure 10.1).

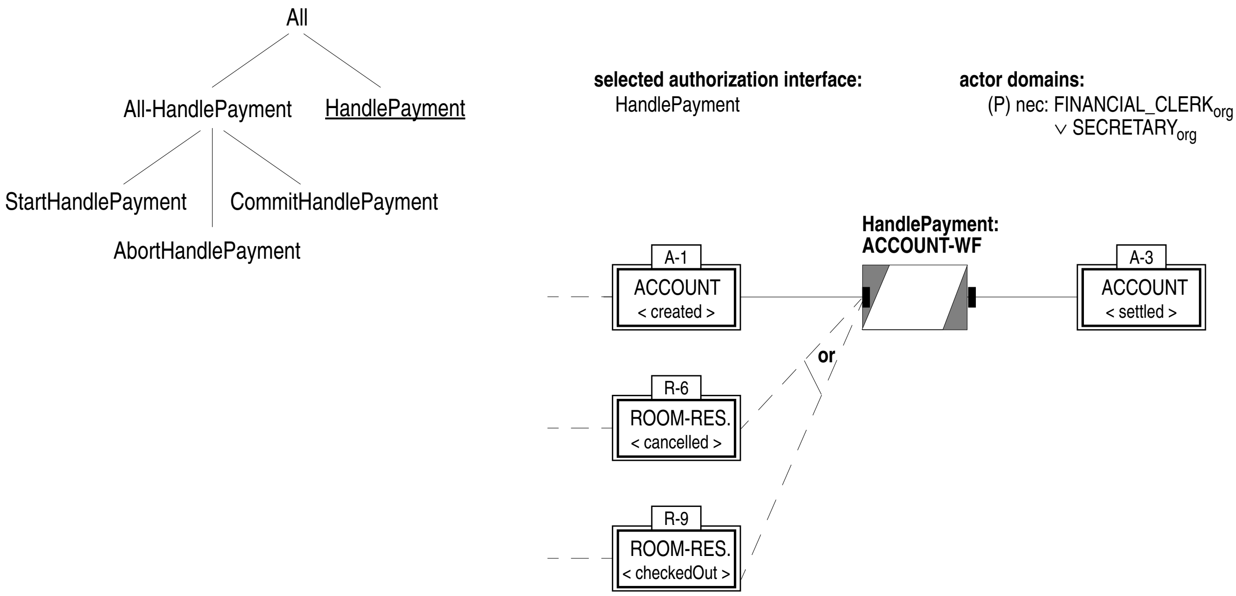


Figure 10.1: Authorization interface HandlePayment at ROOM-RESERVATION-WF

Specialization of Workflow Processes

Workflow-process types are organized in type hierarchies (cf. Subsection 8.3.4). A subtype may specialize structure, behavior, as well as the workflow process it inherits from its supertype. A workflow process may be specialized either by extension, i.e., boxes, worksteps and nested workflow processes added at a subtype extend the inherited workflow process, or by refinement, i.e., boxes, worksteps, and nested workflow processes added at a subtype refine inherited boxes and worksteps. In either case, the protection requirements of the worksteps and nested workflow processes added at a subtype must be determined. To enforce the protection requirements of an inherited workflow process requires to properly organize authorization interfaces for worksteps and nested workflow processes added at a subtype into the specialization hierarchy of the set of inherited authorization interfaces.

If a workstep s (nested workflow process n) added at a subtype *extends* an inherited workflow process w, then by default the protection requirements of workstep s (nested workflow process n) are determined by the inherited authorization interface All. This means, authorization interface S of workstep s (N of nested workflow process n) is modeled

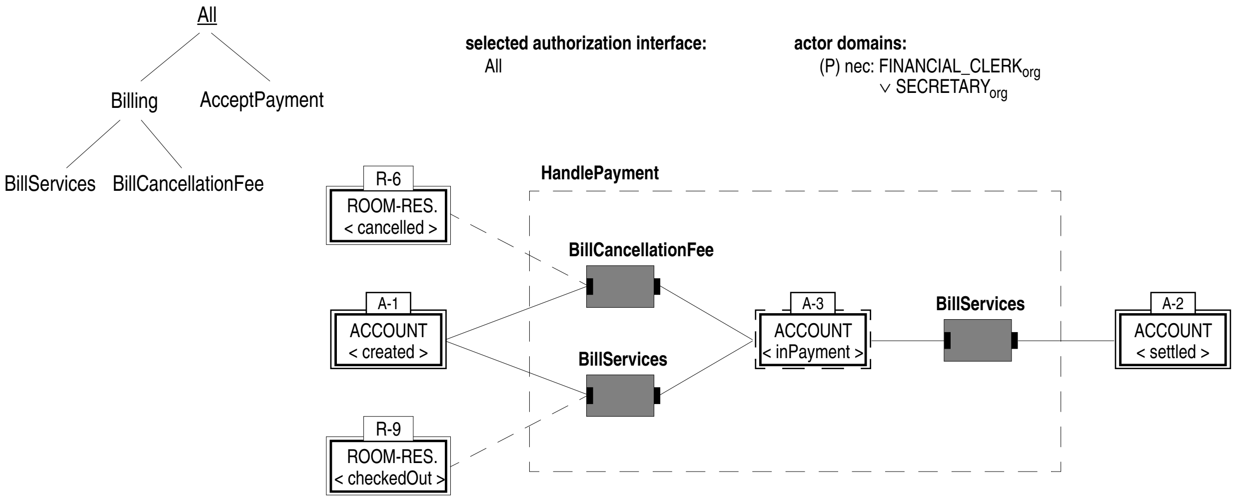


Figure 10.2: Authorization interface All at ACCOUNT-WF

as direct subordinate of the inherited authorization interface All. A security designer may strengthen the system default in that it models authorization interface S of workstep s (N of nested workflow process n) as subordinate of an authorization interface that is direct or indirect subordinate to All.

Example: Figure 10.3 shows authorization interface Finalize defined at workflow-process type RESERVATION-WF. It contains a single workstep Finalize.

Workflow-process type RESERVATION-WF' extends the workflow process it inherits from its supertype RESERVATION-WF by adding worksteps Reject, Apologize, and SaySorry. Figure 10.4 shows authorization interface Finalize at workflow-process type RESERVATION-WF'. The responsible security designer decided to overrule the system default. That is, instead of modeling the authorization interfaces of workstep Apologize and workstep SaySorry as direct subordinates of the inherited authorization interface All, the security designer did model authorization interfaces Apologize and SaySorry as a direct subordinates of the inherited authorization interface Finalize (cf. Figure 10.3). Furthermore, the security designer did model authorization interface Reject as direct subordinate of the inherited authorization interface HandleReservations.

If a workstep s (nested workflow process n) added at a subtype *refines* an inherited element e, i.e., a box or a workstep, of an inherited workflow process w, then by default the protection requirements of workstep s (nested workflow process n) are determined by the inherited authorization interface E of e. This means, authorization interface S of workstep s (N of nested workflow process n) is modeled as direct subordinate of the inherited authorization interface E. However, authorization interfaces do not consider boxes. In fact, the protection requirements of worksteps and nested workflow processes in the refinement of an inherited box b are determined by the authorization interface S' of an inherited workstep s' if b is in the refinement of s' and there is no inherited workstep s'' such that b is in the

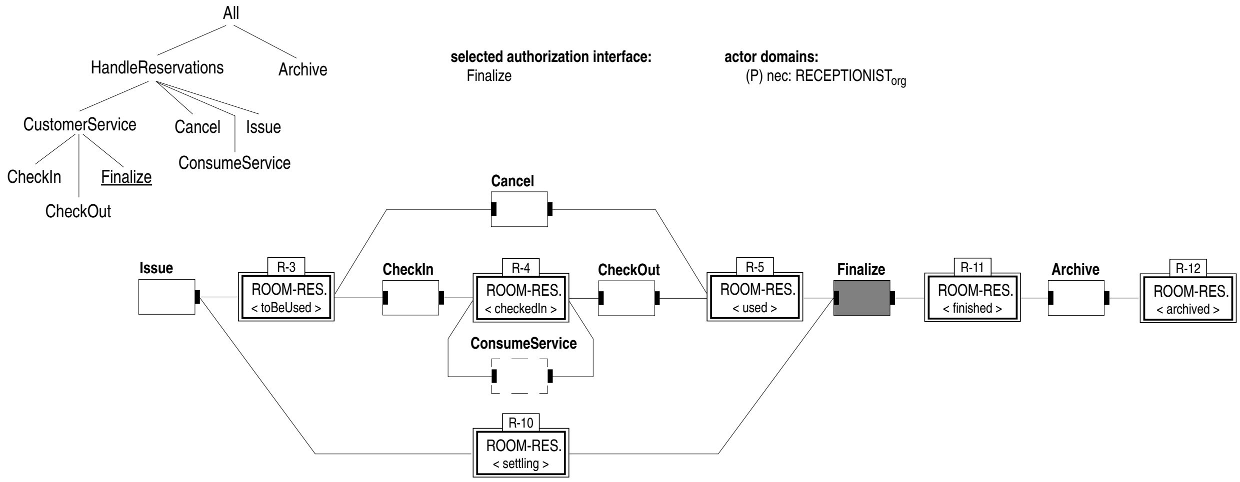


Figure 10.3: Authorization interface Finalize at RESERVATION-WF

refinement of s'' and s'' is in the refinement of s'. If there is no such workstep s', then the protection requirements of worksteps and nested workflow processes in the refinement of a box b are determined by the inherited authorization interface All. Again,a security designer may strengthen the system default in that it models the authorization interface S of workstep s (N of nested workflow process n) as subordinate of an authorization interface that is direct or indirect subordinate to E.

Example: Worksteps Request, Check&Decide, and Confirm added at workflow-process type RESERVATION-WF' refine the inherited workstep Issue, worksteps SaySorry&Thanks and SayThanks the inherited workstep Finalize (cf. Figure 10.4). By default, authorization interfaces Request, Check&Decide, and Confirm are subordinate to the inherited authorization interface Issue, authorization interfaces SaySorry&Thanks and SayThanks to the inherited authorization interface Finalize (cf. Figure 10.4).

10.2.2 Modeling of Authorization Constraints

An *authorization constraint* (AC) specifies an actor domain of a set of public operations that belong to the same authorization interface. In Subsection 6.3.2, we proposed a deontic constraint language L_{auth} to specify authorization constraints as normative statements on the authorization state of a business area. We start this subsection with a short summary of the basic features of L_{auth}.

- L_{auth} supports deontic modalities that allow to distinguish between permitted, obligatory, discretionary, and forbidden actor domains of public operations that are organized into authorization interfaces. Assuming a closed-world security policy for workflow systems (cf. Section 6.2), however, we mainly use the deontic modalities *permission* (**P**) and *obligation* (**O**).

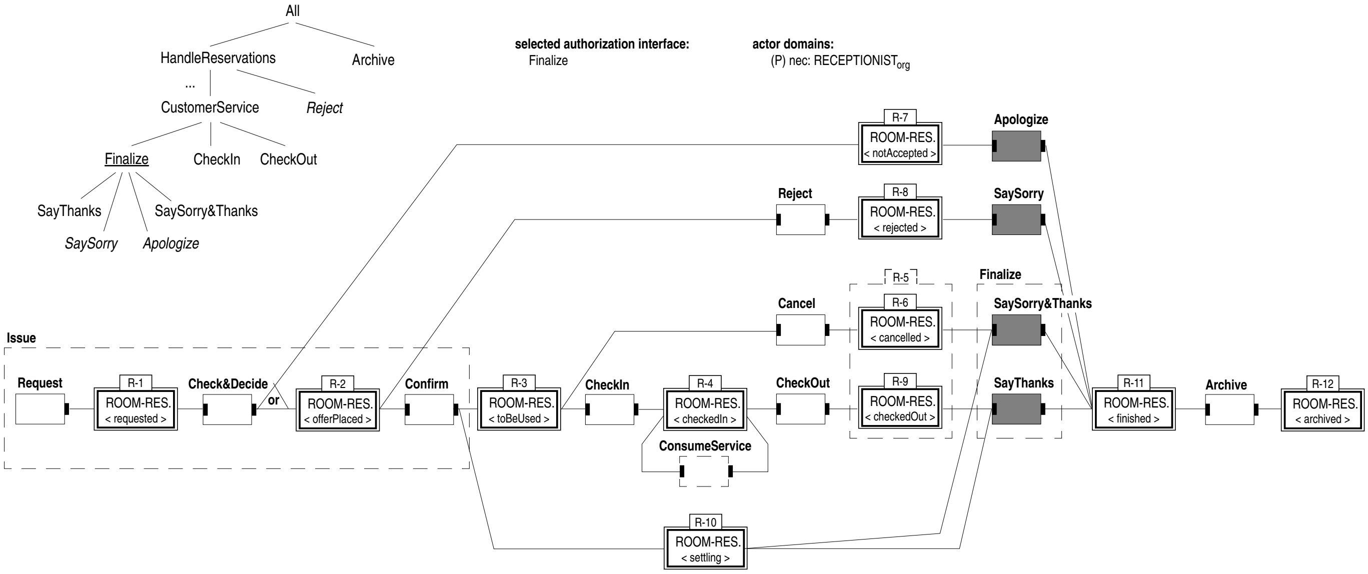


Figure 10.4: Authorization interface Finalize at RESERVATION-WF'

Example: AC 6.1 says that it is *permitted* to authorize receptionists that belong to class *Receptionists*, AC 6.2 says that it is *obligatory* to authorize receptionists that belong to a subclass *SeniorReceptionists* of class *Receptionists*, to invoke worksteps that belong to authorization interface *WaivePayment* on room reservations that are processed according to a reservation workflow-process that belongs to class *Reservation-WFs*.

- L_{auth} supports the modeling of *necessary domains* and *sufficient domains* of actors of public operations organized into a hierarchy of authorization interfaces. A necessary actor-domain D_{nec} defined at an authorization interface f says that actors requesting authorizations for operations that belong to f must belong to D_{nec}. Necessary actor-domains inherit downwards conjunctively, i.e., an actor domain D_{nec} defined at an authorization interface f can be strengthened at a subordinate authorization interface f' of f. A sufficient actor-domain D_{suff} defined at an authorization interface f says that all actors that belong to D_{suff} can obtain authorizations for operations that belong to f. Sufficient actor-domains inherit downwards disjunctively, i.e., an actor domain D_{suff} defined at an authorization interface f can be weakened at a subordinate authorization interface f' of f. This means, the potential of modeling sufficient actor-domains of public operations is the capability to limit the specialization of necessary actor-domains in a taxonomic network.

Example: AC 6.5 specifies position class *WfAdmins* as a sufficient domain of potentially authorized actors of worksteps that belong to authorization interface *Archive* for reservation workflow-processes that belong to class *Reservation-WFs*. As a consequence, the potential actor-domain *WfAdmins* must not be restricted at a subclass of the class of suppliers *Reservation-WFs*, at a subordinate authorization interface of the authorization interface of services *Archive*, or at a subclass of the class of clients *Public*. Consequently, AC 6.6 is in conflict with AC 6.5, since it restricts the sufficient actor-domain *WfAdmins*, inherited from authorization interface *Archive*, to a subclass *SeniorWFAs* of class *WfAdmins* for authorization interface *NegArchive* that is subordinate to authorization interface *Archive*.

- L_{auth} supports the modeling of *type domains* and *class domains* of actors of public operations. Type-level actor domains are specified by type expressions, class-level actor domains by class expressions. The use of type expressions and class expressions, respectively, to model actor domains allows for a flexible modeling of the protection requirements of objects, independently from a particular organization of classes (types) of agents and positions into a class hierarchies (type hierarchies).

  Example: AC 6.3 specifies a type-level actor domain of all worksteps defined at reservation workflow-processes that are instances of type *RESERVATION-WF* by type expression $GUEST \vee RECEPTIONIST \vee WFA$. AC 6.4 specifies the class-level actor domain of worksteps that belong to authorization interface *Cancel* defined for reservation workflow-processes that belong to class *Reservation-WFs* by class expression $VIP\text{-}Guests \cup Receptionists$.

In the advanced conceptual model, the concept of authorization constraints must be extended to cope with new modeling concepts such as type hierarchies, local classes, nesting of workflow processes, and specialization of workflow processes. Subsequently, we review the modeling of authorization constraints in the realm of the advanced modeling concepts and discuss necessary extensions and adaptions of L_{auth}.

- *Type Hierarchies*: The advanced conceptual model allows to organize object types into type hierarchies (cf. Section 7.2). As a consequence, inheritance and specialization of type-level authorization constraints is also supported in a taxonomic network of types of suppliers, i.e., types of workflow processes (cf. Chapter 8), and types of clients, i.e., types of agents and types of positions (cf. Chapter 9), too. Since the definitions on inheritance and specialization of authorization constraints in taxonomic networks given in Subsection 6.3.2 do abstract from a particular kind of taxonomy, these definitions apply to class hierarchies and type hierarchies in the same way. The only extension required is a redefinition of the rule for a consistent specialization of type expressions, considering organization of types into type hierarchies.

- *Local Classes*: The advanced conceptual model maintains objects as members of classes that are properties of composite objects. Such classes are called *local classes*. In fact, a local class collects objects as its members that are direct or indirect components of that composite object of which it is a property (cf. Subsection 7.2.3).

Specifying local classes as actor domains of public operations is a mechanism to represent the concept of *local-authorization integrity*[1]. A local-authorization integrity constraint requires(depending on the deontic modality it associates with the specified actor domain) that *a client may be, must be, may not be, or must not be authorized to request a certain service from a supplier if both the client as well as the supplier (directly or indirectly) belong to the same organizational unit or organization* (cf. Section 10.1).

Local-authorization integrity constraints are modeled by authorization constraints that specify class-level actor domains of public operations. We denote these authorization constraints as *local-authorization constraints*. Local-authorization constraints require extensions to the syntax of L_{auth}, e.g., local classes must be addressed by path expressions starting from the object type at which local-authorization constraints are defined, and extensions to rules for inheritance and consistent specialization of authorization constraints.

Example: Local-authorization constraint AC 10.1 specifies a necessary actor-domain of worksteps that belong to authorization interface *Archive* defined for members of local workflow-object classes *Reservation-WF*. That is, it says that only workflow administrators of a hotel *may be* authorized to archive room reservations that are processed according to the room-reservation workflow process of a hotel. Furthermore, AC 10.1 says that a workflow administrator of a particular hotel h may be authorized to archive room reservations only if they are processed according to the room-reservation workflow process of the same hotel h. Contrast this semantics with that of AC 6.8. In fact, AC 6.8 would allow to authorize a workflow administrator to archive room reservations that are processed according to room-reservation workflow processes of any hotel.

$$\text{HOTEL:} \quad \forall s \in \textit{Reservation-WF}(), r \in \textit{Archive}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{Reception}().\textit{WfAdmins}_{org}()) \tag{10.1}$$

- *Nested Workflow Processes*: The advanced conceptual model allows a workflow process to delegate part of its duty to a nested workflow process (cf. Subsection 8.3.4). Since a nested workflow process is realized by a separate workflow process that is defined at a different workflow-process type than its superordinate workflow process, rules for inheritance and consistent specialization of authorization constraints must be extended to cope with workflow-process boundaries.

- *Specialization of Workflow Processes*: A workflow process w' may specialize another workflow process w adding a set of boxes, worksteps and nested workflow processes that *extend* w or *refine* boxes and worksteps inherited from w (cf. Subsection 8.3.4). If a workflow process w' specializes another workflow process w, w' inherits a set

[1]Note: Local-authorization integrity is an interpretation of the general concept of local referential integrity (cf. Subsection 7.2.3) in the context of authorization bases.

of authorization interfaces from w that determine the protection requirements of inherited worksteps and nested workflow processes. The protection requirements of worksteps and nested workflow processes added at a workflow process w' can be determined in that the authorization interfaces of these worksteps and nested workflow processes are properly organized into the hierarchy of the set of inherited authorization interfaces (cf. Subsection 10.2.1). No extensions to authorization constraints are required to cope with the specialization of workflow processes.

The remainder of this subsection is organized as follows. First, we define the extended syntax of L_{auth} and give a modeling example. Second, we discuss advanced aspects of inheritance and specialization of authorization constraints. Third, we discuss the graphical representation of authorization constraints. Finally, we propose a particular organization of authorization-constraint bases.

$\mathbf{L}_{auth}$ – A Language to Specify Authorization Constraints

L_{auth} is a deontic constraint language used to specify authorization constraints. The basic syntax of L_{auth} has already been presented in Subsection 6.3.2. Subsequently, we present extensions to L_{auth} that allow for a proper specialization of local-authorization constraints, i.e., authorization constraints that specify local classes as actor domains of public operations organized in authorization interfaces.

- A local-authorization constraint is specified at a composite-object type O, i.e., a type of organizational units[2] or the predefined type DATABASE, of which both the suppliers and the clients addressed in a local-authorization constraint are direct or indirect components. That is, typical sentences of L_{auth} expressing local-authorization constraints are, where $\delta \in \Delta$ is a deontic modality:

$$O\colon \quad \forall s \in S, r \in R, c \in C\,(\delta(E_A(s,r,c)) \Rightarrow c \in C')$$

$$O\colon \quad \forall s \in S, r \in R, c \in C\,(c \in C' \Rightarrow \delta(E_A(s,r,c)))$$

- Class domains are modeled by class expressions (cf. Subsection 6.3.2). However, since local classes are not global schema objects, but properties of composite objects, *local classes must be addressed using locality paths* (cf. specification of locality paths for local classes in class-domain constraints, Subsection 7.2.3). For example, if a local-authorization constraint g_l is defined at an object type O, then a locality path $c_1().c_2()$ addresses a local class c_2 defined for members of a local class c_1 that is defined for instances of O. Note: For each local class listed in a locality path, a member-type constraint may be specified in parentheses behind the name of a local class.

[2]Note: Overall organizations are considered as a special kind of organizational units.

A local-authorization constraint defined at an object type O is applied to every instance o_i of O. $S(o_i)$ identifies suppliers that are direct or indirect components of o_i and are contained in S, $C(o_i)$ identifies clients that are direct or indirect components of o_i and are contained in C, $C'(o_i)$ identifies clients that are direct or indirect components of o_i and are contained in C', where $C'(o_i) \subseteq C(o_i)$. Note: Authorization interfaces are interpreted as global classes of public operations.

- Letter v (usually indexed) represents path variables used in locality paths to keep the presentations of local-authorization constraints readable. If a local class c_4 is addressed by a locality path $c_1().c_2().c_3().c_4()$, a path variable v_1 could be bound to subpath $c_1().c_2().c_3()$ allowing to represent the original locality path as $v_1.c_4()$. Path variables are defined globally, i.e., the same path variable could be used in different locality paths. Whenever a set of local-authorization constraints is shown, first definitions of path variables used in locality paths by these authorization constraints are presented. Their presentation is marked with the keywords *path variables* (cf. example, below)

- Pseudo class Public is considered to be a local class that is defined for each organizational unit u as a union class of all local classes of agents and positions defined for u and all organizational units that are direct or indirect components of u.

Authorization constraints model type domains *and* class domains of actors of public operations that are organized in authorization interfaces. For consistency, we require that for each type-level actor domain defined at an authorization interface f an appropriate class-level actor domain is defined at f, too. As a consequence, *only class-level actor domains defined at authorization interfaces must be enforced to ensure that no illegal authorizations are issued* (cf. authorization-state integrity, Section 6.2). In Subsection 6.3.2 (cf. definition of syntax of L_{auth}), we gave a simple rule to automatically derive class-level authorization constraints from type-level authorization constraints. Basically, class-level authorization constraints are derived from type-level authorization constraints in that (a) names of types are replaced by names of the corresponding base classes and (b) each occurrence of the instanceOf operator (symbol $\odot$) is replaced by an occurrence of the memberOf operator (symbol $\in$). To automatically derive local-authorization constraints from type-level authorization constraints is more complicated, since locality paths must be computed to properly address base classes of types of suppliers and clients. We do not give an appropriate algorithm here. Nevertheless, we assume that for each type-level authorization constraint one or several corresponding local-authorization constraint are defined. Note: In the following, we specify only local-authorization constraints to keep examples small and understandable.

Example: The following set of local-authorization constraints determines the protection requirements of worksteps defined for room-reservation workflow processes of type RESERVATION-WF' (cf. Figure 10.4). Workflow objects of type RESERVATION-WF' are components of organizational units of type HOTEL. For the following examples, we assume that organizational-unit type HOTEL is a supertype of type EU-HOTEL (cf. Figure 9.10).

path variables:

$v_1 := Reception();$
$v_2 := FinanceManagement();$

$$\text{HOTEL:} \quad \forall s \in \textit{Reservation-WF}(), r \in \textit{All}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{Guests}() \cup v_1.\textit{Receptionists}_{org}() \cup v_1.\textit{WfAdmins}_{org}() \cup v_2.\textit{Clerks}_{org}() \cup v_2.\textit{Secretaries}_{org}()) \tag{10.2}$$

$$\text{HOTEL:} \quad \forall s \in \textit{Reservation-WF}(), r \in \textit{Archive}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_1.\textit{WfAdmins}_{org}()) \tag{10.3}$$

$$\text{HOTEL:} \quad \forall s \in \textit{Reservation-WF}(), r \in \textit{HandleReservations}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_1.\textit{Receptionists}_{org}() \cup \textit{Guests}()) \tag{10.4}$$

$$\text{HOTEL:} \quad \forall s \in \textit{Reservation-WF}(), r \in \textit{ConsumeService}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \textit{Guests}()) \tag{10.5}$$

$$\text{HOTEL:} \quad \forall s \in \textit{Reservation-WF}(), r \in \textit{CustomerService}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_1.\textit{Receptionists}_{org}()) \tag{10.6}$$

$$\text{HOTEL:} \quad \forall s \in \textit{Reservation-WF}(), r \in \textit{CustomerService}, c \in \textit{Public}$$
$$(\mathbf{O}(E_A(s,r,c)) \Rightarrow c \in v_1.\textit{Receptionists}_{org}()) \tag{10.7}$$

$$\text{HOTEL:} \quad \forall s \in \textit{Reservation-WF}(), r \in \textit{Check\&Decide}, c \in \textit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_1.\textit{Receptionists}_{org}()) \tag{10.8}$$

Inheritance of Authorization Constraints

Authorization constraints support *inheritance*. First, authorization constraints inherit downwards in a three-dimensional taxonomy of domains S, R, and C of potential suppliers, services, and clients, respectively. This form of inheritance has already been defined in Subsection 6.3.2. Although the advanced conceptual model allows to build richer taxonomic networks, i.e., class hierarchies *and* type hierarchies are supported, no extensions to the basic definition of inheritance of authorization constraints is required. Remember that the basic definition of inheritance of authorization constraints abstracts from a particular kind of taxonomy and thus applies to type hierarchies in the same way as to class hierarchies.

Second, authorization constraints inherit downwards in a taxonomy of workflow processes modeled by has-nested-workflow and nested-workflow-of relationships, respectively (cf. Chapter 8). This means, a workflow process w_2 that realizes the specification of a nested workflow process n in a superordinate workflow process w_1 inherits all authorization constraints that determine actor domains of n in w_1. In fact, authorization interface All defined at w_2 is subordinate to authorization interface f that determines the protection requirements of n in w_1 (cf. modeling of authorization interfaces, Subsection 10.2.1). Note: A workflow process w_1 *does not inherit* authorization constraints from a workflow process w_2 if w_1 has a has-independent-workflow relationship to w_2 or vice versa.

Example: Organizational-unit type EU-HOTEL (cf. Figure 9.10) restricts the inherited domain RESERVATION-WF of has-component relationship reservation-wf to ROOM-RESERVATION-WF (cf. Figure 8.7), which extends the inherited workflow process (cf. Figure 10.4) by adding workstep CheckOut_Charge_Bill&Pay as well as nested workflow process HandlePayment (cf. Figure 10.5). Local-authorization constraint AC 10.9 has been defined at organizational-unit type EU-HOTEL. It specifies a necessary domain of potentially authorized actors for nested workflow process HandlePayment, which is the only member of authorization interface HandlePayment. The workflow process defined at type ACCOUNT-WF (cf. Figure 10.2) realizes the nested workflow process HandlePayment. As a consequence, the predefined authorization interface All defined for worksteps of the account workflow-process inherits authorization constraints (and thus actor domains) that are defined for nested workflow process HandlePayment in the superordinate room-reservation workflow process. AC 10.10 shows AC 10.9 inherited by the predefined authorization interface All of account workflow-processes of type ACCOUNT-WF (cf. Figure 10.2).

path variables:

$v_2 := \mathit{FinanceManagement}();$

$$\text{EU-HOTEL:} \quad \forall s \in \mathit{Reservation\text{-}WF}(), r \in \mathit{HandlePayment}, c \in \mathit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_2.\mathit{Clerks}_{org}() \cup v_2.\mathit{Secretaries}_{org}()) \tag{10.9}$$

$$\text{EU-HOTEL:} \quad \forall s \in v_2.\mathit{Account\text{-}WF}(), r \in \mathit{All}, c \in \mathit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_2.\mathit{Clerks}_{org}() \cup v_2.\mathit{Secretaries}_{org}()) \tag{10.10}$$

Third, local-authorization constraints inherit downwards in a taxonomy of types of organizational units. Remember that local-authorization constraints are defined at types of organizational units, which may be organized in type hierarchies, too (cf. Chapter 9). That is, a type of organizational units inherits structure, behavior, *and* local-authorization constraints from its supertype. Of course, a type of organizational units may specialize inherited local-authorization constraints (cf. specialization of authorization constraints, below).

Example: Organizational-unit type EU-HOTEL (cf. Figure 9.10) is a subtype of type HOTEL. Thus, EU-HOTEL inherits ACs 10.2 to 10.8 from its supertype. For example, AC 10.11 shows AC 10.2 inherited by EU-HOTEL.

path variables:

$v_1 := Reception();$

$$\begin{aligned}\text{EU-HOTEL:}\quad & \forall s \in \mathit{Reservation\text{-}WF}(), r \in \mathit{All}, c \in \mathit{Public} \\ & (\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \mathit{Guests}() \cup v_1.\mathit{Receptionists}_{org}() \\ & \qquad \cup v_1.\mathit{WfAdmins}_{org}())\end{aligned} \tag{10.11}$$

Specialization of Authorization Constraints

Authorization constraints support *specialization*. We distinguish two different dimensions of the specialization of authorization constraints: First, *inherited authorization constraints can be specialized*. We distinguish three basic forms of inheritance of authorization constraints (cf. inheritance of authorization constraints, above). For each form of inheritance, we discuss individual criteria for a consistent specialization of inherited authorization constraints. Second, *local-authorization constraints can be specialized along the hierarchy of composite-object types*. Below, we define a numeric measure for local-authorization constraints, called local restrictiveness, that determines whether a local-authorization constraint has been specified at the most-specific type possible. We start the discussion on the specialization of authorization constraints with presenting criteria for a consistent specialization of inherited authorization constraints.

First, authorization constraints inherit downwards in a three-dimensional taxonomy of domains S, R, and C of suppliers, services, and clients, respectively. In Subsection 6.3.2 (cf. specialization of authorization constraints), a set of criteria for a consistent specialization of authorization constraints have been defined. This set of consistency criteria must be extended to ensure a consistent specialization of local-authorization constraints. In fact, a local-authorization constraint h_{nec} (h_{suff}) defined on domains of suppliers, services, and clients S_2, R_2, and C_2, respectively, specializes another local-authorization constraint g_{nec} (g_{suff}) defined on domains S_1, R_1, and C_1, if and only if consistency criteria (1) to (5) are satisfied (cf. specialization of authorization constraints, Subsection 6.3.2) and, in addition, the following criterion is satisfied:

6. h_{nec} (h_{suff}) and g_{nec} (g_{suff}) are defined at the same type of organizational unit O.

Example: Local-authorization constraints AC 10.3 to 10.8 specialize AC 10.2. Furthermore, local-authorization constraints AC 10.5 and AC 10.6 specialize AC 10.4 (cf. specialization hierarchy of authorization interfaces in Figure 10.4).

Note: Since the advanced conceptual model allows to organize types of suppliers, services, and clients into specialization hierarchies, too, the rule of consistent specialization of type domains must be adapted accordingly. That is, a type domain P_2 specializes another type domain P_1, written as $P_2 < P_1$, if the type expression $\hat{t_2}$ that denotes P_2 consists of a proper subset of the types of that type expression $\hat{t_1}$ that denotes P_1. In addition, $\hat{t_2}$ may replace any type O in $\hat{t_1}$ by a set of subtypes of O. For example, $P_2 := O_{11} \vee O_{12}$

specializes $P_1 := O_1 \vee O_2$, where O_{11} and O_{12} are direct subtypes of O_1 and neither O_{11} is a subtype of O_{12} nor O_{12} a subtype of O_{11}. Note: A legal type expression must not contain two types O_1 and O_2, where either O_2 is a direct or indirect subtype of O_1 or vice versa.

Second, authorization constraints inherit downwards in a taxonomy of workflow processes modeled by has-nested-workflow and nested-workflow-of relationships, respectively. In Subsection 6.3.2 (cf. specialization of authorization constraints), a set of criteria for a consistent specialization of authorization constraints have been given. This set of consistency criteria must be modified to ensure a consistent specialization of authorization constraints. In fact, consistency criterion (1) must be relaxed to ensure a disciplined specialization of authorization constraints over workflow-process boundaries. That is, an authorization constraint h_{nec} (h_{suff}) defined on domains of suppliers, services, and clients S_2, R_2, and C_2, respectively, specializes another authorization constraint g_{nec} (g_{suff}) defined on domains S_1, R_1, and C_1, if and only if consistency criteria (1) to (5) (and consistency criterion (6) for local-authorization constraints) are satisfied (cf. specialization of authorization constraints, Subsection 6.3.2, and definition of consistency criterion (6), above), where consistency criterion (1) is redefined as follows:

1. S_2 is identical to S_1, or S_2 specializes S_1, or workflow processes that are elements of S_2 have a nested-workflow-of relationship to workflow processes that are elements of S_1

Example: AC 10.12 specializes AC 10.9 in that it restricts the inherited actor domain to $v_2.Clerks_{org}() \cup v_2.SeniorSecretaries_{org}()$, where $\mathsf{SeniorSecretaries_{org}}$ is a direct subclass of class $\mathsf{Secretaries_{org}}$. Note: Account workflow-processes that are members of local class Account-WF of the finance-management groups of hotels of type EU-HOTEL (cf. Figure 9.10) have a nested-workflow-of relationship to room-reservation workflow processes being members of local classes Reservation-WF of hotels of type EU-HOTEL (cf. Figures 8.7 and 8.8). Furthermore, authorization interface All defined for account workflow-processes is subordinate to authorization interface HandlePayment defined for room-reservation workflow processes (cf. Figures 10.1 and 10.2).

path variables:
$v_2 := FinanceManagement();$

$$\text{EU-HOTEL:} \quad \forall s \in v_2.Account\text{-}WF(), r \in All, c \in Public$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_2.Clerks_{org}() \cup v_2.SeniorSecretaries_{org}()) \tag{10.12}$$

Third, local-authorization constraints inherit downwards in a taxonomy of types of organizational units. In Subsection 6.3.2 (cf. specialization of authorization constraints) a set of criteria for a consistent specialization of authorization constraints have been defined, which have been extended for the specialization of local-authorization constraints in a three-dimensional taxonomy of domains S, R, and C of suppliers, services, and clients, respectively (cf. consistency criterion (6), above). This set of consistency criteria must be modified to ensure a consistent specialization of local-authorization constraints in a taxonomy of types of organizational units. In fact, consistency criterion (6) must be relaxed. That is, a local-authorization constraint h_{nec} (h_{suff}) defined on domains of suppliers, services, and clients S_2, R_2, and C_2, respectively, specializes another local-authorization constraint g_{nec} (g_{suff}) defined on domains S_1, R_1, and C_1, if and only if consistency criteria (1) to (6) are satisfied (cf. specialization of authorization constraints, Subsection 6.3.2, as well as redefinition of consistency criterion (1) and definition of consistency criterion (6) above), where consistency criterion (6) is redefined as follows:

6. h_{nec} (h_{suff}) is defined at a type of organizational units O', g_{nec} (g_{suff}) is defined at a type of organizational unit O, and either O' is identical to O or O' specializes O.

Example: Organizational-unit type EU-HOTEL (cf. Figure 9.10) inherits ACs 10.2 to 10.8 from its supertype HOTEL (cf. inheritance of authorization constraints, above).

EU-HOTEL specializes AC 10.4 to AC 10.13 in that it restricts the inherited actor domain of worksteps that belong to authorization interface HandleReservations from $Receptionists_{org}() \cup Guests()$ to $Receptionists_{org}() \cup VIP\text{-}Guests()$, where VIP-Guests is a direct subclass of local class Guests defined for hotels of type EU-HOTEL (cf. Figure 9.10).

Specializing AC 10.4 to AC 10.13 requires to specialize inherited AC 10.5 to AC 10.14, too. Assuming that AC 10.15 is defined at type EU-HOTEL, a conflict would arise if AC 10.4 and AC 10.5 would have been specialized to AC 10.13 and AC 10.14, respectively. In fact, AC 10.15 specifies local class Guests as a sufficient domain of potentially authorized actors for worksteps belonging to authorization interface ConsumeService. Restricting the necessary domain of potentially authorized actors at authorization interface ConsumeService to local class VIP-Guests, which is a subclass of local class Guests, would thus be illegal.

path variables:
$v_1 := Reception();$

$$\text{EU-HOTEL:}\quad \forall s \in Reservation\text{-}WF(), r \in HandleReservations, c \in Public$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_1.Receptionists_{org}() \cup VIP\text{-}Guests()) \tag{10.13}$$

$$\text{EU-HOTEL:}\quad \forall s \in Reservation\text{-}WF(), r \in ConsumeService, c \in Public$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in VIP\text{-}Guests()) \tag{10.14}$$

$$\text{EU-HOTEL:}\quad \forall s \in Reservation\text{-}WF(), r \in ConsumeService, c \in Public$$
$$(c \in Guests() \Rightarrow \mathbf{P}(E_A(s,r,c))) \tag{10.15}$$

Next, we discuss criteria for a consistent specialization of local-authorization constraints along the hierarchy of composite-object types. We start the presentation with a discussion on the restrictiveness of local-authorization constraints.

The restrictiveness of a local-authorization constraint g depends on the object type at which g has been defined. In general, authorizations that permit clients of type O_c to request services from suppliers of type O_s can be constrained by a local-authorization constraint g defined at a composite-object type O if clients of type O_c and suppliers of type O_s are *direct or indirect* components of instances of O, where composite-object type O is either the predefined object type DATABASE or a type of organizational units. That is, there may be different types of composite objects at which local-authorization constraints on certain authorizations can be defined, where two local-authorization constraints specified on the same kind of authorizations at different types of composite objects have different restrictiveness. The *least specific type* at which a local-authorization constraint on some kind of authorizations can be specified is the predefined object type DATABASE, since all objects are (direct or indirect) components of the database object[3]. The *most specific type* at which a local-authorization constraint on authorizations that permit clients of a particular type O_c to request services from suppliers of a particular type O_s may be specified is that composite-object type O at which a local-authorization constraint on such authorizations can be specified (i.e., clients of type O_c and suppliers of type O_s are (direct or indirect) components of instances of O), such that there is no other composite-object type O' at which a local-authorization constraint on such authorizations can be specified, and instances of O' are direct or indirect components of instances of O.

The *local restrictiveness* of a local-authorization constraint g on authorizations that permit clients of a particular type to request services from suppliers of a particular type is a numeric measure that determines whether g has been specified at the most specific type at which a local-authorization constraint on such authorizations could have been specified. Based on the observation that the locality paths that identify the classes of clients and the class of suppliers, respectively, of a local-authorization constraint g on certain authorizations share a common prefix if g is not specified at the most specific type at which a local-authorization constraint on such authorizations could have been specified, we compute the local restrictiveness of a local-authorization constraint g from the length of the common prefix of the locality paths that identify the classes of clients and the class of suppliers in g. A local restrictiveness of zero of a local-authorization constraint g on some kind of authorizations, i.e., $\triangle(g) = 0$, says that g has been specified at the most specific type at which a local-authorization constraint on such authorizations could have been specified.

Example: AC 10.16 has been defined at organizational-unit type EU-STAY (cf. Figure 9.11). Instances of type EU-STAY are hotel chains consisting of sets of hotels of type EU-HOTEL (cf. Figure 9.10).

AC 10.16 says that each receptionist that belongs to a hotel of a particular hotel chain *may be* authorized to invoke worksteps that belong to authorization inter-

[3]Note: The database object is the only instance of the predefined object type DATABASE and is the root of the composite-object hierarchy into which all objects stored in a database are organized according to their component-of relationships (cf. Subsection 7.2.1).

face CustomerService on room reservations that are processed according to a room-reservation workflow process of any hotel of the same hotel chain. This means, AC 10.16 allows a security administrator to authorize a receptionist of a hotel "Five Wings" to handle room reservations that are processed according to the room-reservation workflow process of a hotel "Seven Nuggets" if both hotels belong to the same hotel chain.

The local restrictiveness of AC 10.16 is one, i.e., $\triangle(AC\ 10.16) = 1$. There may be a local-authorization constraint with a local restrictiveness of zero that specializes AC 10.16.

path variables:

$v_3 := Hotels();$

$$\text{EU-STAY:}\quad \forall s \in v_3.\mathit{Reservation\text{-}WF}(), r \in \mathit{CustomerService}, c \in \mathit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in v_3.\mathit{Reception}().\mathit{Receptionists}_{org}()) \tag{10.16}$$

Given local-authorization constraints g_{nec} and g_{suff} defined at some object type O, where O is the predefined object type DATABASE or a type of organizational units, and authorization constraints h_{nec} and h_{suff} defined at a type of organizational units O', where O' is not a subtype of O or vice versa. Furthermore, the set of deontic modalities $\Delta := \{\mathbf{P}, \mathbf{O}, \mathbf{D}, \mathbf{F}\}$ and the deontic modality $\delta \in \Delta$ are given, where δ is fixed for pairs of authorization constraints g_{nec} (g_{suff}) and h_{nec} (h_{suff}).

$$\begin{array}{lll}
(g_{nec}) & O': & \forall s \in S_1, r \in R_1, c \in C_1\ (\delta(E_A(s,r,c)) \Rightarrow c \in C'_1) \\
(h_{nec}) & O': & \forall s \in S_2, r \in R_2, c \in C_2\ (\delta(E_A(s,r,c)) \Rightarrow c \in C'_2) \\
(g_{suff}) & O': & \forall s \in S_1, r \in R_1, c \in C_1\ (c \in C'_1 \Rightarrow \delta(E_A(s,r,c))) \\
(h_{suff}) & O': & \forall s \in S_2, r \in R_2, c \in C_2\ (c \in C'_2 \Rightarrow \delta(E_A(s,r,c)))
\end{array}$$

A local-authorization constraint h_{nec} (h_{suff}) defined at type of organizational units O' on domains of suppliers, services, and clients S_2, R_2, and C_2, respectively, specializes a local-authorization constraint g_{nec} (g_{suff}) defined at an object type O on domains S_1, R_1, and C_1, where O is the predefined type DATABASE or another type of organizational units, if and only if

1. instances of O' are (direct or indirect) components of instances of O,
2. S_2 and S_1 denote the same local class(es) of suppliers or S_2 specializes S_1,
3. R_2 is identical to R_1 or R_2 specializes R_1,
4. C_2 and C_1 denote the same local class(es) of clients or C_2 specializes C_1,
5. C'_2 and C'_1 denote the same local class(es) of clients, or either C'_2 specializes C'_1 (necessary domain constraints) or C'_1 specializes C'_2 (sufficient domain constraints).

Example: AC 10.17 specializes AC 10.16. In fact, the local restrictiveness of AC 10.17 is zero, i.e., $\triangle(AC\ 10.17) = 0$. This means, AC 10.16 has been specified at the most specific type at which a local-authorization constraint on authorizations that permit receptionists to perform worksteps at room-reservation workflow processes could have been specified.

AC 10.17 says that each receptionist that belongs to a particular hotel may be authorized to invoke worksteps that belong to authorization interface CustomerService on room reservations that are processed according to the room-reservation workflow process of the same hotel. This means, AC 10.17 would prevent a security administrator from authorizing a receptionist of a hotel "Five Wings" to handle room reservations that are processed according to the room-reservation workflow process of any other hotel.

$$\text{EU-HOTEL:} \quad \forall s \in \mathit{Reservation\text{-}WF}(), r \in \mathit{CustomerService}, c \in \mathit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \mathit{Reception}().\mathit{Receptionists}_{org}()) \tag{10.17}$$

Furthermore, local-authorization constraints AC 10.18 and AC 10.19 have been defined at organizational-unit type FINANCE-MANAGEMENT-GROUP (FMGMT). Finance-management groups of this type are components of hotels of type EU-HOTEL (cf. Figure 9.10). AC 10.18 specializes AC 10.10 and AC 10.19 specializes AC 10.12. Note: AC 10.19 specializes AC 10.18, too.

$$\text{FMGMT:} \quad \forall s \in \mathit{Account\text{-}WF}(), r \in \mathit{All}, c \in \mathit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \mathit{Clerks}_{org}() \cup \mathit{Secretaries}_{org}()) \tag{10.18}$$

$$\text{FMGMT:} \quad \forall s \in \mathit{Account\text{-}WF}(), r \in \mathit{All}, c \in \mathit{Public}$$
$$(\mathbf{P}(E_A(s,r,c)) \Rightarrow c \in \mathit{Clerks}_{org}() \cup \mathit{SeniorSecretaries}_{org}()) \tag{10.19}$$

Graphical Specification of Authorization Constraints

Authorization constraints can be graphically represented in workflow-process diagrams, which have been extended for this purpose (cf. Subsection 6.3.2). If local-authorization constraints are depicted, *locality paths* are shown to properly address local classes of workflow objects as well as local classes of actors. If a depicted authorization interface contains worksteps that are implicitly defined to start, commit, or abort a *nested workflow process* n, a horizontal bar filled in light-grey color is shown below the graphical symbol representing n. This allows to distinguish whether a depicted authorization interface contains a nested workflow process n, or the basic worksteps implicitly defined to start, commit, and abort n, or both.

Example: Figure 10.5 is a graphical representation of local-authorization constraint AC 10.17. The selected authorization interface CustomerService contains among other worksteps the three basic worksteps StartHandlePayment, CommitHandlePayment, and AbortHandlePayment implicitly defined for nested workflow process HandlePayment.

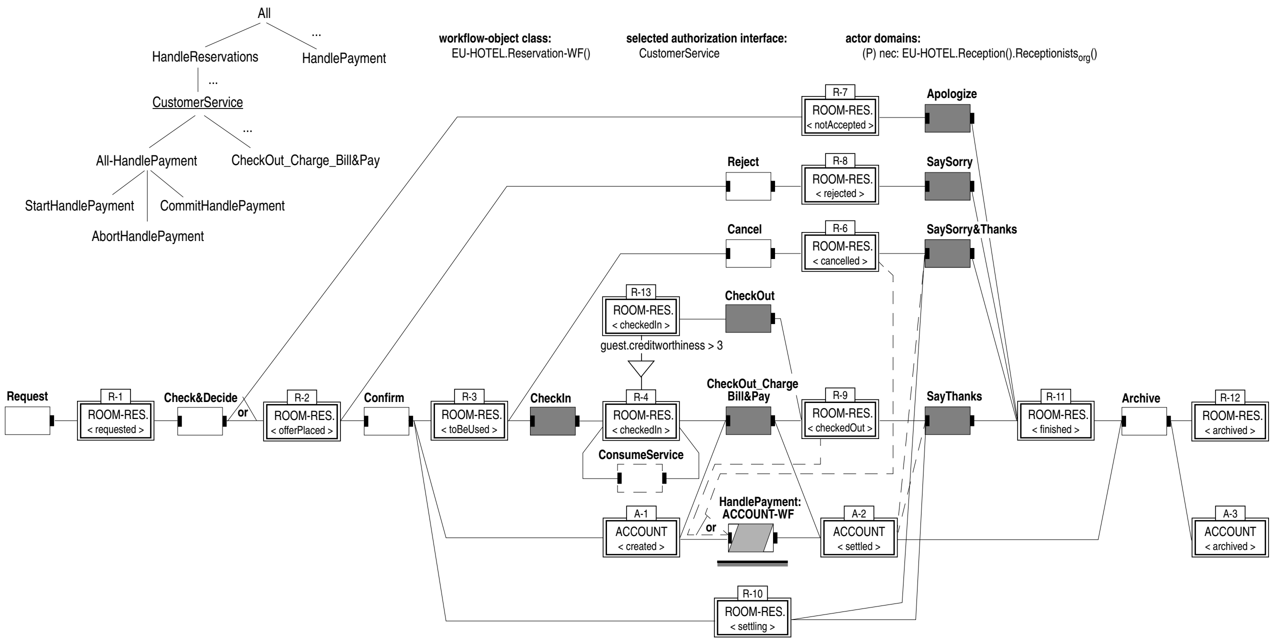


Figure 10.5: Authorization interface CustomerService at ROOM-RESERVATION-WF

Organization of the Authorization-Constraint Base

We maintain authorization constraints in an authorization-constraint base (ACB) that consists of a global authorization-constraint base (GACB) and a set of local authorization-constraint bases (LACBs).

- The *global authorization-constraint base GACB* of a particular ACB b contains all authorization constraints stored in b that determine type-level actor domains of public operations organized in authorization interfaces.

- A *local authorization-constraint base* $LACB_O$ of a particular ACB b contains all local-authorization constraints stored in b that have been defined at object type O, where object type O is either the predefined object type DATABASE or some organizational-unit type.

  Local authorization-constraint bases are organized into a *locality hierarchy* according to the physical structure of organizations (cf. physical structure of organizations, Chapter 9). The root of such a locality hierarchy is the predefined local authorization-constraint base $LACB_{DATABASE}$, since each object is direct or indirect component of the database object (cf. Subsection 7.2.3). In general, a local authorization-constraint base $LACB_{O'}$ is subordinate to a local authorization-constraint base $LACB_O$ if instances of O' are direct components of instances of O.

Example: Figure 10.6 shows the locality hierarchy of local authorization-constraint bases defined for the physical structure of organizations of type EU-CORPORATION (cf. Figure 9.12).

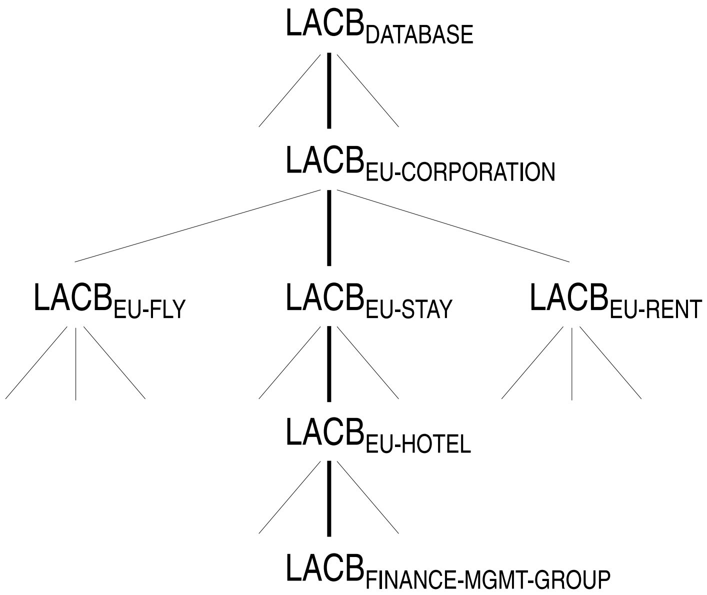


Figure 10.6: Locality hierarchy of LACBs

This organization of an overall authorization-constraint base has positive impact on system performance: First, since for each authorization constraint in $GACB$ there is a corresponding local-authorization constraint in some LACB (cf. implicit definition of local-authorization constraints, above), *only authorization constraints maintained in some LACB must be considered to ensure authorization-state integrity.* Second, since local authorization-constraint bases are organized in a locality hierarchy only a limited set of local-authorization constraints must be considered when legitimacy of an authorization to be issued must be tested. For example, assume we want to test legitimacy of an authorization to be issued to some client c, to request some service from some supplier s, and that the most specific organizational unit to which both c and s belong is of type O. Then only local-authorization constraints stored in $LACB_O$ and in superordinate LACBs of $LACB_O$ must be tested to ensure authorization-state integrity.

Example: A security administrator issues a new authorization for a guest of a hotel of type EU-HOTEL (cf. Figure 9.10) in that it adds a service to the guest card of the guest (cf. authorization-contract type GUEST-CARD, Figures 10.9(b), 6.10, and 10.12). To ensure authorization-state integrity, only local-authorization constraints stored in $LACB_{EU\text{-}HOTEL}$ and in (direct and indirect) superordinates thereof must be considered (cf. Figure 10.6).

Furthermore, the proposed organization of authorization-constraint bases has positive impact on their maintenance, since local-authorization constraints are organized around that types of organizational units where they must be applied to ensure authorization-state integrity.

10.2.3 Modeling of Authorization Contracts

An *authorization contract* is a formal agreement between two or more parties, where one or several parties, the *suppliers*, commit themselves to offer one or several *services* to the remaining parties, the *clients* (cf. Subsection 6.3.3). In this subsection, we review the advanced semantic modeling concepts introduced in Chapter 7, i.e., semantic relationships, local classes, and type hierarchies, in the realm of authorization contracts. Before we go into the details of modeling the structure, the behavior, and classes of authorization contracts, however, we start with a general discussion on the modeling of type hierarchies of authorization-contract types. In general, we require that specializations of object types into subtypes must be exclusive and total (cf. Section 7.1 and Subsection 7.2.1). Enforcing this strict semantics of type specialization is problematic in the context of authorization-contract types. We will shortly sketch the problems and propose a working solution.

Exclusive specialization of an object type O requires that each instance of O is instance of at most one subtype of O. Specializing an authorization-contract type $A_{S,R,C}$—we write $A_{S,R,C}$ as a shorthand to denote that authorization-contract type A is defined between domains S, R, and C of suppliers, services, and clients, respectively—is frequently driven by specific requirements of subdomains of participating suppliers, services, or clients. That is, a subtype of an authorization-contract type $A_{S,R,C}$ usually restricts either one or several of the domains S, R, and C to subdomains. For example, authorization-contract type A_{S,R,C_1} specializes the domain of clients C it inherits from its supertype $A_{S,R,C}$ to subdomain C_1. Specializing an authorization-contract type $A_{S,R,C}$ could result in an overlapping specialization that causes multi-instantiation of authorization contracts. For example, assume that supplier domain S has subdomains S_1 and S_2 and client domain C has subdomains C_1 and C_2. If authorization-contract type $A_{S,R,C}$ is specialized into subtypes $A_{S_1,R,C}$ and A_{S,R,C_1} (cf. Figure 10.7(a)), authorization contracts issued to suppliers of domain S_1, services of domain R, and clients of domain C_1 would be instances of type $A_{S_1,R,C}$ *and* A_{S,R,C_1}. The concept of multiple inheritance could be used to resolve conflicts that might arise from multi-instantiation of objects at the schema level (cf. Figure 10.7(b)). However, since each overlapping specialization of an object type could be transfered to an exclusive specialization that captures the same semantics (cf. Figure 10.7(c)), *we require that specializations of authorization-contract types are exclusive, too.*

Total specialization of an object type O requires that each instance of O is instance of at least one subtype of O. Specializing an authorization-contract type $A_{S,R,C}$ into a subtype A_{S_1,R,C_1} (cf. Figure 10.8(a)) results in a partial specialization of authorization-contract type $A_{S,R,C}$ if we assume that supplier domain S has subdomains S_1 and S_2 and client domain C has subdomains C_1 and C_2. That is, authorization contracts of type $A_{S,R,C}$ to be issued for suppliers and clients of domains S_1 and C_2, S_2 and C_1, and S_2

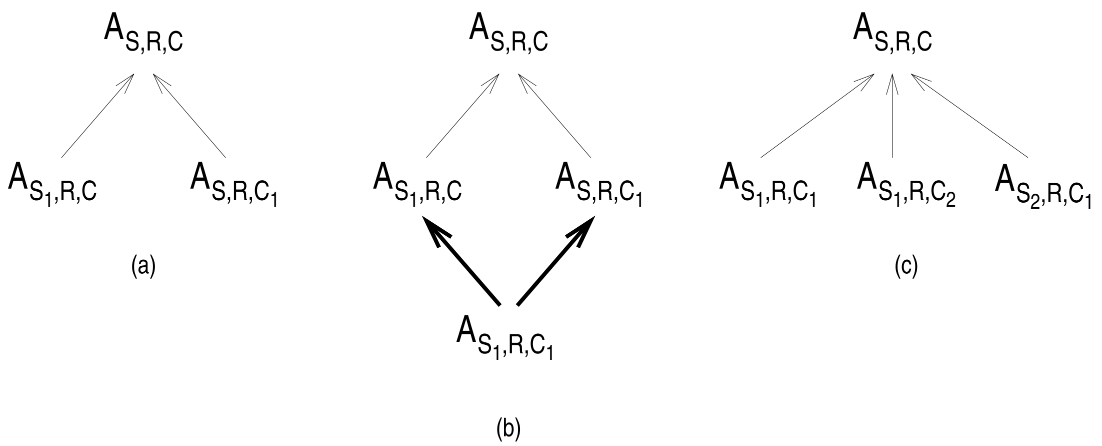


Figure 10.7: Exclusive specialization of authorization-contract types

and C_1, respectively, would not be instance of any subtype of $A_{S,R,C}$. Such conflicts with the requirement of total specializations of object types could be resolved in that further subtypes are introduced for supertypes having direct instances. To resolve the above conflict, authorization-contract types A_{S_1,R,C_2}, A_{S_2,R,C_1}, and A_{S_2,R,C_2} must be introduced as further subtypes of authorization-contract type $A_{S,R,C}$ (cf. Figure 10.8(b)). However, these new subtypes of authorization-contract type $A_{S,R,C}$, introduced to avoid potential conflicts with the requirement for total specialization of object types (cf. *Abstract Supertype Rule for Object Types*, Subsection 7.2.1), would be "empty" in the sense that they are simply clones of their supertype. In fact, requiring total specialization of authorization-contract types would result in a proliferation of authorization-contract types as indicated in Figure 10.8(b)). As a consequence, *we allow for partial specializations of authorization-contract types*[4].

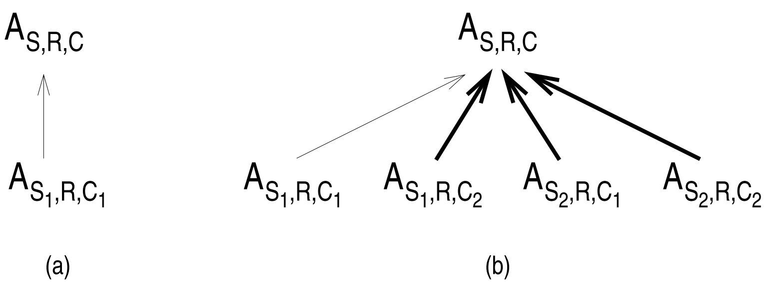


Figure 10.8: Partial specialization of authorization-contract types

Partial specialization of authorization-contract types allows that authorization contract are created as direct instances of supertypes. To ensure consistency, we require that security designers explicitly declare authorization-contract types to be abstract if no direct

[4]Note: This decision is in line with results of Hürsch presented in his work on the abstract-supertype-rule [Hür94]. Hürsch stressed that "the abstract supertype rule should be considered as a guideline rather than a strict rule" and identified situations in which it is even disadvantageous to follow it.

instances must be created from them[5]. Furthermore, we require that security administrators create an authorization contract for a set of suppliers s, a set of services r, and a set of clients c from the most specific authorization-contract type wrt. s, r, and c (*Most-Specific-Type Rule for Authorization Contracts*). The most specific authorization-contract type for a set of suppliers s, a set of services r, and a set of clients c is that authorization-contract type $A_{S,R,C}$ such that each supplier of s is an element of S, each service of r is an element of R, and each client of c is an element of C, and there is no subtype of $A_{S,R,C}$ satisfying these criteria, too. Considering the hierarchy of authorization-contract types depicted in Figure 10.8(a), the most specific authorization-contract type from which to instantiate an authorization contract for a set of suppliers of type S_1, a set of services from authorization interface R, and a set of clients of type C_1 is authorization-contract type A_{S_1,R,C_1}. Instantiating such an authorization contract from supertype $A_{S,R,C}$ violates the *Most-Specific-Type Rule for Authorization Contracts*. In fact, it would neglect specializations of $A_{S,R,C}$ defined especially for suppliers of type S_1 and clients of type C_1 at authorization-contract type A_{S_1,R,C_1}. Note: We assume that preconditions of activities that create new authorizations test adherence to the *Most-Specific-Type Rule for Authorization Contracts* (cf. modeling the behavior of authorization contracts, Subsection 10.2.3; also Figure 10.12).

The *Most-Specific-Type Rule for Authorization Contracts* implicitly separates the domains of the three designated relationships supplier(s), service(s), and client(s) defined at an authorization-contract type O into domains for direct instances of O and domains for indirect instances of O. Assume an authorization-contract type $A_{S,R,C}$ has been specialized into a subtypes A_{S_1,R,C_1} and A_{S_2,R,C_2}, where domains S, R, and C have subdomains S_1, S_2, and S_3, R_1 and R_2, and C_1, C_2, and C_3, respectively. In this case, the domains of relationships supplier(s), service(s), and client(s) are defined by expressions $S_1 \cup S_2$, R, and $C_1 \cup C_2$ for indirect instances of $A_{S,R,C}$ and by expressions $S \setminus (S_1 \cup S_2)$, R, and $C \setminus (C_1 \cup C_2)$ for direct instances of $A_{S,R,C}$. Note: If authorization-contract types would explicitly specify separate domains of the three designated relationships for their direct instances and for their indirect instances, changes to hierarchies of authorization-contract types, e.g., adding or removing subtypes, would require to recompute domains of these relationships at authorization-contract types affected by the changes.

Modeling the Structure of Authorization Contracts

The structure of authorization contracts is determined by a set of *attributes* and a set of *relationships* (cf. Section 6.3.3). In Subsection 7.2.1 advanced concepts of structure modeling, i.e., semantic relationships and type specialization, have been discussed. These concepts of structure modeling apply to the modeling of the structure of authorization contracts in the same way. Below, we review these concepts in the context of authorization contracts.

[5] Note: Supertypes of authorization contracts would *implicitly* be declared to be abstract if we would require total specialization of authorization-contract types.

Semantic Relationships. Using semantic relationships (cf. Subsection 7.2.1) to model references of authorization contracts to other entities, e.g., references to suppliers, services, or clients, allows to express precisely the semantic nature of such relationships in an authorization schema. For example, specifying an object-defining relationship r for authorization contracts of type O expresses that the existence of an instance of O depends on the existence of the object referenced via relationship r.

Example: The EU-Corporation decided to issue charge cards to customers of its divisions EU-Fly, EU-Rent, and EU-Stay (cf. Figure 9.12(b)).—A charge card authorizes a customer to consume certain services offered by the organization that did issue the charge card.—Instead of having each division issuing its own charge card, EU-Corporation decided to issue multi-function charge cards to its customers that can be used as frequent-flyer-card at EU-Fly, as frequent-driver-card at EU-Rent, and as guest cards at hotels of EU-Stay.

Figure 10.9(a) shows a type diagram of object type EU-CHARGE-CARD. A charge card is identified by the value of its key attribute cardNo and is described by the values of attributes expDate (expiration date) and isValid. A charge card is valid until its expiration date did pass by, i.e., the derivation expression of attribute isValid is expDate $\geq$ TODAY. Furthermore, a charge card has a has-constituent relationship holder to that person (cf. specification of type PERSON in Figure 9.2(a)) to whom the charge card has been issued. Has-role relationships asFrequentFlyerCard, asFrequentDriverCard, and asGuestCards model references to instances of authorization-contract types FREQUENT-FLYER-CARD (not shown), FREQUENT-DRIVER-CARD (not shown), and GUEST-CARD, respectively, which represent a charge card as frequent-flyer-card of EU-Fly, as frequent-driver-card of EU-Rent, and as guest cards of hotels of EU-Stay, respectively. Charge cards are components of the EU-Corporation (cf. component-of relationship with domain EU-CORPORATION).

Figure 10.9(b) shows a type diagram of authorization-contract type GUEST-CARD. An instance of authorization-contract type GUEST-CARD represents a charge card as guest card of a particular hotel of hotel chain EU-Stay (cf. component-of relationship with domain EU-HOTEL). That is, a guest card has a role-of relationship to the charge card it represents. Furthermore, a guest card has a general relationship supplier to a room-reservation workflow object, general relationship services to several worksteps of the room-reservation workflow process that belong to authorization interface HandleReservations (cf. class-domain constraint on relationship services of guest cards in Figure 10.15), and has-constituent relationship client to an instance of agent type GUEST (cf. Figure 9.2(c)) that represents the holder of the corresponding charge card as guest of that hotel that issued the guest card. Modeling relationship client as a has-constituent relationship says that the existence of a guest card depends on the existence of the referenced guest object. Note: Has-constituent relationships are object-defining relationships (cf. Subsection 7.2.1).

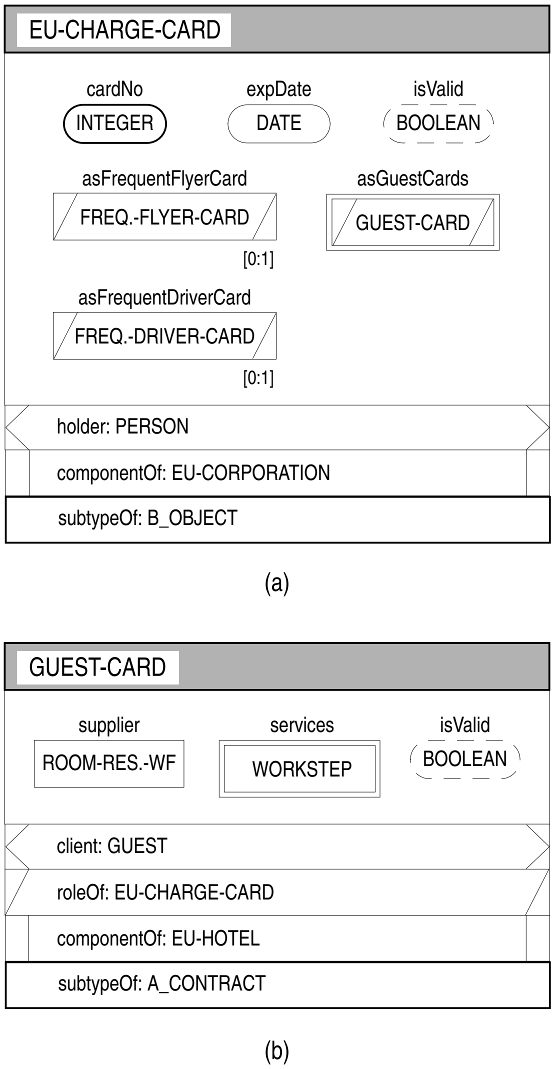


Figure 10.9: Object type EU-CHARGE-CARD and authorization-contract type GUEST-CARD

Note: Authorization contracts are modeled as components of organizations or organizational units of which both the suppliers and the clients are direct or indirect components.

Specialization of the Structure of Authorization Contracts. To specialize the structure of authorization contracts means, to specialize the set of *attributes*, the set of *relationships*, and the set of *local classes* authorization-contract types inherit from their supertypes. In fact, each authorization-contract type defined by a security designer is a (direct or indirect) subtype of the predefined conceptual type A_CONTRACT (Authorization CONTRACT, cf. Figure 7.6), which models the basic structure each authorization contract offers to its environment. That is, each authorization contract is described by the value of a derived, boolean attribute isValid (cf. Figure 10.10). For unconditional authorizations, the derivation expression of attribute isValid is true, i.e., solely the existence of an authorization contract legitimates a request of a client. For conditional authorizations, the derivation expression of attribute isValid models the additional conditions that constrain the situations in which a client of an authorization contract a of type A may request a service given at a from a supplier of a. Subtypes of A_CONTRACT may specialize inherited structure. Note: Criteria for a consistent specialization of the structure of objects have been discussed in Subsection 7.2.1.

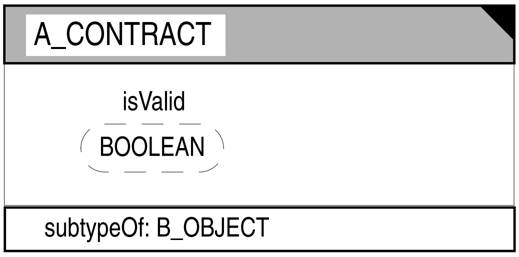


Figure 10.10: Conceptual type A_CONTRACT

Each authorization contract relates one or several suppliers to one or several clients, stating that the client(s) might request one or several services from the supplier(s) (cf. Subsection 6.3.3). Why then did we not specify the designated relationships supplier, service, and client at conceptual type A_CONTRACT? The answer is given by the words "one or several" denoting the multiplicity of the three designated relationships supplier, service, and client of authorization contracts, i.e., these relationships are either single-valued or multi-valued. In fact, these relationships are named suppliers, services, and clients, respectively, if they are multi-valued. Remember that the *Rule of Specialization of the Multiplicity of Properties* (cf. Subsection 7.2.1) requires that the original multiplicity of a property must not be changed when the property is redefined. If we would specify the three designated relationships of authorization contracts at conceptual type A_CONTRACT, we would have to prescribe certain multiplicities of these relationships for *all* authorization contracts, i.e., all authorization contracts would either be individual authorization contracts or some sort of collective authorization contracts. Instead, we require that each concrete subtype of the conceptual type A_CONTRACT defines the three designated relationships of authorization contracts with appropriate multiplicities for its instances. For individual authorization contracts, the three designated relationships are single-valued. For collective authorization contracts, one or several of them are multi-valued.

Example: Authorization-contract type GUEST-CARD (cf. Figure 10.9(b)) is a direct subtype of conceptual type A_CONTRACT. It defines the three designated relationships for its instances as follows: Relationship supplier is a single-valued general relationship, relationship services a multi-valued general relationship, and relationship client a (single-valued) has-constituent relationship. For a full specification of the structure of guest cards refer to Figure 10.9(b).

In large hierarchies of authorization-contract types, naming of individual types tends to be a headache for security designers. Since specification and specialization of structure and behavior of an authorization-contract type is driven by the requirements of certain domains of suppliers, services, and clients, we propose a naming schema that derives a unique name for an authorization-contract type A given a user-defined prefix and the class domains for the three designated relationships of authorization contracts of type A[6]. For example, assume an authorization-contract type A has been defined for suppliers, services, and clients being members of classes S, R[7], and C, respectively. From the user-defined prefix CONTRACT we would derive the unique type name $\mathsf{CONTRACT}_{\mathsf{S(M),R(M),C(S)}}$ if authorization-contract type A defines multi-valued relationships suppliers and services and a single-valued relationship client for its instances. Letters 'M' (multi-valued) and 'S' (single-valued) indicate the multiplicity of a relationship. Considering the multiplicity of the three designated relationships is necessary when deriving the name of an authorization-contract type, since individual as well as collective authorization contracts may be defined for the same kinds of suppliers, services, and clients.

Example: Figure 10.11(a) shows a type diagram of abstract authorization-contract type CONTRACT. Each authorization contract of this type has a single-valued relationship supplier to a value from domain ROOM-RESERVATION-WF and a multi-valued relationship services to values from domain WORKSTEP. The class domains of relationships supplier and services are defined by classes Reservation-WF and All, respectively (cf. Figure 10.16). Furthermore, authorization-contract type CONTRACT is a direct subtype of the predefined conceptual type A_CONTRACT and its instances are maintained as components of the reception of a hotel (cf. component-of relationship with domain RECEPTION).

Authorization-contract type $\mathsf{CONTRACT}_{\mathsf{Reservation\text{-}WF(S),Archive(M),WfAdmins_{org}(M)}}$ (cf. Figure 10.11(b)) is a direct subtype of CONTRACT. It extends its supertype by adding relationship clients with type domain $\mathsf{WFA}_{\mathsf{org}}$. Furthermore, it defines the class domain of relationship clients by class $\mathsf{WfAdmins}_{\mathsf{org}}$ and refines the inherited class domain of relationship services to a class of worksteps Archive being a subclass of class All (cf. Figure 10.16). As a consequence, each authorization contract of type $\mathsf{CONTRACT}_{\mathsf{Reservation\text{-}WF(S),Archive(M),WfAdmins_{org}(M)}}$ authorizes one or several workflow administrators to invoke workstep Archive on room reservations processed by a certain room-reservation workflow process.

Authorization-contract type $\mathsf{CONTRACT}_{\mathsf{Reservation\text{-}WF(S),HandleReservations(M),Receptionists_{org}(M)}}$ (cf. Figure 10.11(c)) is a direct subtype of CONTRACT. It extends its supertype by adding relationship clients with type domain $\mathsf{RECEPTIONIST}_{\mathsf{org}}$. Furthermore, it defines the class domain of relationship clients by class $\mathsf{Receptionists}_{\mathsf{org}}$ and refines the inherited class domain of relationship services to a class of worksteps HandleReservations being a subclass of class All (cf. Figure 10.16). The derivation expression of relationship clients is componentOf.receptionists. That is, each authorization contract of type

[6]This means, the names of abstract authorization-contract types that do not define *all three* designated relationships cannot be derived according to the given naming schema.

[7]Note: The class domain of a relationship service (or services) is defined by a class of worksteps corresponding to an authorization interface defined for members of the class of suppliers S.

$\mathsf{CONTRACT}_{\mathsf{Reservation\text{-}WF(S),HandleReservations(M),Receptionists_{org}(M)}}$ authorizes all receptionists of a hotel to invoke worksteps that belong to authorization interface HandleReservations on room reservations processed by a certain room-reservation workflow process.

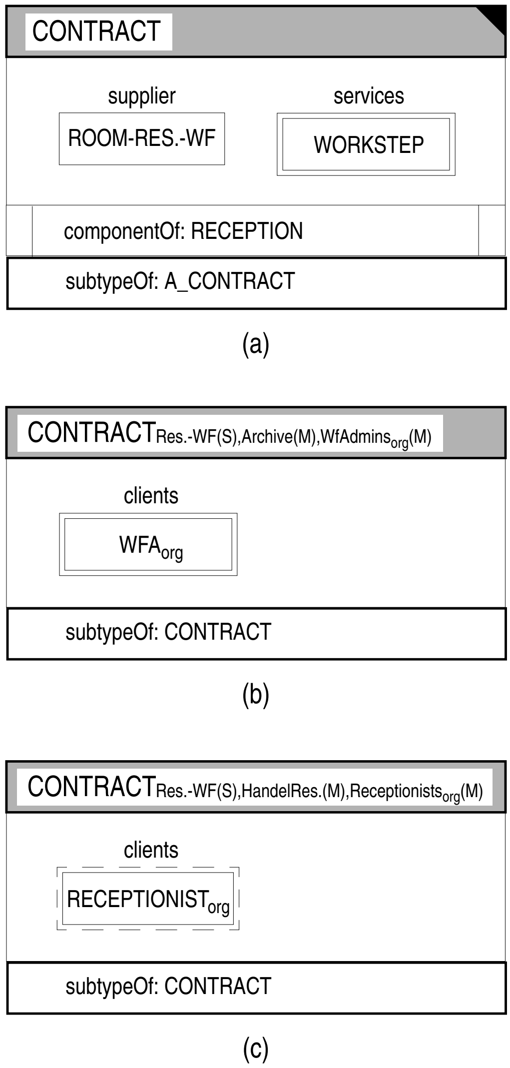


Figure 10.11: Authorization-contract type CONTRACT together with subtypes

Modeling the Behavior of Authorization Contracts

The behavior of authorization contracts is determined by a set of *activities*, by *allowed execution orders* on these activities, and by their *interactions* with other objects (cf. Subsection 6.3.3). In Subsection 7.2.2, advanced concepts of behavior modeling, i.e., synchronization and specialization of the behavior of objects, have been discussed. These concepts of behavior modeling apply to the modeling of the behavior of authorization contracts in the same way. Below, we shortly review these concepts in the context of authorization contracts. First, however, we discuss how activities that create new authorizations can test adherence to the *Most-Specific-Type Rule for Authorization Contracts* (above).

Each authorization-contract type offers an activity isMostSpecificType that takes three arguments s, r, and c, where s is a set of suppliers, r a set of services, and c a set of clients of authorization contracts. Invoking activity isMostSpecificType on an authorization-contract type A with arguments s, r, and c evaluates to 'true' if A is the most specific authorization-contract type for s, r, and c and evaluates to 'false' otherwise (cf. definition of *Most-Specific-Type Rule for Authorization Contracts*, above)

Example: A specification of activity addServices of guest cards has already been given in Figure 6.11(a). Figure 10.12 shows an extended specification of activity addServices that tests adherence to the *Most-Specific-Type Rule for Authorization Contracts* for new authorizations that are created when activity addServices is invoked on a guest card. That is, the precondition of activity addServices evaluates the additional predicate isMostSpecificType on the type of that guest card g on which activity addServices has been invoked, i.e., authorization-contract type GUEST-CARD. The supplier and the client of g as well as the set of services r to be added are passed as arguments to predicate isMostSpecificType.

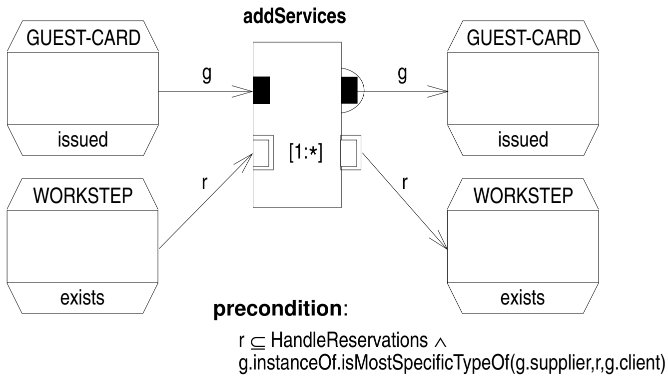


Figure 10.12: Specification of activity addServices

Synchronization of the Behavior of Authorization Contracts. The behavior of authorization contracts could be synchronized with the behavior of other objects by specifying (a) prestates and poststates of ports of activities, (b) domain-restriction constraints on states, and (c) visibility-restriction contstraints on activities defined for authorization constraints or related objects.

Example: Figure 10.13(a) shows the specification of activity destroy defined for charge cards of type EU-CHARGE-CARD (cf. Figure 10.9(a)). A charge card c that resides in state archived can only be destroyed if the authorization contracts that represent c as frequent-flyer card, as frequent-driver card, and as guest cards reside in state archived, too. This means, activity destroy, consider the precondition and the prestates of ports f, d, and g, synchronizes the behavior of each charge card *c* with the behavior of all authorization contracts that are roles of *c*.

Figure 10.13(b) shows a visibility-restriction constraint specified on activity destroy defined for guest cards of type GUEST-CARD (cf. Figure 10.9(b)). It says that activity destroy may only be invoked on guest cards by activity destroy defined for charge cards (cf. Figure 10.13(a)). This means, activity destroy, consider the precondition and the prestates of ports c, synchronizes the behavior of each guest card with the behavior of the charge card of which it is a role.

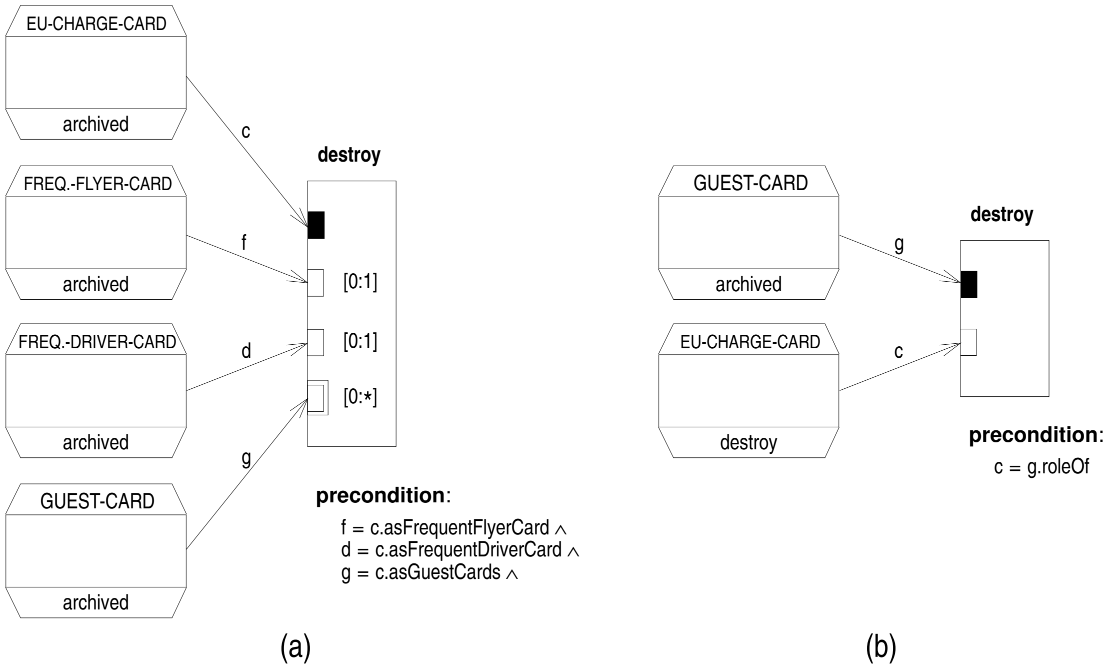


Figure 10.13: Specifications of activities destroy

Specialization of the Behavior of Authorization Contracts. To specialize the behavior of authorization contracts means, to specialize *activities* and *life-cycles* authorization-contract types inherit from their supertypes. In fact, each user-defined authorization-contract type is a (direct or indirect) subtype of the predefined conceptual type A_CONTRACT (Authorization CONTRACT, cf. Figure 7.6) models the basic behavior each authorization contract offers to its environment. In fact, conceptual type A_CONTRACT does not specialize the behavior it inherits from its supertype B_OBJECT (cf. Figure 7.13). Subtypes of conceptual type A_CONTRACT may specialize inherited behavior. Note: Criteria for a consistent specialization of the behavior of objects have been discussed in Subsection 7.2.2.

Example: The life-cycle defined at authorization-contract type GUEST-CARD (cf. Figure 6.10) is an observation-consistent refinement of the life-cycle inherited from the conceptual type A_CONTRACT. Note: Activity issue redefines the abstract activity create, activity destroy the abstract activity destroy. The remaining states and activities are in the refinement of the inherited state exists.

Modeling Classes of Authorization Contracts

An *authorization-contract class* collects a set of authorization contracts as its members (cf. Subsection 6.3.3). Classes of authorization contracts are modeled in the same way as classes of business objects (cf. Subsection 7.2.3). Specifying local classes as domains of relationships guarantees local referential integrity, which in general requires that two objects taking part in a directed relationship must be direct or indirect components of the same composite object (cf. Subsection 7.2.3). Since authorization contracts are relationship objects relating one or several clients to one or several suppliers, specifying local classes as domains of the designated relationships client (clients) and supplier (suppliers) of authorization contracts requires that the client(s) and the supplier(s) related by an authorization contract belong to the same organizational unit or organization.

Example: Figure 10.14 shows the specification of local class GuestCards at organizational-unit type EU-HOTEL (cf. Figure 9.10). A local class GuestCards is an instance of the predefined class type BO_CLASS and collects authorization contracts of type GUEST-CARD (cf. Figure 10.9(b)) as its members (cf. member-type constraint). Local class GuestCards is the base class of member type GUEST-CARD (cf. has-component relationship guestCards with domain GUEST-CARD).

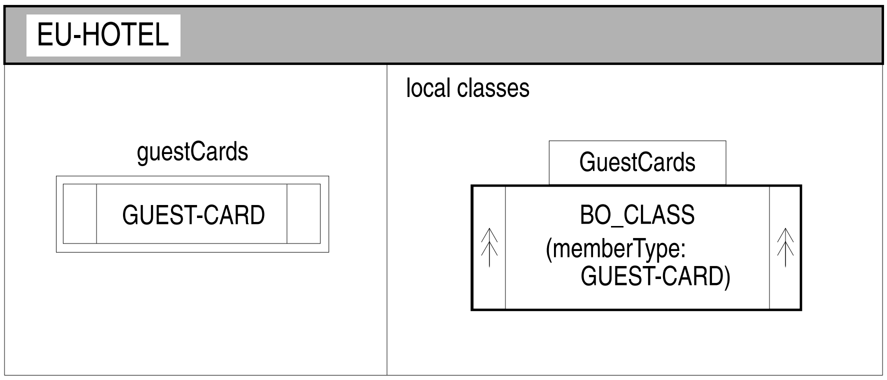


Figure 10.14: Specification of local class GuestCards at EU-HOTEL

Example: Figure 10.15 shows class-domain constraints defined at organizational-unit type EU-HOTEL (cf. Figure 9.10) on relationships supplier, services, and client of guest cards. In fact, the domain of relationship supplier of guest cards is restricted to the member of singleton class Reservation-WF of the same hotel h that did issue the guest cards. The domain of relationship services is restricted to worksteps being members of local class HandleReservations defined for the room-reservation workflow process being member of singleton class Reservation-WF of hotel h. The domain of relationship clients is restricted to members of local class Guests of hotel h. In other words, a guest card issued by a certain hotel h must not authorize guests of hotel h to invoke worksteps on room reservations that are processed according to the room-reservation workflow processes of other hotels than h. Note: Local class GuestCards is the base class of authorization-contract type GUEST-CARD (cf. Figure 10.9(b)). Local class HandleReservations contains all worksteps that belong to authorization interface HandleReservations.

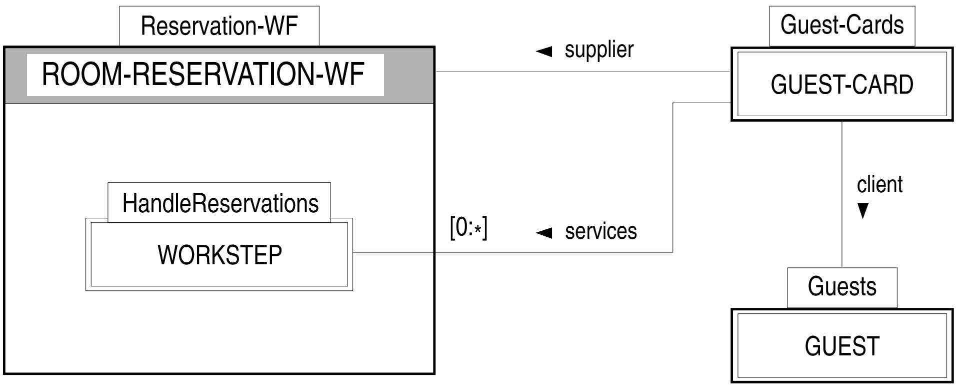


Figure 10.15: Class-domain diagram at EU-HOTEL (partially)

Example: Figure 10.16(a) shows class-domain constraints defined at organizational-unit type EU-HOTEL (cf. Figure 9.10) on relationships supplier and services of authorization contracts of type CONTRACT (cf. Figure 10.11(a)). In fact, the domain of relationship supplier of authorization contracts being members of a local class Contracts defined for the reception of a hotel h of type EU-HOTEL is restricted to the member of singleton class Reservation-WF of the same hotel h. The domain of relationship services for the same authorization contracts is restricted to worksteps being members of local class All defined for the room-reservation workflow process being member of singleton class Reservation-WF of hotel h.

Local classes WfAdminContracts is a subclass of class Contracts. It restricts the inherited member type to $\mathsf{CONTRACT}_{\mathsf{Reservation\text{-}WF(S),Archive(M),WfAdmins_{org}(M)}}$ (cf. Figure 10.11(b)). Furthermore, it restricts the inherited domain of relationship services to local class Archive, which is a subclass of local class All.

Local classes ReceptionistContracts is a subclass of class Contracts. It restricts the inherited member type to $\mathsf{CONTRACT}_{\mathsf{Reservation\text{-}WF(S),HandleReservations(M),Receptionists_{org}(M)}}$ (cf. Figure 10.11(c)). Furthermore, it restricts the inherited domain of relationship services to local class HandleReservations, which is a subclass of local class All.

Figure 10.16(b) shows class-domain constraints defined at organizational-unit type RECEPTION (cf. Figure 9.14) on relationships clients of authorization contracts of type $\mathsf{CONTRACT}_{\mathsf{Reservation\text{-}WF(S),Archive(M),WfAdmins_{org}(M)}}$ (cf. Figure 10.11(b)) and $\mathsf{CONTRACT}_{\mathsf{Reservation\text{-}WF(S),HandleReservations(M),Receptionists_{org}(M)}}$ (cf. Figure 10.11(c)). In fact, the domain of relationship clients of authorization contracts being members of a local class WfAdminContracts defined for a reception is restricted to the members of local class $\mathsf{WfAdmins}_{\mathsf{org}}(\mathsf{M})$ of the same reception. Furthermore, the domain of relationship clients of authorization contracts being members of a local class ReceptionistContracts defined for a reception is restricted to the members of local class $\mathsf{Receptionists}_{\mathsf{org}}(\mathsf{M})$ of the same reception.

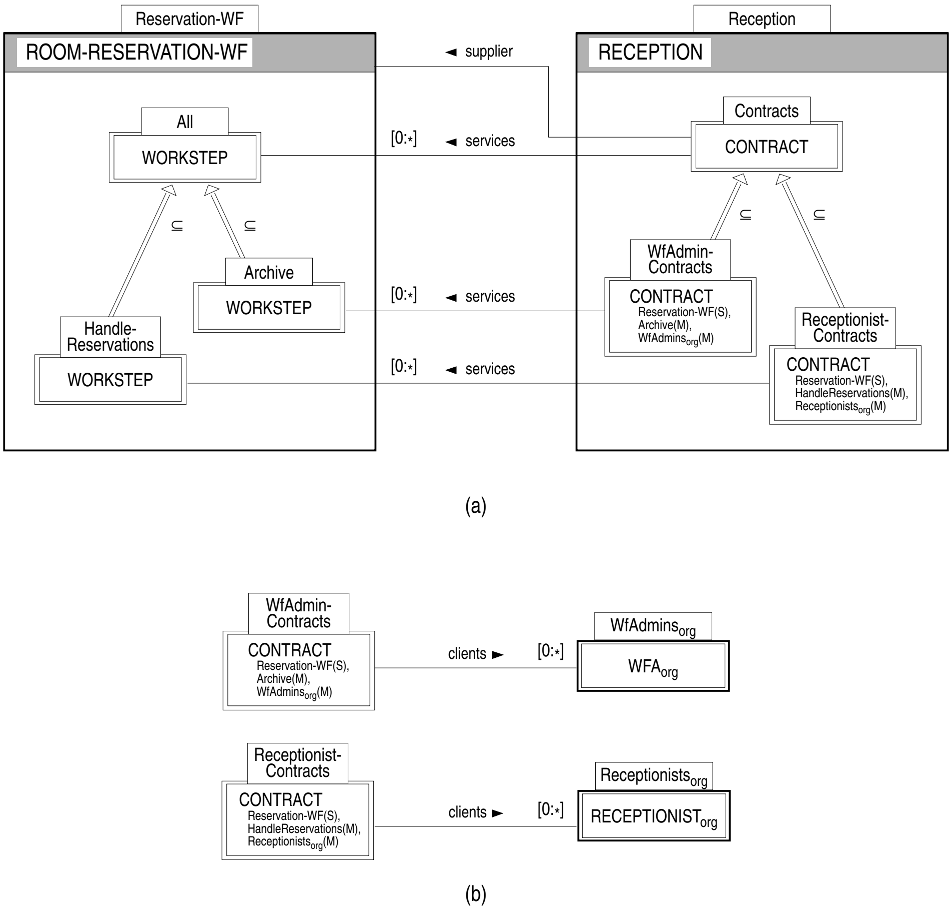


Figure 10.16: Class-domain diagrams at (a) EU-HOTEL and (b) RECEPTION (both partially)

10.2.4 Modeling of Role Constraints

We distinguish role constraints into two categories: First, role constraints that determine legal configurations of roles that may represent an actor or a position. Second, role constraints that restrict the activation of roles during sessions. In Chapter 9 we presented several semantic modeling concepts to represent role constraints that determine legal configurations of roles. For example, a *role-type hierarchy* determines the kind of roles that may represent particular actors (type of roles, mutually exclusive roles) and the order in which these roles must be acquired and abandoned (prerequisite roles), and *cardinality constraint* limit the number of roles of particular types in which actors may exist in parallel. Moreover, we did present an organization model that represents an actor that occupies a certain position within an organization by an agent object and by one individual role for every organizational role of the position she or he occupies. This representation of actors results in a duality of role hierarchies rooted at agent objects and role hierarchies rooted at position objects (cf. Subsection 9.2.1). For the modeling of role constraints that restrict the activation of roles during sessions, we assume that when an actor activates an individual role (agent object) automatically the corresponding organizational role (position object) is activated, too. As a consequence, we consider only individual roles when modeling role constraints on the activation of roles during sessions.

Subsequently, we discuss the modeling of *role-activation constraints* that restrict the sets of roles actors may activate simultaneously during sessions and present the concept of *role-protection groups* that organize roles into different levels of sensitivity, where more sensitive groups of roles may require separate and additional mechanisms to authenticate the identity of actors. Note: In general, role constraints cannot prevent misbehavior, but they allow to reduce the risk by making misbehavior more difficult to appear.

We define the concept of sessions in the context of a role-based authorization control model as follows: A *session* is a process that acts on behalf of a real-world actor within an information system. In general, an actor may have one or several sessions open simultaneously (cf. [SCFY96]). During a session an actor may activate a subset of the roles that represent her or him within an information system. However, the set of active roles of an actor is maintained independently of a particular session of that actor. This means, if an actor has several sessions open simultaneously, each session that actor has open refers to the same set of active roles. The union of the sets of authorizations associated with active roles of an actor determines the authority of that actor during sessions. The set of active roles may change during the existence of a session, whereas the association with the actor is fixed.

Modeling of Role-Activation Constraints

A *role-activation constraint* restricts the set of roles an actor may activate simultaneously. Since each actor can be modeled by a set of roles and the authority of an actor during a session is determined by the union of the sets of authorizations associated with currently active roles, modeling of role-activation constraints helps to avoid unwanted and critical accumulations of authority during sessions. Note: To open several sessions simultaneously does not allow to circumvent role-activation constraints, since all sessions an actor opens simultaneously refer to the same set of active roles for that actor.

Role-activation constraints can be defined for players that may exist in two or several roles of different types in parallel (cf. modeling of role-type hierarchies, Subsection 9.2.1). For example, a role-activation constraint defined for players of type P on roles of types R_1 and R_2 says that although a player p of type P may exist in roles r_1 of type R_1 and r_2 of type R_2 in parallel, p may not activate roles r_1 and r_2 simultaneously during a session. Note: If p may also exist in a role r_3 of type R_3 in parallel to roles r_1 and r_2, the above constraint does allow to activate roles r_1 and r_3 as well as r_2 and r_3, respectively, simultaneously.

Graphically, role-activation constraints are depicted like role-type hierarchies (cf. Subsection 9.2.1). To distinguish role-activation constraints from role-type hierarchies, we depict lines connecting participating types by dashed lines. Note: An unfilled semi-circle denotes that activation of roles is mutually exclusive.

Example: Figure 10.17(a) shows a role-type hierarchy saying that agents of type EMPLOYEE (cf. Figure 9.2(b)) may in parallel exist in roles of agent types $\mathsf{ASSISTANT}_{\mathsf{ind}}$ and STUDENT.

Figure 10.17(b) shows a role-activation constraint saying that employees existing in assistant roles and student roles in parallel are not allowed to activate these roles simultaneously during a session. Of course, employees can activate assistant roles and student roles at different times during the same sessions.

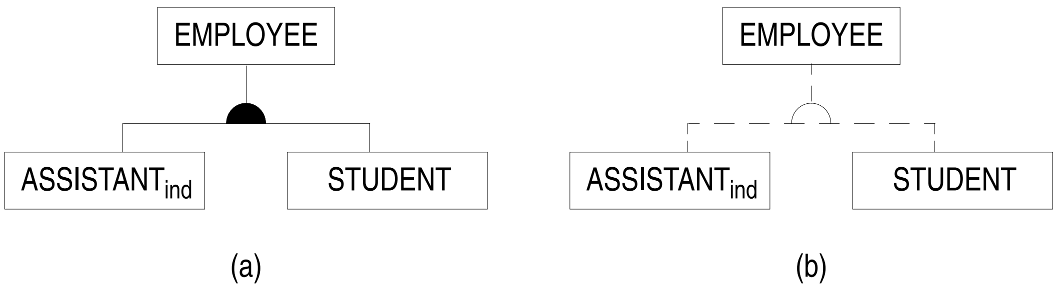


Figure 10.17: Role-type hierarchy *vs.* role-activation constraint

Modeling of Role-Protection Groups

A *role-protection group* is a named set of roles having the same sensitivity, where all roles that belong to a particular role-protection group represent the same actor and the sensitivity of a set of roles is determined by the kind and the number of authentication procedures an actor must follow to verify her or his identity when activating a role. Castano *et al.* [CFMS94] classify authentication mechanisms in three categories: Mechanisms that check the identity of a user through (a) something the user is acquainted with, e.g., passwords, (b) something the user owns, e.g., magnetic cards, or (c) physcial characteristics of the user, e.g., fingerprints, voice.

Since different roles representing the same actor may have different sensitivity, different roles representing the same actor may belong to different role-protection groups. Moreover, role-protection groups may be organized into *sensitivity hierarchies*, where roles that belong to a subordinate role-protection group g are more sensitive than roles that belong to a superordinate role-protection group of g. For consistency, we require that if a role r belongs to a role-protection group g, the subroles of r belong to g, too, or to subordinate role-protection groups of g. The increased sensitivity of roles that belong to a subordinate role-protection group g is taken into account in that activating a role that belongs to g during a particular session, requires the appropriate actor to pass authentication procedures associated with g *and* to pass authentication procedures associated with superordinate role-protection groups of g, too, where authentication procedures of superordinate role-protection groups must be passed first.

Example: Figure 10.18(a) schematically shows a role-type hierarchy indicating the organization of roles that are instances of role types R_1 to R_5 into role-protection groups G_1 and G_2, where G_2 is subordinate to G_1 (cf. Figure 10.18(b)).

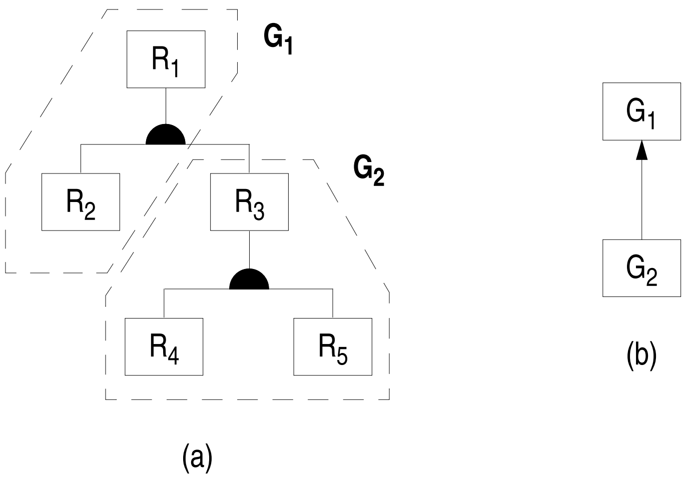


Figure 10.18: Role-protection groups

10.3 Administration of Authorizations

The *administrative* approach to administer authorizations says that objects are owned by organizations independent of who did create them (e.g., [FGS94]). We adopted this approach for the administration of authorizations in workflow systems (cf. Section 6.4). From the knowledge on the control structure of an organization and the knowledge on its physical structure, we derive for each supervisor of an organizational unit with a hierarchical control structure and for each coordinator of an organizational unit with a network control-structure a resource domain and an organization domain[8]. Since the basic model did not know the network as a control structure for organizational units (cf. Chapters 5 and 9), we have to restate how a basic resource domain and a basic organization domain can be computed for each supervisor or coordinator of some organizational unit.

The basic *resource domain* D_r of a supervisor s (coordinator c) contains all objects that belong to the same organizational unit as s (c). The basic *organization domain* D_o of a supervisor s (coordinator c) contains s (c) and all actors that are subordinates of s (are coordinated by c) and belong to the same organizational unit as s (c). Note: In general, supervisors and coordinators of organizational units delegate their responsiblity

[8]Note: The concept of *resource domains* and *organization domains* of managers of organizational units has been adopted from Moffet and Sloman (cf. [MS88]).

to administer authorizations to designated security administrators (cf. separation-of-duty principle, Section 6.4). Modeling of subdomains of their basic resource domains and organization domains allows supervisors and coordinators of organizational units to delegate their responsibility to administer authorizations to different security administrators.

10.4 Summary

This chapter did present advanced concepts for the design of *authorization schemas* of workflow systems. After we gave a management definition of the fundamental concepts discussed in this chapter, we adapted existing concepts for the design of authorization schemas to advanced modeling concepts. That is, *authorization interfaces* and *authorization constraints* have been adapted to cope with the specialization of workflow processes and to guarantee the proper enforcement of protection requirements defined at a workflow process w in nested workflow processes of w, too. In addition, authorization constraints have been adapted to cope with type hierarchies and local classes. Authorization constraints that specify local classes as actor domains of public operations being organized into authorization interfaces are called *local-authorization constraints*, since they require (depending on the deontic modality it associates with the specified actor domain) that clients may be, must be, may not be, or must not be authorized to request certain services from certain suppliers if both the clients and the suppliers belong to the same organizational unit or organization. Furthermore, the modeling of authorizations has been reviewed in the realm of the advanced modeling concepts: *Authorization contracts* may have semantic relationships to other objects, are members of local classes, and are instances of types, which are organized in type hierarchies. That is, authorization contracts are first-class citizens in the presented workflow model. Afterwards, we introduced role constraints that restrict the activation of roles, which represent individual actors within organizations, during sessions. Role constraints that determine legal configurations of roles representing individual actors within workflow systems were introduced in Chapter 9. Finally, basic principles of the administration of authorizations in organizations with mixed hierarchical and network control structures were discussed.

Part IV

Conclusion

Chapter 11

Summary and Future Work

Contents

This chapter is organized as follows. Section 11.1 summarizes the main achievements of the research presented in this thesis. Section 11.2 investigates whether the objectives of the thesis formulated in Section 1.3 have been achieved. Section 11.3 identifies relevant issues for future work.

11.1 Summary

The workflow model presented in this thesis offers a *uniform*, *object-oriented* approach to the *conceptual design* of workflow schemas. The workflow model has been organized into three submodels, i.e., a process model, an organization model, and an authorization model. Before we summarize the main features of the individual submodels, we give a brief defense of our approach to workflow modeling.

Conceptual Design. The design of information systems is a complex process that requires decisions at very different levels. Experiences in database design and application-programs design showed that decomposing the design process into phases helps to master the complexity of the overall design process. In analogy to the phases of the database-design process, we proposed to decompose the workflow-design process into the phases conceptual design, logical design, and physical design.

In the sequel, we highlight the advantages of a conceptual approach to the design of workflow schemas. In fact, we interpret arguments given in favor of a conceptual approach to the design of database schemas (e.g., [BCN92]) in the realm of the design of workflow schemas. The interpretation shows that conceptual design of workflow schemas is worthwhile, too: (a) *Conceptual design allows for a strong influence of end users on design decisions.* An enhanced participation of end users in the design process of information system is said to result in an improved quality of schemas, in projects that more likely converge towards expected results, and in end users that more likely accept information systems. Research conducted in the area of "participatory design" (e.g., [SN93]) gives reason to believe that enhanced user participation is an important issue in the design of workflow systems, too. (b) *Conceptual design concentrates on real-world semantics and neglects implementation details.* As a consequence, decisions on which implementation systems, i.e., which WFMS, which DBMS, and which programming language, to use to implement an information system can be postponed and design results can survive late decisions to change an implementation system. Currently, the workflow management market is characterized by an unrestrained proliferation of WFMS products (e.g., [JB96, p. 12]). As a consequence, the independence of conceptual design from a particular WFMS used to implement (part of) an information system is a very strong argument in favor of a conceptual approach to the design of workflow schemas. (3) *Conceptual schemas ease the understanding of information systems.* Conceptual schemas together with their documentation play a central role during the operation of an information system. As conceptual schemas ease the understanding of information systems they facilitate their maintenance, which is usually longest and most expensive phase of the life-cycle of information systems.

Object-Orientation. Object-orientation is often referred to as being a new way of thinking about the world. A way that is more natural to humans. Actually, object-orientation thinks about the world as a system of interacting objects. Fundamental object-oriented principles are encapsulation of data structure and operations into objects, information-hiding, inheritance, specialization, object identity, polymorphism, and late-binding (e.g., [Mey88]). Benefits of object-orientation for the development of software systems (e.g., extendibility, flexibility, reusability, robustness) have been proved by experiences in the development of application programs and database systems in numerous software projects and are beyond question. However, the question is whether object-orientation is suitable for the development of workflow systems, too.

The development of workflow systems covers their analysis, design, and implementation. For object-orientation to be suitable for the development of workflow systems requires that object-oriented methods are available in all phases of the development of workflow systems. Currently, some WFMSs are already available that allow for an object-oriented implementation of workflow systems, e.g., the commercial WFMS product *In-*

Concert (XSoft Advanced Information Technology, Inc.) (cf. Section 1.4). What about object-oriented methods for the analysis and design of workflow systems? In fact, several authors investigated the suitability of some object-oriented analysis and design methods for the development of workflow systems. For example, Jablonski and Stein [JS95] investigated the Booch method [Boo94], Müller-Luschnat *et al.* [MLHH93] the Coad and Yourdon method [CY91], and Kohl *et al.* [BKMW94, Koh94, Koh96] the Object Modeling Technique (OMT) [RBPE91]. In short, their results have been that object-oriented methods in general are well suited for the analysis and design of workflow systems. However, Jablonski and Stein [JS95] mention that object-oriented methods offer only weak support for the modeling of processes. A criticism that is shared by Scheer [Sch98b, p. 134]. We claim that this is not a criticism of object-orientation *per se*, but is a criticism of the dynamic model of particular object-oriented methods. In fact, many object-oriented analysis and design methods such as the Booch method, the Coad and Yourdon method, and OMT, use some form of state-transition-diagrams to model the dynamic behavior of objects, which in their basic form allow a system to reside in only one state at a time. Enhancements of state-transition-diagrams allow to aggregate states into hierarchies (e.g., [Har88]) making dynamic models more expressive. We model the dynamic behavior of objects using a notation that is based on Petri nets (cf. Chapters 4 and 8). The suitability of Petri nets for the modeling of processes is widely accepted. Van der Aalst [Aal96] did even propose Petri nets as a standard model for the analysis and design of organizational processes.

Furthermore, Georgakopoulos *et al.* [GHS94] criticized that object-oriented methods fail to provide efficient support for workflow designers, since they do not offer workflow-specific object types. As a consequence, workflow designers have to re-define workflow-specific object types from scratch in each project. A solution to this problem could be to offer a set of predefined *conceptual types*, which workflow designers adapt to the requirements of individual projects by specialization. In fact, we presented several conceptual types throughout this thesis (cf. Chapters 7 to 10). However, additional research must be conducted in this area, such that a conceptual-type hierarchy could offer an efficient tool for workflow designers.

We conclude this discussion with the statement that object-orientation is well suited for the development of workflow systems if object-oriented methods employed for the analysis, design, and implementation of workflow systems satisfy the following criteria: (a) adequate support for the modeling of organizational processes and (b) predefined conceptual types of workflow-specific object types are provided. We offer an object-oriented data model for the conceptual design of workflow systems that satisfies these criteria. This research is a contribution to allow for a uniform, object-oriented design of information systems in case a workflow-oriented implementation strategy has been selected (cf. Subsection 1.2.3). Whereas object-orientation is well established in the design of database systems and application programs, few work has been reported on the object-oriented design of workflow systems. This research fills this gap and allows to rely on the same (or on similar) concepts during workflow design, database design, and application-programs design of information systems.

Uniformity. We offer a *uniform*, object-oriented approach to the conceptual modeling of workflow systems. That is, object-oriented concepts can be used to model organizational processes (process modeling), inter-related organizational objects (organization modeling), and protection requirements as well as authorizations (authorization modeling). Obviously, the main advantage of a uniform approach to process modeling, organization modeling, and authorization modeling is that problems concerning the interaction within different subsystems of workflow systems can be avoided. For example, Bussler (cf. [Bus96]) proposed an architecture requiring that WFMSs delegate the maintenance of knowledge on the structure of organizations to an external service. In fact, the commercial WFMS product *WorkParty* (Siemens Nixdorf) realizes such an architecture and uses an independent service called *ORM* to maintain organizational knowledge (cf. Subsection 1.4.2). We have achieved uniformity in that all three submodels of the overall workflow model, i.e., the process model, the organization model, and the authorization model, build on a common, object-oriented data model. As a consequence, conceptual boundaries are limited. A uniform approach to the modeling of all aspects of workflow systems ensures an increased consistency between the different aspects of workflow systems.

In the remainder of this section we summarize the main features of the conceptual workflow-model presented in this thesis. The summary is organized into three subsections, one for each of the submodels of the overall workflow model, i.e., the process model, the organization model, and the authorization model. For a concise presentation of the foundations of the presented object-oriented data model, i.e., Object/Behavior Diagrams, the interested reader is referred to [KS91]. Note: The data model and the graphic notation of Object/Behavior Diagrams have been substantially revised and extended in this thesis.

11.1.1 Process Modeling

The *process model* relies on a powerful approach to process modeling, i.e., Petri nets. Embedding Petri nets into an object-oriented data model combines the power of Petri nets and the benefits of object-orientation. Nevertheless, object-orientation and Petri nets do not offer adequate support for the adaption of organizational processes to changed requirements. We integrate object-orientation and Petri nets into a novel two-schema architecture for WFMSs. The purpose of this schema architecture is to provide efficient support for the change of workflow processes.

Two-schema Architecture. The two-schema architecture for workflow systems proposes to separate knowledge on organizational processes into external knowledge and internal knowledge. External knowledge on organizational processes determines a frame within which an organization may operate, e.g., business rules based on natural facts or law, and is modeled in so-called *business-process schemas.* Internal knowledge on organizational processes restricts this frame and determines the way an organization actually runs its business, i.e., business rules bases on organizational commitments. Internal knowledge on organizational processes is modeled in so-called *workflow-process schemas*. The purpose of the two-schema architecture for workflow systems is to support the evolution of organizational processes, i.e., the adaption of organizational processes to changed requirements in the business environment of organizations. The two-schema architecture builds on the fact that organizations may change internal business rules to react to changes in their business environments but have no influence on business rules of external origin.

Modeling the knowledge on organizational processes in two separate but related schemas requires additional effort during analysis and design of organizational processes but offers several advantages in the long term: (a) *Adaptability*: If organizations have to change their processes in order to react to changes in their business environments, such changes are reflected exclusively within workflow-process schemas. (b) *Reusability*: Since a business-process schema does not incorporate knowledge on organizational processes that is based on organizational commitments, many workflow-process schemas can reuse the same business-process schema. (c) *Out-Sourcing*: Since business-process schemas do not incorporate knowledge on organizational processes that is internal to organizations, their development could be sourced out. That is, organizations might buy business-process schemas off the shelf, reducing the overall effort of the modeling of organizational processes. (d) *Schema Quality*: Business-process schemas are rather stable and mature as they are reused and reused again by many different workflow-process schemas. Building on mature business-process schemas ensures a high quality of workflow-process schemas.

Petri Nets. We rely on Petri nets (e.g., [Mur89, Pet77]) for the conceptual modeling of organizational processes. In fact, object life-cycles are high-level, colored Petri nets (e.g., [Jen92]), and workflow processes build on object life-cycles. Petri nets are well-suited for a conceptual modeling of organizational processes. Van der Aalst (cf. [Aal96]) did propose Petri nets even as a standard paradigm for the modeling of organizational processes. His arguments in favor of Petri nets are: (a) *Petri nets allow to represent the business logic of organizational process by a graphical as well as formal language.* The semantics of classical Petri nets and several enhancements (e.g., color, time) has been defined formally (e.g., [Jen92, Mur89, Pet77]). (b) *Petri nets allow to model nondeterminism.* Petri nets follow a state-based approach to the modeling of processes. That is, Petri nets model the states of business cases explicitly. Consequently, they allow to explicitly distinguish between the enabling of a task and the execution of a task. Nondeterminism means that several alternative tasks are enabled simultaneously and the choice as to which of these tasks to execute next cannot be predicted, e.g., depends on user decision. Event-based paradigms to the modeling of processes, e.g., Activity Networks, cannot model nondeterminism, since they cannot distinguish between the enabling of a task and the execution of a task. In fact, WFMSs that rely on event-based process models have problems to cope with nondeterministic situations, since the workflow engine cannot decide which task to trigger next. That is, whereas in event-based WFMSs the workflow engine is leading, in state-based WFMS the environment is leading, i.e., actors decide what to do next based on the state of business cases. (3) *There are many analysis techniques available for Petri nets.*

The reader is referred to [Jen92, Mur89, Pet77] for a survey of analysis techniques that are available for Petri nets. Note: These analysis techniques cannot be used to analyze the behavior of objects in its entirety. In fact, existing analysis techniques may only be applied to analyze the consistency of allowed execution orders of activities of objects that are specified in object life-cycles. That is, preconditions and postconditions of activities as well as secondary ports of activities (i.e., object interaction) are ignored.

Object-Orientation. Object-orientation offers a flexible approach to the modeling of organizational processes. In fact, fundamental object-oriented principles offer advantages for the modeling of organizational processes, too: (a) *Object Identity*: Object identity allows for referential sharing of objects. In fact, we model organizational processes as networks of inter-related process objects that are either "behaviorally independent" or "behaviorally embedded" (cf. Chapter 8). We call a process object that is "behaviorally embedded" into another process object a nested workflow process. Object identity allows that process objects are created and disposed at runtime. As a consequence, networks of process objects can be modified dynamically. For example, a nested workflow process may be exchanged. (b) *Inheritance and Specialization*: Inheritance and specialization allow for an incremental design of organizational processes and support the reuse and disciplined modification of definitions of organizational processes. Consequently, object-orientation favors the development of libraries of organizational processes.

11.1.2 Organization Modeling

The *organization model* allows for the conceptual design of arbitrary organization schemas. It supports the modeling of process-oriented organization structures, i.e., networks of actors, and allows for a flexible modeling of positions and agents utilizing the role concept from software engineering (e.g., [GSR96]).

Schema Design. A schema contains intensional knowledge that denotes the structure (and possibly behavior) of real-world objects of a particular business area. The design of schemas is a technique that is widely accepted in the database area. Its purpose is to build semantically rich descriptions of reality that act as an effective filter for illegal representations of reality. That is, schemas help to exclude illegal representations of reality from databases. As a consequence, the purpose of the design of organization schemas is to build semantically rich descriptions of the structure of organizations that help to exclude illegal representations of organizations from databases.

Surprisingly, most contemporary WFMSs offer a predefined and fixed organization schema. In the context of a fixed organization schema, organization modeling means to populate the predefined organization schema. That is, to represent each relevant organizational entity as instance of the most-specific object type available. However, since a predefined and fixed organization schema must allow to represent all kinds of organizations, these organization schemas are semantically poor descriptions of reality. A predefined and fixed organization schema cannot capture the semantics of the structure of individual organizations. Bussler [Bus96] investigated the organization modeling capabilities of several WFMSs (i.e., *COSA* (Software-Ley), *FlowMark* (IBM), and *WorkParty* (Siemens Nixdorf)). His results are best described by his statement: "The most interesting question is why WFMS allow to model arbitrary workflows [i.e., process logic] but not arbitrary organization structures ..." Contrary to most contemporary WFMSs, we offer an organization model that allows for the design of arbitrary organization schemas that are semantically rich descriptions of reality. In fact, the organization model builds on a powerful, object-oriented data model that has originally been introduced for the conceptual design of object-oriented database schemas.

Process-Oriented Organization Structure. With the shift to process-oriented thinking in business modeling (e.g., [HC93]), there is a trend to create the structures of organizations around organizational processes. As a consequence, vertical hierarchies of organizations are flattened and management tasks are delegated to employees that are organized into self-managed groups (cf. [Daf95]). Process-oriented thinking in business modeling paved the way for workflow technology. Surprisingly contemporary WFMSs do not provide adequate support for the modeling of process-oriented organization structures (cf. [KU94]).

The organization model we presented in this thesis offers several concepts to model process-oriented organization structures. Self-managed groups can be modeled as organizational units having a *network control structure*. Actors can be modeled by several objects, called *roles* (cf. role modeling, below), that allow to represent actors in different organizational contexts. That is, roles are a concept to increase the horizontal linkage within organizations. Furthermore, we integrate concepts for the modeling of process-oriented organization structures and concepts for the modeling of traditional hierarchical organization structures into a coherent framework. That is, hierarchical and network control structures can coexist within the same organizations.

Separation of Agents and Positions. We model actors of organizations by agents and positions. Whereas an agent models an individual actor, a position models a kind of "virtual actor". Positions are a kind of placeholders for agents within an organization. In fact, a position determines the rank, the authority, and the responsibility of an agent that is assigned to that position. Different agents may be assigned to the same position over time and one agent may be assigned to several positions at a time. Positions help to make the structure of organizations independent of individual actors that join and leave organizations over time and individual actors that change positions within organizations (e.g., [LARB+91]).

The independence of the structure of organizations from individual actors is a strong argument in favor of positions in the context of a formal authorization system. In fact, in a formal authorization system there must be an unbroken connection between the owner or the shareholders of an organization and each actor of the organization such that the legitimacy of an actors authorizations can be re-traced to the authority of the owner (or the shareholders). If the structure of an organization were modeled between agent objects, an actor that leaves the organization could break some connections between the owner of the organization and other actors, e.g., assume a manager leaves an organization. Furthermore, the separation of agents and positions reduces the overall effort to administer the authorizations and responsibilities of individual actors within workflow systems. That is, the authority and responsibility of an individual actor is determined in that the respective agent object is assigned to one or several position objects. Modeling positions as a set of role objects further increases the flexibility of the organization model (cf. role modeling, below).

Role Modeling. We adopted the concept of roles known from semantic data modeling (e.g., [GSR96, KS96b]) for the modeling of positions and agents. The basic principle of roles is that one real-world entity can consistently be represented by several model-world objects, incorporating the knowledge that these objects represent the same real-world entity. For example, a single position can be represented by several role objects, where each role object describes a part of the overall authority and responsibility of that position. Since objects may acquire and abandon roles dynamically, roles allow to cope with the evolutionary nature of positions and agents. For example, the authority and responsibility of a position can be dynamically augmented (diminished) in that the position acquires (abandons) a role.

11.1.3 Authorization Modeling

The *authorization model* allows for the conceptual design of authorization schemas, which are formal descriptions of the protection requirements of organizational processes that are maintained within WFMSs. The authorization model considers authorizations and responsibilities as related but independent concepts and allows for an efficient administration of authorizations. Note: The design of authorization schemas is a novel approach in the design of information systems. Traditionally, the responsibility to issue only authorizations that are legal wrt. the protection requirements of organizations is with the security administrators of information systems. The presented authorization model formalizes the protection requirements of organizational process and makes resulting authorization schemas available for workflow systems.

Schema Design. An *authorization schema* is a formal description of the protection requirements of organizational processes that are maintained within a WFMS. In other words, an authorization schema is a collection of constraints that can act as a filter to exclude illegal authorizations from the authorization base of a workflow system. Authorization schemas are developed using object-oriented modeling concepts: (1) *We model authorizations as first-class citizens, called authorization contracts.* An authorization contract is instance of a particular object type and is member of one or several object classes.

This means, the authorization model allows for a high-level semantic description of the security semantics of a workflow system abstracting from a particular realization of authorizations in a concrete WFMSs, e.g., the realization of authorizations as tuples of an access matrix. As a consequence, a security designer can model authorizations at the conceptual level in the way she or he understands it, e.g., as key cards. (2) *Authorization interfaces extend the concept of information-hiding to the modeling of protection requirements of organizational processes.* An authorization interface contains a set of workflow operations that have similar protection requirements. The protection requirements of workflow operations are defined at authorization interfaces by so-called actor domains. Actor domains are specified using authorization constraints, which are sentences of a deontic constraint language. The deontic constraint language supports four deontic modalities, i.e., permission, obligation, discretion, and prohibition. As a consequence, we can give specific connotations to actor domains. For example, associating deontic modality **P** (permission) with an actor domain D defined at an authorization interface f says that a security administrator *may* authorize actors that are elements of domain D to invoke operations that belong to f. Associating deontic modality **O** (obligation) with an actor domain D defined at an authorization interface f says that a security administrator *must* authorize actors that are elements of domain D to invoke operations that belong to f. Since authorization interfaces and authorization constraints can be organized into specialization hierarchies, these concepts allow for a disciplined sharing and specialization of the protection requirements of organizational processes.

Separation of Responsibilities and Authorizations. Jablonski and Bussler (cf. [JB96, p. 183]) state that contemporary WFMSs determine who *can* (authority) access a workflow operation (or workstep) based on the *responsibility* of actors. This means, contemporary WFMSs do not distinguish between authority and responsibility of actors. Figure 11.1(a) illustrates this situation. Each access of an actor (depicted by a downward arrow) that follows a notification of a workflow engine (depicted by an upward arrow) is implicitly an authorized access. We depict a notification and a corresponding access of an actor by one double-headed arrow to give emphasize to the fact that authorizations and responsibilities are not considered as independent concepts in contemporary WFMSs.

We consider authorizations and responsibilities as two related but independent concepts. An access of an actor is accepted if and only if that actor holds an appropriate authorization. That is, whether an access of an actor is accepted is independent of whether the access of the actor follows a notification of the workflow engine or not. The authorization manager of a workflow system acts as a filter for illegal accesses. This situation is depicted in Figure 11.1(b). This time, we depict accesses of actors and notifications of actors by separate arrows to give emphasize to the fact that we consider authorizations and responsibilities as related but independent concepts. Downward arrows depict accesses of actors and upward arrows depict notifications of actors. Downward arrows ending at the box that represents the authorization manager of a workflow system correspond to unauthorized accesses of actors.

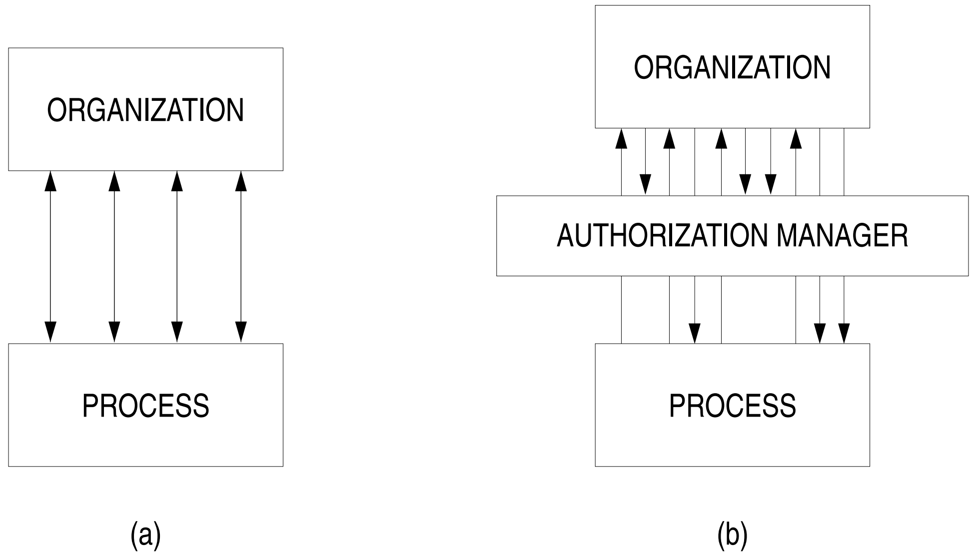


Figure 11.1: Authorization-control architectures of workflow systems

Considering authorizations and responsibilities as related but independent concepts improves security within workflow systems. For example, the separation of authorizations and responsibilities allows to decentralize their administration. Engaging different actors to administer authorizations and responsibilities (separation of duties principle), e.g., engaging security administrators to administer authorizations and managers to administer responsibilities, reduces the risk of threats against the security within workflow systems. Furthermore, the separation of authorizations and responsibilities is a prerequisite for more flexible work-support systems. Contemporary WFMSs assume a model of work in which the workflow engine is leading. That is, in general each access of an actor follows a notification by the workflow engine. Flexible work-support systems must cope with situations in which users are leading, i.e., users decide when to perform what kind of work. The proposed authorization model can be used in such work-support systems, too. Figure 11.1(b) indicates this situation in that there are more downward arrows depicting accesses of users than upward arrows depicting notifications of users.

Administration of Authorizations. Administration of authorizations is a relevant issue of information systems security. We proposed an administrative approach to administer authorizations. This means, the source of authority within an organization is with its owner, which is either a single person or a set of shareholders. The owner of an organization has authority over the use of resources and may delegate its authority to the management of the organization. Moffet and Sloman (cf. [MS88, MS91]) introduced the concepts of organization domain and resource domain of a manager. The organization domain of a manager determines the set of actors to whom a manager may delegate authority, the resource domain of a manager determines the set of resources over which the manager has authority. This means, a manager m may delegate the authority to use a particular resource r to an actor a if r belongs to the resource domain of m and a belongs to the organization domain of m. Whereas Moffet and Sloman offer concepts to explicitly modeling organization domains and resource domains of managers (cf. [MS88, MS91]), we derive this information from the knowledge on the physical structure and control structure of organizations, respectively, represented within the respective organization schema. Furthermore, we extended the approach of Moffet and Sloman to process-oriented organization structures. That is, we allow to derive organization domains and resource domains of coordinators of organizational units with network control structures, too. Note: Managers do not administer authorizations themselves, but delegate this responsibility to designated security administrators (separation of duties principle).

The administration of authorizations is widely acknowledged as an onerous and expensive process (e.g., [FCK95]). We model actors of organizations by agents and positions (cf. organization modeling, above). Whereas each agent represents an individual actor, a position is a kind of "virtual actor", i.e., a placeholder for agents within an organization. In fact, authorizations are not assigned directly to agents but are assigned to positions[1]. The authority of agents is determined in that an agent is assigned to one or several positions. This extra level of indirection reduces the overall effort of administering authorizations within a workflow system. In fact, we model positions by a sets of roles, where each role determines part of the authority of a position. Since roles can be acquired and abandoned dynamically, the workflow model copes well with the evolutionary nature of positions. That is, the authority of a position can be augmented (diminished) dynamically by acquiring (abandoning) roles. Recently a kind of authorization control in database systems has emerged that builds on the same principles, it is called role-based access control (RBAC) (e.g., [RBA95]). In RBAC models, authorizations are assigned to roles and the authority of users[2] is determined in that users are assigned to roles. Sandhu *et al.* [SCFY96] present a family of four reference models $RBAC_0$ to $RBAC_3$, where $RBAC_3$ is the most powerful category of RBAC models. To qualify as an $RBAC_3$ model, an RBAC model must support the modeling of control structures between roles (called role hierarchies in [SCFY96]) and the modeling of role constraints that restrict acceptable configurations of roles an individual user might acquire. In fact, our authorization model (together with the organization model) qualifies as an $RBAC_3$ model. Whereas RBAC models presented in the literature on database security are logical models (e.g., [FCK95, Giu95, LMSY95, NO93, TDH92]), we integrate an RBAC model into a workflow model for the conceptual design of workflow schemas. As a consequence, semantic data modeling concepts are available for the design of roles, too. For example, we explicitly distinguish between role objects, role types, and role classes allowing a more differentiated view of roles in organizations than present logical models (above) do. Furthermore, we offer a graphical notation for the design of organization schemas and authorization schemas that implicitly express semantics that other approaches must express explicitly in form of constraints.

[1]Note: The authorization model does not forbid to assign authorizations directly to agents. Nevertheless, it should be adopted as a basic guideline to assign authorizations in general to positions.

[2]Note: Users are model-world objects that represent real-world actors of organizations.

The concept of positions has originally been introduced to make the structure of organizations independent from individual actors that join and leave the organizations over time or change positions within organizations (cf. [LARB+91]). In fact, the separation of positions and agents is a solid base for a formal authorization system and the administration of authorizations within such an authorization system. Since relationships that determine the control structure of an organization are modeled between positions, the separation of positions and agents ensures that there is at any time an unbroken connection between the owner of an organization and the individual actors via the positions. This connection between the owner and the actors of an organization is a communication channel along which authority (and responsibility) is delegated. An unbroken connection between the owner and the actors of an organization is thus a prerequisite for the administration of authorizations within a workflow system.

11.2 Objectives of the Thesis: Have they been Achieved?

In Section 1.3 we postulated the main objectives of the research presented in this thesis by the following statement:

> "The objective of this research is to develop an authorization model for the conceptual design of secure workflow systems. This authorization model shall be embedded into a workflow model that is comprehensive and flexible wrt. change."

In the sequel we review the individual objectives and say whether the objectives have been achieved. If an objective has been achieved, we add a comment on how this objective has been achieved.

Design of Secure Workflow Systems. Three main requirements for the design of secure workflow systems have been mentioned in Section 1.3: (1) Access rights of actors must be modeled explicitly. (2) Access rights should be considered at the highest level where their semantics are explicit. (3) The protection requirements of organizational processes must be formalized and captured within workflow schemas such that no illegal authorizations may be issued to actors. The authorization model presented in Chapters 6 and 10 of this thesis satisfies all criteria: (1) *Access rights are modeled explicitly.* The workflow model presented in this thesis considers authorizations and responsibilities as related but separate concepts. That is, authorizations are modeled explicitly and independently from responsibilities. This is in contrast to contemporary WFMSs that determine who *can* access a workflow operation (or workstep) based on the *responsibility* of actors (cf. [JB96, p. 183]). (2) *Access rights are modeled at the level of workflow operations (or worksteps).* Access rights (or authorizations) are modeled as authorization contracts. In its basic form, an authorization contract is a formal agreement between a supplier, i.e., a workflow process, and a client, i.e., a position (or in some cases an agent), where the supplier commits itself to offer one or several services to the client. The services a workflow process may offer are its operations, called worksteps. (3) *Authorization interfaces and authorization constraints formalize the protection requirements of organizational processes.* An authorization interface contains a set of worksteps. All worksteps that belong to the same authorization interface have the same or similar protection requirements. The protection requirements of worksteps are defined using authorization constraints. Authorization interfaces and authorization constraints belong to *authorization schemas.* An authorization schema is a formal description of the protection requirements of organizational processes enacted and maintained with in a WFMS. An authorization schema is a part of an overall workflow schema.

Conceptual Design of Workflow Systems. The presented workflow model qualifies as a conceptual model. It allows for high-level descriptions of business areas and does consider neither a particular class of WFMS nor a particular WFMS product. Furthermore, the presented workflow model possesses the qualities required from conceptual models: (a) *Expressiveness*: The workflow model offers a rich set of concepts for the modeling of workflow schemas. It builds on an object-oriented data model that explicitly distinguishes between object types and object classes. Each object is an instances of a particular object type and is member of one or several object classes. Object types model the individual structure and behavior of objects, object classes collect sets of objects as their members. Object classes may define other object classes as domains of relationships defined for their members allowing to explicitly represent referential integrity constraints in schemas. Semantic relationships allow to capture the semantics of references between objects in schemas. Furthermore, object types and object classes may be organized in specialization hierarchies. The set of modeling concepts offered by the underlying data model is further extended with modeling concepts that are specific to the modeling of organizational processes, inter-related organizational objects, and protection requirements as well as authorizations. (b) *Simplicity*: Although the workflow model is very expressive, i.e., it offers a rich set of concepts for the modeling of workflow schemas (above), it has been kept simple in that all its submodels build on the same object-oriented data model. As a consequence, the process model, the organization model, and the authorization model offer the same set of basic modeling concepts. (c) *Minimality*: Each modeling concept offered by the workflow model has a meaning that is distinct wrt. every other modeling concept. (d) *Formality*: Workflow schemas built using the presented workflow model are formal specifications of reality. Furthermore, the presented workflow model satisfies the criteria of graphical completeness and ease of reading wrt. its graphical notation. Note: The presented workflow model builds on Object/Behavior Diagrams (cf. [KS91, KS96b], which have originally been introduced for the *conceptual design* of object-oriented database schemas.

Comprehensive Workflow Model. The presented workflow model is comprehensive, since it supports the modeling of all elements of the business area that are relevant in the context of workflow design, i.e., organizational processes, inter-related organizational objects, and protection requirements as well as authorizations (cf. Section 1.2). The workflow model presented in this thesis consists of three submodels, a process model, an organization model, and an authorization model. The *process model* (cf. Chapters 4 and 8) supports the modeling of organizational processes. The *organization model* (cf. Chapters 5 and 9) supports the modeling of inter-related organizational objects. The *authorization model* (cf. Chapters 6 and 10) supports the modeling of protection requirements as well as authorizations. In addition, the workflow model supports the modeling of inter-related structured objects together with their behavior, since the individual submodels of the overall workflow model build on a data model that allows for the conceptual design of object-oriented database schemas (cf. Chapters 3 and 7).

Flexible Workflow Model. The presented workflow model is flexible, since it supports the adaption of workflow systems to changed requirements as well as the extendibility and the reuse of all components of workflow systems. Building on an object-oriented fundament, the presented workflow model benefits from positive aspects of object-orientation. That is, fundamental principles of object-orientation such as inheritance, specialization, encapsulation, information-hiding, object identity, etc., have proved to make software systems easy to extend, to ease their understanding and maintenance, and to allow for a disciplined reuse of design results and implementations. Nevertheless, object-orientation is weak in supporting the evolutionary nature of organizational processes. In fact, organizations must frequently adapt their processes to changed requirements in their business environments to stay competitive. We proposed a novel schema architecture for workflow systems to offer efficient support for the evolution of organizational processes. In short, the benefits of the proposed schema architecture are: (1) *Static aspects*: The proposed schema architecture models knowledge on organizational processes in two different but related schemas. A business-process schema considers knowledge on organizational processes that is external to organizations, a workflow-process schema considers knowledge that is internal to organizations. The so-called two-schema architecture is based on the observation that organizations change only internal knowledge on organizational processes to react to changes in their business environments, since they may not change external knowledge, e.g., knowledge on organizational processes that is based on natural facts or law. This means, the proposed two-schema architecture separates the knowledge on organizational processes into two schemas with the intention that a large number of changes are reflected only in workflow-process schemas, whereas business-process schemas stay unchanged and can be reused. (2) *Dynamic aspects*: Ellis and Keddara [EK95] coined the term "dynamic change" for the evolution of business cases to changed workflow processes. The proposed two-schema architecture offers a flexible and elegant solution to the dynamic-change problem if changes to organizational processes are reflected only in workflow-process schemas. The basic concepts of dynamic change in the context of the two-schema architecture have been discussed in Section 4.2. For a detailed discussion the reader is referred to [BPS97a, BPS97b].

11.3 Future Work

The historical development of workflow technology and research agendas of leading workflow researchers indicate that current workflow technology is still immature. Currently, we are in a conceptual phase of the development of workflow technology that requires research in architectures, conceptual models, and design methodologies (cf. Section 1.1). The research presented in this thesis is a contribution to the current development of workflow technology. It presents a conceptual workflow model for the design of secure workflow systems and proposes a novel schema architecture for workflow systems that offers flexible support for the evolution of organizational processes. In this section we identify the main issues for future work to extend and complete the work presented in this thesis. Subsection 11.3.1 discusses security-specific issues for future work on the conceptual modeling of workflow systems. Subsection 11.3.2 discusses general issues for future work to prove the applicability of our approach to the modeling of workflow systems.

11.3.1 Security Issues

The aim of conceptual modeling is to build schemas that are semantically rich descriptions of reality. We proposed a workflow model for the conceptual design of workflow schemas with a strong emphasize on the conceptual modeling of security-relevant aspects of workflow systems. That is, we proposed modeling concepts for the design of authorization schemas that capture the protection requirements of organizational processes intended to be enacted using workflow technology. Authorization schemas are part of overall workflow schemas. In the following, we focus on main issues of future work on conceptual modeling of security-relevant aspects of workflow systems.

Authority vs. Responsibility. Unlike contemporary WFMSs, our conceptual workflow model considers authorizations and responsibilities as related but separate concepts[3]. Relevant issues for future work in the context of the modeling of authorizations and responsibilities are: (a) *Object-oriented modeling of responsibilities.* We assume a basic model of responsibilities of actors. That is, the responsibility of actors is determined by assignment rules, which are specified by OQL-like queries. This approach is very powerful wrt. the expressiveness of assignment rules, since OQL queries allow to specify assignment rules as arbitrary predicates on objects maintained within workflow systems. However, this approach is poor wrt. to its analysis capabilities. Since responsibilities are not represented explicitly within the presented workflow model but are derived at the time worksteps are to be performed, the *responsibility state* of a workflow system can hardly be analyzed. Considering responsibilities as first-class citizens of workflow systems (similar as we consider authorizations as first-class citizens of workflow systems) could help to circumvent this problem. Appropriate analysis techniques must be developed, where the problem

[3]Note: Contemporary WFMSs determine who *can* access a workstep based on the *responsibility* of actors (cf. [JB96, p. 183]).

of detecting responsibilities of actors for which appropriate authorizations are missing is certainly an important issue (below). (b) *Co-design of protection requirements, authorizations, and responsibilities.* The design of the protection requirements, the design of authorizations, and the design of responsibilities are closely related tasks. The purpose of a conceptual framework for the co-design of protection requirements, authorizations, and responsibilities is to ensure consistency of authorizations and responsibilities wrt. the protection requirements of organizational processes. For example, defining responsibilities of actors that are in conflict with the protection requirements of organizational processes could be a threat against the availability of workflow systems. That is, if assignment rules frequently select actors that are not authorized to perform required worksteps, a workflow engine may have problems to find actors that possess the required authorizations to perform certain worksteps. As a result, it might happen that business cases are not processed in time. Furthermore, system performance is degraded if workflow engine frequently must re-evaluate assignment rules and consider substitutions. (c) *Checking of authorization constraints.* Authorization constraints are a powerful concept to model the protection requirements of organizational processes. Further research is required to offer a conceptual framework for the enforcement of authorization constraints as well as the detection and correction of violations of authorization constraints. The language we use to specify authorization constraints builds on a deontic constraint language that allows to model violations of deontic constraints as well as to model corrective actions. The deontic constraint language has been presented in [WWMD91]. We consider this deontic constraint language as a starting point to include appropriate concepts into the presented language for the modeling of authorization constraints. Furthermore, we are going to investigate the applicability of active database concepts, i.e., event detection and event-condition-action rules, for the detection and correction of violations of authorization states.

Modeling of Security-Relevant Constraints. Our workflow model organizes authorizations of actors around roles, which represent positions that actors occupy. That is, the presented authorization model (together with the presented organization model) qualifies as a role-based authorization control (RBAC) model. In fact, it qualifies as a $RBAC_3$ model in the family of RBAC reference models presented in [SCFY96], which is the most powerful category of RBAC reference models. $RBAC_3$ models must support constraints that allow the modeling of security principles that are relevant in commercial environments such as separation of duty and least-privilege. Future research aims at identifying further security principles that are relevant in commercial environments and provide appropriate support for their modeling in the presented workflow model.

Substitution. The organization model of our workflow model offers a flexible approach to the modeling of substitutions. In fact, substitutions are modeled as relationship objects and may be defined at different levels of granularity, i.e., positions, roles, worksteps. However, the authorization model takes a rather pragmatic approach. In the case substitution relationships are defined at the level of positions and roles, it assumes that a substitute has all rights associated with the respective position or role, respectively. A more differentiated view of implications of authorizations over substitution relationships is necessary in many cases. Future research aims at associating information with substitution relationships in order to restrict the flow of authorizations over the respective substitution relationships.

11.3.2 General Issues

This subsection discusses general issues for future work. The focus of the issues addressed in the following is to prove the applicability of our approach to the modeling of workflow systems. Using the presented workflow model for the conceptual design of workflow systems in "real" development project requires adequate *tool support* for workflow designers. Furthermore, the translation of conceptual workflow schemas into logical workflow schemas, i.e., the *logical design* of workflow systems, must be supported. Logical workflow schemas can be directly mapped to implementation schemas of particular WFMSs. A problem that might arise during logical design and the subsequent mapping of a logical workflow schema onto an implementation schema of a particular WFMS is that semantics expressed in a conceptual workflow schema cannot be expressed within a logical workflow schema or cannot be mapped onto the implementation schema of a particular WFMS and thus gets lost. The semantic expressiveness of the conceptual workflow model presented in this thesis is very high. In fact, it allows to model several aspects of workflow systems that cannot be captured by the implementation models of contemporary WFMSs. For example, the semantics of authorization schemas cannot be captured by the implementation models of contemporary WFMSs. As a consequence, we propose to develop a *prototype* WFMS that extends the semantic expressiveness of the implementation models of contemporary WFMS products.

Tool Support. The design of large information systems produces workflow schemas that are large in size. The resulting complexity of workflow schemas exceeds the capabilities of humans and requires adequate support. Design tools can offer this support. Design tools separate the presentation of workflow schemas into several diagrams to keep the overall presentation clear and ease the understanding of workflow schemas. Furthermore, design tools allow to cope with the dynamic nature of workflow schemas that must frequently be modified during the design phase.

Recently a project on active object-oriented database design has been finished at the department of business information systems at the Johannes Kepler University of Linz (cf. [Lan97, Obe98]). As part of the project, an editor for the conceptual design of active object-oriented database schemas has been developed, which relies on the graphical notation of Object/Behavior Diagrams (cf. [KS91, KS96b]). The editor is operational an has been presented at the International Conference on Data Engineering in 1998 (cf. [LOKT98]). Since our workflow model and its graphical notation build on the data model and the graphical notation of Object/Behavior Diagrams, too, this editor is a good starting point for the development of an editor for the conceptual design of workflow schemas using the workflow model presented in this thesis.

Logical Design. Logical design follows conceptual design in the design process of workflow systems (cf. workflow design process, Subsection 1.2.4). The purpose of logical design is to translate conceptual workflow schemas into logical workflow schemas, which can be directly mapped onto implementation schemas of particular WFMSs. Jablonski and Stein [JS95] proposed the workflow model of *MOBILE* (cf. [JB96]) as a logical workflow model and showed how to translate a conceptual workflow schema built using the Booch method [Boo94] into a logical workflow schema built using the workflow language of *MOBILE*. A starting point for research in logical design is to offer a mapping of conceptual workflow schemas expressed in the language of our workflow model onto logical workflow schemas expressed in the language of *MOBILE*.

Prototype System. Contemporary WFMSs do not consider authorizations as independent concepts. In fact, they determine who *can* access a workstep based on the *responsibility* of actors (cf. [JB96, p. 183]). Developing a prototype WFMS that considers authorizations and responsibilities as related but separate concepts, that offers concepts for the design of authorization schemas, and that realizes the two-schema architecture for workflow systems is a long-term goal. The purpose is to show the applicability of the concepts presented in this thesis for the development of secure workflow systems. A starting point for the development of a prototype WFMS is the prototype system $TriGS_{flow}$. $TriGS_{flow}$ (cf. [KPRSR95]) is based on *TriGS* (cf. [KRSRV94]), which is an active extension of the commercial object-oriented database system GemStone (GemStone Systems, Inc.). Note: *TriGS* stands for *Tri*ggers in *G*em*S*tone.

Bibliography

[AAAM97] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and Limitations of Current Workflow Management Systems. *IEEE Expert*, 12(5), 1997.

[Aal96] W.M.P. van der Aalst. Petri-net-based Workflow Management Software. In A. Sheth, editor, *Proceedings of the NSF Workshop on Workflow and Process Automation in Infromation Systems: State-of-the-art and Future Directions, Georgia, USA*, May 1996. (http://lsdis.cs.uga.edu/activities/NSF-workflow).

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive Object-Oriented Databases (DOOD'89), Kyoto, Japan*, 1989.

[ADG+92] R. Ahad, J. Davis, S. Gower, P. Lyngbaek, A. Marynowski, and E. Wnuegbe. Supporting Access Control in an Object-Oriented Database Language. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings of the Third International Conference on Extending Database Technology (EDBT'92), Vienna, Austria*, volume 580 of *Lecture Notes in Computer Science*, pages 184–200. Springer Verlag, March 1992.

[AH96] V. Atluri and W.-K. Huang. An Authorization Model for Workflows. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proceedings of the 4th European Symposium on Research in Computer Security (ESORICS'96), Rome, Italy*, volume 1146 of *Lecture Notes in Computer Science*, pages 44–64. Springer Verlag, September 1996.

[AS94] K.R. Abbott and S.K. Sarin. Experiences with Workflow Management: Issues for the Next Generation. In *Proceedings of the International Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 113–120, October 1994.

[Atw93] T. Atwood. ODMG 93: The object DBMS standard. *Object Magazine*, September/October 1993.

[BCN92] C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach.* The Benjamin/Cummings Publishing Company, 1992.

[BJ94] C. Bussler and S. Jablonski. An Approach to Integrate Workflow Modeling and Organization Modeling in an Enterprise. In *Proceedings of the Third IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'94), Morgantown, West Virginia, USA*, April 1994.

[BJ95] C. Bussler and S. Jablonski. Policy Resolution for Workflow Management Systems. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences (HICSS'95)*, pages 831–840, 1995.

[BKMW94] M. Bauer, C. Kohl, H.C. Mayr, and J. Wassermann. Enterprise Modeling Using OOA Techniques. In G. Chroust and A. Benczúr, editors, *Proceedings of the CONnectivity (CON'94), Workflow Management - Challenges, Paradigms and Products*, pages 96–111. Österreichische Computer Gesellschaft, R. Oldenbourg, 1994.

[BM91] E. Bertino and L. Martino. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, pages 33–47, April 1991.

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

[BPS97a] P. Bichler, G. Preuner, and M. Schrefl. Workflow Transparency. In A. Olivé and J.A. Pastor, editors, *Proceedings of the 9th International Conference on Advanced Information Systems Engineering (CAiSE'97), Barcelona, Spain*, volume 1250 of *Lecture Notes in Computer Science.* Springer Verlag, June 1997.

[BPS97b] P. Bichler, G. Preuner, and M. Schrefl. Workflow Transparency. Institutsbericht 97.02, University of Linz, Department of Business Information Systems, Altenberger Str. 69, A-4040 Linz, Austria, 1997.

[Bus96] C. Bussler. Analysis of the Organization Modeling Capability of Workflow-Management-Systems. In *Proceedings of the Conference of the Pacific Research Institute for Information Systems and Management (PRIISM'96), Maui, Hawaii*, January 1996.

[BW94] E. Bertino and H. Weigand. An approach to authorization modeling in object-oriented database systems. *Data & Knowledge Engineering*, pages 1–29, 1994.

[CB97] R.G.G. Cattell and D.K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, 1997.

[CCPP95] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modeling of WorkFlows. In M.P. Papazoglou, editor, *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modeling (OOER'95), Goald Coast, Australia*, volume 1021 of *Lecture Notes in Computer Science*, pages 341–354. Springer Verlag, December 1995.

[CFMS94] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison-Wesley Publishing Company, 1994.

[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.

[CKO92] B. Curtis, M.I. Kellner, and J. Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[CW87] D.D. Clark and D.R. Wilson. A comparison of commercial and military security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In G.M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB'91), Barcelona, Spain*, pages 577–589, September 1991.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, second edition, 1991.

[Daf95] R.L. Daft. *Organization Theory and Design.* West Publishing Company, fifth edition, 1995.

[DAHK91] I. D'Haenens, F. Van Assche, E. Halpin, and B. Karakostas. Experiences with rule-based dynamic modelling. In H.G. Sol and K.M. van Hee, editors, *Dynamic Modeling of Information Systems*, pages 289–301. North-Holland, 1991.

[DGSZ94] G. Dinkhoff, V. Gruhn, A. Saalmann, and M. Zielonka. Business Process Modeling in the Workflow Management Environment LEU. In P. Loucopoulos, editor, *Business Modelling and Re-Engineering, Proceedings of the 13th International Conference on the Entity-Relationship Approach (ER'94), Manchester, UK*, volume 881 of *Lecture Notes in Computer Science*, pages 46–63. Springer Verlag, 1994.

[DGW94] W. Deiters, V. Gruhn, and H. Weber. Software Process Evolution in MELMAC. In D.E. Cooke, editor, *The Impact of CASE on the Software Development Life Cycle*, Series on Software Engineering and Knowledge Engineering, pages 301–323. Word Scientific, 1994.

[Dit95] E. Dittrich. Entwurf und Implementierung von Services zur Bereitstellung von Organisations- und Personalstrukturen für ein Workflow-Management-System (WFMS). Master's thesis, Technische Universität Dresden, Fakultät Informatik, 1995. (in German).

[EK95] C.A. Ellis and K. Keddara. Dynamic Change within Workflow Systems. In *Proceedings of the ACM SIGOIS Conference on Organizational Computing Systems (COOCS'95)*, 1995.

[Ell82] C.A. Ellis. OfficeTalk-D: An Experimental Office Information Systems. In *Proceedings of the First ACM Conference on Office Information Systems*, June 1982.

[Ell93] C.A. Ellis. Formal and Informal Models of Office Activity. In *Proceedings of the IFIP International Computer Congress, Paris*, July 1993.

[EN80] C.A. Ellis and G.J. Nutt. Office Information Systems and Computer Science. *ACM Computing Surveys*, 12(1):27–60, January 1980.

[EN96] C.A. Ellis and G.J. Nutt. Workflow: The Process Spectrum. In A. Sheth, editor, *Proceedings of the NSF Workshop on Workflow and Process Automation in Infromation Systems: State-of-the-art and Future Directions, Georgia, USA*, May 1996. (http://lsdis.cs.uga.edu/activities/NSF-workflow).

[EPQ94] E. Ellmer, G. Pernul, and G. Quirchmayr. Security for Workflow Management. In *Proceedings of the 6th IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems, Washington, D.C., USA*, October 1994.

[Ess93] W. Esswein. Das Rollenmodell der Organisation: Die Berücksichtigung aufbauorganisatorischer Regelungen in Unternehmensmodellen. *Wirtschaftsinformatik*, 35(6):551–561, 1993. (in German).

[FCK95] D.F. Ferraiolo, J.A. Cugini, and D.R. Kuhn. Role-Based Access Control (RBAC): Features and Motivations. In Ross [Ros95], pages 241–248.

[FE95] D.G. Firesmith and E.M. Eykholt. *Dictionary of Object Technology: The Definitive Desk Reference*. SIGS Reference Library Series. SIGS Books, Inc., 1995.

[FGS94] E.B. Fernández, E. Gudes, and H. Song. A Model for Evaluation and Administration of Security in Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):275–292, April 1994.

[FLPG93] E.B. Fernández, M.M. Larrondo-Petrie, and E. Gudes. A Method-Based Authorization Model for Object-Oriented Databases. In B. Thuraisingham, R. Sandhu, and T.C. Ting, editors, *Proceedings of the OOPSLA-93 Conference Workshop on Security for Object-Oriented Systems, Washington DC, USA*, pages 135–150, September 1993.

[GE96] T. Gale and J. Eldred. *Getting Results with the Object-Oriented Enterprise Model*. SIGS Books, 1996.

[GHS94] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *International Journal on Distributed and Parallel Databases*, 2(3), September 1994.

[Giu95] L. Giuri. A New Model for Role-Based Access Control. In Ross [Ros95], pages 249–255.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.

[GSR96] G. Gottlob, M. Schrefl, and B. Röck. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[Har88] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[HC93] M. Hammer and J. Champy. *Reengineering the Cooperation: A Manifesto for Business Revolution*. N. Brealey Publishing, London, 1993.

[Her95] H. Herbst. A Meta-Model for Business Rules in Systems Analysis. In J. Iivari, K. Lyytinen, and M. Rossi, editors, *Proceedings of the 7th Conference on Advanced Information Systems Engineering (CAiSE '95), University of Jyväskylä, Finland*, volume 932 of *Lecture Notes in Computer Science*, pages 186–199. Springer Verlag, 1995.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 10(2):201–260, 1987.

[HK94] H. Herbst and G. Knolmayer. Ansätze zur Klassifikation von Geschäftsregeln. Technical Report 46, Institut für Wirtschaftsinformatik der Universität Bern, June 1994. (in German).

[HK95] H. Herbst and G. Knolmayer. Ansätze zur Klassifikation von Geschäftsregeln. *Wirtschaftsinformatik*, 37(2):149–159, 1995. (in German).

[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, *Information Systems Life Cycle*, pages 29–46. Elsevier Science B.V. (North Holland), 1994.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[HM81] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[Hsu93] M. Hsu. Bulletin of the Technical Committee on Data Engineering: Special Issue on Workflow and Extended Transation Systems, June 1993.

[Hür94] W.L. Hürsch. Should Superclasses be Abstract? In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, pages 12–31, 1994.

[Jaj96] S. Jajodia. Database Security and Privacy. *ACM Computing Surveys*, 28(1):129–131, March 1996.

[JB96] S. Jablonski and C. Bussler. *Workflow Management - Modeling Concepts, Architecture and Implementation.* International Thomson Computer Press, 1996.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering. A Use Case Driven Approach.* Addison-Wesley, 1992.

[Jen92] K. Jensen. *Coloured Petri Nets — Basic concepts, analysis methods and practical use.* EATCS monographs on Theoretical Computer Science. Springer Verlag, 1992.

[Joo94] S. Joosten. Trigger Modelling for Workflow Analysis. In *Proceedings of the CONnectivity (CON'94), Workflow Management - Challenges, Paradigms and Products*, pages 236–247. Österreichische Computer Gesellschaft, R. Oldenburg, 1994.

[Joo95] S. Joosten. Conceptual Theory for Workflow Management Support Systems. Technical report, Centre for Telematics and Information Technology, University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands, July 1995.

[Joo96] S. Joosten. Workflow Management Research Area Overview. In *Proceedings of the Second Americas Conference on Information Systems (SACIS'96), Phoenix, Arizona, USA*, pages 914–916, August 1996.

[JS95] S. Jablonski and K. Stein. Die Eignung objektorientierter Analysemethoden für das Workflow Management. *HMD: Theorie und Praxis der Wirtschaftsinformatik*, 32(185):95–115, September 1995. (in German).

[Kap91] G. Kappel. Reorganizing Object Behavior by Behavior Composition - Coping with Evolving Requirements in Office Systems. In H.-J. Appelrath, editor, *Proceedings Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'91), Kaiserslautern, Germany*, pages 446–453, March 1991.

[KB95] S. Khoshafian and M. Buckiewicz. *Introduction to Groupware, Workflow, and Workgroup Computing.* John Wiley & Sons, Inc., 1995.

[KBC+87] W. Kim, J. Banerjee, H.-T. Chou, J.F. Garza, and D. Woelk. Composite Object Support in an Object-Oriented Database System. In *Proceedings of the International Conference on Object-Oriented Programming Languages and Applications (OOPSLA'87), October 4-8*, pages 118–125, 1987.

[KBG89] W. Kim, E. Bertino, and J.F. Garza. Composite Objects Revisted. In *Proceedings of the ACM-SIGMOD Conference on Management of Data (SIGMOD'89), SIGMOD Record, Portland*, volume 18, pages 337–347, June 1989.

[KBKS96] P. Kueng, P. Bichler, P. Kowalek, and M. Schrefl. How to Compose an Object-Oriented Business Process Model? In S. Brinkkemper et al., editors, *Proceedings of the IFIP TC8, WG 8.1/WG 8.2 Working Conference, Method Engineering – Principles of method construction and tool support, (ME'96), Atlanta.* Chapmann & Hall, London, August 1996.

[KC86] S.N. Khoshafian and G.P. Copeland. Object Identity. In *Proceedings of the International Conference on Object-Oriented Programming Languages and Applications (OOPSLA'86)*, pages 406–416, September 1986.

[KHS94] G. Knolmayer, H. Herbst, and M. Schlesinger. Enforcing Business Rules by the Application of Trigger Concepts. In *Proceedings Priority Programme Informatics Research, Information Conference Module 1, Berne: Swiss National Science Foundation*, pages 24–30, 1994.

[Kir95] S. Kirn. Cooperative Knowledge Processing: The Key Technology for Future Organizations. Technical Report Report No. 39, University of Münster, Institute of Business Informatics, July 1995.

[Koh94] C. Kohl. Die Anwendbarkeit von OOA-Konzepten in der Unternehmensmodellierung. In *Proceedings des EMISA/MOBIS Workshop "Modellierung von*

Geschäftsvorfällen und -prozessen", Münster, Germany, pages 51–53, 1994. (in German).

[Koh96] C. Kohl. Objektorientierte Analysekonzepte in der Unternehmensmodellierung. In G. Vossen and J. Becker, editors, *Geschäftsprozeßmodellierung und Workflow-Management: Modelle, Methoden, Werkzeuge*, chapter 4, pages 63–79. International Thomson Publishing, 1996. (in German).

[KPRSR95] G. Kappel, B. Pröll, S. Rausch-Schott, and W. Retschitzegger. TriGS$_{flow}$ – Active Object-Oriented Workflow Management. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences (HICSS'95)*, pages ppx–ppy, 1995.

[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 (JOOP)*, pages 40–63, July–August 1994.

[KS91] G. Kappel and M. Schrefl. Object/Behavior Diagrams. In *Proceedings of the 7th International Conference on Data Engineering (ICDE'91), Kobe, Japan*, pages 530–539, April 1991.

[KS92] G. Kappel and M. Schrefl. Local Referential Integrity. In G. Pernul and A.M. Tjoa, editors, *Proceedings of the 11th International Conference on Entity-Relationship Approach (ER'92)*, volume 645 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[KS95] J.R. Katzenbach and D.K. Smith. The Wisdom of Teams – Creating the High-Performance Organization. Bookmark 13.0, In [Daf95, p. 464], 1995.

[KS96a] G. Kappel and M. Schrefl. Modeling Object Behavior: To Use Methods or Rules or Both? In *Proceedings of the 7th International Conference on Database and Expert System Applications (DEXA '96)*, volume 1134 of *Lecture Notes in Computer Science*, pages 584–602. Springer Verlag, September 1996.

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

[KU94] S. Kirn and R. Unland. Workflow Management mit kooperativen Softwaresystemen: State of the Art und Problemabriß. Technical Report Report No. 29, University of Münster, Institute of Business Informatics, March 1994. (in German).

[Lan97] P. Lang. *Conceptual Design of Active Object-Oriented Database.* PhD thesis, University of Linz, Department of Business Information Systems, Research

Group for Data & Knowledge Engineering, Altenberger Straße 69, A-4040 Linz, September 1997.

[LARB+91] F. Lehner, W. Auer-Rizzi, R. Bauer, K. Breit, J. Lehner, and G. Reber. *Organisationslehre für Wirtschaftsinformatiker*. Hanser Studienbücher. Carl Hanser Verlag, 1991. (in German).

[LMSY95] E.C. Lupu, D.A. Marriott, M.S. Sloman, and N. Yialelis. A Policy Based Role Framework for Access Control. In *Proceedings of the First ACM/NIST Role Based Access Control Workshop (RBACW'95), Gaithersburg, USA*, December 1995.

[LOKT98] P. Lang, W. Obermair, W. Kraus, and T. Thalhammer. A Graphical Editor for the Conceptual Design of Business Rules. In *Proceedings of the 14th International Conference on Data Engineering (ICDE'98), Orlando, Florida, USA*. IEEE Computer Society Press, February 1998.

[LOS96a] P. Lang, W. Obermair, and M. Schrefl. Situation Diagrams. In *Proceedings of the 7th International Conference on Database and Expert System Applications (DEXA'96)*, volume 1134 of *Lecture Notes in Computer Science*, pages 400–421. Springer Verlag, September 1996.

[LOS96b] P. Lang, W. Obermair, and M. Schrefl. Structural Query Diagrams. Department of Business Information Systems, University of Linz, (manuscript), June 1996.

[LOS97] P. Lang, W. Obermair, and M. Schrefl. Modeling Business Rules with Situation/Activation Diagrams. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97), Birmingham, UK*, pages 455–464. IEEE Computer Society Press, April 1997.

[LR94] F. Leymann and D. Roller. Business Process Management with FlowMark. In *Proceedings of the 39th IEEE Computer Society International Conference (COMPCON'94), Digest of Papers*, pages 230–233. IEEE Computer Society Press, 1994.

[LTP91] P. Loucopoulos, B. Theodoulidis, and D. Pantazis. Business Rules Modelling: Conceptual Modelling and Object-Oriented Specifications. In F. Van Assche, B. Moulin, and C. Rolland, editors, *Object Oriented Approach in Information Systems*, pages 323–342. Elsevier Science Publishers B. V. (North-Holland), 1991.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

[MLHH93] G. Müller-Luschnat, W. Hesse, and N. Heydenreich. Objektorientierte Analyse von Geschäftsvorfallsmodellierung. In H.C. Mayr and R. Wagner, editors, *Proceedings Objektorientierte Methoden für Informationssysteme, Fachtagung der GI-Fachgruppe EMISA, Klagenfurt, Austria*, pages 74–90. Springer Verlag, June 1993. (in German).

[MMWFF92] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The Action Workflow Approach to Workflow Management Technology. In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW'92), Toronto*, pages 281–288, 1992.

[MMWM93] N.M. Mattos, K. Meyer-Wegener, and B. Mitschang. Grand tour of concepts for object-orientation from a database point of view. *Data & Knowledge Engineering*, 9:321–351, 1993.

[MNP+91] P. McBrien, M. Niézette, D. Pantazis, A.H. Seltveit, U. Sundin, B. Theodoulidis, G. Tziallas, and R. Wohed. A Rule Language to Capture and Model Business Policy Specifications. In R. Andersen, J.A. Bubenko Jr., and A. Sølberg, editors, *Proceedings of the Third International Conference on Advanced Information Systems Engineering (CAiSE'91), Trondheim, Norway*, volume 498 of *Lecture Notes in Computer Science*, pages 307–318. Springer Verlag, May 1991.

[MS88] J.D. Moffet and M.S. Sloman. The Source of Authority for Commercial Access Control. *IEEE Computer*, 21(2):59–69, February 1988.

[MS91] J.D. Moffet and M.S. Sloman. Delegation of authority. In I. Krishnan and W. Zimmer, editors, *Integrated Network Management II, Washington D.C., April 1–5*, pages 595–606. IFIP, Elsevier Science Publishers B.V. (North-Holland), 1991.

[MS93] D.R. McCarthy and S.K. Sarin. Workflow and Transactions in InConcert. In Hsu [Hsu93], pages 53–56.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–579, April 1989. (invited paper).

[Mye97] B.A. Myers. Interface Software Technology. In A.B. Tucker Jr., editor, *The Computer Science and Engineering Handbook*, chapter 72, pages 1571–1595. CRC Press, 1997.

[NO93] M. Nyanchama and S. Osborn. Role-Based Security, Object Oriented Databases & Separation of Duty. *SIGMOD-Record*, 22(4), December 1993.

[Obe98] W. Obermair. *Active Object-Oriented Database: From Conceptual Design to Logical Design.* PhD thesis, University of Linz, Department of Business

Information Systems, Research Group for Data & Knowledge Engineering, Altenberger Straße 69, A-4040 Linz, February 1998.

[OSS94] A. Oberweis, G. Scherrer, and W. Stucky. INCOME/STAR: Methodology and Tools for the Development of Distributed Information Systems. *Information Systems*, 19(8):643–660, 1994.

[Oul95] M.A. Ould. *Business Processes: Modelling and Analysis for Re-engineering and Improvement*. J. Wiley & Sons, 1995.

[PB93] G. Pomberger and G. Blaschek. *Grundlagen des Software Engineering: Prototyping und objektorientierte Software-Entwicklung*. Carl Hanser Verlag, 1993. (in German).

[Per94] G. Pernul. Database Security. *Advances in Computers*, 38:1–74, 1994.

[Pet77] J.L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[R+97] Rational Software et al. UML Notation Guide, Version 1.1, September 1997. (http://www.rational.com/uml/html/notation/).

[RBA95] *Proceedings of the First ACM Workshop on Role-Based Access Control (RBAC'95), Gaithersburg, Maryland, USA*. ACM Press, November 1995.

[RBKW91] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A Model of Authorization for Next-Generation Database Systems. *ACM Transactions on Database Systems*, 16(1):88 – 131, March 1991.

[RBPE91] J. Rumbaugh, M. Blaha, W. Premerlani, and F. Eddy. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[Ros95] R. Ross, editor. *Proceedings of the 11th Annual Computer Security Applications Conference (ACSAC'95), New Orleans, Louisiana, December 11–15*. IEEE Computer Society Press, 1995.

[Rup90] W. Rupietta. An Organization & Resources Model for Adapting Office Systems to Organizational Structures. In A. Tjoa and R. Wagner, editors, *Proceedings of the Conference on Database and Expert Systems Applications (DEXA '90)*, pages 346–350, 1990.

[Rup97] W. Rupietta. Organization and Role Models for Workflow Processes. In P. Lawrence, editor, *Workflow Handbook 1997*, pages 165–172. John Wiley & Sons Ltd., 1997.

[RW94] W. Rupietta and G. Wernke. Umsetzung organisatorischer Regelungen in der Vorgansbearbeitung mit WorkParty und ORM. In U. Hasenkamp, S. Kirn, and M. Syring, editors, *CSCW - Computer Supported Cooperative Work: Informationssyteme für dezentralisierte Unternehmensstrukturen*, chapter 7, pages 135–154. Addison-Wesley, 1994. (in German).

[San93] R.S. Sandhu. On Five Definitions of Data Integrity. In *Proceedings of the IFIP WG 11.3 Workshop on Database Security, Lake Guntersville, Alabama, September 12–15*, 1993.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.

[Sch88] M. Schrefl. *Object-Oriented Database Integration.* PhD thesis, Technical University of Vienna, 1988.

[Sch91] M. Schrefl. Behavior Modeling by Stepwise Refining Behavior Diagrams. In H. Kangassalo, editor, *Entity Relationship Approach: The Core of Conceptual Modelling (ER'91)*, pages 119–134. Elsevier Science Publishers B.V. (North-Holland), 1991.

[Sch98a] A.-W. Scheer. *ARIS – Modellierungsmethoden, Metamodelle, Anwendungen.* Springer Verlag, third edition, 1998.

[Sch98b] A.-W. Scheer. *ARIS – Vom Geschäftsprozeß zum Anwendungssystem.* Springer Verlag, third edition, 1998.

[Sci89] E. Sciore. Object Specialization. *ACM Transactions on Information Systems*, 7(2):103–122, April 1989.

[She93] A. Sheth. On Multi-system Applications and Transactional Workflows (A Collection of Papers and Reports), August 1993.

[She97] A. Sheth. From Contemporary Workflow Process Automation to Adaptive and Dynamic Work Activity Coordination and Collaboration. In *Proceedings of the Workshop on Workflow Management in Scientific and Engineering Applications, Toulouse, France*, September 1997. (keynote talk).

[SN93] D. Schuler and A. Namioka, editors. *Participatory Design: Principles and Practices.* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1993.

[SS95] M. Schrefl and M. Stumptner. Behavior Consistent Extension of Object Life Cycles. In *Proceedings of the International Conference on Object-Oriented and Entitiy-Relationship Modeling (OOER'95)*, volume 1021 of *Lecture Notes in Computer Science*, pages 133–145. Springer Verlag, 1995.

[SS97] M. Schrefl and M. Stumptner. Behavior Consistent Refinement of Object Life Cycles. In *Proceedings of the 16th International Conference on Conceptual Modeling (ER'97), Los Angeles, USA*, 1997.

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

[TDH92] T.C. Ting, S.A. Demurjian, and M.-Y. Hu. Requirements, Capabilities, and Functionalities of User-Role Based Security for an Object-Oriented Design Model. In B.M. Thuraisingham and C.E. Landwehr, editors, *Database Security, VI: Status and Prospects*, volume 6, pages 275–296. North-Holland, 1992.

[Thu89] M.B. Thuraisingham. Mandatory Security in Object-Oriented Database Systems. In *Proceedings of the International Conference on Object-Oriented Programming Languages and Applications (OOPSLA'89), October 1–6*, pages 203–210. ACM Press, 1989.

[TS93] R.K. Thomas and R.S. Sandhu. Towards a Task-based Paradigm for Flexible and Adaptable Access Control in Distributed Applications. In *Proceedings of the 1992-1993 ACM SIGSAC New Security Paradigms Workshops, Little Compton, RI*, pages 138–142, 1993.

[TS94] R.K. Thomas and R.S. Sandhu. Conceptual Foundations for a Model of Task-based Authorizations. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop (CSFW'94)*, pages 66–79, June 1994.

[TS96] R. Thomas and R. Sandhu. Task-based Authorization: A Research Project in Next-generation Active Security Models for Workflows. (http://www.oracorp.com/tba.htm), April 1996.

[TYF86] T.J. Teorey, D. Yang, and J.P. Fry. A Logical Design Methodology for Relational Databases Using The Extended Entity-Relationship Model. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[Ver96] F.B. Vernadat. *Enterprise Modeling and Integration – Principles and Applications*. Chapman & Hall, 1996.

[WF94] T.E. White and L. Fischer, editors. *New Tools for New Times: The Workflow Paradigm*. Future Strategies, Inc., 1994.

[Wor95] The Workflow Management Coalition. Coalition Overview, August 1995. (http://www.aiim.org/wfmc/).

[Wor96] The Workflow Management Coalition. Terminology & Glossary. The Workflow Management Coalition Specification — Document Number WFMC-TC-1011, June 1996. (http://www.aiim.org/wfmc/).

[WWMD91] R.J. Wieringa, H. Weigand, J.-J.Ch. Meyer, and F.P.M. Dignum. The Inheritance of Dynamic and Deontic Integrity Constraints. *Annals of Mathematics and Artificial Intelligence*, 3(2–4):393–428, 1991. Special Issue on Deductive Databases, S.A. Naqvi and S. Tsur (eds.).

[WWW^{+}97] D. Wodtke, J. Weissenfels, G. Weikum, A. Kotz Dittrich, and P. Muth. The MENTOR Workbench for Enterprise-wide Workflow Management. In *Proceedings of the ACM-SIGMOD Conference on Management of Data (SIGMOD'97), SIGMOD Record, Arizona, USA*, pages 576–579. ACM Press, 1997.

Curriculum Vitae

Personal Record

Name:	Dipl.-Ing. Peter Bichler
Date of Birth:	7 June 1968
Place of Birth:	Lenzing, Upper Austria
Citizenship:	Austria
Home Address:	A-8160 Weiz, Garrach 43, Styria

University Education

1988–93:	Study of Computer Science at the Johannes Kepler University of Linz, Austria
1993–98:	Ph.D. study at the Johannes Kepler University of Linz, Austria

Professional Experience

1993:	Student associate ("Studienassistent") at the Department of Business Information Systems, Data & Knowledge Engineering, at the Johannes Kepler University of Linz, headed by Prof. Michael Schrefl
1993–97:	Teaching and research associate ("Universitätsassistent") at the Department of Business Information Systems, Data & Knowledge Engineering, at the Johannes Kepler University of Linz, headed by Prof. Michael Schrefl