Definition of Behavior in Object-Oriented Databases by View Integration

DISSERTATION
zur Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

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

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

Verfaßt von: **Dipl.-Ing. Günter Preuner**

Linz, im September 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 30. September 1998

Danksagung

Zuallererst möchte ich mich bei Prof. Michael Schrefl bedanken, der mir durch die Aufnahme in seine Forschungsgruppe die Möglichkeit gegeben hat, diese Dissertation zu verfassen. Wichtige Ideen und Lösungsansätze sind auf Grund zahlreicher Diskussionen mit ihm entstanden und weiterentwickelt worden. Ich bedanke mich auch bei Prof. Gerti Kappel, die sich bereit erklärt hat, die Zweitbeguachtung dieser Arbeit zu übernehmen.

Besonderer Dank gilt den Kolleginnen und Kollegen am Institut für Wirtschaftsinformatik, Forschungsschwerpunkt Data & Knowledge Engineering, für die angenehme Atmosphäre, in der ich in den letzten Jahren arbeiten konnte. Hier möchte ich besonders meinen Kollegen Peter Bichler erwähnen, mit dem meine erste Publikation entstand und der immer bereit war, Ideen zu diskutieren. Natürlich gilt mein Dank auch Heidelinde Kneißl, Peter Küng, Peter Lang, Ingeborg Liedlbauer, Werner Obermair, Thomas Prückler, Margit Rudelstorfer, Roman Schendl und Thomas Thalhammer.

Ich bedanke mich auch bei meinen Eltern, die mich bei meinem Studium so gut unterstützten, und bei allen Freunden und Bekannten, bei denen ich mich speziell in den Monaten vor Vollendung dieser Arbeit viel zu selten sehen ließ.

Günter Preuner

Kurzfassung

Datenbanken für große Organisationen werden üblicherweise von einer Gruppe von Personen entworfen, die potentielle zukünftige Benutzer der Datenbank umfaßt. Diese Benutzer kennen das Anwendungsgebiet der neuen Datenbank auf Grund ihrer täglichen Arbeit. Der Entwurf von Datenbanken durch Integration von Benutzersichten bedeutet, daß mehrere Benutzer ihre Sichten auf die zu erstellende Datenbank definieren. Diese Sichten werden gesammelt und zu einem konzeptuellen Datenbankschema integriert.

Wir behandeln Integration von Sichten im Zusammenhang mit der Modellierung von Geschäftsprozessen und verwenden zur Modellierung von Struktur und Verhalten von Geschäftsobjekten das objektorientierte Datenmodell *Objekt/Verhaltensdiagramme.* Bei der Integration von Sichten konzentrieren wir uns in dieser Arbeit auf das *Verhalten* von Geschäftsobjekten. Das Verhalten von Objekten wird mit Hilfe einer Zwei-Ebenen-Architektur definiert: *Geschäftsprozesse* modellieren das Verhalten von Geschäftsobjekten gemäß naturgegebener Fakten oder anderer dauerhafter Regelungen; *Workflows* hingegen definieren die aktuelle Bearbeitung in einer Organisation auf Grund interner Regeln.

Benutzer definieren ihre Sichten so, wie sie die Bearbeitung von Objekten in der Organisation wahrnehmen und modellieren daher Workflows. Außerdem definieren verschiedene Benutzer verschiedene Sichten, die sich wie folgt voneinander unterschieden können: (1) Sichten repräsentieren das Verhalten verschiedener Entitäten der realen Welt. (2) Verhalten wird unterschiedlich detailliert modelliert oder Teile davon werden weggelassen. (3) Vergleichbare Information wird mit verschiedenen Modellierungskonzepten dargestellt. (4) Aktivitäten werden mit oder ohne Spezifikation im Sinne einer Parameterschnittstelle definiert.

Aufgabe der Integration von Sichten ist es, ein konzeptuelles Datenbankschema mit den vollständigen Geschäftsprozessen zu modellieren. Dabei werden die für Geschäftsprozesse relevanten Informationen aus den Workflows extrahiert, die Objekttypen des integrierten Schemas bestimmt sowie die integrierten Geschäftsprozesse und die Spezifikationen der Aktivitäten ermittelt. Ziel der Integration ist es, Geschäftsprozesse zu modellieren, die den Benutzersichten dahingehend entsprechen, daß jede Bearbeitung eines Objekts entsprechend dem Geschäftsprozeß im konzeptuellen Schema als korrekte Bearbeitung in jeder Benutzersicht beobachtet werden kann.

Abstract

Databases in large organizations are usually designed by a group of people including potential future users of the database who know the application domain from everyday business. The design of databases by *view integration* means that several users or groups of users define their views on the proposed database. These views are collected and integrated to one conceptual database schema.

We treat view integration in the context of modeling business processes and workflows and use the object-oriented data model *Object/Behavior Diagrams* to define the structure and behavior of business objects. In this work, we will focus on the definition and integration of views of the *behavior* of objects. Behavior of business objects is defined based on a two-schema architecture: *Business processes* define the overall behavior of business objects according to external business rules, which mainly reflect long-term business rules due to natural facts or law; *workflows* represent the current processing of business objects in an organization according to internal business rules.

Users define their views as they perceive processing of business objects in everyday business, thereby modeling workflows. Further, views of different users comprise different information: (1) Views model behavior of different entities of the real world; (2) views define processes at different levels of detail or omit parts of the processes; (3) views contain comparable information that is defined using different modeling concepts; (4) views define specifications of activities by providing an interface or refer to activities without defining an interface.

The goal of view integration is to define the complete set of object types and business processes in the conceptual database schema. Thus, information relevant for the business processes is extracted from the workflows, the object types of the integrated schema are determined, and a business process as well as the activity specifications are defined for each object type. The goal of view integration is to define object types with business processes in a way that the processing of objects according to a business process in the conceptual schema can be observed as a correct processing in each view if information that is not defined in the view is ignored.

Contents

Chapter 1

Introduction

Contents

In modern organizatons, information systems are used to store and process information that is needed for reaching the business goals of the organization. An information system does not only comprise data of entities but also has to handle applications that access the data. The field of *business process modeling* deals with the question which activities are performed on the entities and in which order these activities can be executed.

During the *design* of a large information system for an organization, all requirements of the information system have to be defined. No single person will have a complete overview of the information that has to be represented in the information system, i.e., the handled data and the way of processing these data. Therefore, several users from different organizational units of the organization will usually define their view on the entities of the real world that they deal with and on the processing of these entities. In a subsequent step, these views are integrated to build the information system.

This thesis deals with integrating the users' views on the information system. We will assume that the user views are defined in some high-level notation. We will prepare these views such that they fulfill some quality criteria and integrate the prepared views to one single schema that represents all information that has to be handled by the information system.

We will focus on modeling information about structure and behavior of the entities that are treated by business processes and that have to be represented in a database. We will omit discussion about external applications, organizational structures, obligations and permissions of users, etc. Further, we concentrate on an early phase of design, where the views and the integrated schema are defined in a high-level notation.

This introducing chapter is structured as follows: In Section 1.1, a motivation for this work is given, Section 1.2 gives a brief overview of the process of database design and the embedding of view integration in this process. Section 1.3 presents the idea of our integration process; in Section 1.4, we give an overview of related work. Finally, Section 1.5 outlines the topics treated in this work.

1.1 Motivation

Modern organizations have often to deal with information systems containing large quantities of data, with particular characteristics: (1) lots of users have to access data, (2) a plenty of applications use data, (3) users and applications are distributed to different places, (4) users and applications use different parts of the stored data for different purposes.

A room reservation systems for hotels may handle information on room reservations of a hotel chain with several incorporated hotels and cooperations with travel agencies that may book rooms in the hotels. Within the hotels, several users have to access information about reservations, e.g., the booking department, which treats requests, the reception, which is responsible for check-in and check-out of guests and has to deal with inquiries of the guests, the accounting department, which has to account payments, etc. All these users use different information on reservations.

Database systems proved to be very useful to cope with the data requirements of such information systems, i.e., they are meant for handling large quantities of data that are used by different users and applications at the same time. Conventional database systems in the past, which were based on the hierarchical, network, or relational data model, could only store *structural information* on entities of the real world, but no behavior. All the knowledge about processing data in the database was hidden in the applications that accessed the database. As applications did not belong to the database themselves, behavior was defined *outside* the database, i.e., separated from the data.

Object-oriented database systems allow to define object types that specify behavior of entities together with the structure. Entities of the real world are represented by objects; the structure and behavior of similar objects is commonly defined by object types. In difference to former database models, where only structure of entities could be defined within the database, object-oriented database models allow to define structure of objects and the activities to access objects (often called *methods*) together.

In business information systems, the definition of structure and activities of entities seems not to be sufficient. Activities must not be invoked in any order at any time. Yet, the field of *business process modeling* deals with the question, which methods can or must be invoked on objects in which order, e.g., two activities *check-in* and *check-out* defined on reservations must not be invoked in any arbitrary order, but *check-in* always has to be invoked before *check-out*. The database should reflect these ordering requirements, i.e., it should allow to define a business process over the objects stored in the database.

In this work, we will define business objects by object types of an object-oriented schema and the business processes as the behavior of instances of object types. Therefore, the object-oriented model used in this work must allow to define the following information about objects: (1) *structure*, (2) *activities*, and (3) *object life-cycles*. Object life-cycles define possible sequences of activity invocations. In this work, activities and object life-cycles will be commonly refered to as the *behavior* of objects.

Building information systems requires a *design process* during which the structure and behavior of entities of the real world have to be determined and mapped to definitions of object types in a database schema. The database schema will either be defined by a project group which analyses the requirements on the database and develops the database schema in a top-down manner. Such an approach will possibly be chosen if a completely new database schema without experience on which information has to be defined there is built.

In many cases, a new database schema that must be defined reflects information about entities of the real world that have already been existing in the organization. These entities have not existed as database objects so far, but may have been stored in paper forms and records, books, data files etc. The overall processing of these entities is known, too, as these entities have been treated in some way up to present. Yet, knowledge about structure and processing of these entities is often not written down in some complete documentation, but is "distributed" among several users, who have the processing steps that they execute on entities in their minds or written down in some notes or instructions. This means that the people who have processed entities so far and who will use the information system in the future, will have much knowledge about the processing. This knowledge should be used to define the structure and behavior of the database objects of the new system.

Thus, we assume that several users or groups of users define their views on the database schema using some high-level notation, where one view schema is defined by each user (or group of users). The database schema is built by comparing the view schemas, identifying the corresponding and different knowledge, and integrating the views to one single schema. The latter design phase is usually called *view integration*.

View integration is considered useful for several reasons: Sølvberg and Kung [SK93] point out that the "success in system development depends on effective communication." Users are the specialists in the application domain where the information system is built and thus have to participate in specification of the information system. Similarly, Spaccapietra and Parent [SP94] mention that no single user can grasp the whole complexity of a database schema for an organization.

1.2 Process of Database Design

In this section, we roughly mention main issues on the ANSI/SPARC architecture and the process of database design that is commonly used in database literature; further, we will briefly discuss how view integration is embedded into database design. For more information on the ANSI/SPARC architecture and the process of database design in general, the reader is refered to standard textbooks about database systems (e.g., [EN94]).

The ANSI/SPARC architecture [TK78] defines a three-schema structure of databases: a set of external schemas, a conceptual schema, and a physical schema (cf. Figure 1.1). The *conceptual schema* describes the complete

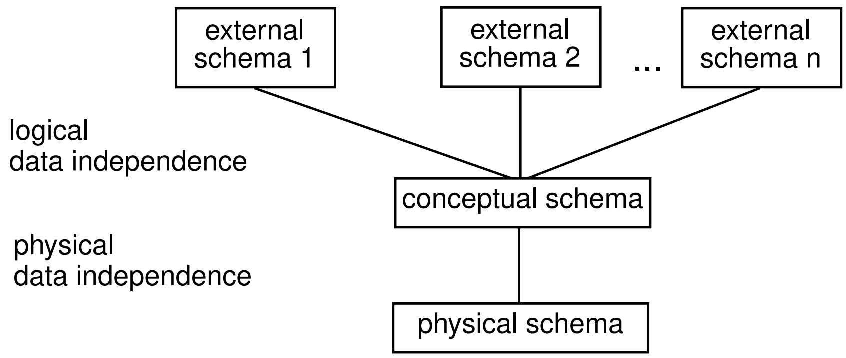


Figure 1.1: ANSI/SPARC three-schema architecture

schema of the whole database in some high-level or implementation data model. The *external schemas* (or *view schemas*) describe parts of the conceptual database schema that are relevant and visible to some user or group of users. The *internal schema* describes the physical structure of the whole database with details about physical storage, indexing, etc. Two kinds of independence are defined: *logical data independence* and *physical data independence*. The former allows to change the specification of the conceptual schema without changing the external schemas, the latter allows to change the physical schema without influencing the definition of the conceptual schema.

As we can see from this architecture, the external schemas or views are defined based on the conceptual schema, where the conceptual schema contains the full definition of the database and the views consider only parts of the conceptual schema relevant to certain users. Notice that the schemas are not necessarily defined in the order (1) external schemas, (2) conceptual schema, and (3) physical schema. Yet, the conceptual schema can be defined first and the external schemas can be derived from the conceptual schema later.

If the conceptual schema of a database is defined by integrating the user views, the user views exist first; the integrated schema has to be defined in a way that the existing views are correct views of the conceptual schema. Correctness criteria will be given later in this chapter. Intuitively, a view can only include information that is derivable from the conceptual schema.

In database literature, usually a process of database design is presented which — in a simplied version — consists of three phases, the *conceptual, logical,* and *physical design* (cf. Figure 1.2). Based on the requirements, a conceptual database schema is defined during conceptual design. The conceptual schema is usually defined with a high-level data model. High-level

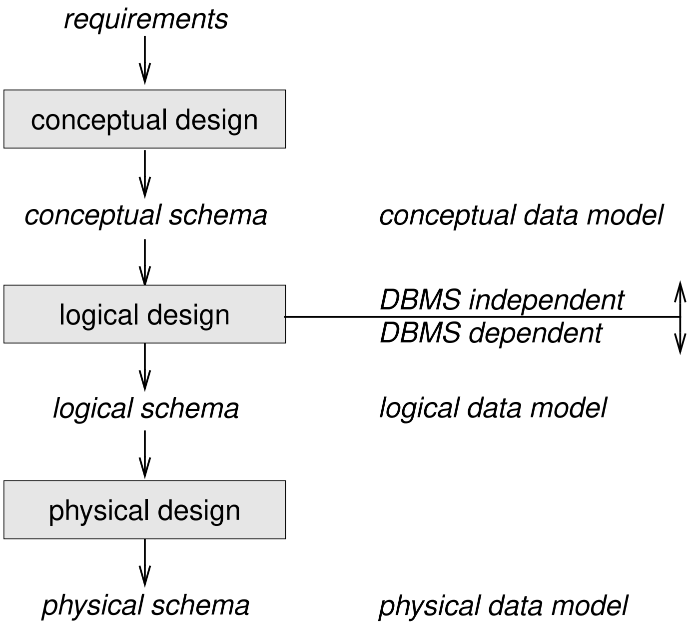


Figure 1.2: Database design process

data models and conceptual schemas are independent from the used database system. During logical design, the conceptual schema is translated into an implementation schema, which depends on the used database system. Thus, the logical schema is not independent from the database system. During physical database design, the physical structure of the schema is defined.

How is view integration embedded in this design process? Usually, views are defined in the phase of conceptual design using a conceptual data model. In this phase, the views are integrated to an integrated conceptual schema, which is again defined using a conceptual data model. The authors of [GLN92] point out that, alternatively, the views could be translated to an implementation schema first and then integrated. In general, view integration takes place at the conceptual level as the most expressive data model is used in that phase. In our approach, we will treat integration of views at the conceptual level, using a conceptual design notation, namely *Object/Behavior Diagrams* (OBD).

Therefore, at the end of the conceptual design phase, one conceptual schema is defined, which can be translated to a logical schema, i.e., the peculiarities of view integration only concern the conceptual design. As this work focuses on view integration, we will only consider the conceptual design of databases.

1.3 Overview of the Integration Approach

In this section, we will give an overview of our integration approach, which is presented in this work. Notice that this section is not intended for a complete description of the integration process, but it should point out the main ideas how views and the conceptual schema that results from integrating the views are related to each other. Therefore, we will restrict the description of the integration approach to those aspects that are needed to convey an idea of the approach to the reader.

There is a set of view schemas, where each view describes a part of the whole conceptual schema. The conceptual schema is defined by integrating the information defined in the views. The scope of our work will be the integration of *behavior* of objects. Behavior of objects in OBD consists of the definition of activities, which are comparable to methods in object-oriented programming languages, and object life-cycles, which show in which order update activities can be performed on an object.

In this chapter, we will not define and use the semantics of OBD in detail. Yet, we will use OBD as far as necessary for this introduction. Behavior is defined by *Behavior diagrams*, which comprise *life-cycle diagrams* to represent the life-cycle of objects and *activity specification diagrams* to show the interface of activities. The notation of life-cycle diagrams is based on Petri nets [Pet77]; yet, in OBD, we will speak of *activities* and *states* instead of *transitions* and *places* and slightly change semantics (these changes are not relevant for the moment; they will be explained in Section 2.2).

The view schemas are defined by different users, who describe their view on the schema. Particularly, they define object types, activities that have to be performed on instances of these object types and allowed sequences of activity invocations. As no user has a complete overview of the complete processing of objects, the conceptual schema is composed from all the views, where correspondences between the views, i.e., features that are defined in several views, and differences between the views are determined and the complete schema is built. Although no view is complete, we require that each view schema is correctly modeled according to the rules of OBD, i.e., each view schema follows the rules of how structure and behavior of object types are defined and how subtypes can be defined by specialization of a supertype.

The goal of the process of view integration is to build a conceptual schema, which obeys the definition of OBD, too, and which includes all information defined in the views. Corresponding parts in the view schemas are represented by the same elements in the conceptual schema, heterogeneities between views are resolved. The logical and physical database design is based on the integrated conceptual schema, such that all objects that populate the database are processed according to the conceptual schema. Each view can *observe* processing, where only those parts of objects processed according to the conceptual schema that are defined in the respective view schema are visible. Further, we require, that all information in the conceptual schema is defined in at least one view, i.e., during view integration, no behavior is specified that is not defined in any view.

1.3.1 Definition of Object Types and Type Hierarchies

We will discuss some basic assumptions on the definition of type hierarchies in the conceptual schema and in the views as well as on the question which object types can be instantiated.

If we speak of *entities*, we mean entities of the real world, which have to be represented by objects in the database. In the conceptual schema, each entity of the real world will usually be represented by *one* object, which is an instance of a certain object type. An exception will be the case of *roles*, where one entity may appear in different contexts. In that case, an entity is represented by one single instance of a role-root type and optionally by instances of related role types.

If we speak of an object type, we mean the definition of structure and behavior (*intension*) as well as the set of objects that are instances of this object type (*extension*). In this work, we will not consider the intension and the extension of object types separately. The distinction between intension and extension (or object types and object classes) was particularly introduced for integration of schemas of *existing databases* where the local database schemas represent the intension, whereas the contents of the databases constitute the extension. There, correspondences between intension and extension may contradict each other: suppose that database d_1 comprises an object type o_1 with extension c_1, whereas database d_2 comprises object type o_2 with extension c_2. The integrator may consider o_2 as a subtype of o_1 (intensional aspect); then, the extension c_2 must be a subset of c_1 (extensional aspect), which may not hold for existing databases. In view integration, no existing extensions exist such that no classes need to be compared or integrated; thus, the mentioned conflict cannot arise.

We will require that an object type o_1 and an object type o_2 that are defined in different views and that represent the same intension must be defined based on the same extension, too. They will be integrated to object type o. It is not allowed that an o_i defines behavior based on a limited class of objects. If different extensions are considered, either (1) the intension of the object types is not the same, such that they represent similar object types in the same type hierarchy with different definition of behavior or (2) the intension is the same, but one view defines behavior which considers only a subset of objects and thus need not necessarily be valid for all instances of the integrated object type o.

Specialization of a supertype to a set of subtypes must be disjoint and complete, i.e., each object that is a direct or indirect instance of an object type that has subtypes is an instance of *at most* (disjointness) and *at least* (completeness) one subtype of this object type, i.e., each instance of such an object type must be an instance of exactly one subtype. This criterion corresponds to the *abstract superclass rule,* which says that all inner nodes of type hierarchies, i.e., all object types that have subtypes, must not have direct instances. Only leaf nodes can have direct instances. An object type that can have direct instances is called *instantiable*, whereas all other object types are *non-instantiable.*

1.3.2 Heterogeneities

Differences between views are generally refered to as *heterogeneities* and are caused by the following reasons:

View limitation: A user may not know about *all* entities that will exist in the database. Therefore, he/she can only define his/her view on the behavior of those entities that he/she perceives. Thus, the set of object types defined in a view is limited to a subset of the object types that appear in the conceptual schema. The instances of missing object types are not considered in a view. Another user will perceive these object types and their instances. The criterion that specialization must be complete requires the following condition to hold: if a subtree of a type hierarchy is omitted due to view limitation, whereas another subtree remains in the schema, the common supertype of the subtrees is omitted, too, as otherwise, the specialization would not be complete any more in the view. This means, only *full subtrees* can be missing in a view schema due to view limitation.

View generalization: A user may know about the existence of an object type, but he/she will *not know about all details* of the behavior of its instances; instead, he/she will know only parts of the behavior in full detail, whereas he/she will perceive some other parts only roughly or not at all. Missing information may be modeled in the views of other users.

View restructuring: A chunk of information can be defined using *different modeling concepts.* Users may define the same information in their views but may represent this information with different concepts. Particularly, users can define one single object type instead of a type hierarchy and represent differences in structure and behavior by different alternative activities, additional conditions, etc. Further, related entities of the real world can either be defined by one object type or by several related object types.

Included workflow aspects: We consider integration of behavior in the context of modeling business processes. In Chapter 3, we will distinguish *business processes* and *workflows.* The behavior specified with object types should reflect the business process, but not the workflow. If workflow-specific aspects are defined in a view, they must be omitted before the view can be considered for integration. We will skip this heterogeneity for the moment as a detailed introduction of the concept of *workflow transparency* is necessary before.

In the following, we will discuss the heterogeneities in more detail. Based on these heterogeneities, we will introduce the overall steps of the integration process and will propose rules which correctness criteria must be fulfilled after each integration step.

We distinguish the following kinds of heterogeneities:

Intra-type heterogeneities: This kind of heterogeneities occurs if an object type is defined in several views, but the behavior of instances of this object type is defined differently in each view. Particularly, information is defined in full detail, only abstracted, or is omitted at all due to view generalization. Further, behavior can be defined using different concepts, i.e., the behavior definition of an object type can vary because of restructuring. Restructuring within an object life-cycle will be omitted in this chapter, as knowledge of more details about definition of object life-cycles will be necessary and restructuring within an object life-cycle is not necessary for understanding the overall integration process.

Inter-type heterogeneities: Different views may contain different object types for several reasons: (1) Object types defined in one view are omitted in another view, as the respective objects are not considered in the latter (view limitation). (2) The object types defined in the views represent equal sets of entities, but they define different sets of object types to represent these entities (view restructuring).

Heterogeneities must be resolved during the integration process. We will deal with the overall steps of the integration process in the remainder of this section: Intra-type heterogeneities are resolved during *intra-type integration*, whereas inter-type heterogeneities are resolved during *inter-type integration*. Although the latter is executed before the former in the integration process, we will start with intra-type integration.

1.3.3 Intra-type Integration

During intra-type integration, a set of equivalent object types is integrated to a single object type of the integrated, conceptual schema. An object type of one view is considered *equivalent* to an object type in another view, if both object types represent the same kind of objects and consider the same set of objects, i.e. they represent the same *intension* and *extension*.

Equivalent types exist if several users define behavior of equal sets of objects, yet no one is able to define complete behavior. In practice, a user will define those parts in detail, which correspond to his/her everyday work, other activities are only known roughly or not at all.

Each of these object types models a part of the object life-cycle of the object type that has to be defined in the integrated schema, where — according to view generalization — it defines some information in full detail, some information abstracted, and some information not at all. Activities and states that are defined in full detail are called *basic*, all others are called *abstract*.

The integrated object life-cycle is defined by gathering all the information from the views and build the object life-cycle of the object type in the integrated schema. The integrated object type must include all information defined in the views and fulfill a correctness criterion as defined below. Thus, it deviates from each view as it defines parts of the object life-cycle that are defined abstracted in the view in full detail and it comprises activities and states that are missing in one view but that are defined in another view.

We assume that the information that can be defined based on the views constitutes the complete object life-cycle for an object type. As the complete definition of activities and states is only provided in the integrated schema, the set of correct sequences of activity invocations is only visible there. In the views, the decision about correct sequences of activity invocations cannot be taken as activities or states may be missing or are defined abstract.

Suppose there is a set of view schemas $v_1, \ldots, v_3$, where v_i defines an object life-cycle of object type o_i, for $1 \leq i \leq 3$. The system integrator, who has to integrate the views, decides that o_1, o_2, and o_3 are equivalent; thus, they are represented by one single object type o in the integrated schema.

The object life-cycles are shown in Figure 1.3. Activities are depicted by small rectangles whereas states are depicted by rectangles with a trapezium at the top and at the bottom. Each activity and each state has a name, which is unique within the object life-cycle. Elements depicted with dashed borders are explained later.

Views v_1, v_2, and v_3 comprise object life-cycles for reservations, defined by the reservation department, the reception, and the statistics department, respectively. The object types are considered equivalent as they all deal with the same set of reservations.

The object life-cycles have correspondences and heterogeneities, which have to be determined by the system integrator: Correspondences between activities and states are indicated by equal names (which will certainly not be usual in practical situations, but is assumed for simplicity of this introduction).

Further, the following heterogeneities will be determined:

- *Granularity conflicts:* Activity use is defined as a single activity in view v_1, whereas it is refined in view v_2. This situation causes both a correspondence (i.e., both views consider the use of rooms) and a heterogeneity at the same time (i.e., one view defines an activity more detailed than another view). Further, state created of view v_2 is refined in view v_1. The correspondence between refinements of an activity or a state and the unrefined activity or state are depicted in the figure by framing the former with a symbol with dashed borders.

- *Different aspects:* View v_3 considers the aspect of statistics, which is not modeled in the other views. Statistics is treated in parallel to the use of the reservation.

In the integrated schema, all knowledge from the views is defined. Corresponding parts have a common representation in the integrated (= conceptual) schema. As the conceptual schema includes all information of the view schemas, information at different levels of detail is integrated at the finest level of detail, information that is defined in one view schema, but that is missing in another view schema, is taken into the integrated schema. The schema that results from integrating the views is depicted in Figure 1.3, too.

Finally, we identify requirements that must be fulfilled by the object life-cycle of the object type in the integrated schema with respect to the object life-cycles of the respective object types in the views:

Objects are processed as instances of the object type o in the integrated schema according to the object life-cycle B defined with o. The views

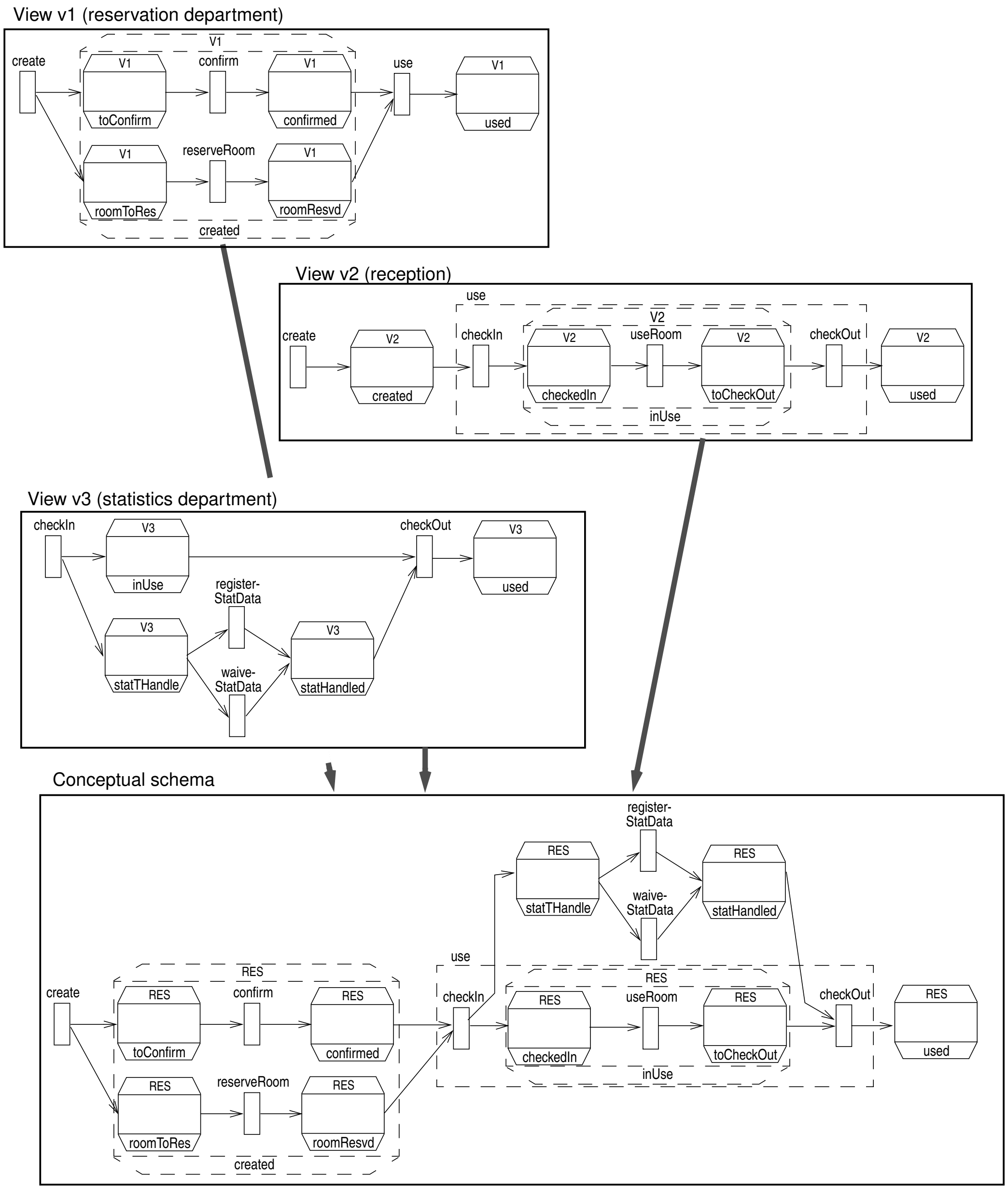


Figure 1.3: Example: intra-type integration

$v_1, \ldots, v_n$ include corresponding object types $o_1, \ldots, o_n$ whose object life-cycles $B_1, \ldots, B_n$ comprise parts of object life-cycle B, where we require that each view v_i should be able to observe the processing of all instances of o as a correct processing according to the object life-cycle B_i defined in the view. To capture this idea formally, we will introduce the notion of *observation consistent specialization* and *observation consistent generalization* and require that object life-cycle B is an observation consistent specialization of object life-cycles $B_1, \ldots, B_n$ and, conversely, that $B_1, \ldots, B_n$ are observation consistent generalizations of B. We will also explain in Chapter 2, that observation consistent specialization allows a subtype to refine activities and states (refinement) and to add new activities and states in new parallel chunks (extension). Refinement and extension are used to represent differences between the object life-cycles $B_1, \ldots, B_n$ and B, too, as motivated by the example.

Further, there should be no refinements and extensions defined in B that are not defined in any view, i.e., we need a "minimal" (or most general) specialization. Therefore, we require that there is no object life-cycle $B' \neq B$, such that B' is an observation consistent specialization of $B_1, \ldots, B_n$, too, and B is an observation consistent specialization of B'. More detailed specifications of this criterion will be introduced later in this work (cf. Section 6.3.1.2).

1.3.4 Inter-type Integration

Inter-type integration is needed to resolve differences between views as object types defined in one view are missing in the other view (view limitation) and as information is represented by different object types (view restructuring). To facilitate the decision whether some entities are considered in one view or several views, the views must be standardized first. During *inter-type standardization*, the view schemas are "normalized" as object types are split to several related object types or object types are split to type hierarchies. After standardization, one-to-one equivalences between object types in different views as well as missing object types can be determined.

1.3.4.1 View Restructuring

As views are defined by different users, they may be structured in a different way although they consider the same entities. Particularly, different entities can be represented by several object types in one view and by one single object type in the other view. Different contexts in which an entity can exist may be defined by a role hierarchy in one view and by a single object type in the other view. Different variants of objects can be defined by one object type or by a type hierarchy with a supertype and several subtypes. If differences between the view schemas exist because of differently structured information, the view schemas must be standardized before they can be integrated.

Our approach of inter-type standardization is based on the concept of *schema standardization*, which has been presented in [Joh94] for the structure of object types. The goal is to define a "standardized" (or normalized) schema from an existing schema, which fulfills the condition that different entities or different roles of entities are represented by different object types and different specializations of an object type are represented by different subtypes of an object type. In the literature, authors point out that standardization should be *information capacity preserving* [MIR93, MIR94], i.e., no information should be added or removed. Notice that schema standardization is performed separately for each view; no parts of the integrated schema are defined in this step.

During schema standardization, a single object type o of the original view schema is replaced by a set of object types O'. If each instance of object type o represents simultaneously n different entities or an entity in n different roles, then object type o will be replaced by a set of n object types O', where O' comprises an object type for each entity or for each role; thus, the instances of the object types of O' are related to each other by a relationship, where any instance of o corresponds to a set of related instances of object types of O'.

If an object type o represents different but similar objects and differences between these objects have not yet been exploited, object type o is replaced by a type hierarchy with object types O', where one object type $o' \in O'$ is a supertype of the other object types in O'. Each instance of o is represented by an instance of exactly one subtype in O'.

As the overall behavior should not change during standarization, we require that correspondences between the activities in the original object type o and the new object types O' are defined in a way that the possible sequences of activity invocations on an instance i of o are equal to the possible sequences of activity invocations on the objects that are instances of the object types in O' and that correspond to i. The result of inter-type integration is the basis for intra-type integration.

We will propose correctness criteria on how specialization hierarchies in different views may deviate from each other, such that intra-type integration is as easy as possible. We will illustrate this correctness citeria in this chapter and discuss them in more detail in Section 4.2.3.

We distinguish the following cases (cf. Figure 1.4 for an illustration):

Equivalent type hierarchies: The type hierarchy in two views is equiva-

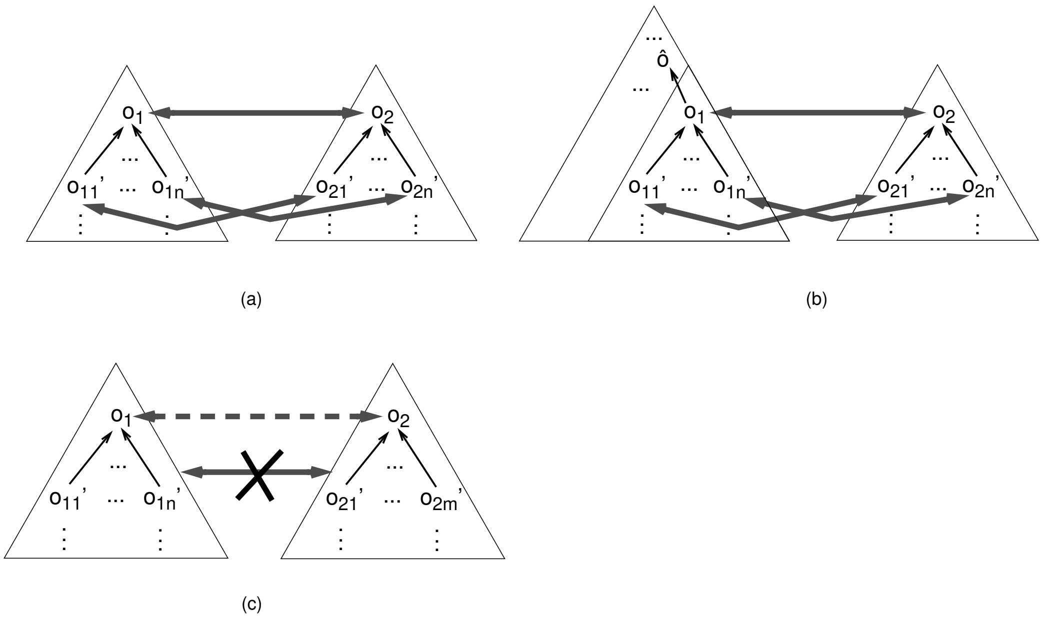


Figure 1.4: Correspondences between type hierarchies in views

lent (cf. part (a) of the figure). This means, that for each object type in the type hierarchy in one view there is an equivalent object type in the type hierarchy in the other view. Supertype–subtype relationships are consistent, i.e., if o_1 is equivalent to o_2, and o_1' is equivalent to o_2' then o_1 is a supertype of o_1' if and only if o_2 is a supertype of o_2'.

Subtree correspondence: The type hierarchy of one view is equivalent to a full subtree of the type hierarchy of the other view (cf. part (b) of the figure). As a special case, the subtree could be the empty tree, such that a type hierarchy exists in only one view.

Disjoint type hierarchies: The type hierarchies consider different intensions and disjoint extensions (indicated by the crossed-out arrow in part (c) of the figure), but their root types are similar, which means that they are different specializations of a common supertype not modelled (cf. part (c) of the figure).

1.3.4.2 View Limitation

View limitation means that object types that are defined in the conceptual schema are missing in a view. As all object types in the integrated schema are defined in at least one view, view limitation can also be considered as the fact that an object type defined in one view is missing in another view. View limitation is reflected by the different kinds of correspondences between type hierarchies in the views as discussed before:

Equivalent type hierarchies: The views define the same object types. Thus, there is no view limitation.

Subtree correspondence: Those object types that are defined in one view but are missing in the other view have been omitted in the latter due to view limitation. As a special case, a type hierarchy can be completely missing in a view.

Disjoint type hierarchies: Each view considers a subtree of the complete type hierarchy and omits the other subtrees as well as the common root object type of the type hierarchy.

1.3.4.3 Integration

The integrated schema contains the complete type hierarchies, i.e., all object types defined in any standardized view are defined in the integrated schema, too. Thereby, heterogeneities because of view limitation are resolved.

We distinguish once more the cases defined before:

Equivalent type hierarchies: If type hierarchies in the views are equivalent, exactly the same type hierarchy is created in the integrated schema. The behavior of each object type is defined as discussed in Section 1.3.3. As type hierarchies are normalized during schema standardization, object types in the integrated schema can be defined by considering exactly one pair of equivalent object types in the views. If views as illustrated in Figure 1.4 (a) are given, object types o_1 and o_2 are integrated to the root object type o in the integrated schema, and all other object types are integrated pair-by-pair, too (e.g., o'_{11} with o'_{21}).

Subtree correspondence: The common object types are integrated, whereas the object types that exist in only one view are taken into the integrated schema without integration. The specialization hierarchy in the integrated schema will be correct, i.e., each object type in the integrated schema is defined by integration if its supertype is defined by integration.

Consider the type hierarchies shown in Figure 1.4 (b): The object types in the subtree of one view are integrated with the object types in the type hierarchy in the other view, e.g., o_1 is integrated with o_2 yielding object type o; object type o'_{11} is integrated with o'_{21}, yielding object types o'_1, where o'_1 is a subtype of o.

As object type o'_1 results from integrating subtypes of o_1 and o_2, whereas o results from integrating o_1 and o_2, it holds that o'_1 is a subtype of o and o'_1 fulfills the criterion of observation consistent specialization of o if correspondences have been determined consistently. Object type $\hat{o}$ has no corresponding object type in the other view, such that it is taken into the integrated schema without integration. As we will discuss in Chapter 6, the resulting type hierarchy in the integrated schema is correct as o_1 is a subtype of $\hat{o}$ in the view, and thus must fulfill the criterion of observation consistent specialization, and o'_1 is an observation consistent specialization of o_1.

Disjoint type hierarchies: In this case, there are no equivalent object types defined in the type hierarchies, such that all of these object types are taken into the integrated schema without being integrated with an object type in the other view.

As a type hierarchy must have a single root type, a new object type is defined as the root type of the type hierarchy that results from integrating the common behavior of the root types of the respective type hierarchies in the views. If views as shown in Figure 1.4 (c) are defined, there will be an integrated schema that takes all object types of the views into the integrated schema without integration and that comprises a new object type $\hat{o}$, that is a supertype of o_1 and o_2 and that contains the common elements (activites, states, etc.) of o_1 and o_2.

The approach defined above allows to define correct integrated schemas, where the specialization criterion between supertypes and subtypes is obeyed. Yet, there is an open question that is considered separately in the following:

Generalization of elements. In the object-oriented data model, an object type should comprise the common elements of its subtypes. If all subtypes of an object type o should include a property, it is originally defined in o and inherited by the subtypes of o. If an element e_1 is defined in a type o_1 and an element e_2 is defined in o_2, where o_1 and o_2 have a common supertype, these features are considered different if no corresponding element e is defined in object type o.

In the approach introduced in this work, object types are defined by integrating two equivalent object types, independently from other object types in

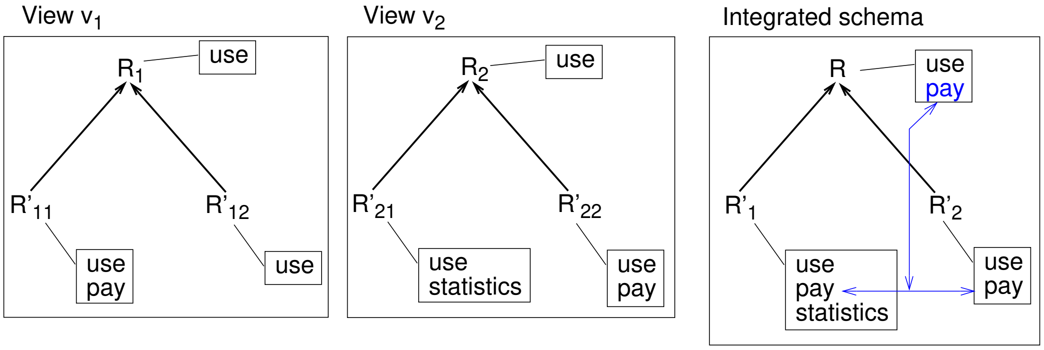


Figure 1.5: Example: generalization of elements

the type hierarchy. As no object type in a view contains a complete specification of behavior, common elements of object types in the integrated schema that should be defined in the supertype may not be recognized.

Consider the following case: Suppose that there are two views v_1 and v_2. View v_1 comprises an object type R_1 with subtypes R'_{11} and R'_{12}; view v_2 comprises an object type R_2 with subtypes R'_{21} and R'_{22} (cf. Figure 1.5). Each object type contains an object life-cycle (we do not depict the whole object life-cycles in the figure, but indicate only the fact some elements are defined with the object life-cycle by listing them next to the name of the object type). The system integrator defines the following object types in the integrated schema: R (for R_1 and R_2), R'_1 (for R'_{11} and R'_{21}), and R'_2 (for R'_{12} and R'_{22}).

The object types in the integrated schemas contain all elements that are defined in at least one of the corresponding object types in the views. Thus, element pay is defined in R'_1 and R'_2, but is not defined in R. In that case, activity pay in R'_1 and activity pay in R'_2 are considered different. If they were equal, they had to be defined in R, too.

This decision cannot be automated. Therefore, our approach will include an integration step *generalization of elements* to extract common elements of different types $O' = \{o'_1, \ldots, o'_n\}$, and to define them in their common supertype o. This step defines an element in object type o only if it is defined in *all* subtypes. Thus, the subtypes of the supertype will fulfill the specialization criterion after this change, too. If common elements are only defined with a subset $O'' \subset O'$, these properties cannot be defined with o as the specialization criterion between o and the object types of $O' \setminus O''$ would be violated. To extract common properties from O'', a new object type $\hat{o}$ that is a subtype of o and a supertype of the object types in O'' must be added; this object type $\hat{o}$ comprises the common elements of the object types of O''.

1.3.5 Specification of Activities

The object life-cycle presented before provides an overview of the *overall behavior* of objects and shows which sequences of activity invocations on objects are allowed. Activities have to be further specified in so-called *activity specifications*.

The activity specification specifies the interface of an activity: it consists of a set of *ports*, which are comparable to parameters as defined for procedures in programming languages. Similar to input and output parameters, there are input ports and output ports. Whereas pre- and poststates in object life-cycles show *when* an activity can be invoked on an object, i.e., in which states an object must reside, and in which states the object resides after execution of the activity, ports show *which parameters* must be provided to an activity and which parameters are returned by the activity. There is one distinguished port, the *primary* port, which defines the pre- and poststates of the object on which the activity is invoked. The pre- and poststates of the primary port are equal to the pre- and poststates defined with the object life-cycle. All other ports are called *non-primary.*

The specification of ports constitutes pre- and postconditions as an activity can only be started if appropriate values are assigned to all input ports, whereas all output ports return some value according to the specification of the port after execution. Further, we allow to define textual pre- and postconditions as boolean expressions with each activity specification.

The difference between states defined with primary ports and textual conditions is that pre- and poststates are mainly defined to represent overall ordering constraints between activities whereas textual pre- and postconditions define conditions on values of parameters and define that the invocation of an activity depends on the properties of the object or some other facts. The decision whether a state or a textual precondition should be defined must be taken by the modeler.

Activities defined in the object life-cycle in a view or in the conceptual schema may have activity specifications and may be invocable. We distinguish the following properties for activities:

Specified vs. unspecified activities: An activity is *specified* if and only if an activity specification is defined for this activity.

Invocable vs. non-invocable activities: An activity is *invocable* if it can be invoked on an object, otherwise it is non-invocable. If an activity is invocable it must be specified.

For activities defined in the object life-cycles in the view schemas and in the conceptual schema, we require the following conditions to hold:

Conceptual schema: All activities defined with an instantiable object type (i.e., defined with the leaf nodes of type hierarchies) must be invocable and thus, specified. Non-invocable activities in instantiable object types are not useful as they could never be started on an object. Activities in other object types (i.e., in inner nodes of type hierarchies) need not be specified as no direct instances of these object types exist, such that these activities are non-invocable. Yet, activity specifications for non-invocable activities may be useful and necessary: If an activity a is defined with object type o as well as in object types o_1 and o_2 that are subtypes of o, the common ports and conditions of a in o_1 and a in o_2 are represented by the activity specification of a. In Chapter 2, we will define rules how activity specifications in an object type can be specialized in the subtypes.

View schemas: All activities in view schemas are only defined to observe processing, thus they are non-invocable. Activity specifications defined for activities in view schemas only exist to define the activity specifications for activities in the conceptual schema based on the activity specifications in the views.

Activity specifications in the views need not be complete, which means that parts of the activity specification defined for an activity a in the conceptual schema are not defined in the activity specification of the corresponding activity in the view. This is no problem as activities are non-invocable in the views. During intra-type integration of object life-cycles, we argued that no view needs to know all pre- and poststates of an activity. As pre- and poststates defined in the object life-cycles are equal to the primary ports of the activity specifications, an activity specification in a view is not complete if the pre- and poststates defined for this activity in the object life-cycle are not complete.

Further, we could argue that for the other parts of the activity specifications, i.e., non-primary ports and textual conditions, the same rule holds. This means, each view could provide an incomplete activity specification for an activity a in that non-primary ports are missing and weaker pre- and post-conditions are defined. Then the integration of the activity specifications for a that are defined in the views constituted the activity specification for a in the conceptual schema.

Yet, we restrict this assumption for the following reason: We assume that each activity that appears in the object life-cycle of o in the integrated schema can be specified by one user, particularly by that user who performs this activity in everyday business. This user will know about the information that has to be provided to the activity and that is returned by the activity and thus is able to define the non-primary ports for the respective activity specification in the view schema. All other views may refer to the activity in the object life-cycle to express ordering relationships, thereby defining further pre- and poststates, but they do not define an activity specification for this activity (thus the activity is *unspecified* in these views); the reason will be that other users may know about the existence of the activity but cannot specify its non-primary ports and detailed textual pre- and postconditions.

Thus, we restrict the definition of views in the following way: The pre- and poststates of an activity in the object life-cycle need not be completely defined in any view (cf. Section 1.3.3). But, for each activity defined with a certain object type in the conceptual schema, there is exactly one view that defines the non-primary ports and textual conditions for this activity. The specification of the primary port can be derived from the object life-cycle.

Activity specifications in *non-instantiable* object types are defined to represent common parts in the activity specifications of the corresponding activity in all subtypes. Thus, the problem of *generalization of elements* arises for activity specifications, too: Consider once more the example illustrated in Figure 1.5: Suppose that activity use is unspecified in object type R_1, R'_{12}, R_2, and R'_{21}, whereas it is specified in R'_{11} and R'_{22}. In the integrated schema, there is an activity specification for use in R'_1 and R'_2, whereas use in R is unspecified at first. Corresponding to generalization of elements in object life-cycles, an activity specification is defined in an object type (i.e., in R) if this activity is specified in all subtypes, where the activity specification in the supertype comprises all common parts of the activity specifications in the subtypes.

1.4 Related Work

Publications exist in several fields related to integration of views and integration of information systems in organizations. We particularly mention work done in *schema integration, enterprise integration* and *definition of views.*

1.4.1 Schema Integration

View integration is often considered within the larger context of *schema integration,* which does not only consider integration of *views* but also integration of *databases* [BLN86]; the former is used to define the global schema for a *proposed* database, whereas the latter is used to integrate the schemas of *existing* databases. Database integration is mainly discussed in the realm of *federated* and *multi-database systems* [LMR90, SL90, HM85]. In [BE96], an overview of *object-oriented* multidatabase systems is given; [Con97] gives an overview of federated database systems and discusses several approaches of schema integration.

A plenty of publications exist in schema integration, but most of this work considers integration of *structure*, either integration in the relational data model or integration of structure of objects in the object-oriented model. Schema integration mainly in the realm of federated database systems is treated in approaches like [GSCS95, GPNS92, KGF98, Joh97, Kri97, LC95, Mir97, PSU98, Sch98, SJB96, YOL97]. In [San98], multi-level integration is introduced: The author assumes that not only the final conceptual database schemas of the databases that have to be integrated are available, but these schemas are defined by stepwise refinement, and the whole chain of refinement steps is available for each schema; then, integration is based on the information about how each schema has been created by top-down refinement. In [QFG97], another approach using a conceptual model is introduced, where OMT [RBPE91] is extended for bottom-up construction of database schemas. Some other works particularly deal with integration of structure in *view integration*, e.g. [BCW91, GLN92, Joh93, Joh94, NEL86, SP94].

Some recent publications deal with integration of *methods* of object-oriented views (cf. [EJ96, Pit97, VA97, TS95]) and of *integrity constraints* ([VA96, CST98]), but little work has been done so far on integration of object life-cycles.

One approach is known [Fra96, FE97] where object life-cycles specified by state charts are integrated. That work emphasizes view integration of dynamic models that are related to each other in one of five ways, namely *parallel, disjoint, alternative, consecutive,* and *mixed*, depending on how objects have to pass the dynamic models defined in the views. The first four possibilities allow merging of the schemas in an automatic way after some start and end states have been determined. Two dynamic models are *mixed* if there are objects that pass both models, but if the models are not related to each other in one of the other four ways. In that case, no automatic integration is possible; instead, the modeler has to examine the correspondences between states of the schemas and to integrate corresponding states until a correct state chart is found.

Another integration approach is used in [Emb98], where behavior of objects is defined by state nets, consisting of states and transitions between states that are invoked if an event arises or a certain condition is fulfilled. The author mentions integration but does not show in detail how integration works.

1.4.2 Enterprise Integration

As we define business information systems, which define business processes of business objects, we have to examine literature about business process modeling and integration of business processes, too. In modeling business processes, the term *enterprise integration* [Ver96] is defined. Enterprise integration deals with coordination of work across organizational boundaries. Three kinds of integration are defined: (1) *physical integration,* which mainly deals with networks and protocols, (2) *application integration,* where common data interchange formats (e.g. EDI) are used to exchange information, and (3) *business integration,* where business processes shall be coordinated.

Halter [Hal96] uses the term *integration of processes* in a completely other sense, in that he requires that several processing steps of business cases are integrated in one workflow, i.e., no business processes are integrated to a new business process, but single processing steps are controlled by one workflow. Works like [Hal96] and [SJHB96] mainly treat integration of different *applications* in one workflow. In [MM98], integration of processes only means that a business process includes processing steps in different organizations.

Integration of business processes in [Sch97] is comparable with application integration in [Ver96]; i.e., data interchange formats like EDI, EDIFACT, etc. are used for exchange of information. Such formats are often used in connection with *middleware*, which is used for asynchronous communication between loosely coupled systems (see e.g. [SZ98] for an actual approach).

In [CCPP96], *semantic workflow interoperability* is introduced. The authors assume that business processes are defined from different perspectives and interact with each other. This approach is comparable with the interaction between object life-cycles of two object types within the same schema, which will be presented for OBD in Section 2.2.1.

As a summary, we found out that literature in this field deals with integration of workflows or processes at another level; no integration of business processes at the conceptual level as proposed in this work is used.

1.4.3 Definition of Views

Many publications deal with deriving views from an existing schema. In the ANSI/SPARC architecture, views are defined to constitute the *external schemas* of a database.

Most approaches in definition of views use a query language to define a query over the database to derive the data visible in the views. Examples are

FUGUE [HZ90], COCOON [SLT91], O_2 [AB91, SAD94], XSQL [KKS92], LIVING IN A LATTICE [HS93], *MultiView* [KR96], and the approach of Bertino [Ber92]. In difference to these approaches, views are defined using a high-level conceptual model in [CE91]. In [SSS98], definition of derived classes in object-oriented databases with object-generating or object-preserving semantics is discussed.

[KR95] focusses on updatable materialized views, [YPS95] discusses the advantages of deductive object-oriented systems in definition of views. In [DE94], views are defined as an extension of the ODMG-93 standard [Cat95]. More recent publications discuss definition of views in data warehouses (e.g. [QW97, AASY97]).

Notice that definition of views differs from integration of views in several respects:

1. During integration of views, the conceptual schema is defined from the view schemas, whereas during definition of views, a conceptual schema exists, from which the views are defined.

2. Integration of views belongs to the *design phase* of a database, where *no actual extensions* of a database exist. Definition of views is used to derive external schemas *during operation* of an existing database in order to make *existing data* available to the views.

3. Views are mostly integrated based on schemas defined in a *high-level model*, whereas views are defined mostly using some *query language*, which depends on the used implementation model (relational, object-oriented, etc.)

Definition of views is not further needed and discussed in this work. For a more detailed discussion of defining views and for an overview over existing approaches the reader is refered to works of Motschnig-Pitrik [MP96] and Dobrovnik [Dob95].

1.5 Outline

This thesis treats integration of behavior of object types in object-oriented views using *Object/Behavior Diagrams*. The work is organized as follows:

Object/Behavior diagrams: In Chapter 2, we present *Object/Behavior Diagrams* (OBD), a graphical design notation for object-oriented databases. The structure of objects is defined in *Object diagrams*, whereas the behavior is defined in *Behavior diagrams*. The latter consist of *activity specification diagrams* and *activity realization diagrams*, where activities are defined, and *object life-cycle diagrams*, in which the overall behavior of objects during their lifetime is defined.

Workflow transparency: View integration of behavior of objects will be treated in the realm of modeling business processes, where object diagrams are used to define the structure of business objects and behavior diagrams are used to define the activities performed on business objects and to define the business process. We will introduce a two-schema architecture to distinguish business processes and workflows to provide for workflow transparency. For defining workflows, we will present a workflow model.

Overview of view integration: In that chapter, we present the basics about our integration approach. We will discuss how views deviate from the conceptual schema and will introduce the main steps of the integration process.

View preparation: During view preparation, the view schemas are modified such that only aspects of business processes are represented and type hierarchies are defined in a standardized form.

Integration of object life-cycles: Views will define parts of the object life-cycles of objects. In this chapter, we deal with defining object life-cycles in the integrated schema from the object life-cycles defined in the views.

Integration of activity specifications: Finally, the activity specifications of activities in the integrated schema must be defined, based on the activity specifications defined in the views.

Conclusion: In this chapter, we conclude our work by summarizing the results of this thesis and discussing future work.

Chapter 2

Object/Behavior Diagrams

Contents

Object/Behavior Diagrams (OBD) are a graphical notation for the conceptual design of object-oriented databases. The design of object-oriented databases comprises the design of the *structure* and the *behavior* of objects. The behavior of objects is modeled at two levels of detail: (1) by activities, which are represented by methods at the implementation layer, and (2) by object life-cycles, which define the dynamics of objects over their lifetime. In OBD, the structure of objects and the interrelationship between objects is defined by *Object Diagrams* whereas the behavior is modeled by *Behavior Diagrams.* OBD is focused on the *conceptual* design of objects, the mapping of the conceptual structure to the physical structure as well as the internal implementation of activities and object life-cycles in a database system are not considered.

Object/Behavior Diagrams have originally been introduced in [KS91a]; the model was refined and extended in subsequent work, mainly in two directions: (1) inclusion of active behavior and (2) modeling business processes.

The concept of *local referential integrity* was introduced in [KS92], where referential integrity, which is known from the relational data model, is extended for the object-oriented model with complex objects. It says that an object may only reference objects that belong to the same composite object.

In [BS94], the first notation of *Active Object/Behavior Diagrams* was introduced. In [BS95], OBD was used to model business processes together with business rules, which define when an activity must be executed (*obligation*), may be executed (*permission*), or must not be executed (*prohibition*). Further, inheritance of business rules was briefly discussed. The problem of specialization of business processes together with its business rules was illustrated by an example dealing with treatment of loans in a bank [KS95].

In a subsequent research project, *Situation/Activation Diagrams* were introduced for the conceptual design of active rules to represent business policies (cf. [LOS96, LOS97, Lan97, Obe98] and a graphical editor was implemented [LOKT98]. In these approaches, the OBD model was extended by *Object-Class Diagrams,* which allow to maintain local referential integrity. [Lan97] further investigates in specialization of business rules. As behavior may be defined by methods and by rules, the modeler may have difficulties to decide which behavior should be implemented in methods and what should be expressed by rules; this problem is further treated in [KS96a].

In [BPS97a, BPS97b], the use of Object/Behavior Diagrams for modeling business processes in a two-schema architecture was introduced (cf. Chapter 3 for details).

[KS96b] summarizes the original definition of Object/Behavior Diagrams together with an overview of the process of analysis and design of object-oriented information systems; furthermore, some of the most important extensions of OBD are described.

Object/Behavior Diagrams are used in this work as originally defined in [KS91a]. Extensions of this definition are used as far as necessary for the contribution of this work.

This chapter is organized as follows: Section 2.1 gives an overview of *Object Diagrams*; it contains the definition of the properties of objects and treats semantic relationships between objects that are supported by OBD. Section 2.2 deals with *Behavior Diagrams* and describes the three kinds of diagrams that are used in OBD to define the objects' behavior at the conceptual level of design. Section 2.3 deals with the specialization of Object/Behavior Diagrams and presents an overview of the rules for specialization on a rather informal level. Finally, Section 2.4 presents a formal definition of those aspects of Behavior Diagrams that are necessary for understanding the remainder of this work.

Example: As a running example that is used throughout this work, we will model a hotel reservation system, in which hotels, rooms, reservations, people, etc. are represented. This example was originally motivated by the example defined in [OHM$^+$88] and was used in the previous publications about OBD in a similar form, too.

2.1 Object Diagrams

Object Diagrams rely on semantic data model concepts [HK87]; they are used to model the structure of objects, i.e. their attributes and relationships, which will be commonly refered to as an object's *properties*. Any property may be defined as *single-valued* or *multi-valued* depending on whether the property contains at most one value or a set of values, respectively.

Properties are considered to be abstract concepts and should not be confused with instance variables as used in several object-oriented programming languages. Rather every property is to be interpreted as a declaration of a public retrieval operation returning an instance or a set of instances of its domain.

OBD supports many of the important types of relationships known from several object-oriented design models (e.g., [Emb98, FS97, RBPE91]). Different from many other approaches, OBD supports the concept of roles. In the past decade, numerous papers have been published in this field. The reader is refered to [KRSR98] for an overview.

In difference to other models, relationships between instances of object types are not depicted by a graph-like notation where object types are represented by nodes and relationships are represented by edges, but they are depicted by nested graphical boxes.

This section gives an overview of the structure and graphical representation of Object Diagrams and introduces the supported types of relationships.

2.1.1 Properties

In OBD, properties are classified in attribute properties and relationship properties. Each attribute has a predefined *data type* like INTEGER, STRING, BOOLEAN, DATE, REAL, etc. as its domain. Further simple datatypes may be defined depending on the application, e.g., ATS for an attribute representing a value in Austrian shillings. The instance of a datatype is called *data value*, e.g., 12 and 'Huber' are data values for data types INTEGER and STRING, respectively. Relationships represent references to other objects and thus have an object type as their domain.

In the graphical notation, an object diagram is depicted by a box consisting of a shaded bar at its top showing the name of the object type and a symbol for each property nested within the box. Each attribute is represented by an oval, each relationship is depicted by a rectangle. The name of the property is depicted above the symbol, the domain is depicted within the symbol. Single-valued properties are depicted by single-line borders, multi-valued properties are depicted by double-line borders.

Example: Figure 2.1 shows the object diagrams of the object types used in our example.

Object type GUEST has two attributes: the single-valued attribute noOfVisits with data type INTEGER and the multi-valued attribute specialWishes with data type STRING. Notice that there exist data types which are usually not represented in conventional programming languages: the data type of attribute price of object type ITEM-OF-BILL is ATS, which means that the domain is an amount in Austrian shillings; the data type of vatRate is %.

Any object is uniquely identified by its *object identifier,* which is defined by the system and which is not visible to the user. Further, the modeler may define *key properties,* which — corresponding to key attributes in the relational model — identify an object of a certain object type. Key properties have to be single-valued attributes. They are depicted by attribute symbols with bold borders.

Example: Every hotel (i.e., every instance of object type HOTEL) can be identified by its name and the name of the city in which it is located.

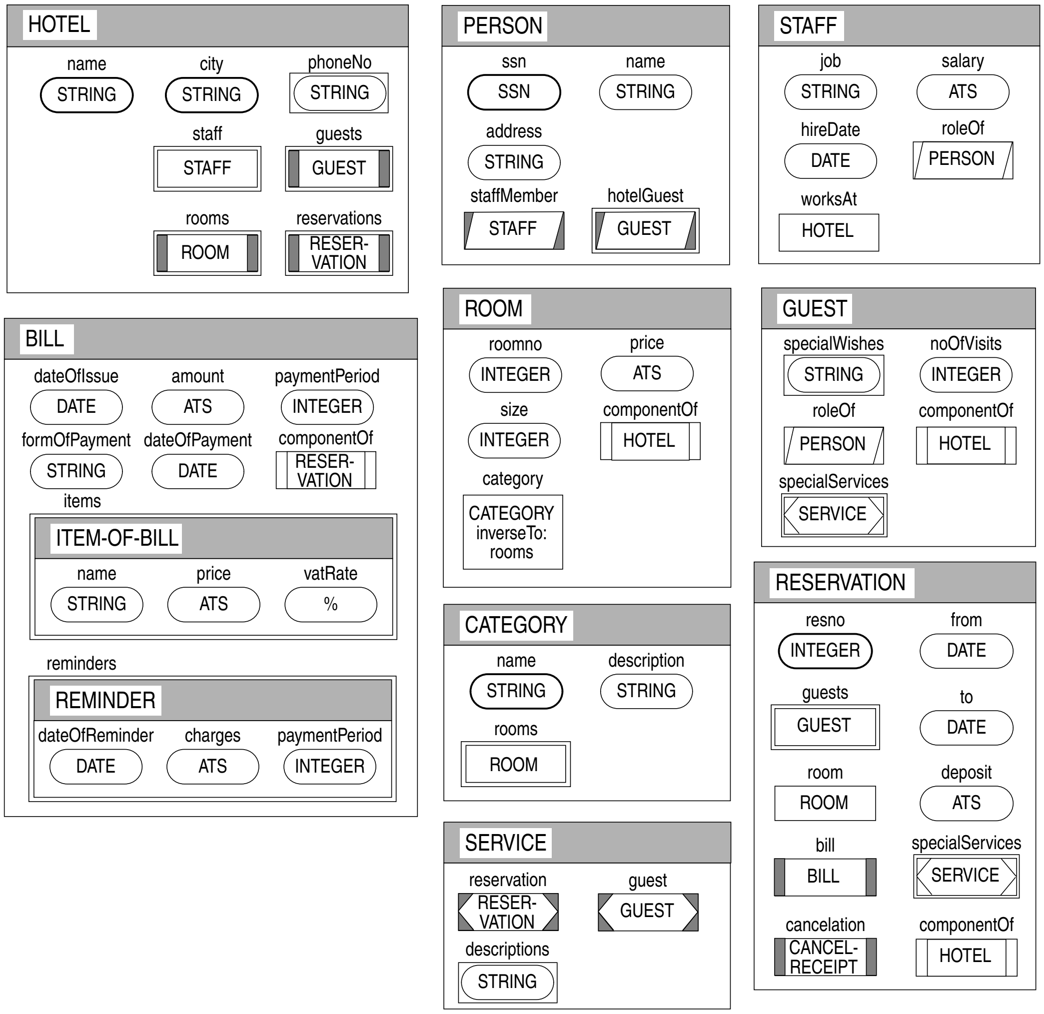


Figure 2.1: Object diagram of the room reservation system

As a convention, names of value types and object types are printed in upper-case letters, names of properties are printed in lower-case letters.

2.1.2 Relationships

Relationships represent references from one object to one or a set of other objects. In OBD, several kinds of relationships with different semantics are provided. *General relationships* describe a relationship between two independent objects; they do not define any hierarchy between the related objects. All other relationships define a hierarchy between the related objects. Therefore, these relationships always exist in pairs: (1) *componentOf* – *hasComponent*, (2) *roleOf* – *hasRole*, and (3) *hasConstituent* – *constituentOf*. All these kinds of relationships will be described in this section.

General relationships. Different from other models like OMT [RBPE91], general relationships in OBD are *directed*. This means, that if a relationship is defined from an object i_1 that is an instance of object type o_1 to an object i_2 that is an instance of object type o_2, no relationship from i_2 to i_1 is defined automatically. Instead, each bi-directional relationship has to be defined explicitly by a pair of directed relationships, one directed relationship with each object type in question. Two directed relationships r_1 and r_2 may be defined as *inverse* to each other to express the fact that an object i_1 refers to an object i_2 by relationship r_1 if and only if i_2 refers to i_1 by relationship r_2.

A general relationship models a directed relationship between two *independent* objects. General relationships are depicted by a rectangle with the name of the relationship placed above and the name of its domain placed within the rectangle.

Example: Every reservation in our reservation system refers to exactly one room. Thus, object type RESERVATION includes a single-valued general relationship called room with domain type ROOM. Notice that this relationship is uni-directional as there is no reference from object type ROOM to object type RESERVATION.

Each room belongs to a certain category, which describes the level of quality of the room. A bi-directional general relationship is defined between ROOM and CATEGORY, in that a multi-valued general relationship rooms is defined with object type CATEGORY and a single-valued general relationship category is defined with object type ROOM. The clause inverseTo: defined with property category in object type ROOM defines the relationship as inverse. This clause must be defined with only one of the related relationships.

***componentOf*- and *hasComponent* relationships.** These relationships support the definition of component object hierarchies. A *componentOf* relationship is a directed relationship from a component object to a superordinate compound object. The object containing the *componentOf* property is refered to as the *component object* whereas the object referenced by the *componentOf* relationship is refered to as *aggregate* or *composite object*. Each *componentOf* relationship has an inverse relationship, called *hasComponent* relationship.

Although various models support some kind of composition, which is often called *aggregation,* there is no consensus about the semantics of aggregation. Kim [KBG89] defines composite objects and composite references considering two dimensions: (1) *dependency,* which states whether a component object's existence depends on the existence of the compound object, and (2) *exclusivity,* which states whether an object may be a component of at most one or of a set of compound objects. Therefore, there are four kinds of composite references.

In OMT [RBPE91] and UML [FS97], there is no definition about in which cases an aggregation relationship is to be prefered to a general relationship. In UML, a special kind of aggregation, called *composition,* can be defined, which represents exclusive and dependent component relationships.

In OBD, an object depends on and belongs exclusively to the object referenced by the *componentOf* relationship. Therefore, any object may contain at most one single-valued *componentOf* property, which is always named componentOf, whereas any object may contain a set of single-valued or multi-valued *hasComponent* relationships, which must have different names for distinction.

As an object may contain a *componentOf*- as well as several *hasComponent* relationships, a component hierarchy may be modeled. Due to transitivity of the *componentOf* relationship, object i_1 is a component of object i_2, if and only if $i_2 = i_1$.componentOf or there exists an object i_3 such that $i_3 = i_1$.componentOf and i_3 is a component of i_2.[1]

In the graphical notation, a *componentOf* relationship is depicted by a rectangle with unshaded side bars, a *hasComponent* relationship is depicted by a rectangle with shaded side bars. Alternatively, the object diagram of object type o_1 that has a *componentOf* relationship with domain o_2 may be depicted nested within the object diagram of o_2. In that case, the *compo-*

[1] We use the usual notation to refer to properties of objects and to represent path expressions, e.g., i.prop represents property prop of object i.

nentOf property need not be depicted explicitly, although it exists and can be refered to.

Example: Every hotel has several components: guests, rooms, and reservations. The structure of the component types GUEST, ROOM, and RESERVATION is defined in separate object diagrams. Every bill contains items and, optionally, reminders in case that the guest does not pay in time. The object diagrams of the component types ITEM-OF-BILL and REMINDER are depicted nested in the object diagram of the composite object type BILL.

***roleOf*- and *hasRole* relationships.** A *roleOf* relationship represents a directed relationship between two objects that represent the same real-world entity, but considering different roles. The object refered to by the *roleOf* relationship represents the entity in a more general context than the *role object,* i.e., the object that contains the *roleOf* property.

Any object may contain at most one *roleOf* relationship, which has to be single-valued, and may contain several *hasRole* relationships, which may be single- or multi-valued. The relationships are inverse relationships to each other, i.e., it holds that $i_2 = i_1.\mathsf{roleOf}$ if and only if there is a *hasRole* property hasRole defined with i_1 such that $i_1 \in i_2.\mathsf{hasRole}$. The existence of a role object depends on the existence of the object referenced by the *roleOf* relationship. As an object may contain a *roleOf* relationship as well as *hasRole* relationships, a *role hierarchy* can be modeled. The object type that includes a *hasRole* relationship is refered to as the *role supertype*, the object type which is refered to by the *hasRole* property is called *role subtype*.

If a *hasRole* relationship is multi-valued, i.e., if a real-world entity may obtain several instances of the same role type, these role objects have to be distinguished by a *qualifier property,* which has to be defined as a property in the role subtype [GSR96]. This qualifier property may either be an attribute or a *componentOf* relationship. Objects refered to by a multi-valued *hasRole* relationship are called *qualified roles,* their types are called *qualified role types.*

Role hierarchies are used to represent a real-world entity in different contexts if the properties and the behavior that are applicable to an entity cannot be fully determined at the time of creation. When an entity is created, an instance of the root type of the role hierarchy is created. During the entity's lifetime, roles may be created and deleted, thereby defining and removing additional properties and behavior for the entity.

In OBD, *roleOf*- and *hasRole* relationships are depicted by a rectangle with a parallelogram as its inner structure. The area between the rectangle and the parallelogram is depicted shaded for *hasRole* relationships and unshaded for *roleOf* relationships.

Example: A human person is represented in the reservation system at least by an instance of object type PERSON, which contains properties that apply to any person. A person may appear in the information system as a staff member, as a guest of a hotel, or both as a staff member and a guest. Thus, object type STAFF and object type GUEST are defined as role types of PERSON.

We assume that each person may be a staff member of only one hotel, i.e., each instance of object type PERSON may be associated with one instance of STAFF only.

Role type GUEST is qualified by property componentOf, which refers to object type HOTEL: each person may be guest of several hotels, but may be guest of each hotel only once (of course, as a guest, it may be related to any number of reservations); thus several instances of role object type GUEST may be defined that are related to the same instance of PERSON, but these instances of GUEST have to be related to different hotels.

***hasConstituent-* and *constituentOf* relationships.** A *hasConstituent* relationship represents a directed relationship between an *association object* and a *constituent object*. The objects refered to by an association object via a *hasConstituent* relationship are called *constituents* of the object as they "constitute" the association object. The existence of the association object depends on the existence of all constituents and the association object is uniquely identified by its constituents.

A *constituentOf* relationship is the inverse relationship and denotes the fact that an object i_2 refered to by the *constituentOf* property of an object i_1 depends on object i_1. Any object may be a constituent of instances of several different object types, i.e., it may comprise a set of *constituentOf* relationships. A *hasConstituent* relationship has to be single-valued, whereas a *constituentOf* relationship may be either single- or multi-valued.

hasConstituent- and *constituentOf* relationships serve to represent relationships between objects by objects, where the constituent objects are the objects to relate and the association object represents the relationships between the constituents. Thus, the relationship between constituents is not expressed by "direct" references but by an association object. In some other models, this concept is called differently (e.g., *associations as classes* in OMT).

hasConstituent and *constituentOf* relationships are represented by a rectangle with a hexogonal inner structure where a *hasConstituent* relationship is distinguished from a *constituentOf* relationship as the area between the rectangle and the hexagon is depicted shaded in the former case and unshaded in the latter case.

Example: Each reservation object is related to a set of guests. For each reservation, a guest may order a special service. In our schema, this service is represented by an instance of object type SERVICE, which is constituted by an instance of RESERVATION and an instance of GUEST. The property descriptions contains a set of strings describing the items of the ordered service.

2.2 Behavior Diagrams

The behavior of objects is defined in Behavior Diagrams. Behavior Diagrams consist of three types of diagrams: (1) A *life-cycle diagram* represents the overall life-cycle of objects of a certain object type containing all update operations defined with the object type, i.e., for each object type, there exists exactly one life-cycle diagram. (2) An *activity specification diagram* is defined for an update operation of an object type, i.e., for an operation defined in the object life-cycle of that object type; activity specification diagrams specify the interface of an activity. (3) An *activity realization diagram* is defined for an update operation defined with an object type; it defines the realization of the operation in a form based on Nassi-Shneiderman diagrams [NS73].

In this section, first, an overview of Behavior Diagrams is given. For more details, the reader is refered to [KS96b]. Second, a formal definition of life-cycle diagrams is presented (cf. Section 2.4) as this formal definition will be needed in the subsequent chapters of this work.

2.2.1 Life-Cycle Diagrams

A life-cycle diagram that is defined with an object type defines the overall behavior of the instances of this object type. Life-cycle diagrams are based on Petri nets [Pet77, Mur89] and consist of *activities,* which correspond to transitions in Petri nets, *states,* which correspond to places in Petri nets, and directed arcs, which connect one activity with one state or vice versa. Activities represent work that can be performed on an object, states show where an object actually resides.

Each instance of an object type is represented by a unique token, the object identifier, which may reside in one or several states. An activity may be invoked on an object if all prestates of the activity are marked with this object. Upon invocation of the activity on an object, the object is removed from all of its prestates; when the execution of the activity is completed, the object is inserted into all of the activity's poststates. To refer to the current states of an object, every object has a multi-valued property inStates.

In Petri nets, a transition is fired automatically if all prestates are marked; the execution of a transition occurs instantaneously. In behavior diagrams, an activity has to be invoked explicitly by the user, by another activity, or some other mechanism like active rules or worksteps (cf. publications mentioned in the introduction of this chapter and Chapter 3 for the definition of worksteps). Further, execution of an activity takes some time. Therefore, start and completion of an activity are distinguished, and during the time between start and completion, the object on which the activitiy is executed resides in an implicit *activity state,* which is named after the activity.

In the graphical notation, an activity is depicted by a small rectangle with the activity's name placed on top of the symbol. A state is depicted by a rectangle with a trapezium at the top and the bottom. The name of the object type and of the state are depicted in the top and the bottom trapezium, respectively.

Example: Figure 2.2 shows the life-cycle diagram for object type RESERVATION. An instance of the object type is created by method create. After execution of create, the object resides in state created. Then, there are two alternatives: the guest may either (1) cancel (i.e., activity cancelRoomRes is invoked) or (2) confirm the reservation (by invoking activity confirm). After activity confirm has been completed, the object enters states toCheckIn and depToHandle; thus, parallelism is introduced, which allows to invoke the following activities checkIn and payDeposit (or waiveDeposit) in any order.

As a life-cycle diagram is defined for each object type, but objects interact with each other and thus should not be considered in isolation, some synchronizations between objects have been introduced: (1) *triggered synchronization constraints,* (2) *hierarchical synchronization constraints,* and (3) *domain restriction constraints.*

Example: Figure 2.3 shows the life-cycle diagram of object type BILL. An instance of this object type has to be a component of an instance of object type RESERVATION (cf. the *componentOf* relationship shown in Figure 2.1). The shading of some activity symbols and the triggSync and hierSync clauses are explained in the next paragraphs.

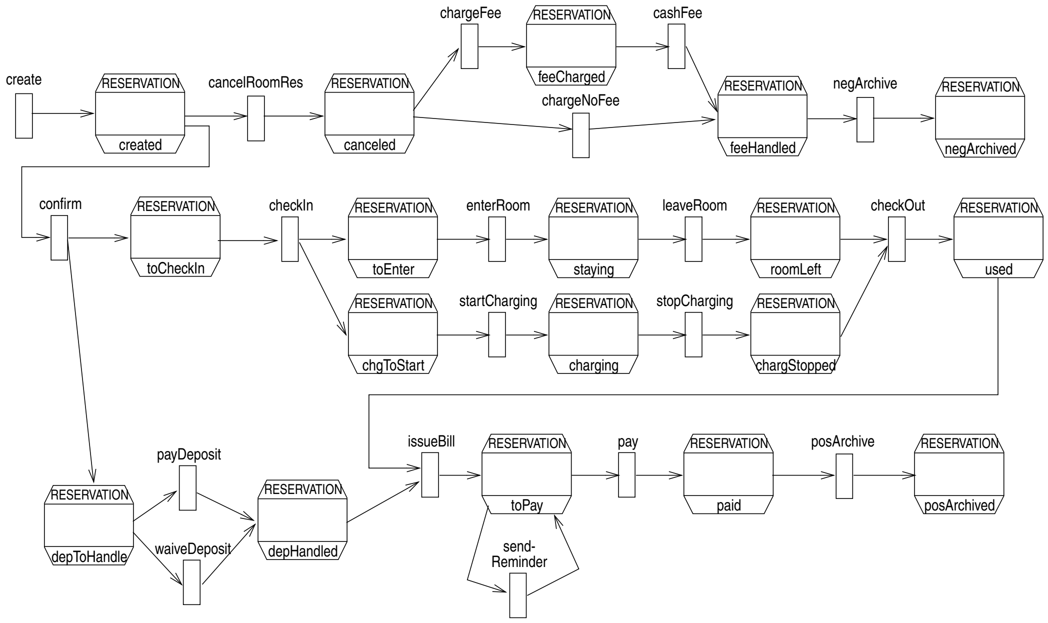


Figure 2.2: Life-cycle diagram for object type RESERVATION

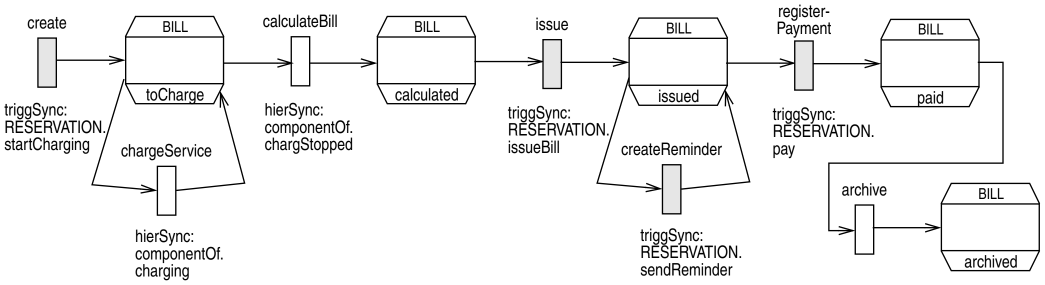


Figure 2.3: Life-cycle diagram for object type BILL

Triggered synchronization constraints. A *triggered synchronization constraint* expresses the fact that an activitiy a_1 of object type o_1 *must* be and *may only* be invoked by activity a_2 of another object type o_2. Such an activity a_1 is also called *restrictedly public activity* and is depicted by a shaded rectangle and a triggSync note, which specifies which activity invokes this activity.

Such constraints are useful if an object calls an activity of a *subordinate* object (i.e., an activity of an aggregate object, role super-object, or constituent object invokes an activity of a component object, role sub-object, or association object). All activities that create instances of an object type (i.e., activities that consume from no state) must always be invoked by another object and thus have to be associated with a triggered synchronization constraint.

Example: Activities create, issue, createReminder, and registerPayment of BILL are invoked by activities startCharging, issueBill, sendReminder and pay, respectively, of RESERVATION and must not be invoked by any other activity. Thus, the activities are depicted by shaded symbols and the triggSync clauses specify which activities invoke them.

The specification means that only activity startCharging invoked on an instance of RESERVATION may trigger creation of an instance of BILL. The created object will be referenced by property bill of the corresponding reservation object. Activity issueBill of a reservation object is the only activity that invokes activity issue on the object referenced by property bill.

Hierarchical synchronization constraints. A *hierarchical synchronization constraint* expresses the fact that an activity of a component, role, or association object may only be invoked if the direct superordinate object (i.e., the object that is referenced by the *componentOf-, roleOf-*, or *hasConstituent* property) resides in a certain state. The constraint is depicted with the activity in question by a hiersync note together with the name of the property that references the superordinate object and the name of the state. As *componentOf-, roleOf-,* and *hasConstituent* relationships are always single-valued properties, the object referenced by a hierarchical synchronization constraint can always be uniquely identified.

Example: Activity chargeService is invoked on an instance of BILL iteratively whenever a service that is not included in the standard service is provided to the guest. As a restriction, the activity may only be invoked if the corresponding instance of RESERVATION, which is referenced by the componentOf relationship, resides in state charging.

Domain restriction constraints. A *domain restriction constraint* is defined with a state s_1 of object type o_1 and contains the name of a *hasComponent* or *hasRole* property p defined with o_1 and a set of states S of the domain type o_2 of p. The set of states S consists of at most one *entry state,* at most one *invariant state,* and at most one *exit state* (refered to by s_e, s_i, and s_x, respectively).

The constraint means that if an instance i of o_1 enters state s_1, all objects referenced by property p of i have to reside in state s_e. Object i can only exit state s_1 if all objects referenced by p reside in state s_x. For all the time, during which i resides in state s_1, all objects referenced by p must reside in state s_i.

A domain restriction constraint is depicted by a state symbol representing state s_1 and a rectangle representing property p depicted within the state symbol. The rectangle is divided into three parts which represent (from left to right) states s_e, s_i, and s_x, respectively.

Example: Consider once more the life-cycle diagram of RESERVATION and BILL (cf. Figures 2.2 and 2.3, respectively). Two domain restriction constraints are depicted in Figure 2.4.

If activity stopCharging of RESERVATION completes and produces the token into state chargStopped, the object refered to by component bill must reside in state toCharge. Further, the bill has to be calculated before check-out, i.e., activity calculateBill of object type BILL has to be executed, which produces the bill object into state calculated. Then, activity checkout can be started, which consumes from state chargStopped. There is no invariant state defined in this example.

The second domain restriction constraint considers state used of RESERVATION. It requires that component bill resides in state calculated if the corresponding reservation object resides in state used. Notice that in this case, the constraint is redundant as the required fact is already enforced by other constraints.

Guidelines for constraints. As a summary, some guidelines for using synchronization and domain restriction constraints concerning creation and archivation of related objects are given:

Creation of objects: The creation of instances of component types and role types should be initiated by the superordinate object, i.e., by the

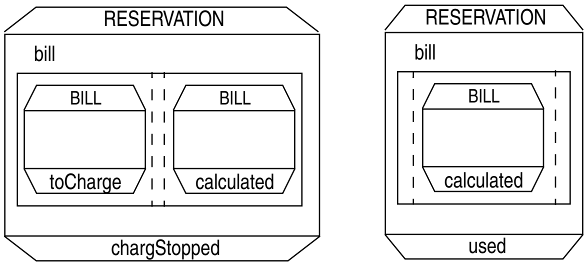


Figure 2.4: Domain restriction constraints for states chargStopped and used of RESERVATION

object that will refer to the instance to create by a *hasComponent, hasRole* or *constituentOf* relationship. Thus, the creation activities of component types, role types, and association types should be associated with a triggered synchronization constraint that restricts invocation to an activity of the superordinate type. Association objects have to be created *after* the creation of *all* constituent objects.

Archivation of objects: Usually, *archive* activities are modeled, which move the object into some *archived* state, which indicates that processing of the object is finished. Similar to other systems where a component of an object, a role of an object, or an association object is deleted if the superordinate object is deleted (*cascading delete*), a component object, role object, or association object should be archived *before* or *at the same time* as its superordinate object. This rule can be enforced by a domain restriction constraint that allows to invoke the archivation activity of the superordinate object only if the subordinate objects have already been archived or by a triggered synchronization constraint between the archivation activity of the superordinate object and its subordinate objects.

Notice that the introduced graphical representation of these constraints is just an abbreviation of the definition of the constraints in the activity specification diagrams.

2.2.2 Activity Specification Diagrams

For each operation defined in the life-cycle diagram that can be invoked on an object, there has to be an activity specification diagram, which defines the interface of the activity, particularly the definition of ports and additional pre- and postconditions. An activity can only be invoked on an object if an activity specification is defined for this activity. We refer to activities that have an activity specification as *specified activities*, whereas all other activities are called *unspecified*.

A port represents a formal parameter of an activity. Each port has a name, a domain, which is either a value type or an object type, and a specification whether it is a primary or a non-primary port. The domain of a port can be either a value type or an object type. If the domain is an object type, a set of states defined with that object type is specified with the domain.

There is one distinguished port, the *primary port,* whose domain is the object type for which the activity is defined. Each port may be single-valued or multi-valued, except the primary port, which is always single-valued.

Ports that read actual parameters provided to the activity are called *input ports,* ports that return parameter values are called *output ports*. A port may be an input port, an output port, or both. The pre- and poststates of the primary input port and the primary output port are those states that are defined as pre- and poststates, respectively, of the activity in the life-cycle diagram. Every activity has a primary output port, all activities except the activities that create a new object have a primary input port.

Non-primary input ports may be defined as *read ports,* this means, that the values of the objects that are assigned to this port are only read, but not modified by the activity. All other input ports are called *transfer ports*. Distinguishing read ports and transfer ports is only useful for ports that have an *object type* as domain. There, the question arises whether the object that is assigned to the port is changed by the activity. For *values,* which have no object identity, this question is not relevant. Therefore, we will define read arcs for all ports with value types as domain.

Every activity has at most one *return port,* which has to be one of the output ports and which represents the return value of the activity.

The activity may be invoked only if the *input port assignment rule* is satisfied: (1) every input port has exactly one object or data value assigned if it is single-valued, or any number n of objects or data values assigned, where $n > 0$, if it is multi-valued. (2) Every data value assigned to an input port belongs to the domain of the port. (3) Every object assigned to an input port must belong to every prestate of the port. (4) Every stated precondition is satisfied.

The activity completes successfully, if it satisfies the *output port assignment rule:* (1) every output port has exactly one object or data value assigned if it is single-valued, or any number n of objects or data values assigned, where $n > 0$, if it is multi-valued. (2) Every data value assigned to an output port belongs to the domain of the port. (3) Every object assigned to an output port belongs to every poststate of the port. (4) Every stated postcondition is satisfied.

An activity specification diagram is depicted by a rectangle with the name of the activity on its top. Input ports and output ports are depicted as squares aligned to the left inner border or the right outer border of the rectangle, respectively. The primary port is depicted by a black square, all other ports are depicted by white squares. The return port of an activity is represented by a semicircle depicted around the square representing the corresponding output port. A port that is an input port and an output port is drawn twice with the same name. If a port's domain is a value type, the port symbol is connected by an arc with a oval representing the value type; if a port's domain is an object type, the port symbol is connected with state symbols, which carry the name of the object type and the required state. The connecting arcs for transfer ports and read ports are called *transfer arcs* and *read arcs*, respectively.

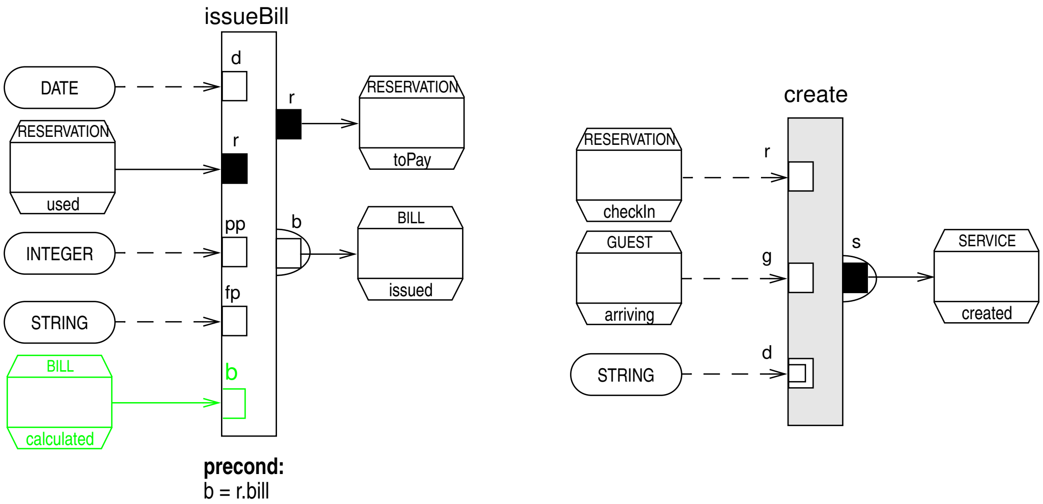
Single-valued ports are depicted by squares with single-line borders, multi-valued ports are depicted by squares with double-line borders. Transfer arcs are depicted by solid lines, read arcs are depicted by dashed lines. Additional pre- and postconditions are annoted textually below the activity symbol.

Example: Figure 2.5 (a) shows the activity specification diagram of activity issueBill of object type RESERVATION. The primary input port p requires an instance of RESERVATION in state used, whereas the primary output port produces the object into state toPay. The prestates and the poststates defined with the primary port correspond to the pre- and poststate depicted in the life-cycle diagram shown in Figure 2.2.

As non-primary ports, the activity requires a date value, an integer value, and a string value for input ports d, pp, and fp. These values will be needed to define the properties dateOfIssue, paymentPeriod, and formOfPayment (cf. Figure 2.1) of the object. These values are only read, thus read-arcs are defined.

Notice that due to the domain restriction constraint that is defined for activity issueBill (cf. Figure 2.4), an implicit input port (depicted in grey color) has to be added that specifies the required prestate calculated of the component object bill. The precondition placed below the activity symbol in Figure 2.5 requires that the object that is assigned to port b is equal to the object referenced by property bill of the reservation object on which activity issueBill is invoked.

The activity returns the instance of BILL, which resides in state issued after completion of issueBill. Notice that port b is an input port as



(a) activity issueBill of RESERVATION (b) activity create of SERVICE

Figure 2.5: Examples of activity specification diagrams

well as an output port. The object changes its state from calculated to issued as activity issueBill triggers activity issue on the BILL object (cf. the triggered synchronization constraint in Figure 2.3).

Figure 2.5 (b) shows the specification of activity create of object type SERVICE. As this activity *creates* the object that is assigned to the primary output port, there is no primary input port defined. An instance of SERVICE is constituted by an instance of GUEST and an instance of RESERVATION (cf. Figure 2.1 for the definition of SERVICE). Thus, the objects that constitute the instance of SERVICE have to be specified at the time of creation. Further, there is a multi-valued port d, to which a set of string values is assigned.

We allow that *non-primary* ports with an object type o as domain refer to *abstract* states of o. Abstract states are used to indicate that the objects that are assigned to the port must reside in at least one state of the subnet N that constitutes the abstract state. This possibility is equivalent to introducing a parallel state in the object life-cycle of o which is marked with an object if and only if the object resides in subnet N. Abstract states will be discussed in detail during *specialization* of object life-cycles (cf. Section 2.3.4).

Example: The activity specification diagram of RESERVATION.create requires an instance of GUEST in state active (cf. Figure 2.6 — all other

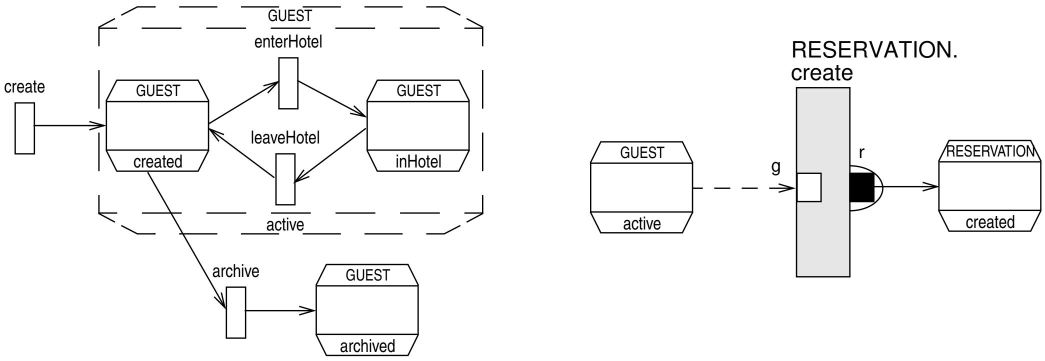


Figure 2.6: Abstract states in activity specification diagrams

ports have been omitted in the figure for simplicity). This state is abstract as indicated in the life-cycle diagram of GUEST. Alternatively, state active could be defined as a parallel state that is a poststate of create and a prestate of archive and thus is marked with an object if and only if this object resides in one of the states (or activity states) of the subnet consisting of created, enterHotel, inHotel, and leaveHotel.

2.2.3 Activity Realization Diagrams

Activity realization diagrams define the realization of an activity using a syntax similar to Nassi-Shneiderman diagrams. The description of activity realization diagrams is omitted in this chapter, as these diagrams will not be considered in this work any more. For details, the reader is refered to [KS91a, KS96b].

2.2.4 Behavior Composition: Compound Activities

Compound activities provide for composition of basic activities. Several specified activities can be combined to a compound activity, which defines a control and data flow between the activities and invokes them. Every compound activity belongs to a certain object type and may invoke activities on objects of this type and of other types. It is defined by an activity specification diagram and an activity realization diagram.

Compound activities were introduced to define complex activities — like complex procedures in programming languages that invoke other procedures — and to predefine higher-level activities that are often needed by the users. They can be defined *without* modifying the object's basic behavior and thus

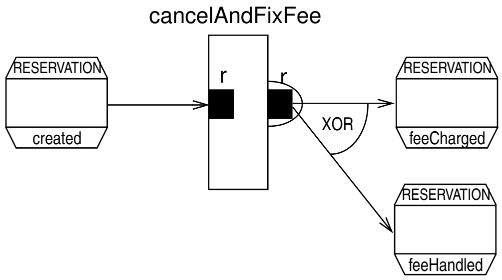


Figure 2.7: Activity specification diagram of a compound activity

without modification of the definition of the object type. In [KS96b], compound activities are motivated by the fact that working steps of human or other agents need not invoke several activities of the life-cycle diagram but can invoke one compound activity instead, which performs the same work. This motivation is partly invalidated by *workflow transparency,* which is presented in Chapter 3.

A compound activity may invoke *alternative activities*; thus the set of prestates from which it consumes and the set of poststates into which it produces depends on the actually chosen alternative. Therefore, any input and output port of a compound activity may have *alternative* pre- and poststates, which are connected by a solid arc marked with the keyword XOR. Thus, the input and output port assignment rules are modified in rule (3) as any object assigned to a input port or output port must belong to exactly one alternative pre- or poststate, respectively.

For each compound activity, there has to be an activity realization diagram, which is not considered in this work. For details, the reader is refered to [KS96b].

Example: Consider once more the life-cycle diagram shown in Figure 2.2. After execution of activity cancelRoomRes, either a fee is charged and the object moves into state feeCharged or no fee is charged and the object moves into state feeHandled. Activities cancelRoomRes, chargeFee, and chargeNoFee make up the compound activity cancelAndFixFee shown in Figure 2.7. The poststate of this activity depends on the decision which one of the alternative activities chargeFee and chargeNoFee is executed. Thus, the poststates are marked with XOR.

2.3 Specialization

Specialization of object types is one of the key concepts of object-oriented models. Specialization provides for inheritance of structure and behavior from the supertype to its subtypes. When the definition of one type is inherited by another type, modifications of the definition should be allowed upto a certain degree.

As there is no common definition in which way the definition of the subtype may vary from the definition of its supertype, we will deal with this subject in the realm of Object/Behavior Diagrams.

In this section, an informal overview of the specialization rules in OBD is given. In the next section, a formal definition of life-cycle diagrams is presented together with specialization rules as far as needed for the remainder of this work.

2.3.1 Specialization of Object Types

The specialization relationship between object types is depicted in a specialization hierarchy. The specialization hierarchy is a *directed, acyclic graph* where each node corresponds to an object type. There is a directed arc from node o_1 to o_2 if and only if o_1 is a specialization of o_2. In that case, o_2 is a direct supertype of o_1 and o_1 is a direct subtype of o_2. More generally, o is a (indirect) supertype of o' (and o' is a (indirect) subtype of o) if and only if there is a directed path from o' to o. As no multiple inheritance is defined in OBD, each node may have one outgoing arc at most, i.e., each object type may have no or one supertype.

In difference to other object-oriented languages, where a common root supertype of the specialization hierarchy has to be defined (e.g., Smalltalk [GR89]) there need not be a common root type of the specialization hierarchy in OBD. Therefore, the specialization hierarchy may consist of several independent graph components; each component is called *type hierarchy.* Further, OBD differs from some other object-oriented languages (e.g., C++ [Lip91], Eiffel [Mey92]), as only single inheritance, but no multiple inheritance is allowed. Multiple inheritance has not yet been considered in OBD as we assume that in many cases multiple inheritance can be adequately replaced by the concept of *roles* (cf. [GSR96] for further discussion).

Object types that can be instantiated are called *instantiable*, object types that must not be instantiated are called *non-instantiable*. According to the *abstract superclass rule* [Hür94], all internal nodes of the specialization graph, i.e., all object types that have subtypes, should be non-instantiable, such that only the most special object types can be instantiated. For instantiable

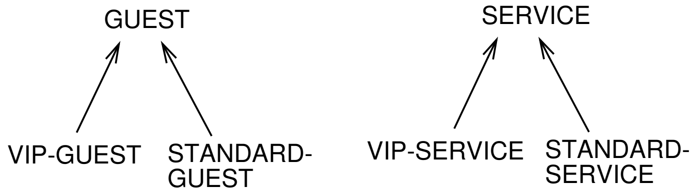


Figure 2.8: Example of a specialization hierarchy

object types it must hold that all activities are specified.

Example: Figure 2.8 shows a small example of a specialization hierarchy. Object type GUEST is specialized to subtypes VIP-GUEST and STANDARD-GUEST; object type SERVICE is specialized to subtypes VIP-SERVICE and STANDARD-SERVICE. Both components of the graph are independent, there is no common supertype. Object types GUEST and SERVICE are non-instantiable, all other object types are instantiable.

2.3.2 Specialization of Object Diagrams

An object diagram is specified for each (instantiable and non-instantiable) object type. With a subtype, the structure inherited from the supertype may be specialized only according to the following rules:

1. The *rule of extension of properties* defines that the subtype inherits all properties of the supertype and may define additional properties.

2. The *rule of original definition of properties* says that every property is originally defined with exactly one object type.

3. The *rule of immediate definition of object-defining properties* requires that any key attribute, *componentOf-*, *hasConstituent-*, and *roleOf* relationship is always originally defined with an object type that has no supertype. This means that a subtype must not define any additional object-defining properties.

4. The *rule of specialization of domains of properties* requires that the domain of a property that is inherited from the supertype (1) may not be redefined if the domain is a value type, (2) may only be specialized to a subtype of the original domain if the domain is an object type.

Specialization due to rules 1 to 3 is refered to as *extension*, specialization according to rule 4 is called *refinement*.

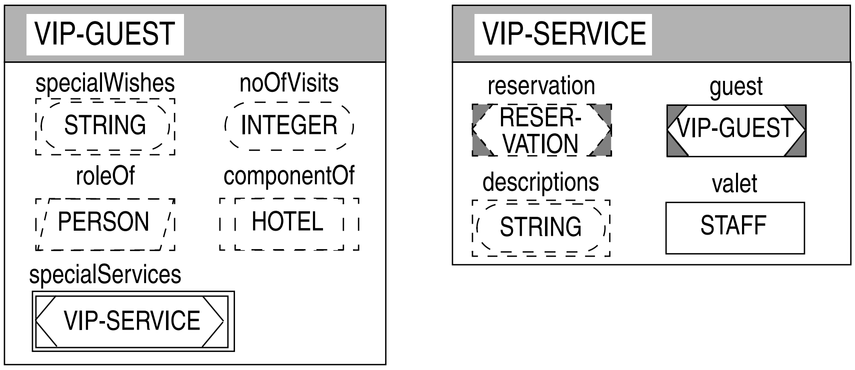


Figure 2.9: Object diagrams of object types VIP-GUEST and VIP-SERVICE

In the graphical notation, only additional and redefined properties have to be depicted in the object diagram. Properties that are inherited without change are depicted with dashed borders or omitted at all.

Example: The object diagrams of object types VIP-GUEST and VIP-SERVICE, which have been defined as subtypes of GUEST and SERVICE, respectively (cf. Figure 2.8), are depicted in Figure 2.9. The properties inherited from the supertypes without change are depicted with dashed borders. Further, according to rule 1, a new property valet is defined with VIP-SERVICE, as a member of the hotel's staff is responsible for the service of a vip guest during his/her stay in the hotel. The domains of properties specialServices in object type VIP-GUEST as well as guest in object type VIP-SERVICE have been refined to a subtype according to rule 4. Properties reservation and guest of object types SERVICE and VIP-SERVICE are object-defining. All of these properties have already been defined with the supertype (i.e., with SERVICE) according to rule 3.

Attributes (i.e., properties with value types as their domains) defined with the supertypes are not modified (cf. rule 4).

Notice that object type STANDARD-SERVICE introduced as a subtype of SERVICE in Figure 2.8 may not define a property valet as thereby the rule of original definition of properties (cf. rule 2) would be violated.

2.3.3 Specialization of Activities

An activity specification defined in the supertype can be specialized in the subtype. We require that if an activity is *specified* in the supertype, the corresponding activity in the subtype is specified, too; i.e., an activity specification must be defined for this activity according to the rules defined below. If an activity in the supertype is unspecified, it may either be refined to a subnet of activities and states in the subtype (see Section 2.3.4 for more details), it may be defined in the subtype as *non-specified*, too, or it may be *specified* in the subtype, i.e., an activity specification is introduced for it.

Like object structure consists of the definition of properties, the specification of activities consists of the definition of ports. Therefore, similar rules apply to the specialization of activity specifications:

1. The *rule of forbidden modification of ports* says that single-valued or multi-valued ports of the supertype have to be single-valued or multi-valued, respectively, with the subtype, too. The return port has to be the same port as with the supertype. Read arcs must not be changed to transfer arcs and vice versa.

2. The *rule of extension of ports* says, that all ports of the activity specification of the supertype are inherited and additional ports may be defined.

3. The *rule of original definition of ports* says that every port of an activity is originally defined with exactly one object type.

4. The *rule of immediate definition of pre- and poststates of ports* requires that no port may get an additional pre- or poststate that has already been defined with the supertype but that has not been defined as pre- or post state of the port in the supertype activity.

5. The *rule of specialization of domains of ports* says that the domain of a port that is inherited from the supertype (1) may not be redefined if the domain of the port is a value type, and (2) may only be specialized to a subtype of the original domain if the domain is an object type.

Pre- and postconditions, which may be defined with an activity specification, may be specialized, too, in that they are defined more restrictive. This means that if the specialized precondition (or postcondition) is fulfilled, then the original, inherited precondition (or postcondition, respectively) has to be fulfilled, too. Notice that as the precondition may get more restrictive it may even be set to FALSE as the most restrictive precondition possible. This means that an activity must not be invoked at all and thus may be omitted in the subtype (we will refer to this case once more in Section 2.3.4.2).

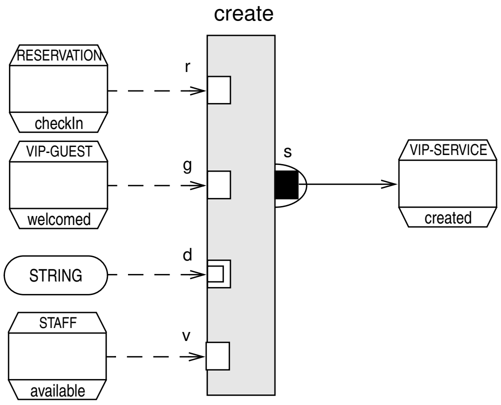


Figure 2.10: Activity specification diagram of activity create of object type VIP-SERVICE

Example: Figure 2.10 shows the specification diagram of activity create defined with object type VIP-SERVICE. Activity create has already been defined with the supertype of VIP-SERVICE, i.e., with object type SERVICE (cf. Figure 2.5).

The specification shown in Figure 2.10 fulfills the rules defined above: The domain of input port g has been specialized from state arriving of GUEST to welcomed of VIP-GUEST (where VIP-GUEST is a subtype of GUEST and state welcomed results from refining state arriving, as explained in Section 2.3.4.1) and an additional port v reading an instance of STAFF has been defined. These modifications reflect the change of the object structure where the domain of property guest has been specialized and an additional property valet has been defined (cf. Figure 2.9).

2.3.4 Specialization of Object Life-Cycles

All instances of a certain object type have to behave according to the life-cycle diagram defined with that object type. As these objects may also be considered as objects of the supertypes, they have to behave in some sense according to the life-cycle diagram of the supertype, too.

Several approaches consider specialization of object life-cycles. In [KS94], *extension* of object life-cycles was discussed for the first time in the realm of OBD. Extension means that activities and states that are not defined in the object life-cycle of the supertype are inserted in the object life-cycle of the subtype. This approach was consolidated in [SS95], where the authors define three kinds of behavior consistency, (1) *observation consistency,* (2) *weak invocation consistency*, and (3) *strong invocation consistency.* These terms will be explained later in this section. In [Sch91, KS96b], rules for stepwise *refinement* of object life-cycles are defined. [SS97] defines a more general approach for refinement by extending the definition of behavior diagrams with labels.

In [vdAB97], specialization of object life-cycles is considered for Petri nets, too, where *protocol, projection, protocol/projection,* and *life-cycle inheritance* are defined, which are comparable to the terms observation and invocation consistency. Several other approaches consider the notion of equivalence and refinement of Petri nets (e.g., [BGV91, vGG90, PRS92]. Some publications [EE94, SHJ$^+$94] treat specialization of life-cycle diagrams specified by statecharts [Har87], where [EE94] distinguishes *observable* and *invocable behavior*, too. In [EKW92], object life-cycles are defined by *state nets.* In the state net of the subtype, triggers and actions may be made more specific and additional behavior may be defined.

In this section, we give a brief informal overview of behavior consistency defined in the publications about OBD [Sch91, KS96b, SS95, SS97]. Specialization of object life-cycles can be regarded as refinement and subsequent extension. Therefore, we will describe these two steps of specialization separately.

In the next section (cf. Section 2.4), a formal definition of *observation consistent specialization* will be given as this kind of specialization will be further considered in the remainder of this work.

2.3.4.1 Refinement

Refinement of object life-cycles was introduced in [Sch91] where object life-cycles of object types were defined by stepwise refinement as known from top-down design of complex systems. At the beginning, there is one single activity called exists and one single state existed, which is the post state of exists.

The life-cycle diagram of any object type may be defined by repeated refinement of activities and states by using a set of *refinement primitives.* All activities and states that are further refined are called *abstract*, whereas all other activities and states are called *basic.* If an activity is abstract, it must be unspecified, i.e., no activity specification is defined for it. Figure 2.11 shows the introduced refinement primitives. We distinguish two sets of refinement primitives, one set for activities and one set for states, where each set supports sequential, parallel, alternative, and iterative refinement.

In the following, an activity or state is refered to as a *source* or a *sink,* if

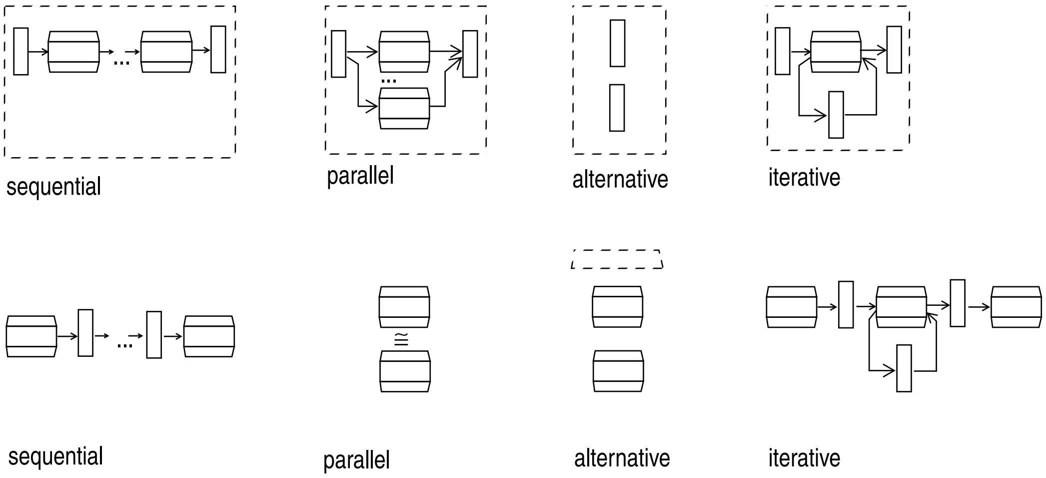


Figure 2.11: Refinement primitives

it has no ingoing or no outgoing arcs, respectively. An abstract state (or an abstract activity) that is refined to a set of activities, states, and connecting arcs, has only states (or activities, respectively) as sources and sinks. An abstract state represents a period of time during which an object resides in this state. This period of time starts when the object comes into some source state of the abstract state and ends when the object leaves the abstract state by leaving the sink states. Similarly, an abstract activity is started on an object if one of its source activities is invoked on the object and ends if one of the sink activities is completed on the object.

Example: Figure 2.12 shows the construction of the life-cycle diagram of object type RESERVATION by stepwise refinement. Every definition of a life-cycle diagram starts with activity exists and its poststate existed. First, activity exists is refined sequentially to activities create and useOrCancel and state created. Activity useOrCancel is further refined alternatively, etc. State existed is refined to the alternative states negArchived and posArchived. In the figure, refinements are depicted either in-line or by a separate diagram. The resulting life-cycle diagram is shown in Figure 2.2.

Besides the *refinement primitiva,* there is a set of *refinement rules,* which have to be obeyed in order to obtain a consistent refinement.

1. *Refinement compatibility rule:* Sources and sinks of state refinements are states, whereas sources and sinks of activity refinements are activities.

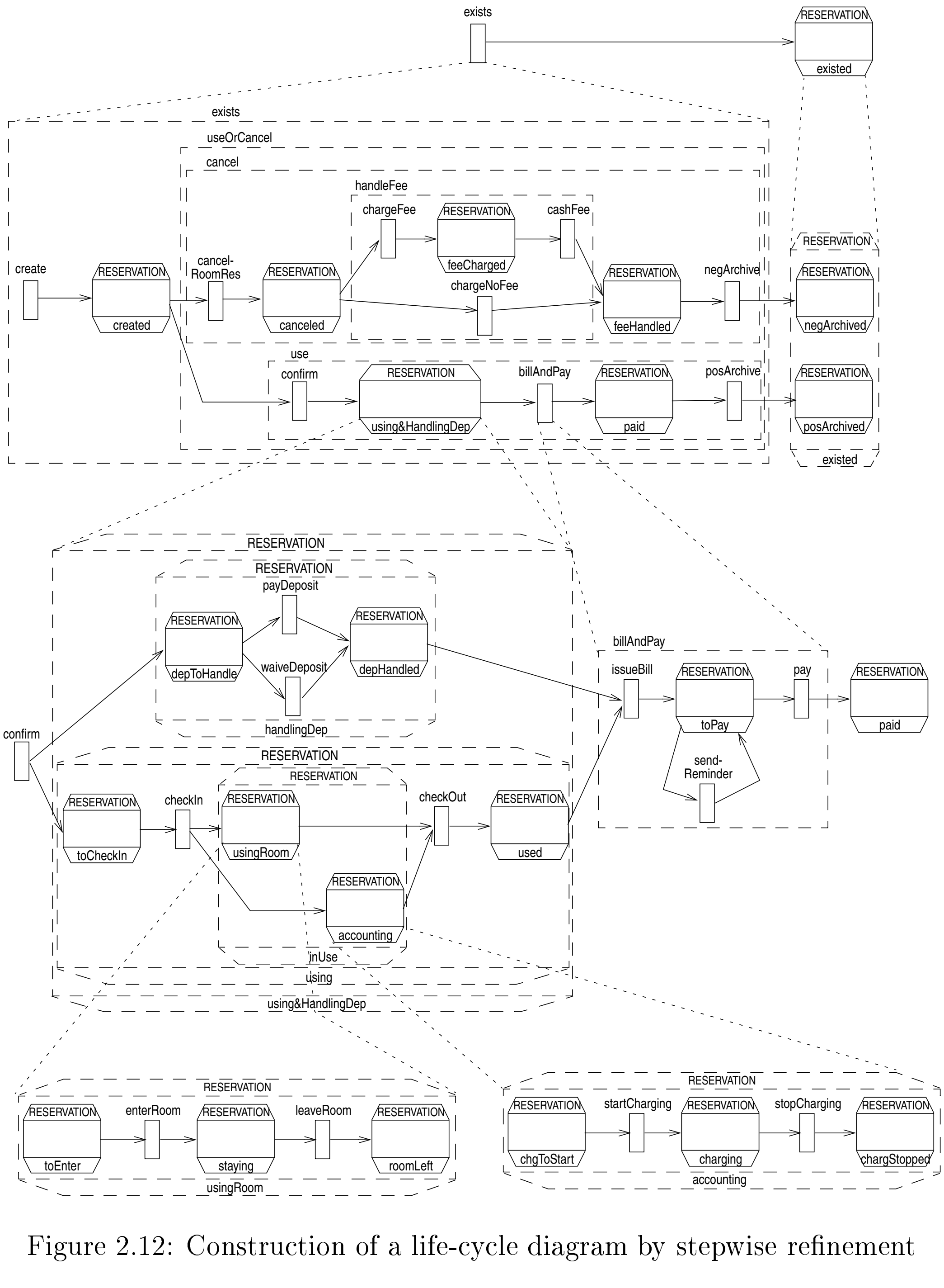


Figure 2.12: Construction of a life-cycle diagram by stepwise refinement

Example: In Figure 2.12, activities chargeFee and chargeNoFee are the sources and activities cashFee and chargeNoFee are the sinks of activity handleFee. States negArchived and posArchived are sources and sinks of state existed.

2. *Transit rule:* Every activity refinement and state refinement has to have at least one source and one sink.

   Example: This rule is fulfilled for all abstract states and activities shown in Figure 2.12, e.g., activity billAndPay has one source issueBill and one sink pay, state using&HandlingDep has two sources depToHandle and toCheckIn and two sinks depHandled and used.

3. *Structured refinement rule:* Only sources and sinks of a refinement may be connected with elements outside the refinement.

   Example: In the abstract activity useOrCancel, only the source activities cancelRoomRes and confirm consume from states defined outside of activity useOrCancel, and only the sink activities negArchive and posArchive produce into states outside the abstract activity.

4. *Pre- and poststate satisfaction rule:* Every prestate of an abstract activity t has to be prestate of each source activity of t. Every poststate of an abstract activity t has to be poststate of each sink activity of t.

   Example: State created is the prestate of the abstract activity useOrCancel and it is the prestate of activities cancelRoomRes and confirm, which are the source activities of the refinement of useOrCancel.

5. *Pre- and poststate specialization rule:* Every prestate of a source activity of an abstract activity t must either be a prestate of t or be a substate of a prestate of t. Similarly, every poststate of a sink activity of an abstract activity t must either be a poststate of t or be a substate of a poststate of t.

   Example: State created is a prestate of activity useOrCancel and a prestate of the source activities of useOrCancel, i.e., of cancelRoomRes and confirm. Activity exists produces into state existed. The sink activities of exists, i.e., activities negArchive and posArchive, produce into states negArchived and posArchived, respectively, which are substates of existed.

Notice that an activity that consumes from (or produces into) an abstract state that is refined into alternative states has to consume from (or produce into, respectively) exactly *one* sink (or source, resp.) state of the refinement of the abstract state. Analogously, if an activity consumes from (or produces into) an abstract state that is refined into parallel states, it has to consume from (or produce into, resp.) *all* sink (or source) states of the refinement of the abstract state.

Example: Activity confirm produces into the abstract state using&HandlingDep. This state is refined to the parallel states using and handlingDep. According to the pre- and poststate specialization rule, activity confirm has to produce into both parallel states. The source issueBill of activity billAndPay consumes from using&HandlingDep and thus has to consume from both sinks of this state.

State existed is refined to alternative states negArchived and posArchived. Thus, the sink activities of exists produce either into negArchived or into posArchived.

6. *Insert and removal specialization rule:* An activity may only insert an object into some source state of state s if it inserts the object also into state s. Analogously, an activity may only remove an object from some sink state of a state s if it removes the object also from state s. This rule says that if an activity a' belongs to the refinement of activity a and a state s' belongs to the refinement of state s and no arc was defined between a and s, then no arc must be defined between a' and s'. Thus, no new arcs can be defined between elements during refinement.

Example: Activities negArchive and posArchive produce into states negArchived and posArchived, respectively, as the more abstract activity exists produces into state existed.

7. *Up-to basic refinement rule:* Every basic activity that has an abstract poststate s must have at least one source state of s as poststate. Every basic activity that has an abstract prestate s must have at least one sink state of s as prestate.

Example: All activities and states of the life-cycle diagram shown in Figure 2.12 that are not further refined are considered basic. The final life-cycle diagram is once more shown in Figure 2.2.

Another approach defined in [SS97] is more general, but it requires an extension of life-cycle diagrams with *labels*. Labels are used to ensure that an object leaves the refinement of an activity or state always entirely; the situation that an activity outside an abstract state consumes a token from one of the substates of the abstract state but leaves another token within the abstract state is not allowed and can be avoided if activities and states are refined using labels.

The approach of [SS97] defines *observation consistent refinement,* which means that if there is an object life-cycle B of an object type o and an object life-cycle B' of object type o', then B' is an observation consistent refinement of B if and only if any processing of an object in B' is a valid processing according to B if refined elements are considered unrefined. As the precise description of this approach needs some formal definitions, it is defered to Section 2.4.

2.3.4.2 Extension

In difference to *refinement*, where elements inherited from the supertype are refined, extension does not further refine inherited elements but adds new activities and states to the object life-cycle. Different restrictions have to be considered depending on whether an observation consistent, a weak invocation consistent, or a strong invocation consistent extension of the object life-cycle of the supertype has to be modeled.

First, an informal introduction of the rules as introduced in [SS95] is given; then we will explain which of these rules have to be considered for different kinds of extension. In the following, we will refer to the life-cycle diagram of the supertype and the subtype as B and B', respectively.

E1. *Partial inheritance:*

- B and B' have to have the same initial state. This state represents a virtual state from which the create activity consumes. It represents the period of time during which the object has not yet been created.
- If there is an arc in B and the activity that is connected with this arc is inherited in B' then the state connected with this arc and the arc must be inherited, too.

E2. *Immediate definition of prestates and poststates:*
If an arc is defined in B' and the activity and the state connected by this arc are inherited from B, then this arc has to be defined with B,

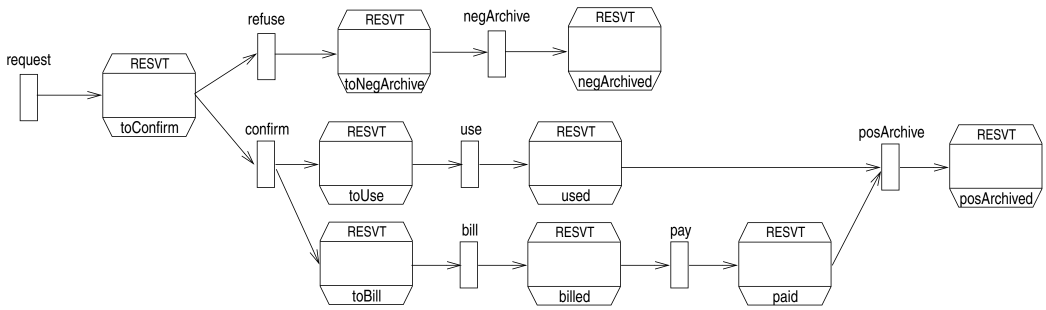


Figure 2.13: Life-cycle diagram of object-type RESVT

too. It is not allowed to add arcs between elements that are inherited from the supertype diagram.

E3. *Parallel extension:*
An activity that is added in B' (i.e., that does not exist in B) must not be connected to states that are inherited from B. This means that no new alternatives may be defined with B' that consume from inherited states.

E4. *Full inheritance:*
All activities of B have to be inherited by B'.

E5. *Alternative extension:*
A state that is added in B' may not be a prestate of an activity inherited from B. This means that no inherited activities may consume from new states, which means that parallel extension is only allowed if it is not synchronized with an inherited activity.

Further, we will only consider *substantial extensions:* B' is a substantial extension of B if a state added in B' is not both prestate of an inherited activity and poststate of an inherited activity.

Example: Figure 2.13 shows the life-cycle diagram of object type RESVT. In the following, object types RESVT-OBCO, RESVT-WICO, and RESVT-SICO will be defined as subtypes of RESVT, which are *observation consistent, weak invocation consistent,* and *strong invocation consistent* extensions of RESVT, respectively.

Observation consistency. B' is an *observation consistent extension* of B if the processing of any object according to B' can be observed as a correct

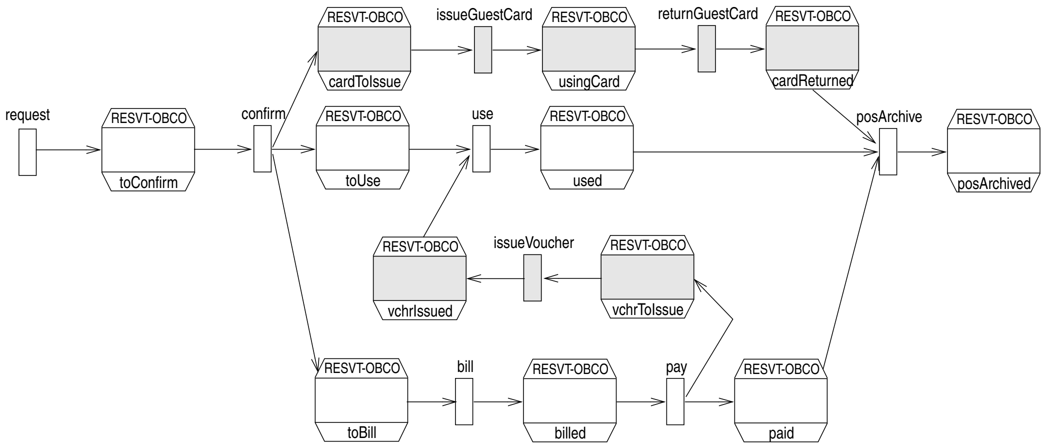


Figure 2.14: Life-cycle diagram of object type RESVT-OBCO

processing according to B if activities and states added in B' are ignored. This means that the behavior of an object of a subtype is consistent to the object life-cycle of the supertype as the supertype may consistently *observe* how the object is processed.

B' is an observation consistent extension of B if and only if rules *E1, E2,* and *E3* are obeyed. Notice that no full inheritance is required; thereby, alternatives defined with the supertype may be removed in the subtype. This corresponds to the possibility to omit activities by specializing their preconditions to FALSE.

Example: The life-cycle diagram of RESVT-OBCO shown in Figure 2.14 is an observation consistent extension of the life-cycle diagram of RESVT. The activities and states depicted shaded have been added by parallel extension (cf. rule *E3*). An alternative that consists of activities refuse and negArchive as well as states toNegArchive and negArchived, which has been specified with RESVT, has been omitted in RESVT-OBCO according to rule *E1.*

Weak invocation consistency. B' is a *weak invocation consistent extension* of B if any processing of an object according to the object life-cycle of B is a correct processing of the object according to the object life-cycle of B'.

B' is a weak invocation consistent extension of B if and only if rules *E1, E2, E4,* and *E5* are obeyed.

Example: The life-cycle diagram of RESVT-WICO shown in Figure 2.15 is a

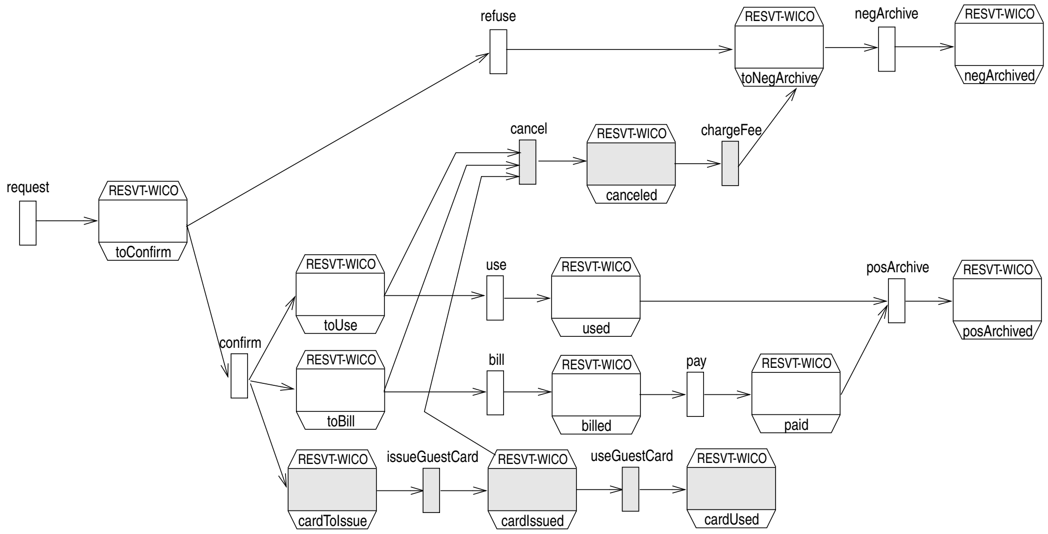


Figure 2.15: Life-cycle diagram of object type RESVT-WICO

weak invocation consistent extension of the life-cycle diagram of RESVT. All activities of RESVT are inherited according to rule *E4.* The elements depicted shaded have been added according to rule *E5:* activities cancel and chargeFee as well as state canceled form a new alternative; activities issueGuestCard and useGuestCard as well as states cardToIssue, cardIssued, and cardUsed form an allowed parallel extension as no inherited activity consumes from a new state.

Strong invocation consistency. B' is a *strong invocation consistent extension* of B if activities defined with B can be invoked on an instance of B' in the same way as on an instance of B although new activities, i.e., activities, which have been added in B', have already been executed on the object.

B' is a strong invocation consistent extension of B if and only if rules *E1* to *E5* are obeyed.

Example: The life-cycle diagram of RESVT-SICO shown in Figure 2.16 is a strong invocation consistent extension of the life-cycle diagram of RESVT. Different from the life-cycle diagram of RESVT-WICO (cf. Figure 2.15), no alternative extension is allowed due to rule *E3.* Thus, the alternative consisting of cancel, canceled, and chargeFee is not allowed.

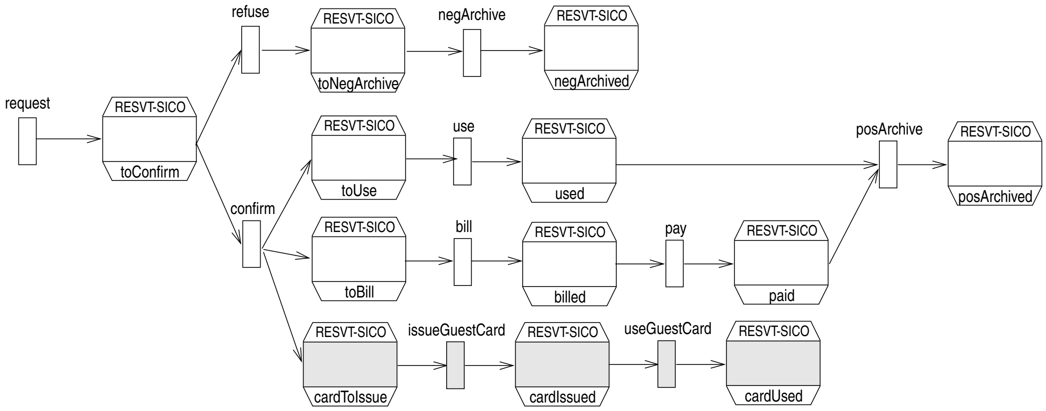


Figure 2.16: Life-cycle diagram of object type RESVT-SICO

2.4 Labeled Behavior Diagrams

This section provides a formal definition of Behavior diagrams, particularly of the life-cycle diagrams. Life-cycle diagrams and their specialization will be needed throughout this work; thus precise definitions of these concepts are useful.

The definitions given in this section are particularly taken from the work of Schrefl and Stumptner. This section gives an overview, further details can be found in [SS95, SS97].

The difference between the previously introduced life-cycle diagrams and the approach defined in [SS97] is the introduction of *labels*, which are necessary to check observation consistent refinement of behavior diagrams.

The idea of labels is taken from the processing of business processes by paper work where different actors work on different carbon copies of a business form. In this analogy, a label corresponds to a particular carbon copy. An activity may collect several copies of a form and distribute them after processing. During the time between processing within activities, a copy resides in a state, in which it is waiting for further processing. As states indicate the progress of processing of the copy, they are colloquially often refered to as *processing states*. A copy may be transfered from an activity into a processing state and vice versa. Analogously, in behavior diagrams, a set of labels is assigned to each activity and each state to show which labels are treated within an activity, which labels reside in some state, and how labels are transfered between activities and states.

Notice, however, that *there exist no actual copies of an object* in OBD, but several activities and states may refer at the same time to the same object by its object identifier.

2.4.1 Basic Definitions

In this section, the basic definitions of labeled behavior diagrams are given, yet without considering specialization. Specialization is treated in the subsequent sections.

Definition 1 (Labeled behavior diagram) *A* labeled behavior diagram *(LBD) B of an object type, where $B = (S, T, F, L, l)$, consists of a set of states $S \neq \emptyset$, a set of activities $T \neq \emptyset$, $S \cap T = \emptyset$, a set of arcs $F \subseteq (S \times T) \cup (T \times S)$, such that $\forall t \in T : (\exists s \in S : (s, t) \in F) \wedge (\exists s \in S : (t, s) \in F)$ and $\forall s \in S : (\exists t \in T : (s, t) \in F) \vee (\exists t \in T : (t, s) \in F)$. L is a set of labels. The labeling function $l : F \rightarrow 2^L \setminus \{\emptyset\}$ (2^X denotes the power set of X) assigns a non-empty set of labels to each arc in F.*

There is one initial state *$\alpha \in S$ such that $\nexists (t, \alpha) \in F$ and there exists a set of* final states *$\Omega \subset S$ such that $\nexists (s, t) \in F : s \in \Omega$. The outgoing arcs of the initial state α carry all labels, i.e., $\forall (\alpha, t) \in F : l(\alpha, t) = L$.*

Example: Figure 2.17 shows the labeled behavior diagram of object type RESVT-L. Label r represents the reservation copy of an imaginary reservation form and is assigned to all arcs that connect activities and states considering reservation and use of a room. Label p represents the payment copy and is assigned to all arcs, activities, and states considering payment.

Activity issue is the activity executed on creation of a new instance of RESVT-L and consumes from a virtual initial state α, which is depicted with dashed borders and which is usually omitted in the graphical representation. There are two final states posArchived and negArchived.

There are some important notes to Definition 1:

- The definition of labeled behavior diagrams includes only a function for labeling of *arcs.* The function $\lambda : S \cup T \rightarrow 2^L \setminus \{\emptyset\}$ representing the labeling of *activities and states* can be derived from the labeling of arcs, in that the set of labels assigned to a state or activity is equal to the set of labels assigned to its incident arcs. Due to the *labeling properties* defined below, the following definition considers only labels of the *incoming* arcs.

$$\lambda(e) := \begin{cases} \bigcup_{\exists (f,e) \in F} l(f, e) & \text{if } e \in (S \cup T \setminus \{\alpha\}) \\ L & \text{if } e = \alpha \end{cases}$$

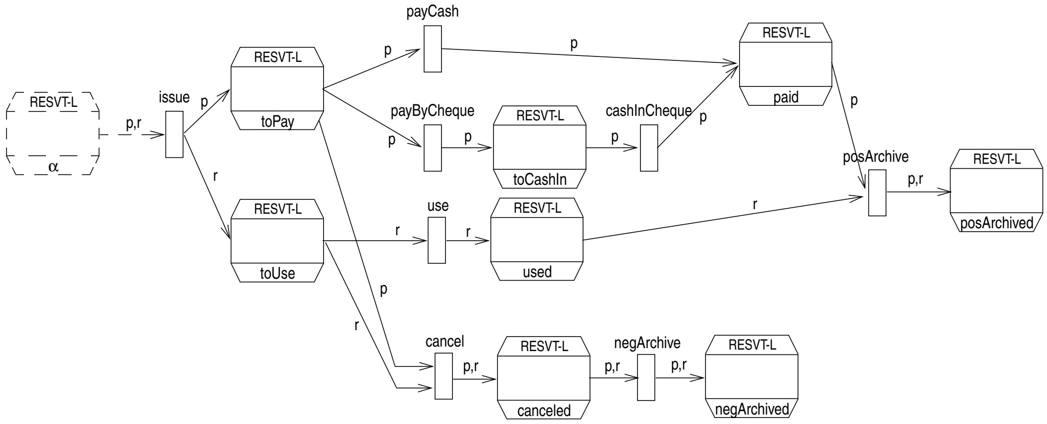


Figure 2.17: Labeled behavior diagram of object type RESVT-L

Example: Activity posArchive is labeled with labels p and r, state used is labeled with r, state α is labeled with p and r.

- In the following, states, activities, and labels are commonly refered to as *elements*.
- We use the notation ${}^\bullet t$ and $t^\bullet$ to refer to the pre- and poststates of an activity t, i.e., ${}^\bullet t := \{s \in S | (s,t) \in F\}$ and $t^\bullet := \{s \in S | (t,s) \in F\}$.

A labeled behavior diagram has to fulfill the following *labeling properties: label preservation, unique label distribution,* and *common label distribution.*

Definition 2 (Label preservation) *A labeled behavior diagram B observes the* label preservation *property if for every $t \in T$ the incoming and the outgoing arcs carry the same set of labels, i.e., $\bigcup_{s \in {}^\bullet t} l(s,t) = \bigcup_{s \in t^\bullet} l(t,s)$.*

Example: The incoming arcs of activity cancel are labeled with p and r, respectively, the outgoing arc is labeled with p and r.

Definition 3 (Unique label distribution) *A labeled behavior diagram B observes the* unique label distribution *property if for every activity the sets of labels of all incoming arcs are disjoint and the sets of labels of all outgoing arcs are disjoint, i.e.,*
(a) $s \in S, s' \in S, t \in T : s \neq s' \wedge (s,t) \in F \wedge (s',t) \in F \Rightarrow l(s,t) \cap l(s',t) = \emptyset$, and
(b) $s \in S, s' \in S, t \in T : s \neq s' \wedge (t,s) \in F \wedge (t,s') \in F \Rightarrow l(t,s) \cap l(t,s') = \emptyset$.

Example: The incoming arcs of activity cancel are labeled with label p and r, respectively. No label is defined with more than one incoming arc.

Definition 4 (Common label distribution) *A labeled behavior diagram B observes the* common label distribution *property if all incident arcs of a state carry the same label set, i.e.,*
(a) $s \in S, t \in T, t' \in T : (s,t) \in F \wedge (s,t') \in F \Rightarrow l(s,t) = l(s,t')$*,*
(b) $s \in S, t \in T, t' \in T : (s,t) \in F \wedge (t',s) \in F \Rightarrow l(s,t) = l(t',s)$*, and*
(c) $s \in S, t \in T, t' \in T : (t,s) \in F \wedge (t',s) \in F \Rightarrow l(t,s) = l(t',s)$*.*

Example: All incident arcs of state toUse carry label p.

The labeling properties correspond to the requirements of handling copies of a form in practice: Each activity gets copies of a form and passes these copies on again. Activities do not create new copies or destroy existing copies (label preservation). An activity may get a copy only once and pass on a copy only once (unique label distribution). A state always describes the processing state of the same set of copies, thus all activities have to produce the same set of copies into that state and have to consume the same set of copies from that state.

At any time, an object resides in a non-empty set of states, its *life-cycle state*. Intuitively, the life-cycle state of an object is composed of the current state of processing of each label of the object. This fact corresponds to the situation in practice where the processing state of a paper form is composed of the processing states of the form's copies.

Definition 5 (Life-cycle state) *A* life-cycle state *(LCS)* σ *of an object is a subset of* $(S \cup T) \times L$*. The initial life-cycle state is* $\sigma_A = \{(\alpha, x) | x \in L\}$*. A life-cycle state* σ *is* final *if* $\forall (e,x) \in \sigma : e \in \Omega$*.*

Example: In the life-cycle diagram shown in Figure 2.17, $\sigma = \{(toPay, p), (toUse, r)\}$ is a valid life-cycle state. The initial life-cycle state is $\sigma_A = \{(\alpha,p),(\alpha,r)\}$, there are two alternative final life-cycle states $\sigma_{\omega_1} = \{(posArchived, p), (posArchived, r)\}$ and $\sigma_{\omega_2} = \{(negArchived, p), (negArchived, r)\}$.

As mentioned before, activities in OBD are not executed instantaneously but execution may take some time. Therefore we distinguish *start* and *completion* of an activity, where an activity can be started if all prestates of the activity are marked, and where an activity can be completed if it has been started before, i.e., if the object resides in the implicit activity state. Notice, that no two activities may start or complete at the same moment of time.

Example: Because of activity states, the following life-cycle state is valid in the life-cycle diagram of RESVT-L: $\sigma_i = \{(payCash, p), (toUse, r)\}$. Although no two activities may start at the same time, one activity may be started while another activity is just executing; starting activity use in life-cycle state σ_i leads to life-cycle state $\sigma_{i+1} = \{(payCash, p), (use, r)\}$. Completing activity payCash in life-cycle state σ_{i+1} leads to life-cycle state $\sigma_{i+2} = \{(paid, p), (use, r)\}$.

In life-cycle state $\sigma_j = \{(paid, p), (used, r)\}$, activity posArchive can be started as both prestates are marked, resulting in life-cycle state $\sigma_{j+1} = \{(posArchive, p), (posArchive, r)\}$.

Definition 6 (Start and completion of activities) *An activity $t \in T$ may be started on an LCS σ, if $\forall s \in {}^\bullet t, x \in l(s,t) : (s,x) \in \sigma$, and its start yields $\sigma' = (\sigma \setminus \{(s,x) | s \in {}^\bullet t \wedge x \in l(s,t)\}) \cup \{(t,x) | \exists s \in {}^\bullet t : x \in l(s,t)\}$. An activity t may be completed on an LCS σ, if $\exists x \in L : (t,x) \in \sigma$, and its completion yields $\sigma' = (\sigma \setminus \{(t,x) | \exists s \in t^\bullet : x \in l(t,s)\}) \cup \{(s,x) | s \in t^\bullet \wedge x \in l(t,s)\}$.*

A behavior diagram defines all legal sequences of life-cycle states. A particular sequence of life-cycle states of an object is called a *life-cycle occurrence.*

Definition 7 (Life-cycle occurrence) *A* life-cycle occurrence (LCO) *γ of an object is a sequence of life-cycle states $\gamma = [\sigma_1, \ldots, \sigma_n]$, such that $\sigma_1 = \sigma_A$, and for $i = 1, \ldots, n-1$ either $\sigma_i = \sigma_{i+1}$, or there exists an activity $t \in T$ such that either t can be started on σ_i and the start of t yields σ_{i+1} or σ_i contains t and the completion of t yields σ_{i+1}.*

A life-cycle occurrence is called complete *if $\sigma_n \subseteq \{(s,x) | s \in \Omega \wedge x \in \lambda(s)\}$. Otherwise, a life-cycle occurrence is called* partial.

Example: In the behavior diagram shown in Figure 2.17,
$\gamma = [\{(\alpha, p), (\alpha, r)\}, \{(issue, p), (issue, r)\}, \{(toPay, p), (toUse, r)\}, \{(toPay, p), (use, r)\}, \{(toPay, p), (used, r)\}, \{(payCash, p), (used, r)\}, \{(paid, p), (used, r)\}, \{(posArchive, p), (posArchive, r)\}, \{(posArchived, p), (posArchived, r)\}]$ is a valid complete life-cycle occurrence.

In the following, only *safe, activity-reduced,* and *deadlock-free* behavior diagrams are considered. These restrictions are quite useful: (1) a labeled behavior diagram has to be safe as we want to observe exactly one processing state for each label, labels cannot be duplicated; (2) activities that are never executed are not useful; (3) the processing of an object should never stop before a final life-cycle state has been reached.

2.4.2 Observation Consistent Refinement

If a labeled behavior diagram is refined, activites and states are replaced by subnets consisting of activities, states, and connecting arcs, whereas labels are replaced by sets of sublabels. All activities, states, and labels may only be added in the life-cycle diagram of the subtype if they belong to the refinement of an inherited element.

The relationship between the elements of the refined behavior diagram and the unrefined behavior diagram is represented by the refinement function h_r:

Definition 8 (Refined behavior diagrams and refinement function) *A labeled behavior diagram $B' = (S', T', F', L', l')$ that is a* refinement *of another labeled behavior diagram $B = (S, T, F, L, l)$ is a labeled behavior diagram with an associated* refinement function *h_r which has the following properties (we write h instead of h_r if it is clear from context that the refinement function is understood):*

$h : S' \cup T' \cup L' \rightarrow S \cup T \cup L$ *(h is total)*
(1) $e \in S' \cup T' \Rightarrow h(e) \in S \cup T$
(2) $x \in L' \Rightarrow h(x) \in L$
(3) $x \in L \Rightarrow \exists x' \in L' : h(x') = x$
(4) $h(e) \in S' \cup T' \cup L' \Leftrightarrow h(e) = e$
(5) $h(\alpha') = \alpha$
(6) $s' \in S' \wedge t' \in T' \wedge h(s') \neq h(t') \wedge ((s', t') \in F' \vee (t', s') \in F')$
$\Rightarrow h(s') \in S \wedge h(t') \in T$

The refinement function requires that each element in B' corresponds to an element in B (therefore, h has to be a total function). Further, the following properties are defined: (1) each activity and state is mapped to an activity or state (and not to a label); (2) each label is mapped to a label; (3) each label of the original behavior diagram has at least one corresponding label in the refined behavior diagram; (4) unrefined elements are mapped onto themselves; (5) the initial state of the refined behavior diagram is mapped to the initial state of the unrefined behavior diagram; (6) sources and sinks of subnets that result from refinement have to be states, if a state has been refined, and have to be activities, if an activity has been refined.

Example: The labeled behavior diagram of object type RESVT-REF shown in Figure 2.18 is a refinement of the labeled behavior diagram of object

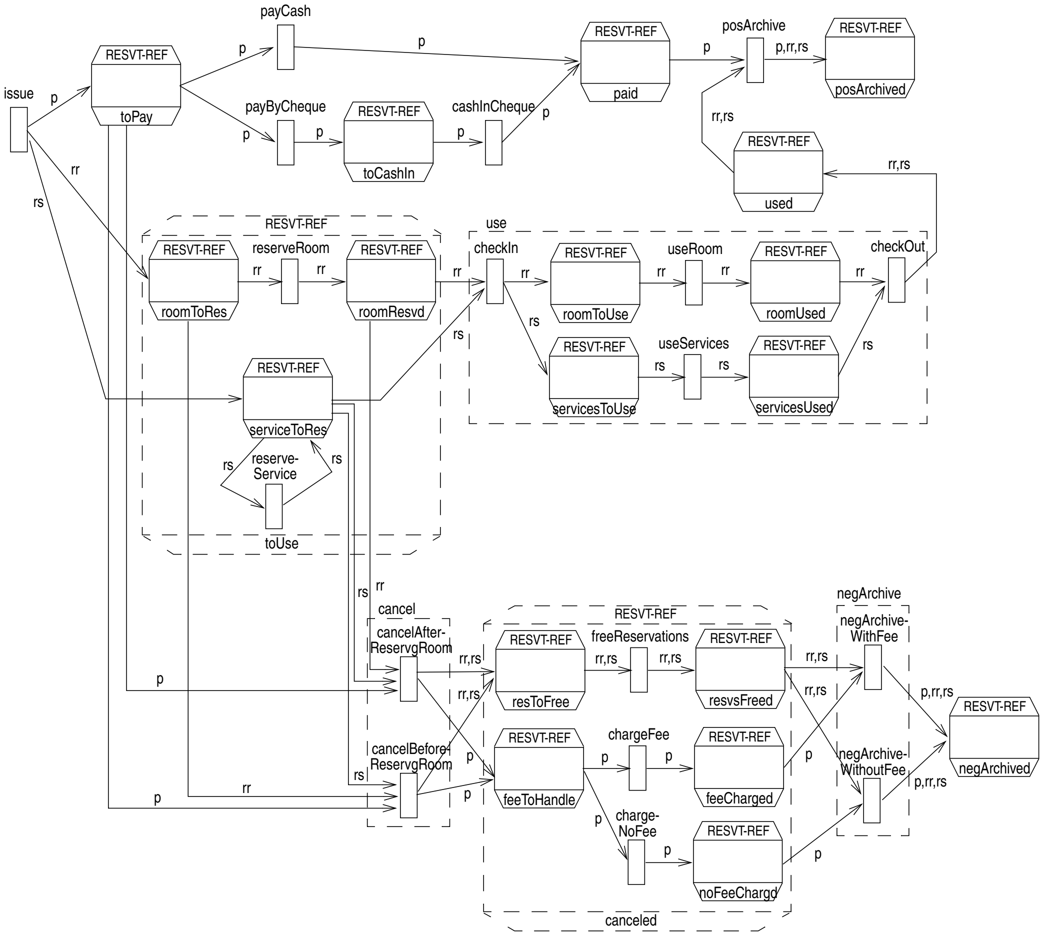


Figure 2.18: Labeled behavior diagram of object type RESVT-REF

type RESVT-L shown in Figure 2.17: Activities use, cancel, and negArchive, states toUse and canceled, as well as label r of RESVT-L have been refined in the behavior diagram of RESVT-REF. The refinement of activities and states is shown in Figure 2.18 by depicting the original, unrefined element with dashed borders around the subnet that constitutes the refinement. Further, label r has been refined to sublabels rr and rs representing the reservation of a room and the reservation of services, respectively. All activities and states that are not depicted within another element depicted with dashed borders are inherited without change.

For the definition of *observation consistent refinement,* we need to check whether each life-cycle occurrence according to the refined behavior diagram is a valid life-cycle occurrence of the original behavior diagram if refined elements are considered unrefined. Therefore, we have to define the *abstraction of a life-cycle state* and the *abstraction of a life-cycle occurrence.*

Definition 9 (Abstraction of a life-cycle state) *The* abstraction of a life-cycle state σ' *of an object type* O' *to object type* O*, written* $\sigma'/_O$*, is defined as* $\sigma'/_O = \{(h(e'), h(x'))|(e', x') \in \sigma'\}$.

Example: Consider the behavior diagram in Figure 2.18: $\sigma = \{(\mathit{toPay}, p), (\mathit{reserveRoom}, rr), (\mathit{serviceToRes}, rs)\}$ is a valid life-cycle state. According to the refinement function h, which maps all elements of RESVT-REF to elements of RESVT-L, the abstraction of σ is $\sigma'/_O = \{(\mathit{toPay}, p), (\mathit{toUse}, r)\}$.

Definition 10 (Abstraction of a life-cycle occurrence) *The* abstraction of a life-cycle occurrence $\gamma' = [\sigma'_1, \ldots, \sigma'_n]$ *of an object type* O' *to object type* O*, written* $\gamma'/_O$*, is defined as* $\gamma'/_O = [\sigma'_1/_O, \ldots, \sigma'_n/_O]$.

Example: Consider the following life-cycle occurrence, which is valid for an instance of RESVT-REF:

$$
\begin{array}{lr}
\gamma = [& \\
\{(\alpha, p), (\alpha, rr), (\alpha, rs)\}, & (\sigma_1) \\
\{(\mathit{issue}, p), (\mathit{issue}, rr), (\mathit{issue}, rs)\}, & (\sigma_2) \\
\{(\mathit{toPay}, p), (\mathit{roomToRes}, rr), (\mathit{serviceToRes}, rs)\}, & (\sigma_3) \\
\{(\mathit{toPay}, p), (\mathit{reserveRoom}, rr), (\mathit{serviceToRes}, rs)\}, & (\sigma_4) \\
\{(\mathit{toPay}, p), (\mathit{roomResvd}, rr), (\mathit{serviceToRes}, rs)\}, & (\sigma_5) \\
\{(\mathit{toPay}, p), (\mathit{checkIn}, rr), (\mathit{checkIn}, rs)\}, & (\sigma_6) \\
\{(\mathit{toPay}, p), (\mathit{roomToUse}, rr), (\mathit{servicesToUse}, rs)\}, & (\sigma_7) \\
\{(\mathit{toPay}, p), (\mathit{useRoom}, rr), (\mathit{servicesToUse}, rs)\}, & (\sigma_8) \\
\{(\mathit{toPay}, p), (\mathit{useRoom}, rr), (\mathit{useServices}, rs)\}, & (\sigma_9) \\
\{(\mathit{toPay}, p), (\mathit{useRoom}, rr), (\mathit{servicesUsed}, rs)\}, & (\sigma_{10}) \\
\{(\mathit{toPay}, p), (\mathit{roomUsed}, rr), (\mathit{servicesUsed}, rs)\}, & (\sigma_{11}) \\
\{(\mathit{toPay}, p), (\mathit{checkOut}, rr), (\mathit{checkOut}, rs)\}, & (\sigma_{12}) \\
\{(\mathit{toPay}, p), (\mathit{used}, rr), (\mathit{used}, rs)\}, & (\sigma_{13}) \\
\{(\mathit{payCash}, p), (\mathit{used}, rr), (\mathit{used}, rs)\}, & (\sigma_{14}) \\
\{(\mathit{paid}, p), (\mathit{used}, rr), (\mathit{used}, rs)\}, & (\sigma_{15}) \\
\{(\mathit{posArchive}, p), (\mathit{posArchive}, rr), (\mathit{posArchive}, rs)\}, & (\sigma_{16}) \\
\{(\mathit{posArchived}, p), (\mathit{posArchived}, rr), (\mathit{posArchived}, rs)\} & (\sigma_{17}) \\
] &
\end{array}
$$

The abstraction of this life-cycle occurrence to object type RESVT-L is equivalent to the life-cycle occurrence shown in an example above. Notice, that due to abstraction, several subsequent life-cycle states may be equal (consider, e.g., the abstraction of life-cycle states σ_3 to σ_5), which is allowed according to Definition 7.

Definition 11 (Observation consistent refinement) *A labeled behavior diagram B' of object type O' is an* observation consistent refinement *of a labeled behavior diagram B of object type O if and only if for any possible life-cycle occurrence γ' of B' holds: $\gamma'/_O$ is a life-cycle occurrence of B.*

Example: The labeled behavior diagram of RESVT-REF is an observation consistent refinement of the labeled behavior diagram of RESVT-L.

In [SS97], rules are defined to check whether a behavior diagram B' is an observation consistent refinement of B (cf. Table 2.1). *Note:* The inverse function of h is defined as follows: $h^{-1}(e) := \{e' \in S' \cup T' \cup L' | h(e') = e\}$.

R1. *Pre- and poststate satisfaction:*

Rules (a1) (and (b1)) say that an activity that consumes from (or produces into, respectively) a refined state, has to consume (or produce) all refined labels of that state. This rule is illustrated in Figure 2.19 (a): Suppose that the elements of the unrefined behavior diagram are depicted in grey color, whereas the elements of the refined behavior diagram are depicted in black color. Originally, activity t consumed label x from state s. Activity t has been refined to a subnet with a source activity t'; state s has been refined to a subnet with states s' and s'' (and possibly some other elements); label x has been refined to labels x' and x''. The rule says that if t' consumes a label x', it has to consume all other labels that result from the refinement of the original label x, too, from a state that belongs to the refinement of s. Thus, t' has to consume x'', too, which is depicted by the dash-dotted line.

Rules (a2) (and (b2)) say that an activity that belongs to a subnet that results from refinement of an activity and that consumes from (produces into, respectively) a state defined outside the activity, has to consume from (produce into) all states from which the unrefined activity consumed (or into which the unrefined activity produced, respectively). In Figure 2.19 (b), activity t is refined to a subnet which contains activity t'. As activity t' consumes from state s' and t consumes from state $\hat{s}$, activity t' has to consume from a state $\hat{s}'$ belonging to the refinement of $\hat{s}$, too.

Example: In our example behavior diagrams of RESVT-L and RESVT-REF (cf. Figures 2.17 and 2.18), the pre- and poststate satisfaction rule is obeyed. Consider, for example, activity issue: in RESVT-L, this activity produces label r into state toUse; in RESVT-REF, label r is refined to rr and rs; activity issue produces both sublabels into the refinement of state toUse. Thus, rule (b1) is obeyed.

In RESVT-L, activity cancel consumes from states toPay and toUse. In RESVT-REF, activity cancelAfterReservgRoom, which results from refinement of activity cancel, consumes from state toPay and state roomResvd, which belongs to the refinement of state toUse. Thus, rule (a2) is obeyed.

R2. *Pre- and post state refinement:*
This rule says that if there is an arc between a subnet that constitutes the refinement of a state and a subnet that constitutes the refinement of an activity, and this arc carries a certain label, then there has to be an arc between the unrefined state and the unrefined activity, too, which has to be labeled with the unrefined counterpart of the label. In Figure 2.19 (c), activity t' consumes label x' from state s'. The rule requires that there is an arc between state s and activity t and there has to be a label x assigned to this arc, where x' is a sublabel of x.

Example: Activity cancelAfterReservgRoom produces labels rr and rs into state resToFree and label p into state feeToHandle. Pre- and poststate refinement is fulfilled as in RESVT-L, activity cancel produces labels p and r into state canceled and activity cancelAfter-ReservgRoom results from refining activity cancel, states resToFree and feeToHandle result from refining state canceled, and labels rr and rs result from refining label r.

As already mentioned before, *labeling* of behavior diagrams provides for a more general and less restricted form of refinement compared to the approach of [Sch91]. At the end of this section, advantages of refinement with labeled behavior diagrams are discussed:

1. The restrictive definition of sources (whose incoming arcs may only connect them to elements outside the refinement) and sinks (whose outgoing arcs may only connect them to elements outside the refinement) is not necessary any more. Instead, activities within and without the refinement may consume from or produce to a state within the refinement.

R1. Pre- and poststate satisfaction

(a1) $t \in T, s \in S, t' \in T', s' \in S'$:
$h(t') = t \wedge h(s') = s \wedge (s,t) \in F \wedge (s',t') \in F' \wedge$
$x \in l(s,t) \wedge x'' \in h^{-1}(x)$
$\Rightarrow \exists s'' \in S' : h(s'') = s \wedge (s'',t') \in F' \wedge x'' \in l'(s'',t')$

(a2) $t \in T, s \in S, \hat{s} \in S, t' \in T', s' \in S'$:
$h(t') = t \wedge h(s') = s \wedge (s,t) \in F \wedge (s',t') \in F' \wedge (\hat{s},t) \in F$
$\Rightarrow \exists \hat{s}' \in S' : h(\hat{s}') = \hat{s} \wedge (\hat{s}',t') \in F'$

(b1) $t \in T, s \in S, t' \in T', s' \in S'$:
$h(t') = t \wedge h(s') = s \wedge (t,s) \in F \wedge (t',s') \in F' \wedge$
$x \in l(t,s) \wedge x'' \in h^{-1}(x)$
$\Rightarrow \exists s'' \in S' : h(s'') = s \wedge (t',s'') \in F' \wedge x'' \in l'(t',s'')$

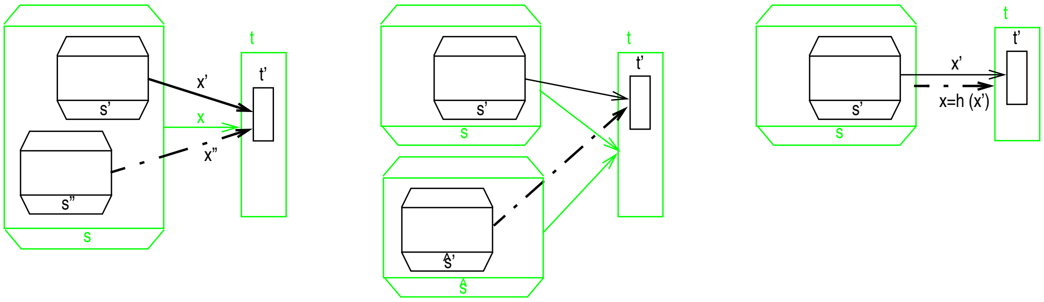
(b2) $t \in T, s \in S, \hat{s} \in S, t' \in T', s' \in S'$:
$h(t') = t \wedge h(s') = s \wedge (t,s) \in F \wedge (t',s') \in F' \wedge (t,\hat{s}) \in F$
$\Rightarrow \exists \hat{s}' \in S' : h(\hat{s}') = \hat{s} \wedge (t',\hat{s}') \in F'$

R2. Pre- and poststate refinement

(a) $s' \in S' \wedge t' \in T' \wedge (s',t') \in F' \wedge h(s') \neq h(t') \wedge x' \in l'(s',t')$
$\Rightarrow (h(s'),h(t')) \in F \wedge h(x') \in l(h(s'),h(t'))$

(b) $s' \in S' \wedge t' \in T' \wedge (t',s') \in F' \wedge h(s') \neq h(t') \wedge x' \in l(t',s')$
$\Rightarrow (h(t'),h(s')) \in F \wedge h(x') \in l(h(t'),h(s'))$

Table 2.1: Rules for checking observation consistent refinement



(a) Rule R1. (a1): Prestate satisfaction (b) Rule R1. (a2): Prestate satisfaction (c) Rule R2. (a): Prestate refinement

Figure 2.19: Illustration of rules for checking observation consistent refinement

Example: Consider the refinement of state toUse shown in Figure 2.18: State serviceToRes, which belongs to the refinement of toUse is prestate and poststate of activities inside and activities outside the refinement of toUse. Further, there is an arc between state roomToRes and activity cancelBeforeReservation although state roomToRes is no sink state of the refinement of toUse.

2. With labels, the pre- and poststate satisfaction rule can be checked. This means that alternative and parallel states within a state refinement can be distinguished (as they carry the same labels in the first case and different labels in the second case).

Example: Consider the refinement of state canceled in Figure 2.18. As states feeCharged and noFeeChargd carry the same labels, they are alternative states, whereas states feeCharged and resvsFreed are parallel as they carry different labels.

2.4.3 Observation Consistent Extension

If a labeled behavior diagram is extended, new activites, states, and labels are added and — due to the rule of partial inheritance — activities and states may be omitted. Different to refinement, inherited elements must not be refined.

The relationship between extended behavior diagram and the unextended behavior diagram is represented by the extension function h_e:

Definition 12 (Extended behavior diagrams and extension function) *A labeled behavior diagram $B' = (S', T', F', L', l')$ that is an* extension *of another labeled behavior diagram $B = (S, T, F, L, l)$ is a labeled behavior diagram with an associated* extension function h_e *which has the following properties (we write h instead of h_e if it is clear from context that the extension function is understood):*

$h : S' \cup T' \cup L' \rightarrow S \cup T \cup L \cup \{\varepsilon\}$ *(h is total)*
(1a) $e \in S' \Rightarrow h(e) \in S \cup \{\varepsilon\}$
(1b) $e \in T' \Rightarrow h(e) \in T \cup \{\varepsilon\}$
(2) $x \in L' \Rightarrow h(x) \in L \cup \{\varepsilon\}$
(3) $x \in L \Rightarrow \exists x' \in L' : h(x') = x$
(4) $h(e) \neq \varepsilon \Leftrightarrow h(e) = e$
(5) $h(\alpha') = \alpha$

The extension function is a total function that maps every element of B' either to an element to B (if the element has been inherited) or to ε (if the element was added in B'). Further, the following properties are defined: (1a) each activity is mapped to an activity or ε; (1b) each state is mapped to a state or ε; (2) each label is mapped to a label or ε; (3) each label of the original behavior diagram has a corresponding label in the extended behavior diagram; (4) all elements that are not mapped to ε are mapped onto themselves; (5) the initial state of the extended behavior diagram is mapped to the initial state of the unextended behavior diagram. Notice that elements of the original behavior diagram may be omitted in the extended behavior diagram.

Example: The labeled behavior diagram of object type RESVT-SPEC shown in Figure 2.20 is an extension of the labeled behavior diagram of RESVT-REF (cf. Figure 2.18). In RESVT-SPEC, a guest card is introduced, which is issued, used, and finally returned again. Thus, label c and several activities and states (which are depicted shaded in Figure 2.20) have been added. Elements payByCheque, toCashIn, and chashInCheque have been omitted.

For the definition of *observation consistent extension,* we need to check whether each life-cycle occurrence according to the extended behavior diagram is a valid life-cycle occurrence of the original behavior diagram if extended elements are ignored. Therefore, we have to define the *restriction of a life-cycle state* and the *restriction of a life-cycle occurrence.*

Definition 13 (Restriction of a life-cycle state) *The* restriction of a life-cycle state σ' *of an object type O' to object type O, written $\sigma'/_O$, is defined as* $\sigma'/_O = \{(e', x') | (e', x') \in \sigma' \wedge h(e') \neq \varepsilon \wedge h(x') \neq \varepsilon\}$.

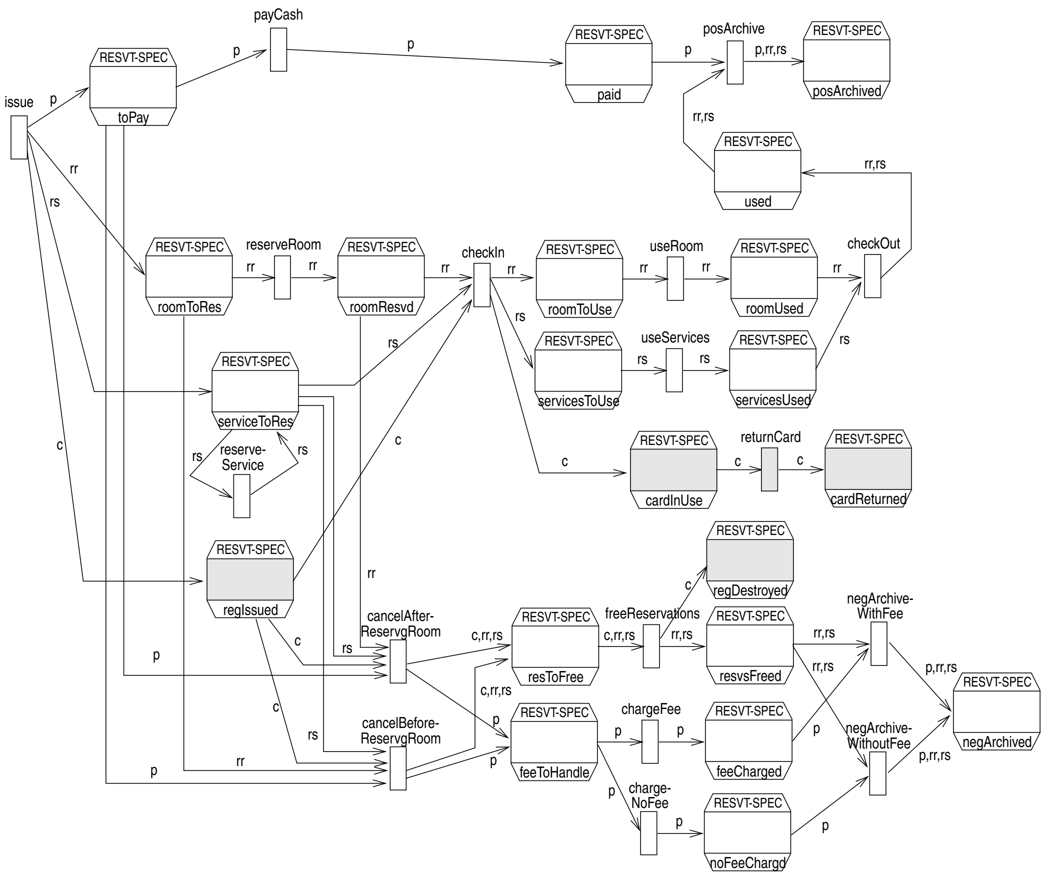


Figure 2.20: Labeled behavior diagram of object type RESVT-SPEC

Example: In the object life-cycle of RESVT-SPEC, $\sigma = \{(toPay, p), (roomToRes, rr), (serviceToRes, rs), (regIssued, c)\}$ is a valid life-cycle state. The restriction to object type RESVT-REF yields $\gamma' = \{(toPay, p), (roomToRes, rr), (serviceToRes, rs)\}$.

Definition 14 (Restriction of a life-cycle occurrence) *The* restriction *of a life-cycle occurrence $\gamma' = [\sigma'_1, \ldots, \sigma'_n]$ of an object type O' to object type O, written $\gamma'/_O$, is defined as $\gamma'/_O = [\sigma'_1/_O, \ldots, \sigma'_n/_O]$.*

Definition 15 (Observation consistent extension) *A labeled behavior diagram B' of object type O' is an* observation consistent extension *of a labeled behavior diagram B of object type O if and only if for any possible life-cycle occurrence γ' of B' holds: $\gamma'/_O$ is a life-cycle occurrence of B.*

Example: The labeled behavior diagram of RESVT-SPEC is an observation consistent extension of the labeled behavior diagram of RESVT-REF.

Similar to the previous section treating refinement, rules are presented to check whether a behavior diagram is an observation consistent extension of another behavior diagram (cf. Table 2.2). For more details, the reader is refered to [SS95, SS97].

E1'. *Partial inheritance:*
Rule (a) requires that the initial states of B and B' are the same and that all labels of B are inherited by B'. Rules (b) and (c) require that if an activity of B is inherited by B', all incident arcs with their assignment of labels have to be inherited, too. Figure 2.21 (a) illustrates the rule: The top half of the figure shows a part of B and the bottom half shows a part of B'. Activity t is inherited (depicted by the bold face arrow). Then all incident arcs and labels are inherited, too (depicted by the bold dotted arrows). As the incident arcs are inherited, the pre- and poststates of t must be inherited, as well.

Example: The behavior diagram of RESVT-SPEC inherits all labels of RESVT-REF; no incident arcs of inherited activities are omitted. As no full inheritance is required, activities and states may be omitted: activities payByCheque and cashInCheque as well as state toCashIn are not inherited.

E2'. *Immediate definition of labels:*
In B', no arcs may be added between elements inherited from B.

Further, no inherited label may be added to an inherited arc. Figure 2.21 (b) shows two possible errors: (1) State $s1$ and activity t are inherited and an arc that is not defined in B is added in B'. (2) Activity t, state $s2$, as well as labels x' and x'' are inherited. In B', the inherited label x'' is added to the inherited arc between t and $s2$. *Note:* Label x'' is depicted in the upper half to show that it is defined in B but not assigned to arc $(t, s2)$ of B.

Example: The rule is fulfilled in our example as no new arcs are defined between inherited elements. Further, only *new* labels, but no inherited labels are added to inherited arcs (e.g., the new label c is added to the inherited arc between cancelAfterReservgRoom and resToFree).

E3'. *No label deviation:*
Incident arcs of activities and states that are added in B' must not be labeled with labels inherited from B. This rule enforces parallel extension as defined in rule *E3.* in Section 2.3.4.2 as an arc that connects an inherited state with an activity that is added in B' would have to be labeled with an inherited label due to the rule of common label distribution. Figure 2.21 (c) shows two possible errors: state $s1$ and label x are inherited, activity t and state $s2$ are added. The arc between $s1$ and t is not allowed. Further, the new arc between t and $s2$ is correct but the assignment of label x to the arc is not allowed.

Example: All new states and activities are only labeled with the new label c.

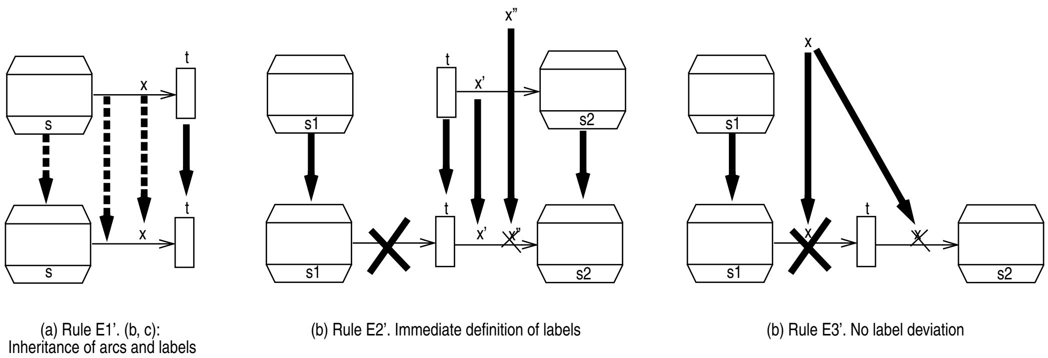


Figure 2.21: Illustration of rules for checking observation consistent extension

E1'. Partial inheritance
(a) $\alpha' = \alpha$, $L \subseteq L'$
(b) $(t \in T' \wedge t \in T \wedge (s,t) \in F) \Rightarrow ((s,t) \in F' \wedge l(s,t) \subseteq l'(s,t))$
(c) $(t \in T' \wedge t \in T \wedge (t,s) \in F) \Rightarrow ((t,s) \in F' \wedge l(t,s) \subseteq l'(t,s))$
E2'. Immediate definition of labels
(a) $((s,t) \in F' \wedge s \in S \wedge t \in T \wedge x \in L \wedge x \in l'(s,t))$
$\Rightarrow ((s,t) \in F \wedge x \in l(s,t))$
(b) $((t,s) \in F' \wedge s \in S \wedge t \in T \wedge x \in L \wedge x \in l'(t,s))$
$\Rightarrow ((t,s) \in F \wedge x \in l(t,s))$
E3'. No label deviation
(a) $(s,t) \in F' \wedge t \in T' \wedge t \notin T \Rightarrow l'(s,t) \cap L = \emptyset$
(b) $(t,s) \in F' \wedge t \in T' \wedge t \notin T \Rightarrow l'(t,s) \cap L = \emptyset$
(c) $(s,t) \in F' \wedge s \in S' \wedge s \notin S \Rightarrow l'(s,t) \cap L = \emptyset$
(d) $(t,s) \in F' \wedge s \in S' \wedge s \notin S \Rightarrow l'(t,s) \cap L = \emptyset$

Table 2.2: Rules for checking observation consistent extension

2.4.4 Observation Consistent Specialization

A behavior diagram is specialized by refinement and subsequent extension. Thus, the problem of specialization may be divided into the problems of refinement and extension described in the previous sections. All differences between a behavior diagram B and a behavior diagram B' that is a specialiation of B are caused either by the refinement or by the extension step.

The specialization function h_s represents the relationship between elements of the original and the specialized behavior diagram.

Definition 16 (Specialized behavior diagrams and specialization function) *A labeled behavior diagram $B' = (S', T', F', L', l')$ that is a* specialization *of another labeled behavior diagram $B = (S, T, F, L, l)$ is a labeled behavior diagram with an associated* specialization function *h_s which has the following properties (we write h instead of h_s if it is clear from context that the specialization function is understood):*

$h : S' \cup T' \cup L' \to S \cup T \cup L \cup \{\varepsilon\}$ *(h is total)*
(1) $e \in S' \cup T' \Rightarrow h(e) \in S \cup T \cup \{\varepsilon\}$
(2) $x \in L' \Rightarrow h(x) \in L \cup \{\varepsilon\}$
(3) $x \in L \Rightarrow \exists x' \in L' : h(x') = x$
(4) $h(e) \in S' \cup T' \cup L' \Leftrightarrow h(e) = e$
(5) $h(\alpha') = \alpha$
(6) $s' \in S' \wedge t' \in T' \wedge h(s') \neq h(t') \wedge ((s', t') \in F' \vee (t', s') \in F')$

$\Rightarrow h(s') \in S \wedge h(t') \in T$

The specialization function is a total function that maps each element of B' either to an element to B (if the element has been inherited) or to ε (if the element has been added in B'). Further, the following properties are defined: (1) each activity and state is mapped to an activity, state, or ε; (2) each label is mapped to a label if the label has been inherited or to ε if the label has been added; (3) each label of the original behavior diagram has at least one corresponding label in the specialized behavior diagram; (4) all elements that are not mapped to ε and that are not refined are mapped to themselves; (5) the initial state of the specialized behavior diagram is mapped to the initial state of the original behavior diagram. (6) States (activities) have to be refined to subnets with states (or activities, respectively) as sources and sinks.

Example: The labeled behavior diagram of object type RESVT-SPEC (cf. Figure 2.20) is a specialization of the labeled behavior diagram of RESVT-L (cf. Figure 2.17).

The specialization function expresses modifications due to refinement and due to extension. Therefore, four differents kinds of modifications may be described by h_s :

Inheritance without change: An element e is inherited and not refined, i.e., $h(e) = e$.

Refinement: An element e in B is inherited and refined to a set of elements E; then $\forall e' \in E : h(e') = e$.

Extension: An element e' is added in B', there is no corresponding element in B; then $h(e') = \varepsilon$.

Elimination: If an activity or state e of B is omitted in B' (because of the rule of partial inheritance), then no element in B' is mapped to e by function h, i.e., $\nexists e' \in S' \cup T' : h(e') = e$. Notice that labels must not be omitted during specialization.

Example: Consider the behavior diagrams of RESVT-L and RESVT-SPEC: (1) Activity issue is inherited without change. (2) State toUse has been refined to a subnet consisting of states roomToRes, roomResvd, and serviceToRes as well as activities reserveRoom and reserveService. (3) Label c, activity returnCard, and several states (e.g., regIssued) have been added by extension. (4) Activities payByCheque and cashInCheque as well as state toCashIn have been omitted by elimination.

The *generalization of a life-cycle state* and the *generalization of a life-cycle occurrence* correspond to the previously defined restriction and abstraction of life-cycle states and life-cycle occurrences. During generalization, extended elements have to be omitted and refined elements have to be considered unrefined.

Definition 17 (Generalization of a life-cycle state) *The* generalization *of a life-cycle state σ' of an object type O' to object type O, written $\sigma'/_O$, is defined as $\sigma'/_O = \{(h(e'), h(x'))|(e', x') \in \sigma' \wedge h(e') \neq \varepsilon \wedge h(x') \neq \varepsilon\}$.*

Example: In the object life-cycle of RESVT-SPEC, $\sigma = \{(toPay, p), (roomToRes, rr), (serviceToRes, rs), (regIssued, c)\}$ is a valid life-cycle state. The generalization to object type RESVT-L yields $\gamma' = \{(toPay, p), (toUse, r)\}$.

Definition 18 (Generalization of a life-cycle occurrence) *The* generalization *of a life-cycle occurrence $\gamma' = [\sigma'_1, \ldots, \sigma'_n]$ of an object type O' to object type O, written $\gamma'/_O$, is defined as $\gamma'/_O = [\sigma'_1/_O, \ldots, \sigma'_n/_O]$.*

Definition 19 (Observation consistent specialization) *A labeled behavior diagram B' of object type O' is an* observation consistent specialization *of a labeled behavior diagram B of object type O if and only if for any possible life-cycle occurrence γ' of B' holds: $\gamma'/_O$ is a life-cycle occurrence of B.*

Example: The behavior diagram of RESVT-SPEC is an observation consistent specialization of the behavior diagram of RESVT-L.

There are no further rules necessary to check for observation consistent specialization. It holds that a labeled behavior diagram B' is an observation consistent specialization of a labeled behavior diagram B if and only if there exists a labeled behavior diagram $\hat{B}$, such that $\hat{B}$ is an observation consistent refinement of B and B' is an observation consistent extension of $\hat{B}$. If B and B' as well as the specialization function h_s are given, $\hat{B}$ is defined as $\hat{B} = (\hat{S}, \hat{T}, \hat{F}, \hat{L}, \hat{l})$, where $\hat{S} := \{s \in S'|h(s) \neq \varepsilon\}, \hat{T} := \{t \in T'|h(t) \neq \varepsilon\}, \hat{F} := F' \cap ((\hat{S} \times \hat{T}) \cup (\hat{T} \times \hat{S})), \hat{L} := \{x \in L'|h(x) \neq \varepsilon\}, \hat{l} := \{(f, X)|f \in \hat{F} \wedge X = l'(f) \cap \hat{L}\}$. The function h_r representing the refinement between $\hat{B}$ and B is defined as follows: $h_r : \hat{S} \cup \hat{T} \cup \hat{L} \to S \cup T \cup L$, where $h_r(e) := h_s(e)$.

Definition 20 (Observation consistent generalization, restriction, abstraction) *A labeled behavior diagram B is an* observation consistent generalization (restriction, or abstraction) *of B' if and only if B' is an observation consistent specialization (extension, or refinement, respectively) of B.*

2.4.5 Unlabeled Behavior Diagrams

Finally, a formal definition of *unlabeled behavior diagrams* is given as unlabeled behavior diagrams are refered to in the process of view integration, too. The difference between labeled and unlabeled behavior diagrams is only the fact that no labels are defined in the latter. Therefore the definitions of *start* and *completion of an activity* as well as the definition of the *life-cycle state* have to be modified.

Definition 21 (Unlabeled behavior diagram) *An* unlabeled behavior diagram (ULBD) $\breve{B}$ *of an object type, where* $\breve{B} = (S, T, F)$*, consists of a set of states* $S \neq \emptyset$*, a set of activities* $T \neq \emptyset$*,* $S \cap T = \emptyset$*, a set of arcs* $F \subseteq (S \times T) \cup (T \times S)$*, such that* $\forall t \in T : (\exists s \in S : (s,t) \in F) \wedge (\exists s \in S : (t,s) \in F)$ *and* $\forall s \in S : (\exists t \in T : (s,t) \in F) \vee (\exists t \in T : (t,s) \in F)$*. There is one* initial state α *such that* $\nexists (t, \alpha) \in F$ *and there exists a set of* final states $\Omega \subset S$ *such that* $\nexists (s,t) \in F : s \in \Omega$*.*

Example: Consider once more the behavior diagram of RESVT-L shown in Figure 2.17 and omit all labels. The resulting behavior diagram fulfills Definition 21.

Definition 22 (Unlabeled life-cycle state) *An* unlabeled life-cycle state σ *of an unlabeled behavior diagram is defined as* $\sigma \subset S \cup T$*. The initial life-cycle state is defined as* $\sigma_A := \{\alpha\}$*.*

Example: In the unlabeled version of the behavior diagram of Figure 2.17, $\sigma = \{toPay, toUse\}$ is a valid life-cycle state.

Definition 23 (Start and completion of an activity) *An activity* t *can be* started *on a life-cycle state* σ *if* ${}^\bullet t \subseteq \sigma$*, its start yields the life-cycle state* $\sigma' = (\sigma \setminus {}^\bullet t) \cup \{t\}$*. An activity* t *can be* completed *on a life-cycle state* σ *if* $t \in \sigma$*, its completion yields the life-cycle state* $\sigma' = (\sigma \setminus \{t\}) \cup t^\bullet$*.*

Example: The completion of activity issue in state $\sigma_i = \{issue\}$ yields $\sigma_{i+1} = \{toPay, toUse\}$.

The definition of *life-cycle occurrence* needs not be modified.

Chapter 3

Workflow Transparency

Contents

In modern organizations, entities treated in daily business that are needed to fulfill the business goals of an organization will be represented in an information system. We will refer to the objects defined in an information system that represent such entities as *business objects*. Corresponding to restrictions defined on the invocation of activities in object-oriented models by object life-cycles, activities may not be invoked on business objects in any arbitrary sequence, but certain rules have to be obeyed. Such rules are usually refered to as *business rules*; the process that represents the necessary activity invocations and restrictions due to business rules is usually called *business process*.

No standard definition for the term *business process* exists in the literature. The *Workflow Management Coalition* defines *business processes* in its document *Terminology & Glossary* as follows [Coa96]:

> "A set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within

> the context of an organisational structure defining functional roles and relationships."

In OBD, we used Petri nets to model object-life cycles. Several other approaches introduced in literature define business processes using dynamic models that correspond more or less to some kind of Petri nets (cf. e.g., [vdA97, GK96, OS96, AAH98]). [DO96] gives an overview of using Petri nets in applied computer science, particularly for business processes. [vdA96] named "three good reasons for using a Petri-net-based Workflow Management System".

In a previous work [BPS97b], we introduced a two-schema architecture for modeling the processing of business objects, where *business processes* and *workflows* are distinguished. This distinction will be relevant for the process of view integration, too, as discussed in Section 5.1 in more detail.

The two-schema architecture is motivated in Section 3.1. In Section 3.2, the model for representing workflows is briefly presented.

3.1 A Two-Schema Architecture for Business Rules

For modeling the processing of business objects in an information system, we capture the idea of Herbst and Knolmayer, who distinguish *external* and *internal* business rules [HK95]: *External* business rules are defined outside the organization and cannot or can only hardly be influenced by the organization. Such rules are either *natural facts,* which are valid at any time at any place (e.g., a room may only be reserved if it exists in the hotel), or *norms,* like laws etc., which may vary from time to time and which may be different in different countries or regions (e.g., the category of a room depends on the existence of certain marks of quality). *Internal* business rules are defined within the organization, they are only valid for the organization that defines them and may be changed by the organization at any time (e.g., every customer has to pay for the hotel bill at the time of check-out). We subsume external and internal business rules under the term *business policy.* We assume that external business rules are changed rarely, but internal business rules may change frequently due to market forces, internal decisions, continuing improvement of the process (continuing adaptation of processes is discussed in [AS95]), etc.

The *business goal* is defined to be a processing state that should be reached by every business object that is processed within an organization. A business object is processed successfully if it reaches one of the final processing states that are defined for the object. On the way from creation to one of the final processing states, the object has to be processed according to the external business rules and the actual internal business rules.

Usually, internal business rules must not contradict external business rules, but they may further restrict them. For example, a law may exist as an external business rule that restricts the price for a room in a hotel depending on the size and the category to a range between ATS 1.000,– and ATS 2.000,–. The management of a hotel decides as an internal business rule to charge for those rooms a price between ATS 1.700,– and ATS 1.900,–. The prices fixed by the hotel do not contradict the law as they are defined within the allowed range, but they further restrict the regulations defined by law.

We introduce a *two-schema architecture* for the representation of processes treating business objects: (1) the *business-process schema* represents the restrictions on the processing of business objects according to the *external* business rules, (2) the *workflow schema* represents the definition of the process according to *internal* business rules.

The business process and the workflow contain *activities* and specify *ordering constraints between activities*. The ordering of activities in the business process depends solely on the restrictions defined by external business rules, whereas the ordering of activities in the workflow depends on the current internal business rules. As the internal business rules have to obey all external business rules, all ordering constraints defined with the business process have to be represented in the workflow, too, and further ordering constraints may be defined.

The *business process* describes the business goal by defining the final processing states and all activities and restrictions that are necessary to lead a business object to one of the final processing states. The activities describe overall business-process activities, that define *what* has to be done with an object, which preconditions due to external business rules exist for the invocation of an activity and which information has to be provided to be able to perform the activity.

The *workflow* determines the actual flow of work within an organization. It defines *how* activities have to be performed in an organization by human or machine agents, e.g., which additional restrictions hold according to internal business rules, how information that is required by a business-process activity is obtained etc.

As the workflow schema reflects all rules necessary for actual processing of business objects, it is the complete instruction how agents have to do their work. But, as the business goal is only specified through the business process, the execution of business-process activities is sufficient for successful processing of business objects; the existence a workflow is no prerequisite for reaching the business goal. Therefore, the definition of the workflow could be ignored for some business objects. In practice, a workflow could be defined that instructs the employees who have to process certain business objects in daily business; the manager is allowed to omit this workflow and may invoke the business-process activities in an ad-hoc manner, nevertheless, he/she must not contradict the restrictions of the business process.

The two-schema architecture has the following advantages: (1) The separation of external and internal business rules and the representation in different schemas enables the management and the modeler to recognize at a glance which business rules may be modified or omitted by the organization and which business rules have to be obeyed at any time as they cannot be influenced by the organization. (2) If internal business rules are modified, the resulting change of the workflow schema may be *verified* with respect to the external business rules (and with the business-process schema). The representation of external and internal business rules in one common schema allows only *validation* of the whole set of rules with respect to the environment. (3) As internal business rules are valid within a smaller environment than external business rules and internal business rules may change more frequently than external business rules do, there may be several workflow schemas that obey to the same business-process schema.

In a previous work [BPS97b], we found out, that the two-schema architecture provides for some kind of *process independence* where the overall processing of business objects represented by the business process is modeled independently from the actual processing in an organization. This kind of independence is comparable to *data independence* in database systems where the ANSI SPARC architecture introduces logical and physical data independence, as well as to *distribution transparency* in distributed database systems [CP84].

We define *workflow transparency* as the quality of a workflow system to consider the specification of a business process independently from the workflow(s) implementing it.

For the two-schema architecture, we use the following models: The business-process schema is represented by the OBD model, where the structure of business objects is defined with business-object types by Object Diagrams, whereas the behavior of business objects, i.e., the business process, is defined by Behavior Diagrams. For modeling the workflow schema, we present a model in the following section.

3.2 The Workflow Model

A workflow describes the handling of business objects, whose overall behavior is defined in the business process, due to a set of internal business rules, where restrictions defined with the business process must not be violated. The workflow defines which business-process activities are invoked, in which order these activities are invoked, which preconditions have to be fulfilled, how actual parameter values for business-process activities are specified, and which additional activities that are not defined with the business process have to be executed by the actors.

A workflow does not define a representation of business objects of its own, but treats *business objects*, i.e., objects that are instantiated from the business-object types defined in the business-process schema. A business object is treated according to a workflow if it has been assigned to it. In the following, we will only consider business objects that are assigned to a considered workflow as a member.

A workflow consists of *worksteps,* which define the basic units of work and *boxes,* which are containers for business objects. In each workstep, some work is done on a business object; each workstep has one or several inboxes, which specify the set of business objects that may be treated in the workstep.

Similar to behavior diagrams, a workflow is represented by several diagrams: (1) The *workflow diagram* provides an overview of the workflow. (2) For each workstep, a *workstep specification diagram* represents the interface of the workstep. (3) Finally, for each workstep, a *workstep realization diagram* is defined to specify the realization of the workstep.

In this section, we will present an overview of our workflow model: Section 3.2.1 gives an overview of the basic elements of workflows. The subsequent sections deal with the diagrams used for representing workflows: Section 3.2.2 introduces *workflow diagrams*, Section 3.2.3 presents *workstep specification diagrams,* and Section 3.2.4 completes the description with *workstep realization diagrams.*

3.2.1 Basic Construction Elements

First of all, the basic construction elements of workflows are introduced: *boxes* and *worksteps.*

3.2.1.1 Boxes

Each box is specified by a *box specification,* which comprises (1) the name of a business-object type, (2) the name of a state that is defined with the business process of this business-object type, and (3) an optional boolean condition that refers to properties of instances of the business-object type and to some global information (like SYSDATE etc.). If no condition is specified, TRUE is defined as default. A box contains all business objects that fulfill the box specification, i.e., all objects that (1) are instances of the specified business-object type, (2) actually reside in the specified state, and (3) fulfill the condition.

A box is depicted by a rectangle with double-line borders. The business-object type and the state are shown within the rectangle, the optional boolean condition is written below the rectangle. Each box has a name, which has to be unique within the workflow; it is depicted within a small rectangle on top of the box symbol.

Example: We extend our running example (cf. Figures 2.1 and 2.2 shown in Chapter 2). Figure 3.1 shows the workflow for instances of business-object type RESERVATION.

The boxes are named R-1 to R-17. Each box holds instances of business-object type RESERVATION. Box R-1 holds all business objects that reside in state created; box R-3 holds all business objects that reside in state created and that fulfill the condition that one of the guests that are refered to in property guests of the reservation object has visited the hotel more than five times.

We allow to define box hierarchies, where one box b_1 always contains a subset of the objects of another box b_2. Box b_1 will be refered to as *subbox* of b_2, whereas b_2 is called *superbox* of b_1. Of course, the subbox relationship is transitive. Subboxes are defined for the same business-object type and the same state as their direct superbox but with a more restrictive condition. Thus, if a business object qualifies for a box, it qualifies for the superbox of this box, too. A subbox relationship is depicted by a line that connects the box symbols of the superbox and of the subbox and by a tringle whose top points to the superbox symbol.

Example: Box R-1 contains all instances of RESERVATION in state created, boxes R-2 and R-3 contain subsets of the objects in box R-1 due to a boolean condition.

3.2.1.2 Worksteps

Each workstep has a name, which is unique within the workflow, and has one primary input port and one primary output port. Further, a set of boxes is

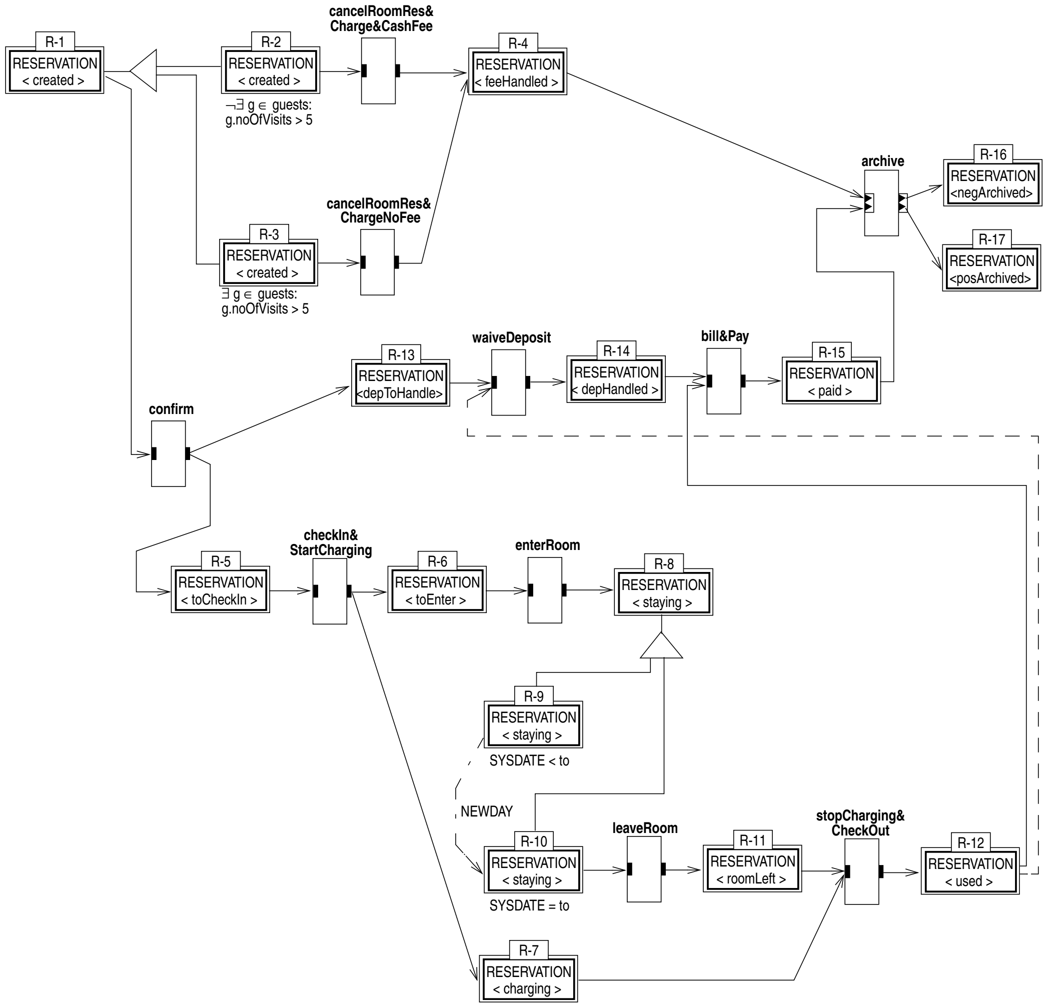


Figure 3.1: Workflow diagram for business-object type RESERVATION

assigned to the workstep's primary input port (and output port) as *inboxes* (and *outboxes,* respectively).

A workstep is depicted by a rectangle with the workstep's name at the top and two small black rectangles representing the primary input port and the primary output port. The inboxes (and outboxes) are connected to the primary input port (or primary output port, respectively) by a directed arc.

Example: In our example, workstep confirm has one inbox R-1 and two outboxes R-5 and R-13.

Similar to compound activities, which may have alternative pre- and post-states (cf. Section 2.2.4), a workstep may have *alternative in- and outboxes*: For each port, there may be several alternatives, each alternative consists of one or more boxes. If a port has alternative boxes, a triangle is depicted within the port symbol for each alternative.

Example: Workstep archive has alternative inboxes and outboxes. One alternative for the input port comprises box R-4, the other alternative comprises box R-15.

A workstep *without* alternative inboxes can be invoked on a business object if this business object is member of all of the workstep's inboxes. If *alternative inboxes* are defined, a workstep can be invoked on a business object if this business object is member of all boxes of exactly one alternative. We say that *a business object qualifies for a workstep* if this workstep can be invoked on the business object.

Notice, that worksteps are *not* invoked automatically on a business object, but each workstep has to be invoked by the human or machine agent that is assigned to the workstep.

Example: Workstep stopCharging&CheckOut can be invoked on a business object if the business object is a member of box R-7 and of box R-11. Workstep archive can be invoked on a business object that is member either of box R-4 *or* of box R-15.

Box membership is solely determined by the box specification. Hence a workstep does not explicitly remove the business object from the inboxes, but it may invoke a business-process activity on the business object such that this object changes its life-cycle state and thus does not qualify for the inbox any more.

Arcs connecting an inbox with a workstep's input port are distinguished in *transfer arcs* and *read arcs*. If an inbox is connected with the workstep by a transfer arc, the considered business object will change its state while treated in the workstep, such that it does not qualify for the inbox any more. If an inbox is connected with the workstep by a read arc, the treated business object will *not* stop qualifying for this box during its treatment in the workstep. Read arcs are often used to enforce sequential execution of activities that could be invoked in any order according to the business process. The state defined with the box that is connected with the workstep by a read arc is not left during execution of the workstep but enforces some synchronization only.

Example: Consider workstep waiveDeposit, which is connected with inbox R-13 by a transfer arc and with inbox R-12 by a read arc. The workstep can be invoked on a business object that qualifies for both inboxes. This business object will be modified in a way that it loses membership in box R-13, but does not lose membership in box R-12.

After execution of a workstep *without alternative outboxes* on a business object, the business object qualifies for all outboxes of the workstep. If the workstep has alternative outboxes, the handled business object will qualify for all outboxes of exactly one alternative.

Example: After having executed workstep confirm on a business object, the business object will qualify for boxes R-5 and R-13. After execution of workstep archive, the treated business object will qualify either for box R-16 or for box R-17.

3.2.2 Workflow Diagrams

A *workflow diagram* presents an overview of the workflow by depicting the worksteps with their primary input- and output ports, the boxes with their specification, and the arcs between workflows and boxes. Further details about worksteps are omitted, they will be specified in separate diagrams.

The workflow diagram specifies how business objects are treated to fulfill the restrictions defined by the business process. Business-process activities that could be executed in parallel may be invoked sequentially in the workflow; alternatives defined in the business process may be omitted in the workflow; business-process activities may be invoked in the workflow only if preconditions are fulfilled that are more restrictive than the preconditions defined in the business process.

Further, several worksteps may invoke the same business-process activity, and one workstep may contain several business-process activities (from which one or more are invoked for each business object).

Example: Consider the business process shown in Figure 2.2 and the workflow shown in Figure 3.1. The business process allows to execute activities concerning the use of a hotel room (i.e., activities checkIn, enterRoom, checkOut, etc.) and activities concerning the deposit (i.e., activities payDeposit, waiveDeposit) in parallel. The workflow restricts the processing: worksteps checkIn&startCharging, enterRoom, leaveRoom, and stopCharging&checkOut invoke the business-process activities concerning the use (match the names of the worksteps and the business-process activities to find out which worksteps invokes a business-process activity), whereas workstep waiveDeposit invokes business-process activity waiveDeposit. Due to the defined read-arc between box R-12 and workstep waiveDeposit, the parallelism is restricted to a sequence: Workstep waiveDeposit can only be invoked on objects that qualify for box R-12, i.e., on objects that reside in state used.

The alternative of requiring a deposit is omitted in the workflow as business process activity payDeposit is not used; instead, waiveDeposit is executed on all business objects.

Workstep checkIn&startCharging invokes two business-process activities, checkIn and startCharging. The business-process activity chargeNoFee may be invoked in workstep cancelRoomRes&Charge&CashFee and cancelRoomRes&ChargeNoFee (see later in this section).

The subbox hierarchy comprising boxes R-1, R-2, and R-3 is introduced to distinguish processing of reservations according to some property: if a customer who has already stayed in the hotel for several times is involved in a canceled reservation, no cancelation fee is charged. Otherwise, the decision whether a fee is charged is made within the workstep (i.e., workstep cancelRoomRes&Charge&CashFee). This internal rule is not specified with the business process, but it is defined only with the workflow.

A *transition arc* may be specified between two boxes b_1 *(source-box)* and b_2 *(target box)* if a business object that qualifies for b_1 and does not qualify for b_2 may at some time lose its membership in b_1 and become a member of b_2, although no workstep of the workflow diagram has been executed on this business object in between. The reason may be that the conditions defined with the boxes contain time-dependent predicates (see example below) or a business-process activity is invoked on the business object from outside the workflow (e.g., by an active rule that is specified with the business-process activity). A transition arc is a directed arc from box b_1 to b_2 with the name of the event that may cause the transition.

Example: Consider boxes R-9 and R-10, which both contain instances of RESERVATION in state staying: According to the boolean conditions specified with these boxes, an object resides in box R-9 if the current date is lower than the value of attribute to, whereas it resides in box R-10 on the day specified in to. This means that an object "moves" from source box R-9 to target box R-10 if SYSDATE switches to to, although no activity is performed on the object. The label NEWDAY indicates that the transfer of an object can occur if a new day starts, i.e., the value of the global variable SYSDATE is incremented by one day.

A workflow is *(temporarily or absolutely) blocked* for a business object if no workstep can be invoked on a business object. A workflow is *temporarily blocked* if some transition arcs are defined that may cause the workflow to get alive again; it is *absolutely blocked* if no such transition arc is defined or none of the defined transitions is applicable to the object, as the box specification of the target boxes cannot be fulfilled by the object at any time.

Example: Consider boxes R-8, R-9, and R-10. If a business object resides in boxes R-8 and R-9, no workstep can be started, i.e., it is temporarily blocked. On the departure day specified by property to of the reservation object, the business object moves from box R-9 to R-10, and workstep leaveRoom can be invoked.

Further, assume that a guest stays longer than planned. If the current date (SYSDATE) is later than the date specified by property to, the business object loses membership of boxes R-9 and R-10. In that case, it has to be processed in an ad-hoc manner as no internal business rule has been specified for this case; thus, the object is absolutely blocked.

3.2.3 Workstep Specification Diagrams

The *workstep specification diagram* depicts the interface of a workstep and those parts of the business process whose activities may be invoked during execution of the workstep.

The first part of the workstep specification contains the definition of the primary ports together with the associated boxes as defined in the workflow diagram. Further, secondary ports may be defined (depicted by white rectangles), by which additional information that is needed or produced by the workstep is specified. Secondary ports require (or produce) either *internal* or *external* information. Internal information is information held within the information system, e.g., another business object; external information is information that is not represented within the system, but that has to be provided from outside, e.g., a credit card, a passport, etc. This distinction is similar to the distinction of information flow and material flow in [KS86]. For *alternative* secondary ports, a similar notation as for alternative primary ports is used.

Secondary ports may be *optional*: if an input port is optional, the workstep can be executed even if no information is provided for this port; if an output port is optional, the workstep does not necessarily produce information into this output port. A port is declared as optional by putting the name of the port within brackets.

In difference to primary input ports, where the information is required as a prerequisite for invocation of the workstep, information provided by secondary input ports need not be specified before starting the workstep. But, it is sufficient that the required information is supplied "in time" when it is requested within the workstep. This rule is particularly useful for optional and alternative ports, as the agent who executes the workstep may decide during workstep execution whether information or which information, respectively, is requested via the secondary port.

The second part of the diagram specifies (1) which activities of the underlying business process may be invoked in the workstep, (2) the states between these activities, (3) the states used in the specification of the inboxes and outboxes, and (4) the arcs connecting these activities and states. Notice that this specification means that the specified activities are *statically* included in the workstep realization, but not all of them are necessarily executed for successful execution of the workstep.

A workstep may invoke compound activities. This fact is depicted by drawing a rectangle that represents the compound activity around the activities that are invoked by the compound activity.

Example: Figure 3.2 shows the workstep specification diagram of workstep cancelRoomRes&Charge&CashFee. There is one inbox connected with the primary input port and one outbox connected with the primary output port. As additional information, a means of payment is required via a secondary port. This port is optional as a means of payment needs to be provided via input port mp only in case that the clerk decides during execution of the workstep to charge a fee. The workstep produces in output port mp the consumed credit card, the change (if the fee is paid cash), or nothing (if no fee has been required). In any case, a receipt is handed to the customer.

The lower part of the figure shows the considered part of the business process: the workstep may invoke activities cancelRoomRes, chargeFee,

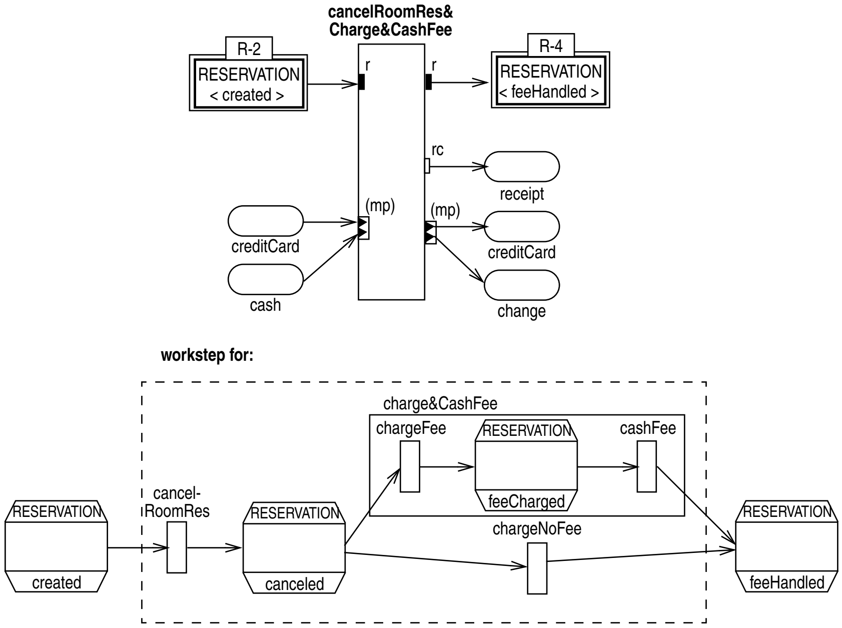


Figure 3.2: Workstep specification diagram cancelRoomRes&Charge&CashFee

cashFee, and chargeNoFee. Activities chargeFee and cashFee are invoked by the compound activity charge&CashFee.

3.2.4 Workstep Realization Diagrams

The *workstep realization diagram* shows how a workstep is carried out by a human or machine agent. A workstep may comprise different kinds of operations: invocations of business-process activities, database queries, calculations, and external operations (such as collecting cash). These operations can either be formally specified (such as invocations of activities or predefined database queries) or informally specified (such as external operations). We omit the description of workstep realization diagrams. They are described in more detail in [BPS97a].

Chapter 4

Overview of View Integration

Contents

View integration is the process of merging the views defined by some user-groups to an integrated conceptual schema, where the conceptual schema is made up by integrating all information defined in one or more views. In this work, we treat integration of *behavior* of objects, i.e., integration of object life-cycles and activity specifications defined in view schemas.

In this chapter, we first discuss how views are related to the conceptual schema in Section 4.1. Section 4.2 presents an overview of the integration process and its main steps. In Section 4.3, we will briefly discuss integration strategies.

4.1 Relationship Between Views and the Conceptual Schema

During our integration process, existing views are standardized to some normalized form and integrated to an integrated conceptual schema. We define a sequence of process steps that first change each view, and second integrate the views to a conceptual schema.

In this section, we discuss how a view schema can deviate from a conceptual schema. We will illustrate a process of stepwise derivation, where we start with the conceptual schema v_0 and define an intermediate view schema v_i from schema v_{i-1} in each step i. We will discuss how a schema v_i is related to v_{i-1} and how potential objects that are defined in v_i are related to objects in v_{i-1}.

Notice that the steps of view derivation shown in this section are never performed in this order in our integration approach. Yet, they are discussed here to illustrate which deviations of a view from the conceptual schema can be handled. For our integration process, this means that a view schema can lead to a conceptual schema only if it could have been derived from this conceptual schema by the steps of view derivation shown here.

We start with the conceptual schema v_0. The following steps are applied (for illustration of the whole process, the process steps as well as the relationships between object types and instances are summarized in Figure 4.1):

1. *View generalization:* During view generalization, the set of object types is not changed, but information about the instances is generalized. This means, that all objects in the conceptual schema are visible in the generalized view, too, but at a more generalized level. The result of view generalization are *generalized schemas*. Formally, v_1 is defined from the conceptual schema v_0. Each object type o_0 in v_0 corresponds to exactly one object type o_1 in v_1 and vice versa, where o_1 is an observation consistent generalization of o_0. Further, for each instance i_0 of o_0, there is exactly one instance i_1 of o_1 and vice versa.

2. *View limitation:* Objects considered in the generalized schema may not be visible in a view. If all instances of an object type are omitted in the view, the object type will be removed, too. During view limitation, the limited schema v_2 is defined from the generalized schema v_1 by omitting object types, whose instances are not considered in the view. Formally, each object type o_2 in v_2 corresponds to exactly one object type o_1 in v_1, whereas each object type o_1 in v_1 corresponds to at most one object type o_2 in v_2. If an object type of v_1 is defined in v_2, too, then all of its instances are represented in v_2. This means, if i_1 is an instance of o_1 and o_1 corresponds to o_2, then there is exactly one instance i_2 in o_2, such that i_1 corresponds to i_2 and vice versa.

3. *Intra-type restructuring:* In this step, definition of behavior within the object types is restructured, where restructuring must be information-

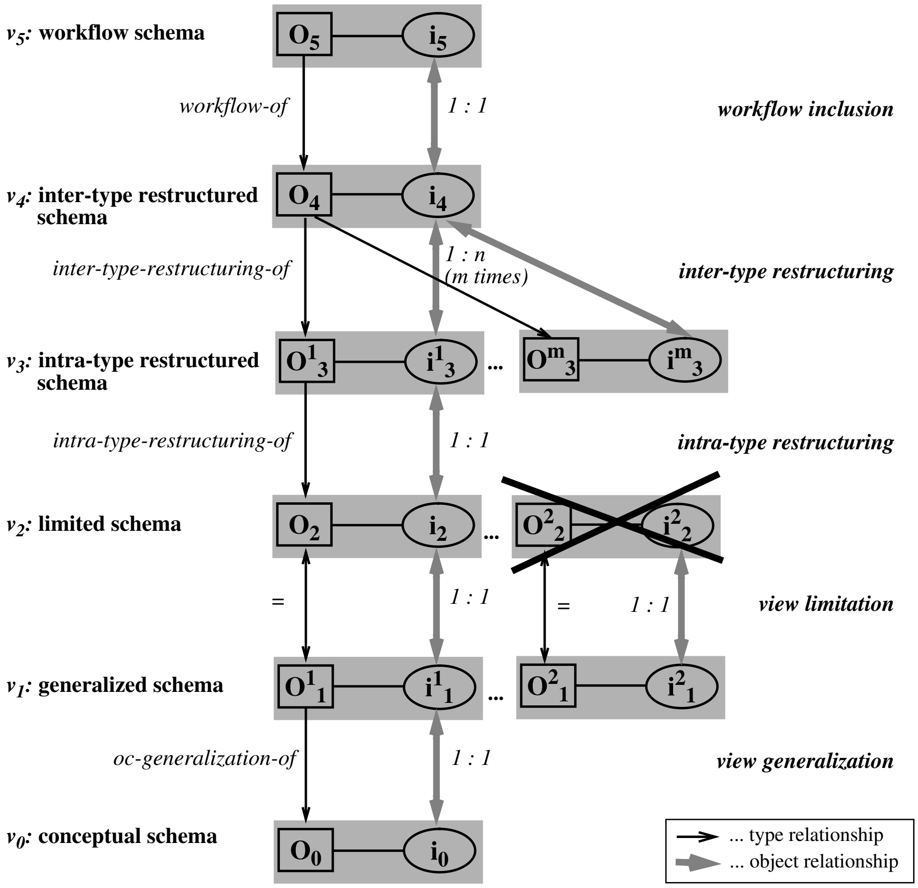


Figure 4.1: Process of view definition

capacity preserving, i.e., no information is added or removed. Any object type o_2 in the limited schema v_2 may be restructured resulting in object type o_3 in the intra-type restructured schema v_3. There is a one-to-one mapping between the object types in v_2 and the object types in v_3 and the instances of corresponding object types o_2 and o_3.

4. *Inter-type restructuring:* Definitions of object types can be merged, which means that *a set of* object types in the intra-type restructured schema v_3 is replaced by *one* object type in the inter-type restructured schema v_4. In Figure 4.1, this fact is indicated by a type relationship *inter-type-restructuring-of* between one object type o_4 and a set of object types $o_3^1, \ldots, o_3^m$. Inter-type restructuring must be information capacity preserving, too. We consider three kinds of inter-type restructuring:

    (a) *Merging types within the same type hierarchy:* A part of the type hierarchy is replaced by a single object type. Thus, a set of object types $o_3^1, \ldots, o_3^m$ is replaced by one object type o_4, where each direct instance i_3 of one of the object types $o_3^1, \ldots, o_3^m$ corresponds to one instance of i_4 and vice versa.

    (b) *Merging of different, related object types:* Object types $o_3^1, \ldots, o_3^m$ that are related to each other by a relationship are merged to a single object type o_4. Then, a set of instances of $o_3^1, \ldots, o_3^m$ corresponds to one single instance i_4 of o_4 and vice versa. In our integration approach, we allow only merging of object types that are related to each other by component or role relationships. Handling general relationships is not treated in this work. Further, we will focus on merging of object types where the *hasComponent-* or *hasRole* relationship is *single-valued*; otherwise, the structure and behavior of *a set of* component or role objects had to be represented in the object type that results form merging. Problems that arise in this situation are further discussed in Section 5.2.1.2.

    (c) *Replacing association types by direct relationships:* Relationships that are represented by an association type can be replaced by a direct relationship between the related object types. If o_3^1 and o_3^2 are related to each other, where this relationship is represented by an association type o_3^3, the former are defined in the inter-type restructured schema by object types o_4^1 and o_4^2, respectively, whereas object type o_3^3 is not defined in v_4. The structure and behavior defined with o_3^3 are defined either with o_4^1 or o_4^2. Each instance of o_3^1 and o_3^2 is represented by an instance of o_4^1 and o_4^2, respectively, whereas the properties of an association object i_3^3 are held in v_4 either by an instance of o_4^1 or o_4^2, depending on which one of these object types includes the properties of o_3^3. We assume that the object type o_4^i that includes the properties of i_3^3 will be related to the other object type o_4^j by a single-valued direct relationship. Otherwise, this o_4^i had to represent structure and behavior of a set of association objects (which causes similar problems as multi-valued *hasComponent*- and *hasRole* relationships mentioned before).

5. *Workflow inclusion:* According to our approach of *workflow transparency*, business processes and workflows are defined separately. If workflows and business processes are integrated in one schema, the inter-type restructured business process schema v_4 is enhanced by workflow aspects resulting in the workflow schema v_5. Each object type o_5 in v_5 represents the workflow for exactly one object type o_4 in v_4 and each o_4 corresponds to exactly one o_5. Each instance i_5 of o_5, which represents an object that behaves according to a workflow, is related to exactly one instance i_4 of o_4, which represents behavior according to the business process, and vice versa.

4.2 Integration Steps

Each given view will miss some information that should be defined in the conceptual schema. Therefore, we require that several views are defined, where all information that should be defined in the conceptual schema is defined in at least one view; the conceptual schema results from integration of the views, where information that is defined in one view and missing in the other view is defined in the integrated schema.

The integration steps that have to be applied on a view schema to obtain an integrated schema are presented in this section. The process with all the steps and the intermediate schemas is once more illustrated in Figure 4.2. Each integration step changes the view schemas to obtain the integrated schema at the end of the process; each step corresponds to a step for view definition introduced in Section 4.1, where the steps for view definition were applied on a schema during derivation of a view schema from a conceptual schema, i.e., they were applied in the reverse order.

We define the following overall integration steps:

1. *Extraction of business processes:* If a *workflow* is defined in a view schema, the business process has to be extracted. This step corresponds to *workflow inclusion*.

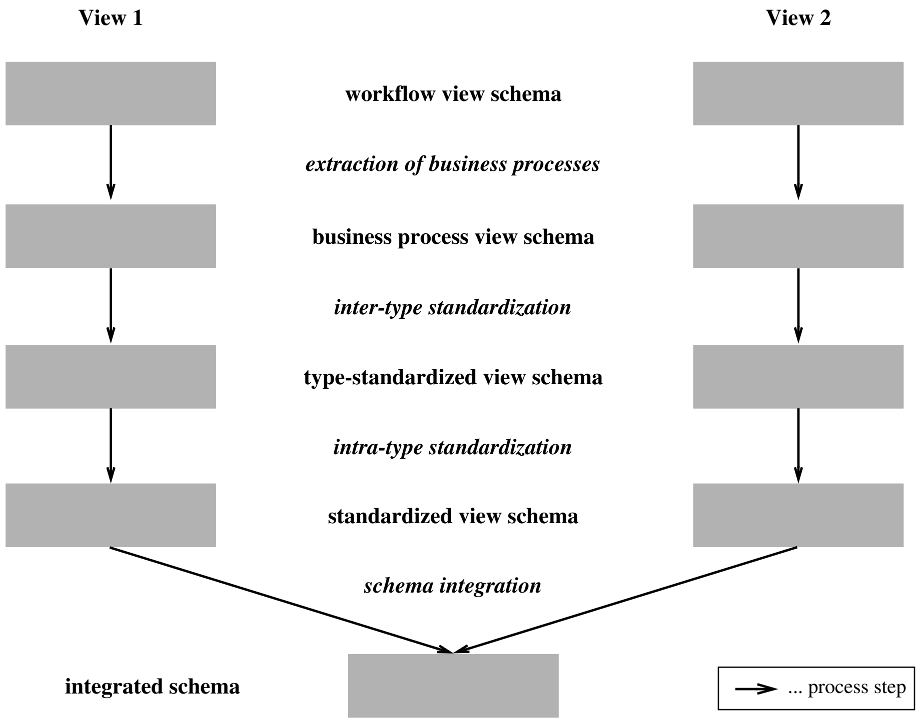


Figure 4.2: Overview of view integration

2. *Inter-type standardization:* An object type can be replaced by a set of object types to separate different concepts. This step corresponds to *inter-type restructuring.*

3. *Intra-type standardization:* Behavior within each object type is restructured without changing the overall behavior. This step corresponds to *intra-type restructuring.*

4. *Schema integration:* During schema integration, the integrated schema is defined from the views: (1) The type hierarchies are defined with all object types that are defined in the views. Object types that are defined in one view, but missing in the other view, are included in the integrated schema; thus, this step corresponds to *view limitation.* (2) The behavior of object types is defined where information defined at different levels of specialization in the views is integrated at the most specialized level. This step corresponds to *view generalization.* In difference to the other steps, schema integration does not change the view schemas, but forms a new schema, the integrated schema.

From this first description, we can see, that steps 1 to 3 can be executed within each schema without considering the other schemas to be integrated; in step 4, a new schema is built, which constitutes the integrated schema. We will refer to steps 1 and 2 as *view preparation* and present them in a chapter of its own (cf. Chapter 5). Steps 3 and 4 will be considered for object life-cycles in Chapter 6 and for activity specifications in Chapter 7.

In many publications on schema integration, the step *schema translation* is introduced to translate the given schemas into a canonical model. In this work, we omit the step of schema translation and assume that our canonical model are *Object/Behavior Diagrams* (OBD) and all view schemas are defined in OBD.

We will assume that *two* views must be integrated during integration. This assumption is further discussed in Section 4.3. In the following, the steps of the integration process are once more surveyed:

4.2.1 Extraction of Business Processes

A view schema may not only define the *business process*, but include information about the workflow within an enterprise. This situation occurs if users define the process as they perceive and execute it in the current organization; hence they include workflow-specific information, which reflects internal business rules. The workflow aspects must be removed from the schema, such that the pure business process remains. The result of this step is a *business process view schema*, where each object type of the *workflow view schema* is changed to one object type in the business process view schema. Extraction of business processes is discussed in Section 5.1.

4.2.2 Inter-type Standardization

The goal of inter-type standardization is to define a standardized schema, where different concepts are represented by different object types. In this phase, object types are split to a set of object types if they comprise different concepts.

There are three forms of inter-type standardization:

1. *Splitting entities and contexts:* We distinguish splitting of entities and of contexts: (1) An object type can be split into a set of object types if it represents different kinds of real-world entities (splitting entities). (2) An object type can be split into a set of object types if it represents real-world entities in different contexts (splitting contexts).

   Example: In our example, object types RESERVATION and BILL are defined, where the latter is a component type of the former. In a view, reservations and bills may be represented by one single object type. Further, two object types PERSON and STAFF are defined, where the latter is a role type of the former (cf. Figure 2.1). Each person of the real world is represented by an instance of PERSON; if a person is member of a hotel's staff, it is represented by an instance of STAFF, as well. In a view, object types PERSON and STAFF may not be distinguished but there may be only one object type that jointly represents people in their role as person and as staff member. Such object types are split during inter-type standardization.

2. *Explicit definition of relationships:* Instances of object types can be related by a general relationship. This relationship can be defined by *direct relating*, where each involved instance has a property refering to the related objects, or it can be defined by explicitly representing the relationship by an association type. If a direct relationship is defined, the properties of the relationship have to be defined in one of the related object types. During splitting object types, an association type is introduced instead of direct relationships.

   Example: The relationship between reservations and guests (i.e., between object type RESERVATION and GUEST) is represented by an assocation type SERVICE. A view could omit object type SERVICE and define a direct relationship between RESERVATION and GUEST.

In both case (1) and case (2), an object type o of the business process view schema is split to several object types O', where each instance i of the original object type corresponds to a set of instances I' of the new object types. Correspondences between the activities of o and the activities in the object types of O' have to be defined. As a correctness criterion we require, that the sequences of activity invocations in o and O' correspond to each other: For each sequence of activity invocations on i, there must be a corresponding sequence of activity invocations on I' and vice versa. Sequences of activity invocations are corresponding in this sense if the following condition holds: if a sequence of activities of o can be invoked on i, then the corresponding activities in O' can be invoked on instances I' in the same order and vice versa.

We will find out (cf. Section 5.2) that splitting of entities and contexts is mainly useful if *component* or *role* relationships are defined between the object types that result from splitting, where the *hasComponent*- and *hasRole* relationships are single-valued. Further, for explicit definition of relationships, the properties of the relationship itself will have to be defined with that object type that refers to the other object type by a *single-valued* direct relationship.

3. *Subtyping:* An object type o in the view schema is split to a type hierarchy consisting of one object type o' and several subtypes O'' of o'. Each instance of o is represented by exactly one instance of one $o'' \in O''$ (notice that only leaf nodes of type hierarchies can be instantiated according to the abstract superclass rule). Corresponding instances of o and an o'' must behave the same.

   Example: There is a type hierarchy defined in our example (cf. Figure 2.8) where object type SERVICE has two subtypes STD-SERVICE and VIP-SERVICE, which represent different kinds of services. A view could only define one object type SERVICE that represents all different kinds of services and that is standardized by replacing it by a type hierarchy.

Inter-type standardization is treated in detail in Section 5.2. Inter-type standardization is defined in a comparable form as *schema standardization* for structure of object types in [Joh93, Joh94]. In [KFM+96], similar operations are subsumed under the term *augmentation*. Johannesson [Joh94] points out that view schemas should be transformed during schema standardization such that afterwards, there are only simple correspondences between object types of the view schemas (i.e., one object type of view schema 1 corresponds to at most one object type of view schema 2 and vice versa).

During inter-type standardization, a *type-standardized view schema* is defined. In the type-standardized schema, we assume that different entities or entities in different contexts are represented by different object types, relationships that constitute objects of their own are represented by association types, and different but similar kinds of objects are represented by different subtypes of an object type in a type hierarchy.

4.2.3 Correspondences between Object Types

The goal of inter-type standardization is to define object types in a way that simple 1 : 1 correspondences between object types in the views can be determined such that the remaining integration process must only consider pairs of comparable object types. Therefore, object types in the views must be compared, where the following kinds of correspondences can be determined.

For each object type o in one of the two type-standardized view schemas one of the following statements must hold:

1. *Equivalence:* Object type o in one view represents the same intension and extension as an object type o' in the other view. This means, both object types are views on the same object type of the integrated schema. We say that o and o' are *equivalent*.

2. *Similarity:* Object type o in one view and object type o' in the other view are *similar* if all of the following conditions hold: (1) Object types o and o' represent different specializations of an object type that is not defined in either view, i.e., their intensions are similar in the sense that they should have a common supertype that represents the common parts of their structure and behavior. (2) Object type o and object type o' have disjoint extensions. (3) Object type o and o' are root types within their view, i.e., there is no object type o'' in the respective view such that o'' is a supertype of o (or o', respectively).

   Similarity is used to represent the fact that type hierarchies that are defined in different views consider no common extensions but they have common properties that should be reflected by a common supertype in the integrated schema. Object types o and o' are the root types of the specialization hierarchies in the views.

3. *Uniqueness:* Object type o is unique if it does not correspond to any object type in the other view by equivalence or similarity.

The type hierarchies defined in the views must fulfill correctness criteria so that they can be integrated to a type hierarchy in the integrated schema. The following correctness criteria state how the object types in the type hierarchies of the views must correspond to each other. These criteria may not be fulfilled for one or more of the following reasons: (1) The system integrator determined wrong correspondences. He/she must review and correct the determined correspondences. (2) The system integrator has not standardized the view schemas correctly, i.e., further standardization operations must be applied. (3) The view schemas contain contradicting information or define schemas in a way that they cannot be transformed by the proposed process of inter-type standardization. Case (3) will be considered once more later in this section.

The following correctness criteria must hold between object types and type hierarchies in the views:

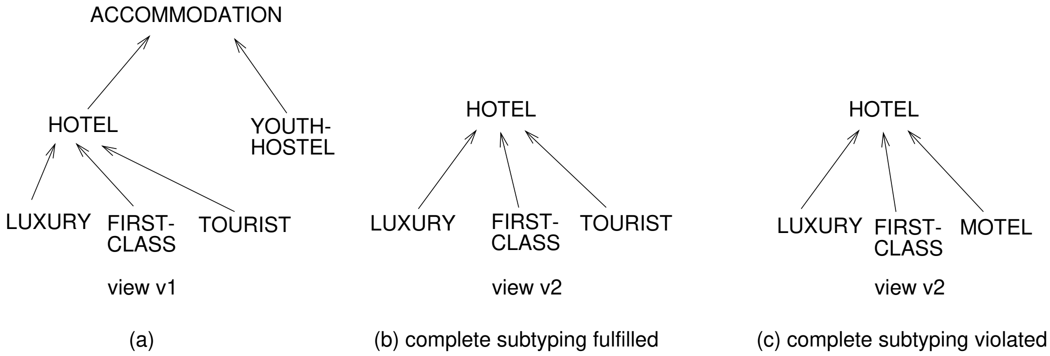


Figure 4.3: Example: complete subtyping

1. *Consistency:* Type hierarchies in the views are not contradictory, i.e., if o_1 and o_2 are equivalent and o'_1 and o'_2 are equivalent then o'_1 is a direct subtype of o_1 if and only if o'_2 is a direct subtype of o_2.

   Example: Suppose that there are two views v_1 and v_2; view v_1 defines two object types HOTEL-1 and MOTEL-1, whereas view v_2 defines HOTEL-2 and MOTEL-2. In both views, MOTEL-i is a subtype of HOTEL-i. The system integrator detects an equivalence between HOTEL-1 and HOTEL-2 as well as between MOTEL-1 and MOTEL-2, which obeys to the criterion of consistency. These correspondences would lead to a violation of consistency if MOTEL-2 were a supertype of HOTEL-2.

2. *Complete subtyping:* If an object type o_1 in one view is equivalent to object type o_2 in the other view and o'_1 is a subtype of o_1, there must be a subtype o'_2 of o_2 that is equivalent to o'_1. This means that if the root type of a specialization hierarchy or of a subtree of a specialization hierarchy is equivalent to an object type in the other view, then there must be equivalences between all of their subtypes.

   Example: Consider Figure 4.3: The type hierarchy of view v_1 is shown in part (a). If view v_2 is defined as shown in part (b), complete subtyping is obeyed. If view v_2 is defined as shown in part (c), the criterion is violated.

3. *Supertype compliance:* If o_1 is equivalent to o_2, then at least one of the following conditions must hold: (1) o_1 has no supertype, (2) o_2 has no supertype, (3) the supertype of o_1 is equivalent with the supertype of o_2. This means that the specialization hierarchy of one view is equal

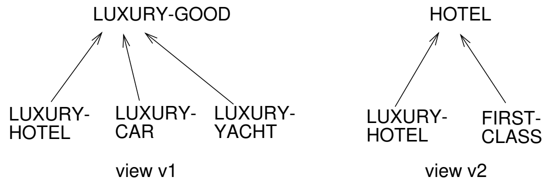


Figure 4.4: Example: violated supertype compliance

to or is a sub-tree of the specialization hierarchy defined in the other view.

Example: Consider once more Figure 4.3: View v_1 and view v_2 in part (a) and part (b), respectively, fulfill supertype compliance; e.g., the supertypes of object type LUXURY in v_1 and v_2 are equivalent; object type HOTEL has a supertype in view v_1, but no supertype in v_2.

In Figure 4.4, supertype compliance is violated as object type LUXURY-HOTEL has supertype LUXURY-GOOD in one view and supertype HOTEL in the other view.

4. *Disjointness of unique object types:* If an object type o_1 is unique, one of the following conditions must hold: (1) Its extension is disjoint from the extensions of all object types in the other view. (2) If the extension of o_1 overlaps with the extension of an object type o_2 in the other view, there must be a subtype o'_1 of o_1 that is equivalent with o_2. In case (1), object type o_1 is only defined in one view and there are no object types in the other view that consider the objects represented by o_1. In case (2), object type o_1 is defined in only one view but its subtypes are represented in the other view, too.

Example: Consider Figure 4.3 (a) and (b): Object type ACCOMMODATION is unique although it has indirect instances that are considered in the other view, too. But there is a subtype HOTEL of ACCOMMODATION that is equivalent to HOTEL in view v_2 and that covers exactly those objects that are common in v_1 and v_2.

Suppose that there is an object type HIGH_STANDARD_HOTEL defined in view v_1 and an object type LOW_STANDARD_HOTEL defined in view v_2, where the former considers hotels of category *5-stars, 4-stars,* and *3-stars,* whereas the latter considers hotels of category *3-stars, 2-stars,* and *1-star*. Both object types are unique as they are neither equivalent (as they do not consider the same extension) nor similar (as their extensions are not disjoint). Yet, they cannot be defined in the integrated schema as hotels of category 3-stars had to be defined as instances of both object types.

The above correctness criteria are based on an object-oriented model like OBD, which supports only single inheritance at the moment. If a view schema cannot be standardized in a way that the correctness criteria are fulfilled, the view schemas cannot be integrated. Models that support multiple inheritance allow to integrate type hierarchies that do not fulfill the above correctness criteria. We follow the model of OBD in this work and omit multiple inheritance. This drawback will not severely restrict the core topics of this work.

Example: The type hierarchies shown in Figure 4.4 can be integrated if multiple inheritance is supported. Then, a common supertype for LUXURY-GOOD and LUXURY-HOTEL could be introduced and object type LUXURY-HOTEL could be defined as a direct subtype of both LUXURY-GOOD *and* HOTEL.

If two view schemas cannot be integrated, at least one of the view schemas must be changed. In practice, this change will be performed by the system integrator, possibly together with the users who modeled the respective views. If view schemas define contradicting information (e.g., one user defines hotels as special kind of motels, whereas another user argues that motels are special kinds of hotels), redefinition of views may be a non-trivial task where users may have to meet and to discuss design.

Contradictions between views can in some cases be resolved by small changes in the view schemas, which may be done by the system integrator without involving the users, e.g., redundant information can be omitted if it causes violations.

Example: Consider once more the example shown in Figure 4.3: The type hierarchies of view v_1 and v_2 in part (a) and part (c) of the figure cannot be integrated as they violate the criterion of complete subtyping. Suppose that the integrated schema should define object type HOTEL with four subtypes LUXURY, FIRST-CLASS, TOURIST, and MOTEL. The system integrator will change the views by introducing an object type MOTEL as a subtype of HOTEL in view v_1, where MOTEL has the same structure and behavior as HOTEL, and by introducing object type TOURIST as a subtype of HOTEL in view v_2. Notice, that these changes are no regular changes according to the integration process, but a change of the view schemas beyond the integration process.

4.2.4 Intra-type Standardization

The definition of behavior within an object type can be restructured. This restructuring does not add or remove information, such that the overall behavior of instances of this object type does not change. During intra-type standardization, the object life-cycles and the activity specifications are normalized similarly as object types are normalized during *inter-type standardization*. The result of this step is a *standardized view schema*. Intra-type standardization of object life-cycles and of activity specifications are treated in Section 6.4.2 and Section 7.4.2, respectively.

4.2.5 Schema Integration

During schema integration, the standardized view schemas are integrated to an integrated schema. We use different integration approaches for the following correspondences between object types:

Equivalent object types: If o_1 and o_2 are equivalent, they are represented by one single object type o in the integrated schema. We require that o is an observation consistent specialization of o_1 and an observation consistent specialization of o_2, as motivated in the introduction and further discussed in Section 6.3.1. For integration, we will define *integration by specialization* for object life-cycles and activity specifications later in this work (cf. Chapter 6.3.1 and Chapter 7.1.2.2, respectively).

Notice that if two object types o_1 and o_2 are considered equivalent, there must be an equivalent object type for any subtype of o_1 and any subtype of o_2. If o_1' is a subtype of o_1 and o_2' is a subtype of o_2, where o_1 is equivalent to o_2 and o_1' is equivalent to o_2', then there will be two object types o and o' in the integrated schema, where o (or o') results from integrating o_1 and o_2 (or o_1' and o_2', respectively) and o' is a subtype of o. In our approach, o and o' will be defined in different integration steps independently from each other. The system integrator has to check whether the specialization hierarchy is defined correctly in the integrated schema, i.e., whether o' is an observation consistent specialization of o. The necessity to check this condition is not considered as a severe restriction because it will be fulfilled if the system integrator integrates the object life-cycles of o_1 and o_2 and of o_1' and o_2' in a consistent way.

Similar object types: Similar object types need not be integrated by specialization as no equivalent object types exist for them. They are taken into the integrated schema without being integrated with another object type. Yet, as similar object types o' and o'' shall have a common supertype o in the integrated schema with the common properties of o' and o'', a new object type o has to be defined in the integrated schema. The behavior of o is defined during *integration by generalization* (cf. Section 6.3.2 for object life-cycles and Section 7.1.2.1 for activity specifications).

Unique object types: Unique object types need not be integrated with any other type, their definition is taken into the integrated schema without integration.

Generalization of elements. The integrated schema must be a correct schema according to the rules of OBD, which reflect the fundamental rules in object-oriented data models. For definition of elements in supertypes and subtypes, we require that the rule of original definition of elements is obeyed: If there is an object type o with subtypes o_1 and o_2, and there is an element (activity, state, label, or port) e_1 in o_1 and an element e_2 in o_2, then e_1 and e_2 correspond to each other if and only if there is an element e in o, such that e_1 results from inheriting e in o_1 and e_2 results from inheriting e in o_2. Elements in object types that are not inherited from a common supertype are considered different.

If object types are defined in the integrated schema by integrating pairs of equivalent object types in the views, a schema may result, where e_1 in o_1 and e_2 in o_2 are considered to correspond to each other but there is no element e defined with the common supertype o such that the above rule is not fulfilled.

During *generalization of elements,* an element is defined with an object type if all its subtypes define this element. The decision whether elements in different types correspond to each other cannot be automated such that correspondences will be determined by the system integrator. We will discuss generalization of elements in more detail for object life-cycles in Section 6.5.2 and for activity specifications in Section 7.4.4.

4.3 Integration Strategies

If a set of n views is given, several approaches to integrate them are discussed in literature [BLN86], namely *binary* and *n-ary* integration-processing strategies: The former assumes that exactly *two* schemas are integrated in each step: These schemas may be view schemas or intermediate schemas result-

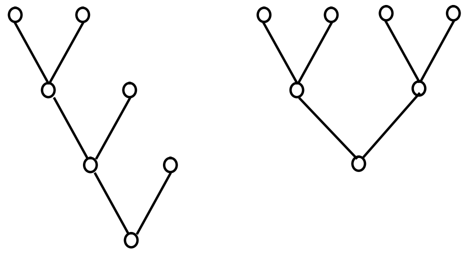


Figure 4.5: Binary integration strategies [BLN86]

ing from a previous integration step. The latter allow to integrate several schemas (view schemas or intermediate schemas) in one step. The binary integration strategies are distinguished in two types (cf. Figure 4.5): the *ladder* strategy, which starts with integration of two view schemas in the first integration step, and continues by integrating one view schema with the intermediate schema in each further integration step (cf. part (a) of the figure), and the *balanced* strategy, which integrates the schemas symmetrically (cf. part (b) of the figure).

We will describe our integration process using a binary integration strategy. We do not fix whether the ladder or the balanced approach has to be used; this decision will be taken by the system integrator. In practical use, views will be defined by departments or similar sub-units within an organization; probably, each view is defined to represent the view of a certain organizational unit on the database. Integration may follow the organizational hierarchy, i.e., the view of a superordinate organizational unit will result from integrating the views of the subordinate units; thus a more or less balanced strategy may be chosen.

Although we define and present the whole integration process based on the assumption that a binary strategy is used, we assume that the approach can easily be extended to an n-ary approach. We will review the integration steps presented before and consider how the selection of the integration strategy influences them:

The steps of extraction of business processes from workflows, inter-type standardization, and intra-type standardization cannot be influenced as these steps modify each view separately. In these steps, no schemas are integrated such that it does not matter whether a binary or n-ary strategy is selected.

For the process step of integration, we distinguish equivalent object types, similar object types, and unique object types:

Equivalent object types: Equivalent object types are integrated by specialization. If a binary strategy is chosen and there is a set of more than two equivalent object types, then two of them are integrated and the result is integrated with the third equivalent object type, etc. In each step, the definition of the object type becomes more specialized, as new refinements are defined, new extensions are added, and alternatives that do not appear in each object type are omitted. If an $n-$ary strategy is used, correspondences between n object life-cycles can be determined and all refinements, extensions, and omitted alternatives of the integrated object life-cycle can be determined at once. Yet, in any case, it must hold that the integrated schema is an observation consistent specialization of the view schemas, no matter whether it is defined in one $n-$ary integration step or in several binary steps.

Similar object types: Similar object types are integrated by generalization, i.e., the object types are not changed, but a new common supertype is defined.

Suppose that there are three views v_1, v_2 and v_3 where each view defines one object type o_1, o_2, and o_3, respectively. Further, suppose that all these object types are similar, i.e., the intuitively correct conceptual schema comprises o_1, o_2, and o_3 with one common supertype o. If a binary integration approach is used, these views are integrated in two phases: (1) views v_1 and v_2 are integrated to v_{12} and (2) the intermediate schema v_{12} is integrated with v_3 to v_{123}. In phase (1), object types o_1 and o_2 are considered similar, such that a common supertype o_{12} is introduced. In phase (2), object types o_{12} and o_3 are considered similar; thus, a new object type o_{123} is defined that is a supertype of o_{12} and o_3. Suppose that we change the order of integration, i.e., views v_2 and v_3 are integrated to an intermediate schema first and then, v_1 is integrated with the intermediate schema. The resulting type hierarchy will be different. In a n-ary approach, o_1, o_2, and o_3 were integrated by generalization in one step, such only one common supertype were defined.

Unique object types: Unique object types are integrated into the conceptual schema without being integrated with any other object type. Therefore, it is not relevant *when* they are taken into the integrated schema.

We conclude that the n-ary strategy is more powerful than the binary one (which is intuitively clear as the latter is a special case of the former). Nevertheless, we will follow the binary approach in this work, but we assume that this approach can easily be extended to an n-ary approach. The only change will be to extend integration by specialization and integration by generalization to be able to handle more than two views:

If n views $v_1, \ldots, v_n$ are given and each view defines an object type o_i, where all o_i are equivalent, integration by specialization must be applied according to the following rules: (1) Each activity, state, or label is defined at the most refined level of granularity in only one view of the object type, whereas it is defined abstracted in all other views. It is integrated at the most refined level of granularity. (2) Each view-specific extension is defined in the integrated schema. (3) Alternatives can only be defined in the integrated schema if they are defined in all views (cf. Section 6.3.1 for details).

Similarly, for integration by generalization of object types $o_1, \ldots, o_n$, where a common supertype o is defined, the following rules apply: (1) Each activity, state, or label is defined at the lowest level of granularity. (2) A label is only defined in o if it is defined in every o_i. (3) All view-specific alternatives are integrated.

Chapter 5

View Preparation

Contents

We will discuss the steps of view preparation in this chapter. View preparation consists of *extraction of business processes from workflows* and *inter-type standardization*. After these steps, only aspects of business processes are defined in the view schemas and correct type hierarchies are formed according to the criteria defined in Section 4.2.3, such that the system integrator can focus on the inner definitions of object types, i.e., further integration is based on comparing and integrating *one* object type from each schema. We will present *extraction of business processes* in Section 5.1 and *inter-type standardization* in Section 5.2.

5.1 Extraction of Business Processes

The view schema may consider workflow aspects. As workflow aspects must not be defined in the business-process schema due to workflow transparency, they must be removed from the view schemas.

Workflow aspects may be included in the definition of the object life-cycle and in the activity specifications. We will discuss extraction of the business process from the workflow for object life-cycles and for activity specifications separately in this section.

Workflow aspects in object life-cycles occur if a user defines a workflow diagram instead of an object life-cycle representing the business process (cf. Section 3.2.2). Workflow aspects in activities occur if worksteps instead of business-process activities are defined in the views (cf. the introduction of *workstep specification diagrams* in Section 3.2.3 and *workstep realization diagrams* in Section 3.2.4).

The result of this step is a business-process schema with object life-cycles and activity specifications that include only aspects of business processes, but no aspects of workflows.

5.1.1 Extraction of Object Life-Cycles

If an object life-cycle in a view does not represent a business process, but considers workflow aspects, the workflow aspects have to be omitted from the schema. Workflows may define the following constraints and workflow-specific behavior:

Additional order dependencies: Activities that could be invoked in any order in the business process can be sequentialized in the workflow. If an object life-cycle in a view enforces the sequential execution of activities, although they could be executed in any order according to external business rules, the sequence has to be replaced by parallelism.

Omitted alternatives: Alternatives that are allowed according to external business rules may be omitted in the workflow. Such alternatives have to be inserted into the object life-cycle.

Defining new alternatives: Workflows may introduce new alternatives as they may distinguish objects that reside in the same state by additional conditions defined with boxes (cf. boxes R-2 and R-3 defined in the example shown in Figure 3.1). The worksteps that are executed on objects in these boxes may invoke the same activities of the business process. Such worksteps can be merged in this integration step such that the extracted business process does not contain these alternatives.

Definition of compound activities: One workstep can invoke several business-process activities. If an activity in a view schema represents several activities of a business process, the system integrator may split the activity.

Definition of workflow-specific activities: The workflow defines how the work is performed within the organization. Worksteps invoke business-process activities and describe which activities have to be performed in addition to the business-process activities, e.g., defining parameters for the business-process activities, performing some work on external documents, e.g., cashing money, making out a receipt, etc.) If these additional activities are defined in a view schema, they have to be replaced by the overall business-process activity.

Definition of transition arcs as activities: In workflow diagrams, transition arcs can be defined that represent the movement of objects from one box to another box although no workstep is performed on the objects (cf. Section 3.2.2). If a modeler of a view schema represents such transfers by an activity, this activity has to be omitted.

Definition of read arcs: Read arcs are defined in workflows to define ordering constraints that are not defined in the business process. If a read arc is defined in a view schema, it represents a workflow aspect and has to be removed. Notice that read arcs are not allowed in the business process schema, such that defined read arcs violate the syntactical correctness of the object life-cycle.

Example: Consider the object life-cycle of RES-WF shown in Figure 5.1; it includes several workflow aspects, which have to be extracted:

1. According to a supposed internal business rule, a deposit is required in any case. During extraction of the business process, the possibility of waiving the deposit is introduced (cf. activity waiveDeposit). Thus, an alternative that was omitted because of internal business rules, is inserted.
2. The alternative activities payHighDeposit and payLowDeposit will represent the same overall business-process activity. They may have been introduced as, according to some internal business rule, different rates for the deposit are used, depending on the rating

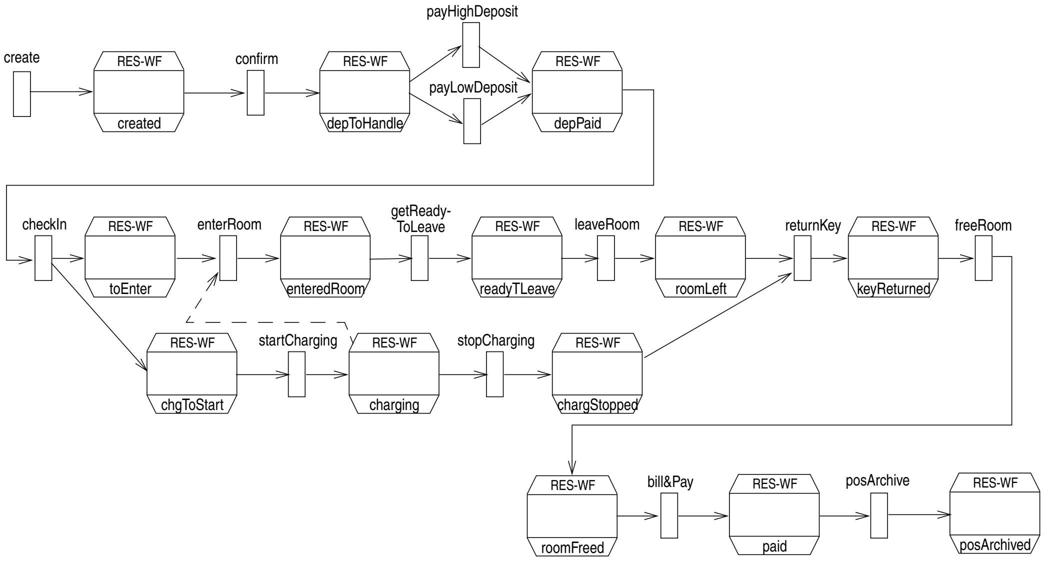


Figure 5.1: View RES-WF including workflow aspects

of the guest. These two alternative activities are replaced by one single activity payDeposit.

3. The deposit need not be necessarily handled before check-in. According to the business process, deposits can be handled at any time before the bill is issued. Therefore, the workflow-specific order dependency is replaced by parallelism: Activity confirm produces into two states, such that activities checkIn as well as payDeposit and waiveDeposit are enabled.

4. Activity bill&Pay is defined with the view, which represents a workstep, where billing and payment is performed together. The business process will introduce two activities issueBill and pay in sequence.

5. Activities returnKey and freeRoom may represent work that has to be done if a customer checks out. These activities represent external operations, which are only defined within a workflow but which are not modeled in the business process. Thus, these activities are replaced by the overall business-process activity checkOut.

6. Suppose that activity leaveRoom can be invoked on the departure day only according to an internal business rule. In the view, an object stays in state enteredRoom until the day of departure, when activity getReadyToLeave moves the object into state readyTLeave.

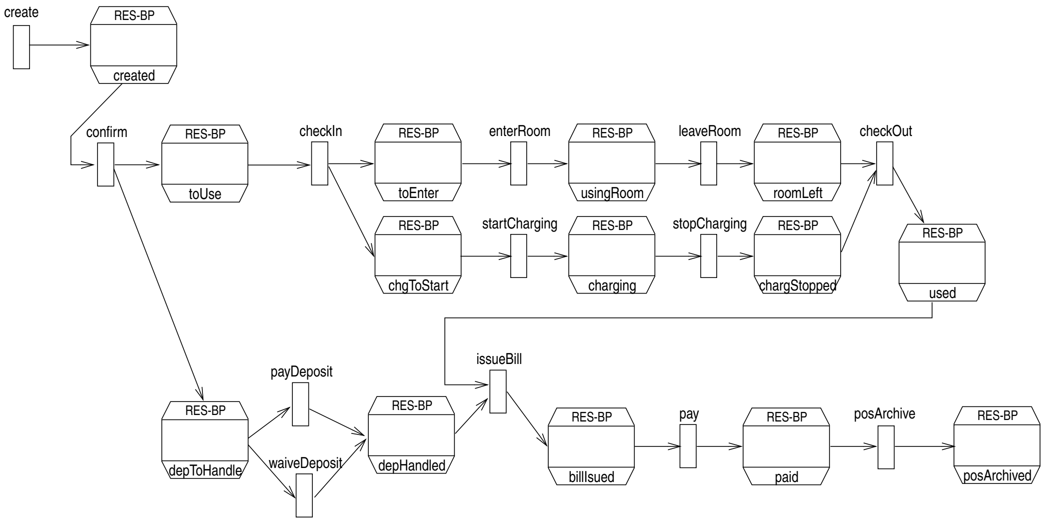


Figure 5.2: View RES-BP after business process extraction and schema improvement

This activity represents a transition arc, which was defined in Figure 3.1 for a similar purpose and is omitted.

7. The read arc defined between state charging and activity enterRoom, which represents a synchronization due to an internal business rule, is omitted.

View RES-BP without workflow aspects is depicted in Figure 5.2.

5.1.2 Extraction of Activity Specifications

In this step, workflow aspects defined in activity specifications are determined and removed. The following list gives an overview of workflow aspects:

Consideration of external information: A workstep specification diagram may consider ports that provide *external* information used for execution of a workstep, e.g., a passport, money, etc. This information will be treated in this integration step in one of two ways: If the port represents external information that is represented in the conceptual schema by some object type, this object type can be used as the domain of the port. If the external information is not considered in the schema, the external port will be omitted. Ports will particularly be omitted if they represent some "material flow", i.e., a port is an input port and an output port at the same time, where some external documents go in and come out again.

Definition of optional or alternative ports: Workstep specifications can comprise optional and alternative ports, which are not allowed in activity specifications. Such ports have to be replaced. Optional and alternative ports indicate that (1) several alternative activities with different ports or different domains should be defined instead of one single activity or (2) the domain of the port should be a more general object type.

Information calculated in the workstep: A workstep that invokes a business-process activity has to provide all actual values for the ports of the business-process activity. These values can either be provided to the workstep by input ports or determined within the workstep. In the latter case, no input port is defined in the workstep specification diagram for those values. If an activity in a view does not consider some information that is calculated in a workstep, but that has to be provided to the business-process activity by a port, a port is inserted.

Example: Consider the specification of activity pay depicted in Figure 5.3 (a): An alternative port mop is defined as an input port and an output port, which considers the means of payment used for paying for a reservation. Output port rec represents a receipt that is handed to the guest. Port rec represents external information that is not considered in the business process; thus this port is omitted. The alternative domains of mop are replaced by object type MOP, which represents different means of payment. The output port mop can be omitted at all as it defines some flow of material in a workstep: a credit card or ec card is provided via an input port and returned to the customer via the output port. If only information flow is considered, but no flow of material, the output port is not necessary. An additional port d for the date of payment is specified; the date is usually calculated in the workstep realization and therefore it may have been "forgotten" in the specification of the activity. The resulting business-process activity pay is shown in Figure 5.3 (b).

5.2 Inter-type Standardization

In this step, an object type is split into a set of related object types and a type hierarchy is defined from one single object type. The definition of

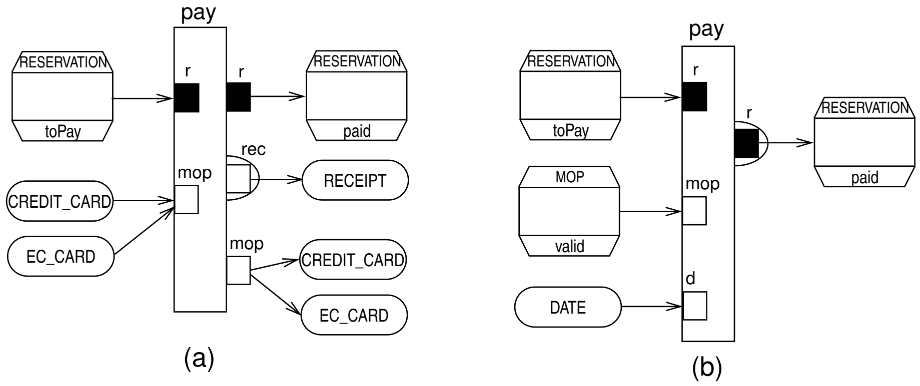


Figure 5.3: Extraction of the business-process activity from the specification of activity pay

structure and behavior of the instances of the original object type must be split, too. The system integrator has to examine which parts of the original definition of behavior have to be assigned to which one of the object types that result from splitting.

We distinguish three kinds of splitting object types, which will be treated in more detail in separate sections:

Splitting entities and contexts: An object type o that is defined in a view schema where each instance represents different entities of the real world or entities in different contexts is split to a set of object types O' where each instance of object type $o' \in O'$ represents one entity or one context in which entities can occur.

Explicit definition of relationships: Relationships between instances of object types o_1 and o_2 can either be represented by a property defined with each o_i whose domain is the other o_i (*direct relationship*), or by an association object that is an instance of an association type o_r with two *hasConstituent* properties with domain o_1 and o_2, respectively. During explicit definition of relationships, an association type is defined instead of an existing direct relationship.

Subtyping: An object type o in a view schema is replaced by a type hierarchy consisting of one object type $\hat{o}$ and a set of direct subtypes of $\hat{o}$. Thereby, different variants of objects that are instances of o are replaced by instances of different object types.

5.2.1 Splitting Entities and Contexts

An object type o is given that has to be split into a set of object types O'. Each instance of o is represented by several instances of object types of O', where the behavior of the latter instances is equivalent to the behavior of the corresponding instance of o.

We distinguish splitting entities and splitting contexts as mentioned before. There are the following assumptions that have to hold for splitting entities and contexts:

Splitting entities. We assume that object type o represents related entities where one entity can be considered as an aggregate (or *superordinate*) entity whereas the other entity is a component (or *subordinate entity*). Then, object type o can be split to object types o_a and o_c (i.e., the object type of the aggregate and the object type of the component) that are related to each other by a component relationship, i.e., o_c is a component type of o_a. Each instance of o then corresponds to one instance of o_a and a possibly empty set of instances of o_c.

We assume that only component relationships are considered during splitting as components are *exclusively assigned* to an aggregate object, the relationship between the component and the aggregate is *immutable*, and the components cannot be created before the aggregate object has been created.

Example: Suppose that there is a bill issued for each reservation. An object type RES_WITH_BILL is defined in a view schema, which represents structure and behavior of reservations and the related bills. Yet, reservations and bills are different entities, where a bill is exclusively assigned to exactly one reservation at any time and a bill cannot be assigned to different reservations during its lifetime. During splitting entities, object type RES_WITH_BILL will be split to object type RESERVATION and object type BILL, where the latter is a component type of the former.

In the case of *general relationships,* our approach would not work as the relationship between objects can be changed such that there is no unique mapping between an instance of o and instances of the object types in O'. The system integrator will have to change a view schema if an object type should be split to several object types that are related to each other by general relationships. This change is not subject to any correctness criteria, but the system integrator will have to validate the change with respect to the expected application.

Splitting contexts. If an entity can appear in different contexts, *roles* are defined to represent the contexts. Thus, each entity is in any case represented by an instance of a role supertype (*superordinate context*), whereas the optional contexts are defined as instances of a related role subtype (*subordinate context*). Therefore, object type o is split to o_e and o_r (for the object type of the entity and the object type of the role), where the former is the role supertype and the latter is the related role subtype. Each instance of o then corresponds to one instance of o_e and optionally to one instance of o_r (in case of non-qualified roles) or to a set of instances of o_r (in case of qualified roles).

Example: Suppose that a view comprises object type PERS_STAFF whose instances represent staff members. Yet, as a person may obtain and lose staff roles during his/her lifetime, the object type should be split to object type PERSON and role type STAFF. Then, each instance of STAFF is exclusively assigned to an instance of PERSON at each moment of time (exclusive assignment) and cannot be related to different instances of PERSON during its lifetime (immutability). But, notice that an instance of PERSON can be related to different instances of STAFF during its lifetime, one at each moment of time, even if the *hasRole* relationship defined in PERSON is single-valued. If the *hasRole* relationship is multi-valued, each instance PERSON can be related to several instances of STAFF at the same time (see *qualified* and *non-qualified roles* in Section 2.1.2).

For simplification, we refer to o_a and o_e commonly as *superordinate types* and to their instances as *superordinate objects*, whereas we refer to o_c and o_r as *subordinate types* and to their instances as *subordinate objects.*

We will find out that for splitting of entities and splitting of contexts, the same rules hold in this approach. Therefore, we will consider both cases in common and will always speak of *aggregates* if we mean aggregate objects or instances of the role supertype and speak of *components* if we mean components or roles; further, we will only refer to object type o_a and o_c if we mean o_a or o_e and o_c or o_r, respectively.

Basics of splitting. If an object type o in the business process view schema represents different entities, each *activity* in the behavior definition of o concerns either entities of the superordinate type or of the subordinate type, or both. An activity *concerns* an entity if it changes properties of an instance i of o that can be assigned to that entity or it moves i from one state to another state, where these states can be considered to represent processing states of that entity. This means, that the system integrator will decide whether an activity should be assigned to the object life-cycle of o_a, to the object life-cycle of o_c, or to both object life-cycles.

For splitting, the following correctness criteria must hold:

1. There is an object type o_a such that for any instance i of o, there has to be an instance i' of o_a, respectively. Instances i and i' are created at the same time.

2. Instances of o_c (i.e., components) that correspond to i can be created at a later point of time. More precisely, if an activity of o that considers a subordinate entity is invoked on an instance i of o, there has to be a corresponding instance i' of o_c on which the corresponding activity can be invoked, too. For that it is necessary that such an instance i' exists. Therefore, we require that the first activity of o that is invoked on i and that concerns a subordinate entity corresponds to a create activity of o_c.

3. Each activity of o corresponds to at most one activity of o_a and at most one activity of o_c. If an activity a_a of o_a and an activity a_c of o_c correspond to the same activity of o, they always have to be invoked at the same time.

4. We require that the possible sequences of activity invocations on an instance i of o correspond to the possible sequences of activity invocations on the corresponding instances of o_a and o_c. This means that if an activity can be invoked on i, the corresponding activities can be invoked on the respective instances of o_a and o_c and vice versa.

Example: Consider Figure 5.4: Part (a) shows the object life-cycle of PERS_STAFF, which represents people in their context as person and as staff member. Object type PERS_STAFF is split into two object types PERSON and STAFF, whose object life-cycles are shown in part (b). All activities, states, and labels of PERS_STAFF are assigned to one or both object life-cycles after splitting (correspondences are indicated by equal names). For simplicity, we assume that any person can get at most one job during his/her life.

For each instance of PERS_STAFF, there is a corresponding instance of PERSON, which is created at the same time (by activity birth). An instance of STAFF is created if activity createJob is invoked. The role object is optional: no instance of STAFF is created if getNoJob is executed. Thus, each instance i_{PS} of PERS_STAFF corresponds to exactly one instance i_P of PERSON and at most one instance i_S of STAFF. Each

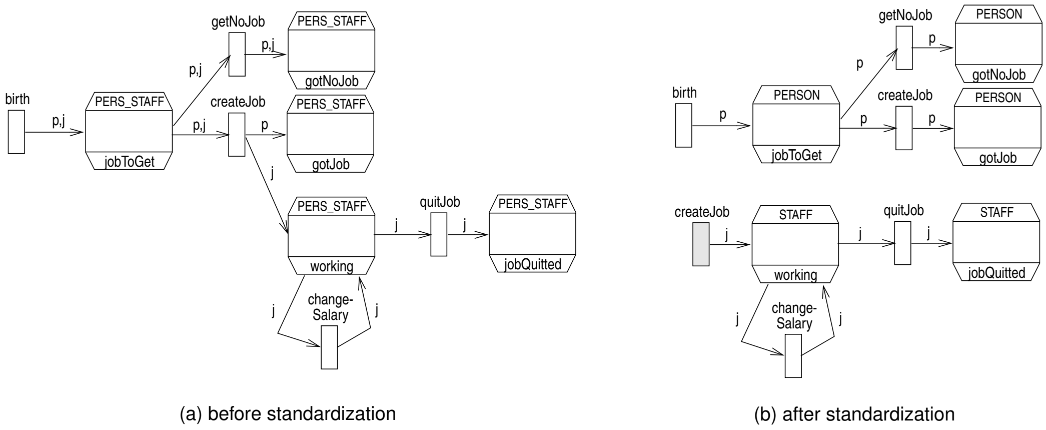


Figure 5.4: Example: correctness criteria for splitting contexts

sequence of activity invocations that can be executed on i_{PS} can be executed on the respective instances i_P and i_S, too, and vice versa.

When the system integrator splits object types, he/she must follow these correctness criteria. We will discuss the problems that may encounter and give design guidelines that should help the modeler to find a correct splitting. These guidelines will be applicable in most cases. Yet, the system integrator is not enforced to use these guidelines as long as the above correctness criteria are obeyed.

We will distinguish two cases of splitting entities: (1) *single-valued components,* i.e., o_a refers to o_c by a single-valued *hasComponent* relationship, and (2) *multi-valued components,* i.e., o_a refers to o_c by a multi-valued *hasComponent* relationship.

5.2.1.1 Single-valued Components

First, we discuss how object life-cycles have to be split in case of single-valued components.

The object life-cycle of o contains activities and states that concern (1) the superordinate entity, (2) the subordinate entity, or (3) both. In case (1), the respective activities and states should be assigned to o_a, in case (2), they should be assigned to o_c, and in case (3), they should be assigned to both o_a and o_c.

Activities of (1), which concern the superordinate entity, and activities of (2), which concern the subordinate entity, are either executed independently from each other or they are synchronized with each other. The idea of splitting is the following. Consider an object life-cycle of o: activities of (1) and activities of (2) will belong to different parallel chunks in o, so that activities concerning aggregates and activities concerning components can be executed independently from each other. If an activity of (1) and an activity of (2) must be synchronized with each other, *synchronization states* are introduced into which an activity of (1) produces and from which an activity of (2) consumes or vice versa. Activities of case (3), which concern the superordinate *and* the subordinate entity are represented after splitting by one activity a_a in o_a and one activity a_c in o_c. As a_a and a_c are tightly coupled they must always be executed at the same time.

As a correctness criterion we require the following condition to hold: Suppose that there is an instance i of o that corresponds to an instance i_a of o_a and to at most one instance i_c of o_c. If an activity is invoked on i and this activity is assigned to o_a (or o_c or both), then this activity must be invocable on i_a (or i_c or both), too. If an activity can be invoked on i_a (or i_c or both), it must be invocable on i, too.

The following problems must be resolved during splitting:

1. *Extraction of entity aspects:* The system integrator has to find out which activities and states of o concern aspects of the superordinate or the subordinate entity. All activities of o must be assigned to o_a, o_c, or both. An activity that is assigned to both object types represents a synchronization between the object life-cycles. Further, the system integrator must determine which states have to taken into the object life-cycles of o_a or o_c and which states are synchronization states, which cannot be defined in the new object life-cycles as they define a synchronization between activities that are defined in different object life-cycles after splitting. We will restructure the object life-cycle of o to separate aspects of different entities.

2. *Definition of synchronizations:* Synchronization states cannot be defined in the new object life-cycles as they are connected with activities of o that are assigned to different object life-cycles after splitting. If an activity of o_a and an activity of o_c are coupled as they correspond to the same activity in o, they must always be executed at the same time.

3. *Different creation time for superordinate and subordinate objects:* In o, only one instance is defined for the related entities with one creation time. After splitting, the superordinate and the subordinate objects may be created at different points of time.

4. *Optional definition of subordinate objects:* The *hasComponent* relationship defined with o_a may be optional, i.e., whereas for each instance of

o, there must be an instance of o_a, the creation of a related instance of o_c is optional. If for an instance i of o, there is a related instance i_a of o_a, but no related instance of o_c, then no activity of o that concerns a subordinate entity can be invoked on i.

5. *Mutable* hasComponent- *and* hasRole *relationships:* Single-valued *hasComponent*- and *hasRole* relationships can be changeable, i.e., at any point of time, only at most *one* instance of o_c is related to an instance of o_a, but different instances of o_c may be related to an instance of o_a in sequence. Notice that this case is no contradiction to our assumption that *componentOf* relationships are immutable. We assume that one component can only be assigned to one aggregate and cannot change this relationship to another aggregate object. Yet, an aggregate object can have different components during its lifetime, i.e., *hasComponent* relationships are mutable.

6. *References from other object types:* Object types may refer to a state s of the object life-cycle of o in activity specifications as the domain of a port p. As object type o does not exist any more after splitting, this reference must be changed to one of the new object types.

1. Extraction of entity aspects. The object life-cycle of o should be restructured in a way that activities that belong to the superordinate entity are separated from the activities that belong to the subordinate entity by different parallel chunks.

We use the concept of labels to extract entity aspects. Remember that labels were used to distinguish behavior of different aspects of objects. Here, labels can be used in o to distinguish behavior of the superordinate entity, the subordinate entity, and labels that exist in order to provide a correct labeling for synchronization states. As activities and states with different labels constitute different parallel chunks of the object life-cycle, the parts of the object life-cycle that concern superordinate and subordinate entities can be clearly distinguished.

We assume that an unlabeled object life-cycle of o is given. This object life-cycle must be labeled first. To fulfill the labeling properties, the system integrator may have to insert redundant states. Redundant states must not further restrict the possible sequences of activity invocations. Inserting labels and redundant states will be discussed in more detail during integration of object life-cycles (cf. Section 6.4.2.3).

After labeling and inserting states, there is a set of labels L defined in the object life-cycle of o, where L_a, L_c, and L_s are the sets of labels that consider aspects of the superordinate entity, the subordinate entity, and synchronization aspects, respectively. Labels that represent synchronization aspects only exist to provide for a correct labeling if synchronization states are defined. It has to hold that (1) $L = L_a \cup L_c \cup L_s$, (2) $L_a \cap L_c = \emptyset$ and $L_s \cap (L_a \cup L_c) = \emptyset$, and (3) for each activity in o holds that it is labeled with at least one label from $L_a \cup L_c$.

The object life-cycle of o_a consists of the set of labels L_a as well as all activities, states, and arcs that are labeled with at least one label of L_a. The object life-cycle of o_c consists — with three exceptions that are treated later — of labels L_c as well as all activities, states, and arcs that are labeled with at least one label of L_c.

Example: Consider the object life-cycle of PERS_STAFF shown in Figure 5.5. Object type PERS_STAFF describes persons that become member of the staff of some enterprise. For simplicity, we assume that a person can only become member of an enterprise's staff once in his/her lifetime. The object type has to be split into two object types PERSON and STAFF, where the latter is a role type of the former.

First, labels and redundant states are introduced: Labels j and p are assigned to PERSON, label r is assigned to STAFF, whereas labels s1 and s2 are synchronization labels. Activity education will be assigned to PERSON as it concerns aspects of a human being in the context of PERSON, but not in the role of STAFF. Thus, labels r (which is assigned to STAFF and s2 (which is only introduced for synchronization) should not be assigned to education. Therefore, the redundant state jobToGet has been introduced to which labels r and s2 are assigned. Further, redundant states notProtected, gotJob, and jobArchived are defined to obtain the labeling properties (cf. Section 2.4.1). States notProtSync and toRetireSync are synchronization states as the activities that produce to or consume from these states are assigned to different object life-cycles after splitting. The enriched object life-cycle is shown in Figure 5.6; added states are depicted with dashed borders, synchronization states are depicted shaded.

The object life-cycle of PERS_STAFF shown in Figure 5.6 is split to the object life-cycles of PERSON and STAFF as shown in Figure 5.7: All activities, states, and arcs labeled with j or p are included in the object life-cycle of PERSON whereas activities, states, and arcs labeled with r are included in the object life-cycle of STAFF. Activity getJob of PERS_STAFF corresponds to the create activity createJob of STAFF. As activity getJob is labeled with label j, a corresponding activity getJob is defined with PERSON, too.

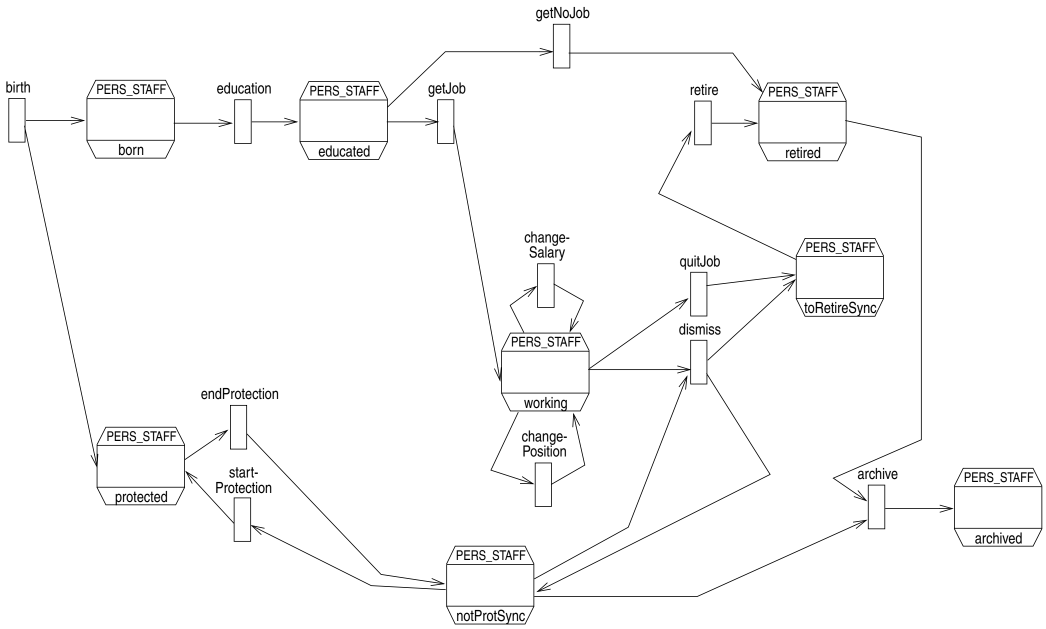


Figure 5.5: Original object life-cycle of PERS_STAFF

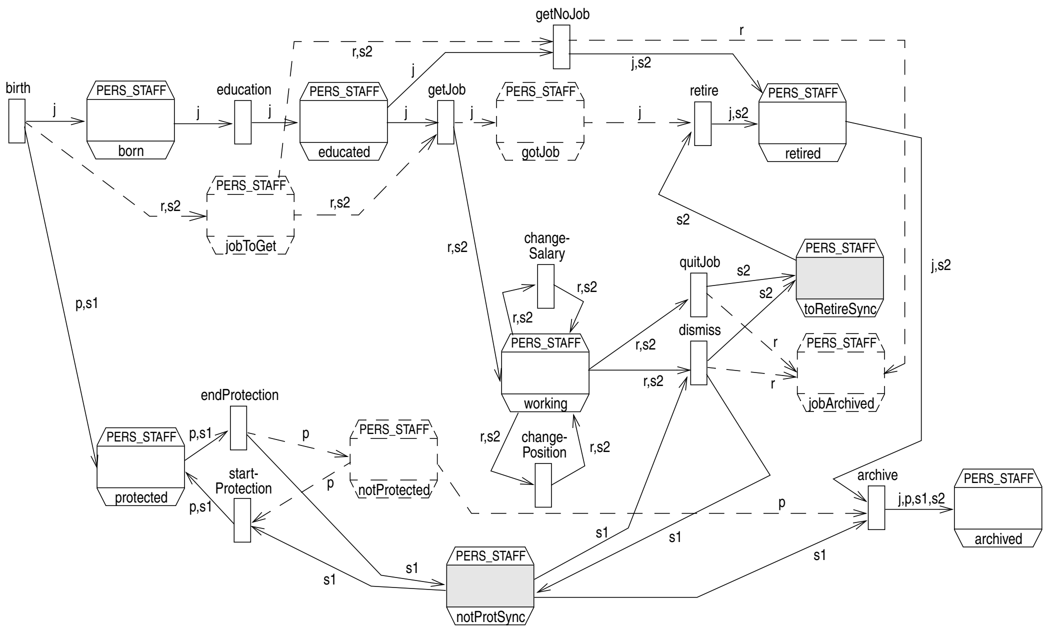


Figure 5.6: Labeled object life-cycle of PERS_STAFF

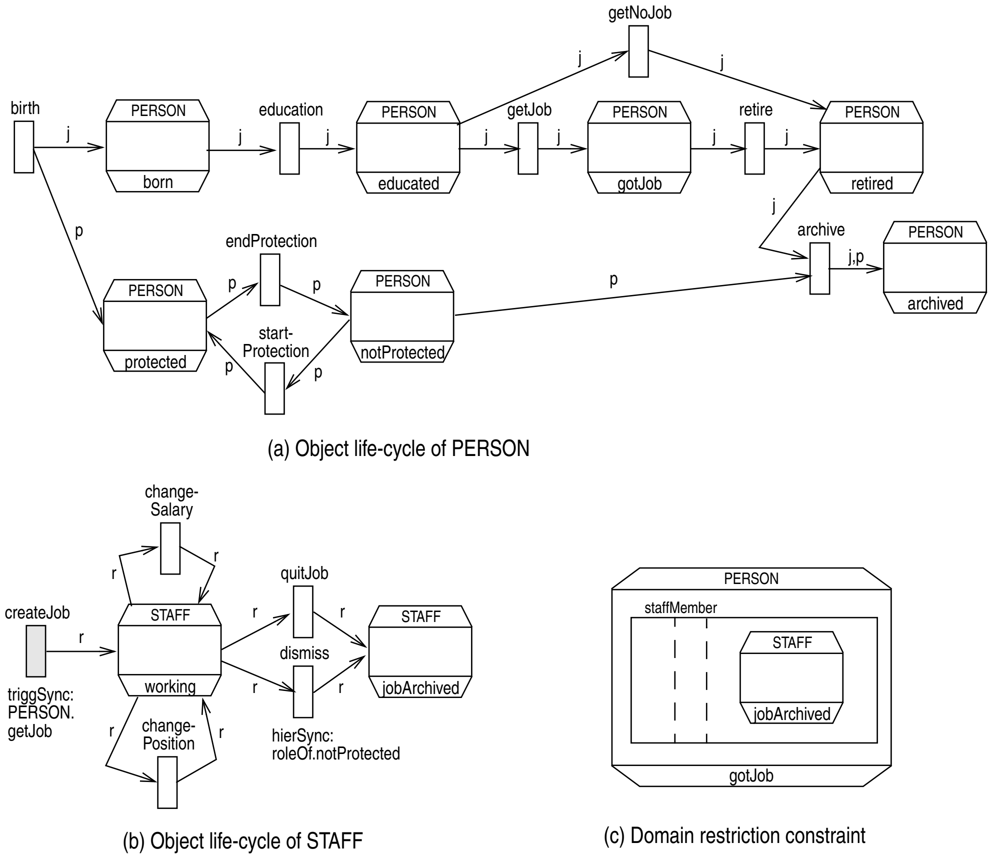


Figure 5.7: Object life-cycles of PERSON and STAFF

The fact that activities birth and getNoJob are not assigned to the object life-cycle of STAFF although they are labeled with label r will be explained later. Further, ignore the triggered synchronization constraint and the hierarchical synchronization constraint in the object life-cycle of STAFF in part (b) of the figure and the domain restriction constraint shown in part (c) for the moment.

2. Definition of synchronizations. Synchronizations between activities are defined by synchronization states. If a state is defined in the object life-cycle of o that is only labeled with synchronization labels, it is included neither in o_a nor in o_c. Thus this synchronization has to be defined in another way. Further, we required before that a pair of activities, one of them defined in o_a and the other one defined in o_c, that correspond to the same activity in o must always be invoked at the same time to obtain behavior equivalence.

Therefore, we define the following guidelines for synchronization:

- *Activity synchronizations:* Activities a_a and a_c in o_a and o_c, respectively, that have to be invoked at the same time are synchronized by a *triggered synchronization constraint*, which is defined with a_c and refers to the activity state of a_a.

  Example: Activity getJob of the object life-cycle of PERS_STAFF is assigned to both PERSON and STAFF. Thus, activity getJob in PERSON must be synchronized with activity createJob in STAFF by a triggered synchronization constraint, which is depicted with createJob.

- *Downwards state synchronizations:* If an activity a_c that concerns the subordinate entity can only be invoked if another activity a_a concerning the related superordinate entity has been executed before, a synchronization state is defined in o into which activity a_a produces and from which activity a_c consumes. This synchronization state is removed during splitting. Thus, the synchronization must be represented by a *hierarchical synchronization constraint* that is defined with activity a_c in o_c. This hierarchical synchronization constraint must refer to a state s_a in o_a, into which activity a_a produces. Thus, the system integrator must determine such a state that guarantees that the set of possible sequences of activity invocations stays the same. (Notice that an abstract state can be used as s_a and additional preconditions can be specified with a_c if the specification of a single basic state is not sufficient.)

Example: In our example, activity dismiss of PERS_STAFF can only be invoked on an object if this object resides in the synchronization state notProtSync. State notProtSync is a poststate of endProtection and a prestate of startProtection and archive. There is another state notProtected which is a prestate and poststate of these activities, too. Synchronization state notProtSync is replaced by a hierarchical synchronization constraint that enforces that activity dismiss of STAFF can only be invoked on an object if the related instance of PERSON resides in state notProtected.

- *Upwards state synchronizations:* This is the inverse case of *downwards state synchronization*: An activity a_a that concerns the superordinate entity can only be invoked on an instance of o if an activity a_c that concerns the subordinate entity has been executed before. In the object life-cycle of o, a synchronization state will be defined, which must be removed during splitting. In that case, a *domain restriction constraint* is defined with a prestate of a_a in o_a that refers to a state s_c in o_c, into which activity a_c produces. This domain restriction constraint enforces that activity a_a can only consume from state s_a if the related instance of o_c resides in state s_c.

Example: Synchronization state toRetireSync in the object life-cycle of PERS_STAFF is defined as a poststate of quitJob and of dismiss and as a prestate of retire. It represents the fact that activity retire can only be invoked if dismiss or quitJob has been executed before. State jobArchived is a poststate of quitJob and dismiss, too, such that for any point of time before activity retire is invoked on an object, state jobArchived is marked with an object if and only if toRetireSync is marked with this object, too. During splitting, state toRetireSync is replaced by a domain restriction constraint that is defined with state gotJob of PERSON that requires that activity retire is invoked on an instance of PERSON only if the related instance of STAFF resides in state jobArchived (cf. Figure 5.7 (c)).

3. Different creation time for superordinate and subordinate objects. One of the major advantages of roles and components is that they need not be defined as soon as the superordinate object is created, but at a later point of time. We assume that creation of an instance of the subordinate type o_c is initiated by the respective instance of the superordinate type o_a, i.e., an activity in o_a invokes the create activity in o_c; this fact is represented by a triggered synchronization constraint.

In the object life-cycle of o, each create activity c (i.e., each activity that consumes from the virtual state α) is labeled with all labels L. Thus, it is labeled with the labels L_c that concern the subordinate entity, too. According to the above rule, activity c would have to be included in o_a and o_c. Yet, the system integrator considers the subordinate object to be created at a later point of time. Therefore, we define one of the exceptions to the rule about splitting the object life-cycle of o, which we have introduced above: Create activities in o need not be assigned to object type o_c as a create activity; instead, the subsequent activities in o can be considered as create activities in o_c. The *subsequent* activities in o are those activities that consume from a state into that a create activity of o produces and that is labeled with at least one label of L_c.

Example: Instances of PERS_STAFF are created by activity birth. An instance of PERSON is created by activity birth at the same time as the corresponding instance of PERS_STAFF. Instances of STAFF are created by createJob, which is synchronized with getJob in PERSON by a triggered synchronization constraint; thus an instance of STAFF is created at a later point of time. Both getJob of PERSON and createJob of STAFF correspond to activity getJob in PERS_STAFF, which is a subsequent activity of birth.

4. Optional definition of subordinate objects. If the *hasComponent-* or *hasRole* property in the superordinate object type o_a is optional, it is possible that no instance of o_c is related to an instance of o_a. For instances of o that represent only a superordinate entity, but no subordinate entity, no activity that is assigned to o_c is invoked.

Suppose that each instance of object type o represents an instance of o_a and *optionally* an instance of o_c. There will be an activity c defined with o that is considered as the create activity of the subordinate entity. Thus, activity c will be assigned to both o_a and o_c during splitting (cf. item 3. above). There is at least one label $x \in L_c$ (where L_c comprises the set of labels in o that are assigned to o_c) that is assigned to activity c in o. Further, there must be a state s in o that is labeled with x and from which activity c consumes. If c is invoked on an instance of o, label x is consumed from state s. Yet, if *no* subordinate entity is represented by an instance of o, activity c is *not* invoked. Yet, to avoid a deadlock in that an object remains in state s, there must be some other activity c' that consumes label x from state s although activity c' is not assigned to o_c during splitting.

Thus, we see that there may be activities in o that are labeled with a label $x \in L_c$ but that do not concern the subordinate entity. This fact leads to the second exception of the splitting rule: If an activity c' consumes a label of L_c but it is invoked only if an instance of o *does not represent* a subordinate entity and thus does not correspond to an instance of o_c, then c' is not included in the object life-cycle of o_c.

Example: Activity getNoJob is invoked if an instance of PERSON is not related with an instance of STAFF. Although this activity does not concern the subordinate entity, it consumes label r to produce it into the final state jobArchived. According to the second exception, it is not included in the object life-cycle of STAFF.

5. Mutable *hasComponent-* and *hasRole* relationships. Different instances of o_c can be related to an instance of o_a, one after the other. We require that an instance of o_c may only be replaced by another instance of o_c if the former has already reached a final processing state, i.e., no further activities can be executed on this instance. In object type o, this fact is represented by a loop, that allows to invoke those parts of the object life-cycle that concern the subordinate entity iteratively. All labels of L_c are assigned to activities and states in this loop, as they are needed upon any new execution of the loop.

The object life-cycle of o_c does not include this loop as the fact that several subordinate objects can be created is expressed by several subsequent instantiations of the object life-cycle of o_c. Yet, in o_a, a loop will be defined as the creation of an instance of o_c is initiated by an activity of o_a, which can be invoked several times. This loop will consist of one activity for creation of a subordinate object and one activity for archivation of the subordinate object. The latter activity moves an instance of o_a into a state where a new subordinate object can be created.

To provide for freeness from deadlocks in the object life-cycle of o, the labels L_c in the loop have to be forwarded to a final state if the superordinate object is archived. This requires that an activity consumes from a state labeled with labels of L_c although this activity does not concern aspects of the subordinate entities. Therefore, the third exception on splitting object life-cycles says that activities that remove labels of L_c from the loop are not included in o_c.

Example: Figure 5.8 (a) shows an object life-cycle of PERS_STAFF, where several roles may be defined in sequence. The object life-cycles that result from splitting are shown in part (b) of the figure. Activity getJob of PERSON invokes activity createJob of STAFF. Further, loseJob of PERSON and archiveJob of STAFF have to be invoked at the same time. This

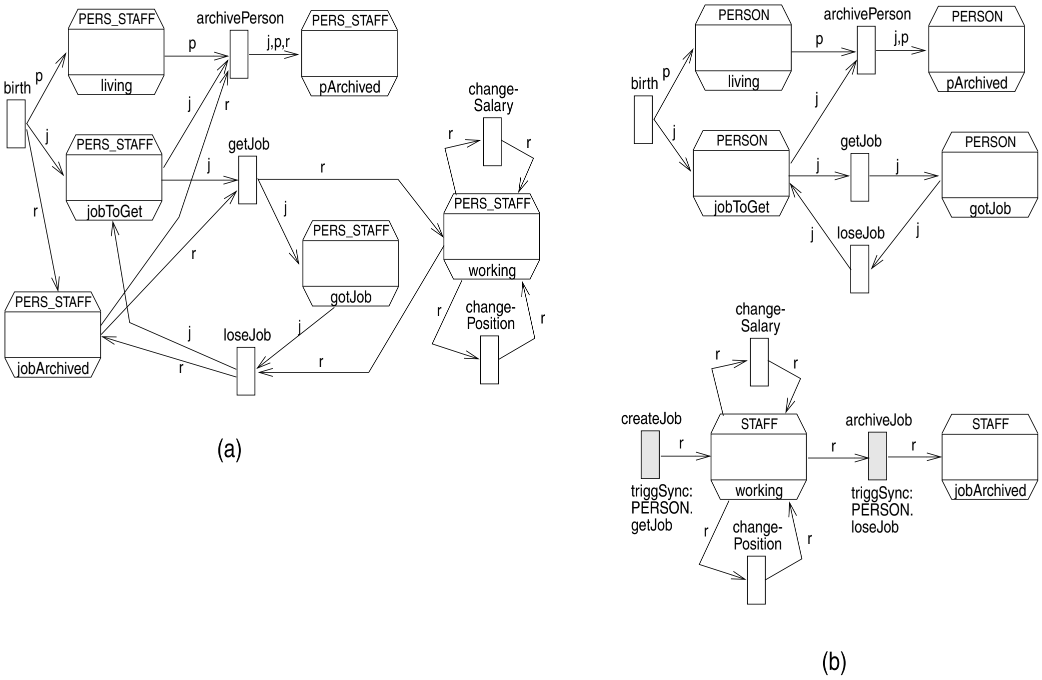


Figure 5.8: Object life-cycles of PERS_STAFF, PERSON and STAFF

fact is represented by a triggered synchronization constraint. Notice that activity archivePerson consumes and produces label r although it does not consider staff aspects, but it moves a token in states jobToGet and jobArchived into a final state.

6. References from other object types. If there is an object type with an activity a that refers to a state s of o as the domain d of a port, this domain must be changed to that object type o_a or o_c to which state s is assigned. If state s is a synchronization state and thus is omitted after splitting, the domain of port p must be changed to a state that is equivalent to state s, i.e., to a state that is marked with an object i if and only if state s is marked with i. If such a state does not exist, there are two possibilities: (1) State s is no synchronization state, but must be assigned to o_a or o_c, such that it remains in the behavior definition of o_a or o_c after splitting and can be referenced from other object types. (2) There is another state s' of o that is defined in o_a or o_c and that is marked with an instance of o at any time when s is marked with this object. In order to preserve information capacity, further restrictions must be specified by a textual condition defined with a.

Example: Suppose that a port p of an activity refers to synchronization state notProtSync (cf. Figure 5.6). After splitting, port p refers to notProtected, which is marked with an instance of o at any time when notProtSync is marked with this instance. Yet, during execution of dismiss, notProtected is marked whereas notProtSync is not marked. Thus, a precondition dismiss NOT IN p.states is added (where $p.states$ is an implicit property that returns the set of states in which the object that is assigned to port p resides).

5.2.1.2 Multi-valued Components

We speak of *multi-valued components,* if each instance of o corresponds to exactly *one* superordinate entity and a *set* of subordinate entities. This means that o is split to two object types o_a and o_c where one instance i_a of o_a is related to a set of instances I_c of o_c. Instances i_a and I_c together correspond to one instance of o.

The idea of splitting is the same as used for single-valued components. Yet, the fact that several subordinate entities must be considered at the same time is new. With single-valued components, different subordinate entities can be related to a superordinate entity only one after the other. With multi-valued components, different subordinate entities may be related with a superordinate entity at the same time.

We will examine several variants how multi-valued components may be expressed in an object life-cycle of o and deal with the question how these variants can be handled during splitting. To come to the point, we will consider splitting with multi-valued components not very useful in general, but only in restricted situations as the assumptions that have to be taken make splitting inadequate in many cases.

We show three variants how aggregate objects with multi-valued components can be handled in one single object type o and briefly discuss how the object life-cycle of o can be split to object life-cycles of o_a and o_c. In difference to Section 5.2.1.1, where guidelines could be presented about how to split object types, we will omit these guidelines here as the way of splitting depends on how the object life-cycle of o handles the subordinate entities such that no general procedure can be presented.

Example: We illustrate the three variants by an example shown in Figure 5.9. A person may take several roles as a guest as shown in the object diagram in Figure 2.1, where GUEST is a qualified role type of PERSON. There is an object type PERS_GUEST defined in a view, whose instances represent one person and a set of guest roles of this person.

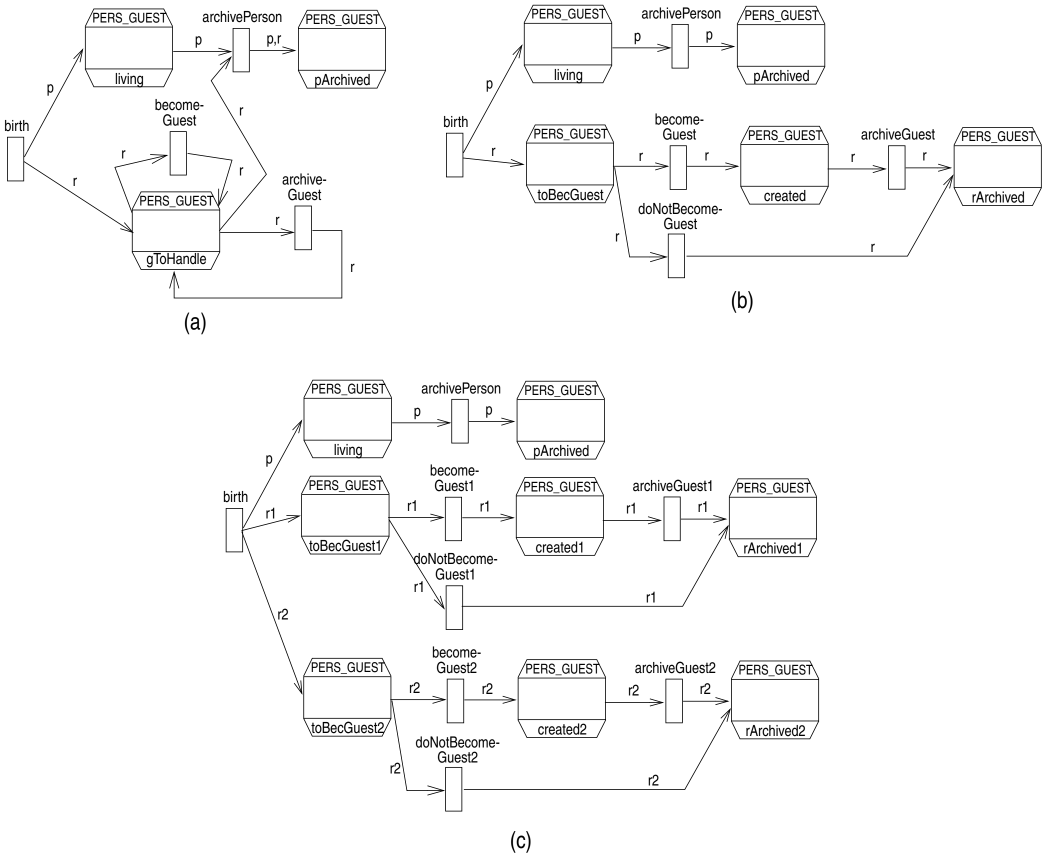


Figure 5.9: Object life-cycles of PERS_GUEST

The variants how the object life-cycle of PERS_GUEST could be defined in a view are shown in parts (a), (b), and (c) of the figure. Notice, that variants (a), (b), and (c) have to be considered independently from each other.

Definition of loops. Suppose that a set of activities in o concerns the subordinate entities. As subordinate entities exist independently from each other, activities concerning subordinate entities can be invoked in almost any order, yet, on different subordinate entities, which are represented by the same instance of o. But, for each subordinate entity, the activities have to be invoked in a special order. This order cannot be represented in the object life-cycle as the object life-cycle has to allow to invoke any of these activities at any point of time for different subordinate entities. To represent the allowed sequences of activities on each entity, rather complex conditions had to be defined with the activities. Thereby, the principle idea of object life-cycles that says that restrictions in the sequences of activity invocations are primarily represented by states between activities would get lost. Thus, this variant is inadequate unless very simple object life-cycles are considered.

During splitting, the object life-cycle of the component type o_c would have to be generated by analysing the defined conditions, which may be a very complex or even impossible task.

Example: Consider the object life-cycle shown in Figure 5.9 (a): A person may take a set of guest roles. Each guest role is created by activity becomeGuest and archived by archiveGuest. At any point of time, a person may obtain or lose a new guest role. Therefore, there is one state gToHandle, where both activities can be invoked. But this object life-cycle allows to invoke archiveGuest before becomeGuest has been executed. To avoid this, a precondition had to be defined. Suppose that further activities are defined on the role aspect of a person. All these activities would have to consume from and produce to state gToHandle. The sequence of activity invocations had to be enforced by preconditions if no information shall be lost.

Synchronized execution of activities. We assume that subordinate entities are processed synchronously, this means, each activity is performed on all subordinate entities that are related to the same superordinate entity at the same time. Then, each activity executed on subordinate entities has to be defined only once in o. This case is not considered useful as it is not typical for independent roles or components that they are executed in a synchronized way. If such a case arises, object type o could be split to o_a and o_c, where o_a has to contain all activities of o_c, too, to synchronize the activity invocations in the subordinate instances of o_c by triggered synchronization constraints.

Example: Consider part (b) of Figure 5.9: There are two activities becomeGuest and archiveGuest, again, that treat the guest roles of a person. Here, we assume that all guest roles are treated synchronously, all guest roles are created and archived at the same time.

Replication of activities and states. In this case, we assume that each instance of o represents only a statically limited number n of subordinate entities. Then, an object life-cycle can be defined with o, that replicates the activities, states, and labels that concern subordinate entities n times and defines them in parallel chunks. This way corresponds to the possibility to define several single-valued properties instead of one multi-valued property in object structure if the number of refered objects is limited. This case is not useful as (1) the maximum number of subordinate entities is statically limited, thereby compromising flexibility, and (2) a part of the object life-cycle has to be replicated several times, which makes the schema more complex.

During splitting, the activities, states, and labels that concern the component type o_c are defined only once as the fact that *a set of* instances of o_c are related to an instance of o is represented by multiple instantiation of the object life-cycle of o_c.

Example: In part (c) of Figure 5.9, we assume that one person can obtain at most two guest roles. For each of these roles, the activities, states, and labels are replicated. For distinction, their names are marked with an index.

5.2.2 Explicit Definition of Relationships

Relationships between objects can be represented by direct relationships defined with the respective object types or by association objects with constituent relationships. This discussion deals with the question how to transform direct relationships to a relationship with association objects.

There are two object types o_1 and o_2 defined in a view, where o_1 has a property with domain o_2 and vice versa. Thereby, a *direct relationship* r is defined. Such direct relationships can be replaced by relationships represented by an *association type* o_r. This means that o_r includes two *hasConstituent* relationships with domain o_1 and o_2, respectively.

If o_1 and o_2 are related by the direct relationship r, information about the properties of the relationship is optionally stored either with o_1 or with o_2. Without loss of generality, we will assume that object type o_1 keeps the information about the relationship r.

If an association type is defined, information about the relationship can be removed from o_1 and the general relationship in o_1 is replaced by a *constituentOf* relationship, resulting in o_1'. In o_2, the general relationship is replaced by a *constituentOf* relationship, too, resulting in object type o_2'. The information about the relationship is defined in the new association type o_r, which has two *hasConstituent* relationships with domain o_1' and o_2', respectively.

Association types have some properties that are important here: *hasConstituent* relationships defined with an association type are (1) single-valued, (2) immutable, and (3) mandatory. These properties are the same as for *componentOf*- and *roleOf* relationships. Further, *constituentOf* relationships, which are defined with the constituent types, are single-valued or multi-valued and they may be optional; these properties are equal to those of *hasComponent*- and *hasRole* relationships. Therefore, the process of explicit definition of relationships will be similar to extraction of entities and contexts, which was treated in Section 5.2.1.

We will distinguish two cases in this section: the *constituentOf* relationship defined in object type o_1 is single-valued (cf. Section 5.2.2.1) or multi-valued (cf. Section 5.2.2.2).

5.2.2.1 Single-valued Relationships

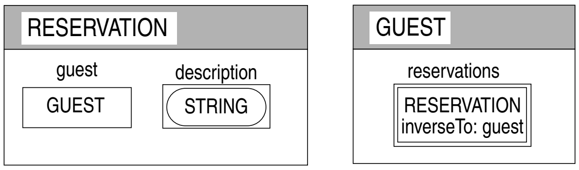
Each instance of o_1 is represented by an instance of o'_1 and at most one instance of o_r, and vice versa, whereas each instance of o_2 corresponds to one instance of o'_2, and vice versa.

The goal of this step is to extract the behavior of o_r from the behavior diagrams of o_1 and to adapt the object life-cycles of o_1 and o_2 as far as necessary. We will use our knowledge from *splitting of entities and contexts* and will discuss how the process defined there has to be modified for definition of association types.

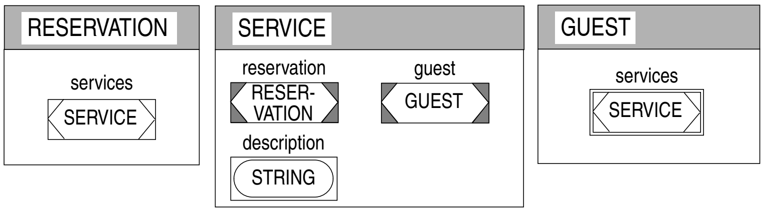
Example: Suppose that two object types RESERVATION and GUEST are defined (cf. Figure 5.10 (a)). There is a single-valued relationship defined with RESERVATION with object type GUEST as domain. Further, a property description is defined with RESERVATION that describes the relationship between an instance of RESERVATION and an instance of GUEST. During explicit definition of relationships, an association type SERVICE is defined and the object diagrams of RESERVATION and GUEST are changed as shown in part (b) of the figure. Notice that property description is moved from RESERVATION to SERVICE.

The main difference between the *explicit definition of relationships* and *splitting entities and contexts* is that an instance of an association type depends on *two* instances of constituent types, i.e., one instance of o'_1 and one instance of o'_2, whereas in component and role relationships, an instance of o_c only depends on one instance of o_a. Thus, the process of splitting is modified as follows:

Extraction of relationship aspects: Labels, activities, and states concerning the relationship must be extracted from the object life-cycle of o_1. These elements constitute the object life-cycle of o_r.



(a) direct relationship between RESERVATION and GUEST



(b) explicit definition of a relationship by association type SERVICE

Figure 5.10: Object diagrams before and after explicit definition of relationships

Definition of synchronizations: Synchronization states and synchronization labels may exist in o_1 such that triggered and hierarchical synchronization constraints as well as domain restriction constraints may exist between o_1' and o_r.

Creation of instances of the association type: An activity of o_1' will initiate the creation of an instance of o_r. Hence, each create activity of o_r will be synchronized with an activity a of o_1' by a triggered synchronization constraint. As upon creation of an instance of o_r, both constituent objects must be fixed, activity a must provide both constituent objects to the create activity of o_r by non-primary ports. One of these constituent objects is the instance i on which a has been invoked, the second constituent object must be determined in a.

Before splitting, when a general relationship exists between o_1 and o_2, the relationship between an instance i_1 of o_1 and an instance i_2 of o_2 will be established either (1) by i_1, (2) by i_2, or (3) by an instance i_3 of another object type o_3, where o_3 is likely to be a superordinate type of o_1 and o_2. In case (1), an activity a_1 of o_1 that is executed on i_1 will determine an instance i_2 of o_2 that has to be related with i_1; to notify this instance i_2 that it is related to i_1, there will be an activity a_2 defined with o_2 that is invoked on i_2 and that gets instance i_1 via an input port. In case (2), the same rule holds the other way round.

In case (3), an instance i_3 of o_3 determines which instances i_1 of o_1 and i_2 of o_2 are related to each other and will invoke an activity a_1 on i_1 and an activity a_2 on i_2, where a_1 receives i_2 via an input port and a_2 receives i_1 via an input port. The decision which case applies can be taken after examining the activity specifications.

After splitting, the domain of these input ports has to be changed from o_1 and o_2, respectively, to o_r, as o_1' and o_1' are not any more related directly, but they are related indirectly via the association type o_r.

Optional and mutable *constituentOf* relationships: The rules defined for the process of splitting entities and contexts hold for association types, too.

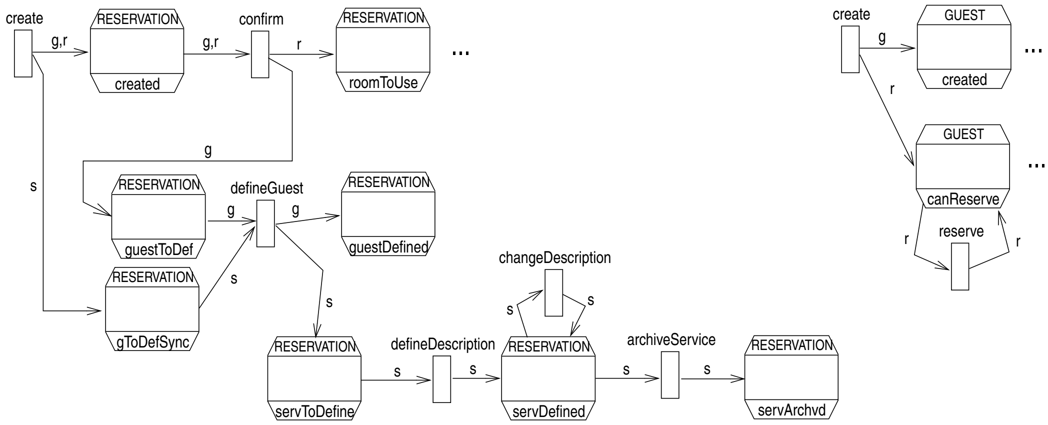
Example: Consider the example shown in Figure 5.11. In part (a) of the figure, parts of the object life-cycles of RESERVATION and GUEST are shown. Activity reserve of object type GUEST receives an instance of RESERVATION via a non-primary port. Label s of RESERVATION and all activities and states labeled with s, except activity create, are assigned to SERVICE.

Part (b) of the figure shows the new definition of the object life-cycles. Activity createService of SERVICE is invoked by activity defineGuest of RESERVATION. Thus, a triggered sychronization constraint is defined with createService. Activity reserve of GUEST is invoked to notify the related instance of GUEST that an instance of SERVICE has been created.

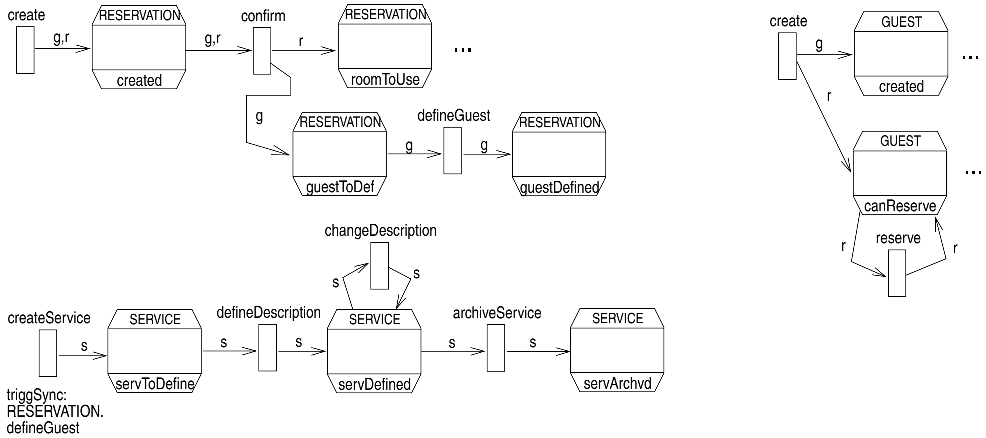
5.2.2.2 Multi-valued Relationships

In this case, a direct relationship is defined between object types o_1 and o_2, where o_1 comprises a multi-valued relationship with domain type o_2. In this case, an instance i_1 of o_1 represents the properties of a set of relationships and the object life-cycle of o_1 must handle behavior of all relationships; thus, all properties of a set of relationships must be defined in the object structure of o_1 and the behavior of all relationships must be defined in the object life-cycle of o_1. Therefore, similar problems will arise as discussed for multi-valued components in Section 5.2.1.2, such that no general rules are defined for this case and no more details are presented here.

Example: Suppose that any instance of RESERVATION can be related with a set of instances of GUEST, where for each related instance of GUEST, a set of strings is necessary for recording the description of the services. This causes a problem in definition of structure as something like a set of multi-valued properties were necessary for property description.



(a) Part of the original object life-cycle of RESERVATION and of GUEST



(b) Part of the new object life-cycle of RESERVATION, GUEST, and SERVICE

Figure 5.11: Replacing direct relationships by relationships with association types

5.2.3 Subtyping of Object Types

During subtyping, an object type o defined in a view schema is replaced by a type hierarchy with an object type $\hat{o}$ and a set of direct subtypes $\hat{o}_1, \ldots, \hat{o}_n$ of $\hat{o}$. Each instance of o has to correspond to exactly one instance of one of the $\hat{o}_i$. According to the abstract superclass rule, no direct instances of $\hat{o}$ exist.

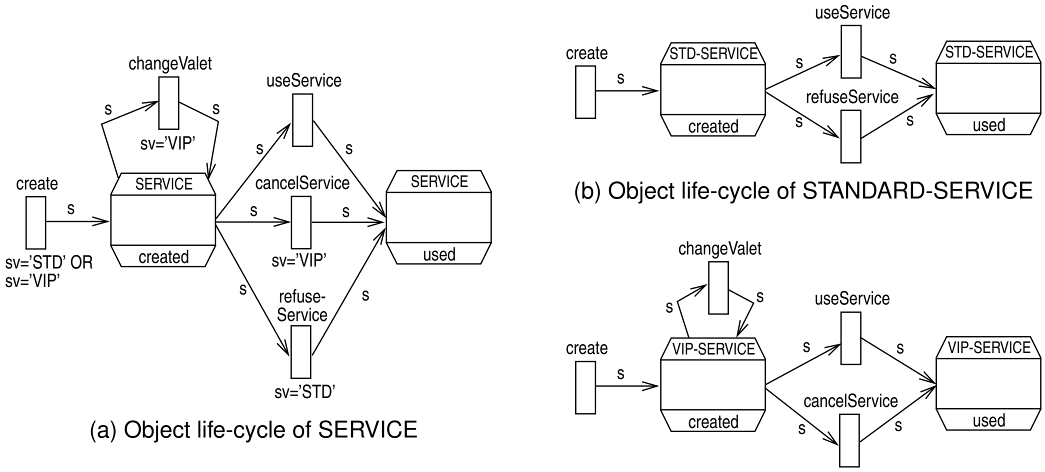
As an object is an instance of a certain object type, and the type membership of an object cannot be changed during the object's lifetime, we require that for each instance of o, there is a corresponding object that is an instance of one $\hat{o}_i$, and it can be decided at the time of creation, from which $\hat{o}_i$ this object is instantiated. This decision is based on a set of properties P (the so-called *discriminating properties*) defined with o: for each $\hat{o}_i$, a boolean condition c_i over values of these properties is defined such that all instances of o can be disjointly and completely classified. For each instance of $\hat{o}_i$, the actual values for properties P must be determined at the time of creation and must not be changed during the instance's lifetime to a value such that condition c_i is not any more true for this instance.

Example: Suppose that there is an object type PERSON with attributes sex, countryOfBirth, and age. It may be useful to define four subtypes AUSTRIAN_MAN, AUSTRIAN_WOMAN, FOREIGN_MAN, and FOREIGN_WOMAN according to the value of sex and countryOfBirth, which can be determined upon creation and do not change. Subtyping based on attribute age is not useful as the age changes continuously and at some time, an object had to change its type.

The object life-cycle of $\hat{o}$ is equal to the object life-cycle of o. The object life-cycle of each $\hat{o}_i$ must be a correct observation consistent specialization of $\hat{o}$. As no information is added during subtyping, no activities, states, and labels are added or refined during specialization. The only allowed operation is omitting alternatives. Thus, we determine for each $\hat{o}_i$, which alternatives can be chosen for its instances. Alternatives that do not apply to the instances of $\hat{o}_i$ are omitted from the object life-cycle of this $\hat{o}_i$. The decision about the alternatives is taken from the preconditions defined with the activities in the object life-cycle of o. If an activity has a precondition p such that this p is never true for instances of $\hat{o}_i$ because of c_i, this activity can be omitted in $\hat{o}_i$.

As the correctness criterion, we require that for each instance i of o, there is an instance i_i of $\hat{o}_i$, and it holds that any possible life-cycle occurrence γ of i is a valid life-cycle occurrence for i_i, too, and vice versa.

Example: Consider the object life-cycle of SERVICE shown in Figure 5.12 (a) and suppose that an attribute sv is defined with SERVICE that states



(a) Object life-cycle of SERVICE

(b) Object life-cycle of STANDARD-SERVICE

(c) Object life-cycle of VIP-SERVICE

Figure 5.12: Subtyping of object type SERVICE

whether an instance of SERVICE represents a standard or a vip service (sv='STD' or sv='VIP', respectively). The value for sv is defined by an input port sv of the create activity that must only take one of the values 'STD' or 'VIP'.

Object type SERVICE is split to three object types GUEST-SERVICE, STD-SERVICE, and VIP-SERVICE, where STD-SERVICE and VIP-SERVICE are direct subtypes of GUEST-SERVICE.

The object life-cycle of GUEST-SERVICE is equal to the object life-cycle of SERVICE, from which the object life-cycles of STD-SERVICE and VIP-SERVICE are observation consistent specializations, where alternatives are omitted. Depending on the type of service, certain alternatives in the object life-cycle can be or must not be invoked.

References from other object types. If there is an object type that contains an activity a that refers to object type o in a port p, the domain of this port must be changed to $\hat{o}$. If activity a defines a precondition for port p that restricts the domain of p to one of the new subtypes $\hat{o}_i$, the domain of p is set to $\hat{o}_i$.

Example: Suppose that there is an activity chargeService defined with the object life-cycle of object type BILL, which requires an instance of object type SERVICE in port s. After subtyping, the domain of s must be changed to GUEST-SERVICE.

Suppose, instead, that a precondition s.sv='VIP' is defined with charge-Service, which requires an instance of SERVICE, whose property sv has value 'VIP'. After subtyping, the domain of port s is set to VIP-SERVICE and the precondition is omitted.

Chapter 6

Integration of Object Life-Cycles

Contents

In this chapter, we deal with the problem of integrating object life-cycles. Integrating object life-cycles means that an object life-cycle for the integrated schema is defined based on two object life-cycles of different views. We will present two kinds of integration, namely *integration by generalization* and *integration by specialization.* Informally, the former integrates common behavior of two object life-cycles if a common supertype must be defined for two similar object types, whereas the latter integrates all behavior of two object life-cycles if an object type must be defined for two equivalent object types. Therefore, the integrated object life-cycle will be a *generalization* or a *specialization*, respectively, of the object life-cycles to integrate. These approaches were originally presented in [PS98a] and [PS98b], respectively.

For this chapter, we assume that the view schemas are inter-type standardized according to view preparation introduced in Chapter 5. The object life-cycles of all object types in the views must be intra-type standardized to obtain standardized view schemas. Then, the object life-cycles in the integrated schema can be defined by comparing and integrating the object life-cycles of the views.

We will discuss the process steps of *intra-type standardization* and *integration* for object life-cycles in this chapter. Before the steps are defined in detail, we will motivate and explain the overall idea of the integration of object life-cycles.

This chapter is structured as follows: In Section 6.1, heterogeneities between object life-cycles are discussed; Section 6.2 deals with correspondences between elements of the object life-cycles to integrate. Section 6.3 discusses how the integrated object life-cycle is defined for both types of integration (i.e., integration by generalization and by specialization). Finally, in Section 6.4, the steps of the integration process are discussed in more detail. The chapter ends with Section 6.5, where further issues are discussed.

6.1 Heterogeneities

As object life-cycles defined in the views are probably modeled by different users, several heterogeneities may occur between the object life-cycles to integrate B_1 and B_2:

Naming conflicts: Equivalent elements can carry different names (*synonyms*), whereas different elements can carry the same name (*homonyms*).

Inconsistent states and labels: Object life-cycles may define comparable information, but they use different labeling functions and different redundant states.

In previous work [PS98a, PS98b], we discussed the situation that unlabeled object life-cycles are defined by the users and labels and redundant states are inserted during integration. In that case, no labels are defined at all in the views. We will consider this possibility in this chapter, too.

Activity–port conflicts: A single activity in one object life-cycle corresponds to a set of alternative activities in the other object life-cycle, where different variants of processing are defined via different values for some ports of the single activity in the former, whereas the variants are expressed by different alternative activities in the latter. Further, more generally, a set of alternative activities in one object life-cycle may correspond to a set of alternative activities in the other object life-cycle. Different variants of processing are represented by different alternatives and different values for ports.

Condition–state conflicts: Pre- and postconditions of activities can be defined by pre- and poststates or textual pre- and postconditions. Two views may deviate from each other as they define conditions differently (by states vs. textual conditions).

Granularity conflicts: A single activity or a single state in one object life-cycle can correspond to a subnet of activities, states, and arcs in the other object life-cycle. A single label in one object life-cycle can correspond to a set of labels in the other object life-cycle. Further, subnets of activities and states or sets of labels may correspond to each other. Granularity conflicts and their treatment are further investigated in Section 6.2.

View-specific aspects: Labels that are defined in one object life-cycle can be missing in the other object life-cycle. Activities and states that constitute parallel chunks of an object life-cycle can be defined in one object life-cycle but may be missing in the other one.

View-specific alternatives: Alternative chunks defined in one view may be missing in the other view.

During the integration process, these heterogeneities have to be detected and resolved as discussed in this chapter.

6.2 Correspondences

The object life-cycles to integrate will comprise corresponding elements. These correspondences have to be considered during integration as elements that correspond to each other should have some common representation in the integrated object life-cycle. In order to be able to determine such correspondences, the object life-cycles have to be in a standardized form. Therefore, we will consider correspondences only when comparing *standardized view schemas*, which result from intra-type standardization (cf. Section 4.2). In standardized view schemas, only the following heterogeneities remain: naming conflicts, granularity conflicts, view-specific aspects, and view-specific alternatives.

The following correspondences between elements of object life-cycles can be determined:

Element equivalence: Two elements are equivalent if they represent the same information. Thus, an activity (a state, a label) in one view can only be equivalent to an activity (a state, or a label, respectively) in the other view.

Element inclusion: An activity or a state in one view may correspond to a subnet consisting of a set of activities, states, and connecting arcs in the other view. An element inclusion where an activity or state corresponds to a subnet is only possible if the single activity or state can be refined to the subnet according to the rules of observation consistent specialization (and, conversely, the subnet can be abstracted to a single activity or state, respectively, according to observation consistent abstraction). Further, a label in one view may correspond to a set of labels in the other view.

Set correspondence: A subnet of activities, states, and arcs in one view may correspond to a subnet in the other view; a set of labels in one view may correspond to a set of labels in the other view. As we will explain later, set correspondences are only allowed in integration by generalization. Further, a set correspondence between two subnets is only allowed if both subnets can be abstracted to an activity or both subnets can be abstracted to a state according to observation consistent abstraction.

No correspondence: An activity, state, or label in one view may have no correspondence to an element in the other view. Such activities, states, and labels are refered to as *unique* in the remainder of this work. Unique activities and states may occur because of alternatives that are only defined in one view (view-specific alternatives) or because of additional aspects considered in a view (view-specific aspects). View-specific alternatives do not cause the existence of unique labels, whereas view-specific aspects are labeled with unique labels as they are defined in a view as parallel extensions of those parts that are defined in the other view, too, such that new labels are introduced.

The fact that elements correspond to each other by inclusion or by set correspondences represents a granularity conflict at the same time. We will refer to all elements that have corresponding elements in the other view as *common* elements, no matter whether the correspondence is an equivalence, inclusion, or set correspondence.

Correspondences between elements have different meaning for object life-cycles of *similar* object types, which must be integrated by *generalization* and object life-cycles of *equivalent* object types, which must be integrated by *specialization*:

Correspondences between elements of similar object types. If two object types are similar, both object types are defined in the integrated schema and a common supertype is defined by integration by generalization. Correspondences between elements in object life-cycles of similar object types mean that the common supertype should contain elements to express these correspondences. Yet, these corresponding elements may be defined in the given object types in a completely different way (i.e., one object life-cycle defines a single activity, state, label, whereas the other object life-cycle defines a subnet, or the object life-cycles define different subnets to represent a similar concept). These differences do not cause any problem at all as the set of objects treated in these object life-cycles is disjoint (yet, an activity of the supertype cannot homogeneously be invoked on instances of the subtypes if the specification of this activity is specialized in the subtypes or the activity is refined in the subtypes).

Example: Suppose that two similar object types R1 and R2 must be integrated. Both object life-cycles define activities considering payment (cf. Figure 6.1); yet, payment is defined differently such that a set correspondence will be determined. In the common supertype, which is defined by integration by generalization, one single activity pay will be defined to represent the fact that the concept of payment is defined in both views.

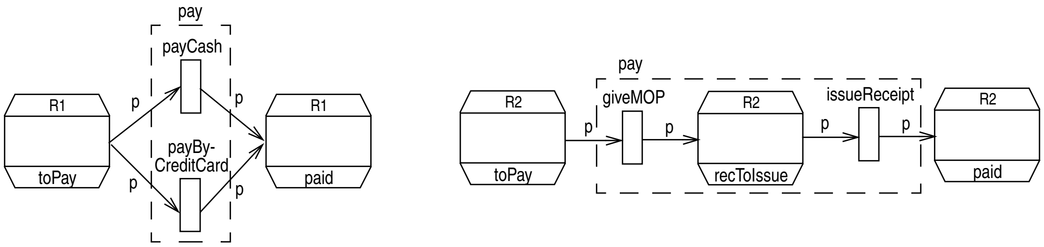


Figure 6.1: Example for a set correspondence

Correspondences between elements of equivalent object types. Correspondences between elements in object life-cycles of equivalent object types mean that both views define a view of the *same elements* in the integrated schema. This means, the view schemas do not include different, comparable elements, but they include views of the same elements. As mentioned in the introduction, at most one view defines an activity specification for an activity that is defined in the integrated object life-cycle; another view may refer to this activity but must not define an activity specification for it. Further, abstract activities are always unspecified. Therefore, correspondences must be considered in the following manner:

- *Equivalences:* If an element in one view is equivalent to an element in the other view, one element exists in the integrated schema, whose execution is observed by both views. If equivalent *activities* are considered, either both activities are unspecified or one activity is unspecified and one is specified.

- *Inclusion:* If an element in one view corresponds to a subnet of elements in the other view, there is a subnet in the integrated schema, where the latter view perceives this subnet in full detail, whereas the former view perceives this set of elements only *abstracted* to a single element. If a single activity corresponds to a subnet, the single activity is *unspecified.*

  If the single activity is specified, *no* inclusion can be determined; yet, in this case, a conflict arises. This case can be considered as a special case of *set correspondence* (see below) or as an *activity–port conflict* (see later in this chapter). In the latter case, the conflict will be resolved before correspondences are determined.

- *Set correspondence:* Set correspondences are not allowed for equivalent object types as they cannot occur if two object life-cycles model the behavior of the same object type. A set correspondence means that an abstract activity or an abstract state is refined differently in two views but there can be only one most refined object life-cycle in the integrated schema. Supposed set correspondences indicate design errors as follows: (1) At least one subnet does not model a pure business process, but considers internal business rules. (2) The subnets consider different aspects (i.e., different labels), and thus, they do not correspond to each other at all. (3) A single element in one subnet corresponds to a single element or a set of elements in the other subnet, i.e., correspondences have not been detected at the finest level of granularity and several equivalences or inclusions should be determined instead of one set correspondence. (4) The views consider processing of different objects and the respective subnets do not correspond to each other but constitute different alternatives that are invoked depending on some condition. (5) Activity–port conflicts have not been detected, such that the activities should be standardized before.

Example: Consider once more Figure 6.1: If these subnets are defined in views of object life-cycles of equivalent types, no decision about how the object life-cycle should be defined in the integrated schema can be taken automatically. One reason for the set correspondence could be that activities giveMOP and issueReceipt are worksteps representing external operations that should be replaced by one single business-process activity pay.

In the above cases, the heterogeneities between the views can be resolved during the integration process. But, the views could be contradictory as they define different ways how to process objects. Such differences cannot be resolved during integration. In this case, the system integrator has to decide to take the definition of one view and to ignore the respective part of the object life-cycle defined in the other view. Notice, that the other view must be changed such that it can fulfill the criterion that it is an observation consistent generalization of the integrated object life-cycle.

6.3 Types of Integration

We defined two kinds of integration before: (1) *integration by specialization* and (2) *integration by generalization*. In this section, the integration method is discussed for both cases.

In this section, we assume that two behavior diagrams B_1 defined in view v_1 and B_2 defined in view v_2 are given, where v_1 and v_2 are the *standardized* *view schemas* to integrate. During integration, heterogeneities due to *granularity conflicts, view-specific aspects, view-specific alternatives,* and *naming conflicts* are resolved.

6.3.1 Integration by Specialization

During integration by specialization, an object life-cycle $\tilde{B}$ for object type $\tilde{o}$ is defined in the integrated schema, which represents the integration of the object life-cycles of B_1 of o_1 and B_2 of o_2 in view v_1 and view v_2, respectively, where $\tilde{B}$ has to the fulfill correctness criteria defined in this section.

We will first give a motivating example to show how the object life-cycles of equivalent object types can deviate from each other and how these heterogeneities are resolved. This discussion will help us to define a correctness criterion for integration by specialization.

6.3.1.1 Motivating Example

In this section, we present a small example where three views on the business process of room reservations are integrated. We will point out how the integrated object life-cycle should be defined intuitively, and will discuss how the integrated object life-cycle should *not* look like. Finally, this discussion should motivate the correctness criteria defined in Section 6.3.1.2.

Consider the object life-cycles shown in Figure 6.2: Each object life-cycle shows a view on the object life-cycle of object type RESERVATION. Object types V1, V2 and V3 are considered equivalent, such that their object life-cycles are integrated by specialization. View v1 represents the view of a user who has to handle requests for reservations and is responsible for all work before the user actually arrives at the hotel. View v2 is defined by the members of the reception, who deal with check-in, the use of the room, and check-out. View v3 is defined by a person who is in charge of statistics; he/she has to select randomly a set of guests to ask them about their expectations on the stay, their evaluation of the hotel, etc. If a guest is selected for statistics, activity registerStatData is executed, otherwise activity waiveStatData is executed for the respective instance of RESERVATION.

First, correspondences are determined: for simplicity, equivalent elements carry the same name; inclusions between one activity or state and a subnet are indicated by an activity or state symbol that frames the respective subnet. Further, there is an inclusion between label r in view v2 and view v3 and labels rs and ro in view v1.

When considering the correspondences and differences between views, several topics about definition of views and integration of views arise:

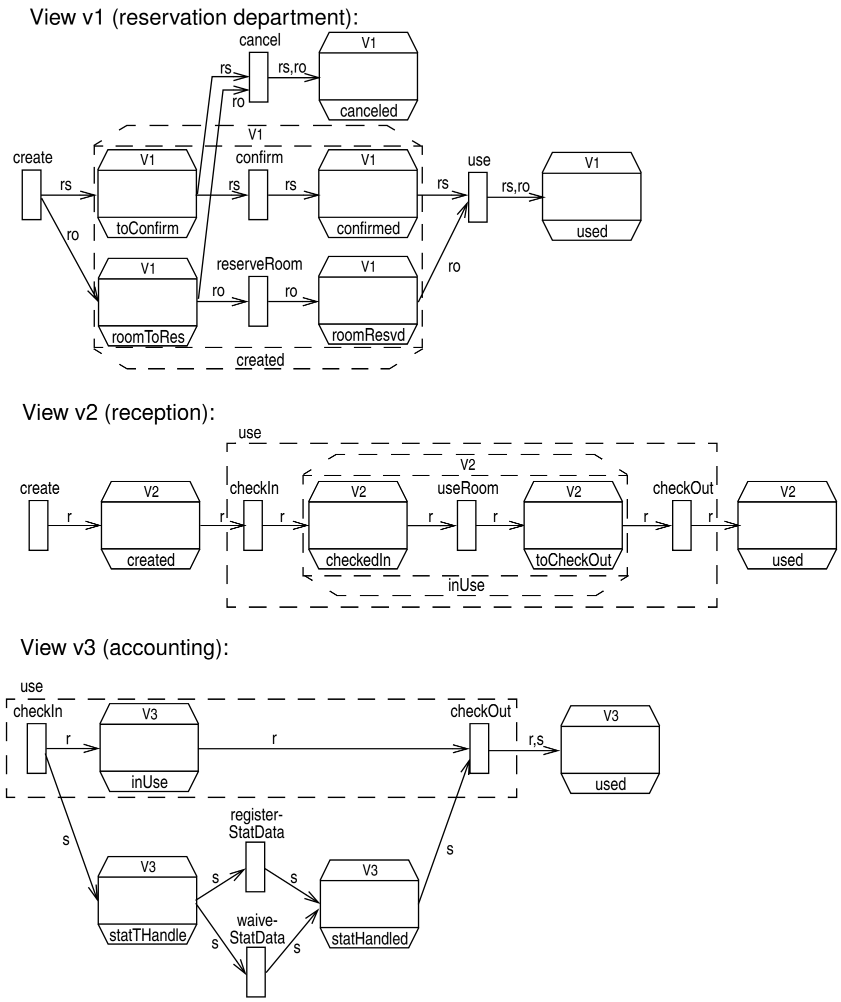


Figure 6.2: Three views of reservations

Granularity. Users who perform some work in an organization and who have to define a view on the business process, will know about their own work in more detail than about work done by other users. This means that granularity conflicts, where an abstract element is defined in one view that corresponds to a subnet in the other view, are quite usual in view integration.

In our example, activity use is abstractly defined in view v1 as the respective user will know that the room will be used, but does not know which activities are executed during use of a room. Further, the abstract state created is defined in view v2, as the reception group does not know what happens to a reservation object between creation (activity create) and check-in (activity checkIn).

View-specific aspects. For the same reason, elements may be completely missing in a view. An element is called missing in our approach if it is not defined in a view, neither in full detail nor abstracted. In our integration approach, elements that are defined in one view but missing in another view are defined in the former as view-specific aspects, which means that they are represented by a parallel chunk that is defined in parallel to the common aspects of the life-cycle diagram. This parallel chunk must be labeled with unique labels. These unique labels indicate that the parallel chunk is unique in this view.

In views v1 and v2, there is no information about statistics, which is defined in v3. Therefore, label s does not appear in v1 and v2. The reason will be that the users who defined v1 and v2 are not aware of the person who has to handle statistics.

View-specific alternatives. View-specific alternatives are alternatives that are labeled with at least one common label and that are not defined in all views. This means, that a view provides an alternative that is not supported by the other views. According to observation consistent specialization, view-specific alternatives must be removed in the integrated object life-cycle as a view that does not include this alternative could not observe processing of an object in the integrated object life-cycle according to this alternative. There may be several reasons for view-specific alternatives: (1) A user defines an alternative in a view although this alternative cannot be invoked on any object. (2) The compared object types are not equivalent. (3) An alternative has been "forgotten" in a view, perhaps because it is not used in the current flow of work in the organization, i.e., its object life-cycle does not model a pure business process, but a workflow. In cases (2) and (3), the decision about equivalence of object types must be revised or the workflow aspects must be extracted from the view schema, respectively. In case (1), the view-specific alternative must be omitted as it is obviously unnecessary. We will only consider case (1) in the following as cases (2) and (3) have to be handled in other steps of the integration process (i.e., by inter-type standardization and extraction of business processes as discussed in Sections 4.2.2 and 4.2.1).

In view v1 of our example, a view-specific alternative consisting of activity cancel and state canceled is defined.

Pre- and poststates. Each activity in the view and in the integrated schema has pre- and poststates. Different users may have different views on the pre- and poststates of an activity, where each view defines necessary pre- and poststates, but neither view needs to know the complete set of pre- and poststates of an activity. An activity can actually be started if all prestates that are defined in any view are fulfilled. Consider activity checkOut in views v2 and v3: this activity can be invoked in state toCheckOut according to v2 and in state statHandled in v3. In the integrated schema, we define that an object must be in both states toCheckOut *and* statHandled so that checkOut can be started.

Somebody may argue that a user who executes an activity in everyday business should know *when* he/she can execute the activity, i.e., the user should know about the prestates that must hold on an object such that the activity can be executed. Then, this user should be able to define the complete prestates of this activity in a view schema. Consider the definition of activities checkIn and checkOut in view v2: Although the modeler of v2 (i.e., the receptionist) performs these activities, he/she defines activity checkIn only with an abstract prestate and activity checkOut without considering prestate statHandled, which is defined in v3.

Yet, we argue that someone who performs an activity need not necessarily be aware of *all* prestates and need not be aware of prestates *in full detail*. Instead, the satisfaction of prestates may be ensured by other users: In the example views, the receptionist is not aware of the statistics aspect and thus cannot verify whether a reservation of a guest who wants to check out is in state statHandled. The statistician, who selects a subset of the guests as a random sample to ask them about their stay, has to make sure himself that he catches the selected guests during the time between check-in and check-out. The receptionist does not consider state statHandled as a prestate of checkOut as at any time, when a hotel guest comes to the reception to check out, it is guaranteed that statistics have been handled before.

Notice that the view schemas may reflect how the processing of objects *should* work, i.e., it describes the *correct processing.* Yet, there may be violations in everyday business, e.g., the statistician cannot get hold of the guest before check-out and thus cannot ask him/her; then, some undesirable ad-hoc exception handling may be started (e.g., as only a subset of guests is asked, the guest who escaped will be assigned to that subset of guests who are *not* asked; or a questionnaire is sent to the guest's home address). An information system that contains the whole business process within a database and controls all activities performed on an object can check whether all preconditions for an activity are fulfilled and can inform the receptionist if activity check-out cannot be started as the statistics have not been handled so far.

Another reason for prestates that are not perceived by a user who performs an activity in everyday business is that the *workflow* ensures that prestates are fulfilled if an activity is started: Someone does some work on a business case, which may be represented by a paper form in conventional work, as soon as the paper form lies in his/her inbox. Then, this user will know that the prestates to perform the activity are fulfilled without knowledge about what happened before and which prestates are actually defined for this activity. In the database, these business cases are represented by instances of an object type that are processed according to the life-cycle diagram of this object type. In the analogy with paper forms, one paper form corresponds to one object.

In our example, activity checkIn can only be executed on an object if the object resides in states confirmed and roomResvd. Yet the receptionist who executes checkIn does not know about these two states as he/she does not know how a reservation is processed before check-in. Consider the process shown in Figure 6.3 (a): This process shows how reservations have been treated in everyday business. It is modeled neither as an object life-cycle of OBD nor as a workflow diagram as it neither represents a business process nor a workflow according to our definition. In everyday business, a reservation is created, it is confirmed, and a room is reserved. After confirming and reserving, someone puts the reservation into a file; if a guest comes to the reception to check-in, the receptionist reads this file to check whether a reservation exists. Yet, the receptionist does not know, *when* the reservation entry has been put into the file and what happened before. The only thing that he knows is that he can execute check-in if the reservation is recorded in the file.

As we represent only the aspects of business processes in the conceptual schema, but do not define the actual flow of work, we will omit activity writeResFile in the business process. The reason is that writeResFile is an activity that exists only as the processing of business cases is not supported

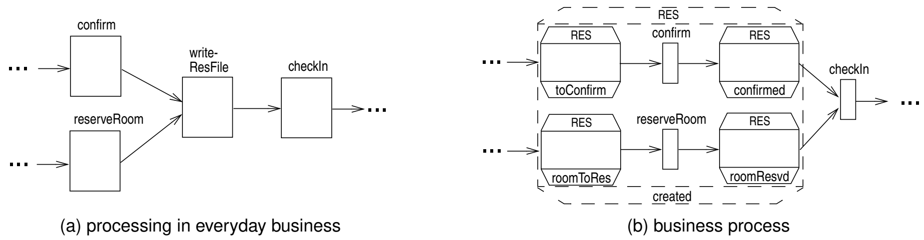


Figure 6.3: Processing in everyday business and business process

by an integrated information system. If reservations are handled in an information system, where business processes are defined, workflows with boxes are defined based on the business processes, and the contents of personal to-do lists of users is defined based on some authorization concept, no such intermediate activities like writeResFile are necessary any more. Therefore, the process shown in Figure 6.3 (a) will be transformed into a business process as shown in part (b) of the figure. Yet, the receptionist who does not know about prestates confirmed and roomResvd and the activities performed before can only specify an abstract prestate for checkIn, such that the view schema of view v2, shown in Figure 6.2, will result.

The integrated object life-cycle. The integrated object life-cycle must contain all information of the views at the finest level of granularity and the views must be able to observe correct processing of objects that are treated in the integrated object life-cycle. Therefore we require that the integrated object life-cycle is an *observation consistent specialization* of the respective object life-cycles in the views (and, conversely, the views are observation consistent generalizations of the conceptual schema). As processing of objects in the integrated schema according to a certain alternative cannot be correctly observed in a view if this alternative is not defined in this view, we require that view-specific alternatives are omitted in the integrated object life-cycle (which follows the rules of observation consistent specialization). For our example, the integrated object life-cycle is shown in Figure 6.4.

Observation consistent specialization allows to define all elements that are refined in one view and unrefined in another view at the most detailed level of granularity, and allows to define all elements that are defined in one view as a view-specific aspect, but are missing in another view; view-specific alternatives are excluded. If observation consistent specialization were not required, the views could not observe processing of objects according to the integrated schema. Consider the object life-cycle in Figure 6.5: If states roomToRes

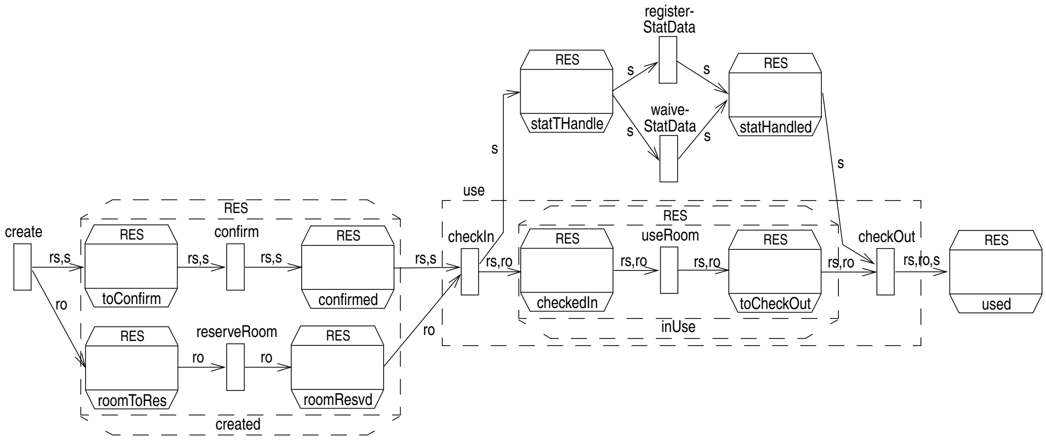


Figure 6.4: Integration of the example views

and roomResvd as well as activity reserveRoom of view v1 were not defined in the integrated object life-cycle, view v1 could not derive information when an object resides in one of these states or is processed by this activity as activity reserveRoom were never invoked on a reservation object in the integrated schema. If only activity use were defined in the integrated object life-cycle instead of checkIn, checkedIn, useRoom, toCheckOut, and checkOut, information from view v2 would be lost. The activities that are executed at the reception desk were missing and could not be invoked on a business object. If the integrated object life-cycle contained a view-specific alternative, which is defined in view v1 in our example and consists of activity cancel and state canceled, views v2 and v3 could not observe behavior of an object that is processed according to this alternative.

It is not sufficient to require that the integrated object life-cycle is an observation consistent specialization of the object life-cycles in the views. As elements in the views may correspond to each other, we require that only corresponding elements between the views are represented by the same elements in the integrated schema. We will refer to this criterion as *correspondence compliance*. In the following, we will only consider *correspondence compliant* object life-cycles as integrated object life-cycles.

Consider Figure 6.6 as an example: In part (a) of the figure, there are two parts of an object life-cycle, where the upper one is defined in view v1 and the lower one is defined in v2. Suppose that equal names indicate equivalences: The integrated object life-cycle can be defined equally as the views, as both views define only equivalent elements. But consider part (b) of the figure: Both views define activity payCash, but activities payByCheque and

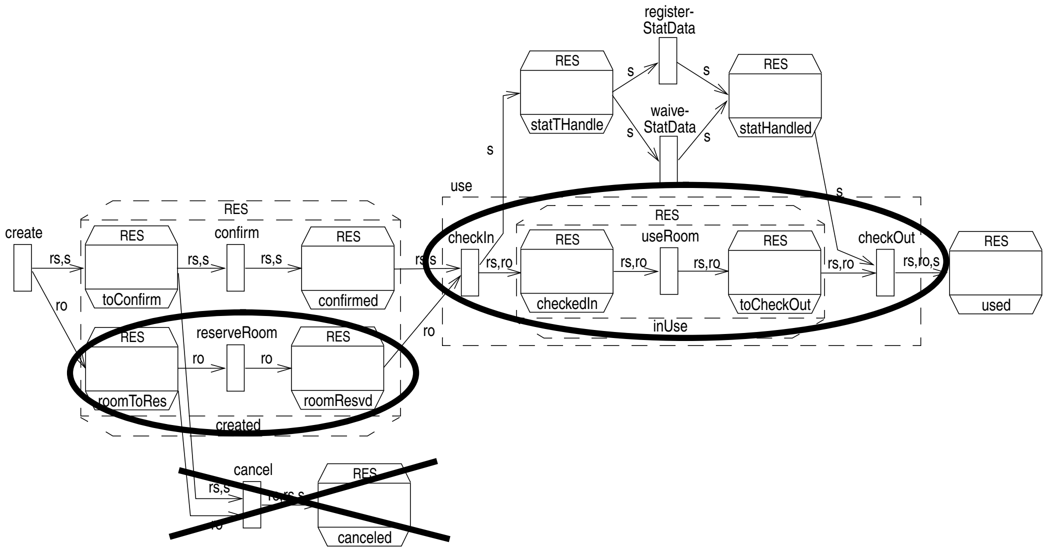


Figure 6.5: Violation of observation consistent specialization

payByCreditCard are view-specific alternatives. It is not allowed to represent these two activities by the same activity in the integrated schema. Thus, only activity payCash will be defined in the integrated schema, whereas the other activities are omitted as view-specific alternatives.

The criterion of *observation consistent specialization* is not sufficient for correctness of the integrated object life-cycle. In the introduction of this work, we argued that the specialization must be “minimal”, this means, the integrated object life-cycle $\tilde{B}$ must only be specialized as far as necessary to fulfill the criterion of observation consistent specialization of B_1 and B_2, i.e., there must not be another object life-cycle B' that is an observation consistent specialization of B_1 and B_2, too, but that is an observation consistent generalization of $\tilde{B}$. We will refer to this “minimal” specialization as *most general* specialization.

We introduced how an object life-cycle can be changed during observation consistent specialization (cf. Section 2.4.4): (1) by inheritance without change, (2) by refinement, (3) by extension, and (4) by elimination of alternatives. Our correctness criterion means that there is no B', such that $\tilde{B}$ results from B' by inserting refinements and extensions, or by removing alternatives. Only inheritance without change is allowed, which means that B' is equal to $\tilde{B}$.

Consider the object life-cycle in Figure 6.4, which is the correct integrated object life-cycle RES with most general specialization, whereas the object life-

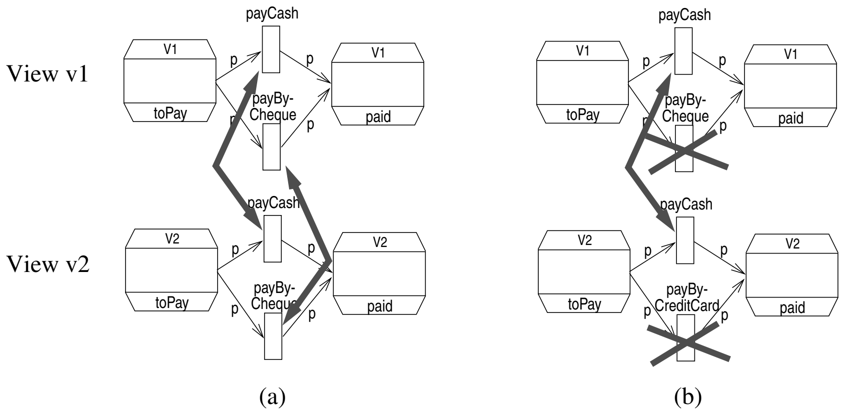


Figure 6.6: View-specific and common alternatives

cycle RES* shown in Figure 6.7 is an observation consistent specialization of the views, too, but with unnecessary specialization: State used is refined to toArchive, archive, and archived by observation consistent refinement; an additional label p with several activities and states is defined by observation consistent extension. Further, activity waiveStatData is omitted in RES* (and therefore depicted with dashed borders). Object life-cycle RES* is no most general observation consistent specialization of the object life-cycles defined in views v1, v2, and v3 as there is an object life-cycle RES (as defined in Figure 6.4) that is an observation consistent specialization of the object life-cycles in the views and an observation consistent generalization of RES*.

Suppose that $\tilde{B}$ is an observation consistent specialization of B_1 and B_2. We want to find out whether there is another object life-cycle B' that is an observation consistent specialization of B_1 and B_2, too, and that is an observation consistent generalization of $\tilde{B}$. We need not consider any arbitrary object life-cycle B'; instead we require that B' and $\tilde{B}$ are defined based on the same set of correspondences between B_1 and B_2. This requirement is quite intuitive as it is not useful to compare $\tilde{B}$ and B' if they were defined based on completely different assumptions about correspondences.

This means that such a more general object life-cycle B' is only found if each element $\tilde{e}$ in $\tilde{B}$ is defined based on the same elements in B_1 and B_2 as element e' in B' where $\tilde{e}$ corresponds to e' according to a specialization function between $\tilde{B}$ and B', i.e., $h(\tilde{e}) = e'$.

Consider the following example as a motivation (cf. Figure 6.8):

Suppose that the object life-cycles of V1 and V2, which are shown in part (a) of the figure, must be integrated, where elements with equal names are equivalent. Then, the intuitively correct integrated object life-cycle RES

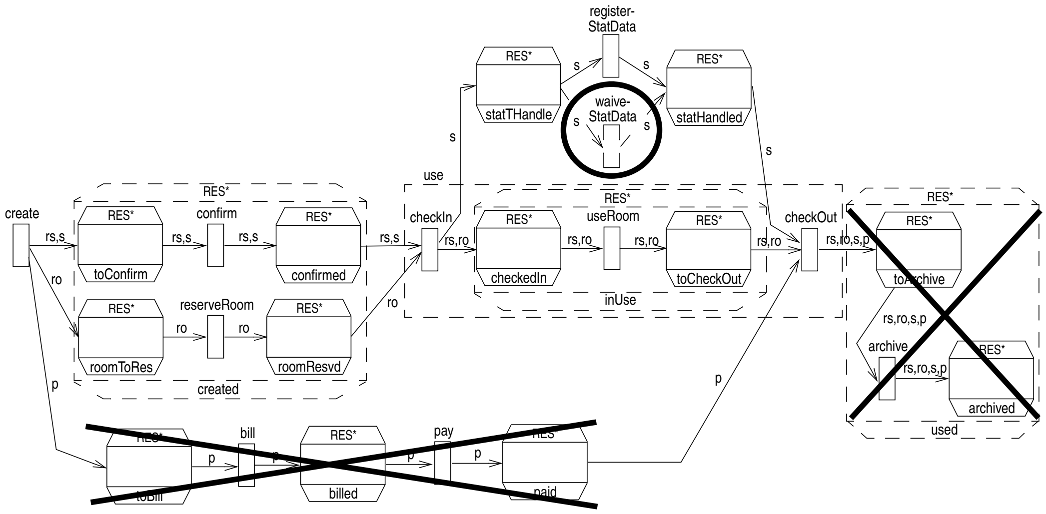


Figure 6.7: Violation of most general specialization

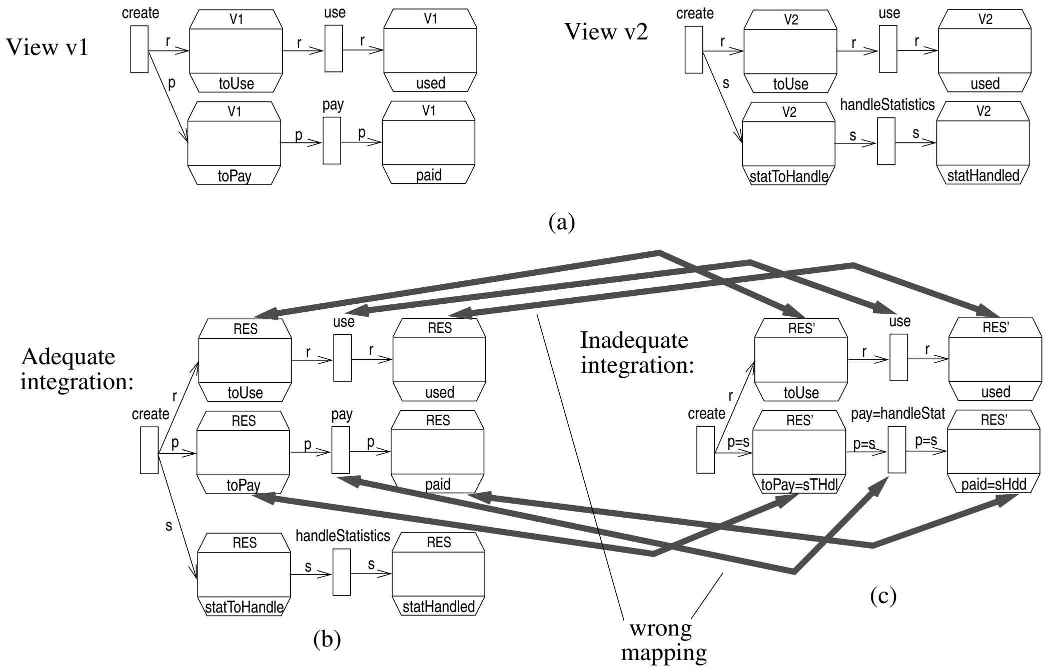


Figure 6.8: Adequate and inadequate specialization

will be defined as shown in part (b) of the figure. When looking for an object life-cycle, which is a more general specialization of V1 and V2 than RES is, we could try to define an object life-cycle RES' as shown in part (c). There, labels p of V1 and s of V2 as well as the activities and states labeled with p and s are considered equivalent as indicated by the names of the elements. We could define a specialization function between RES' and V1 as well as a specialization function between RES' and V2 (these functions can be easily determined by comparing the names of the elements in the figure); further, we could define a specialization function h between RES and RES' and argue that RES' is a more general specialization than RES. This function h could be defined as indicated by the dark-grey arcs between the elements of RES and RES' in the figure. Label s and all activities and states labeled with s could be considered as an observation consistent extension.

This case need not be considered as RES and RES' consider different correspondences. The mapping h is completely unintuitive, e.g., as activity handleStatistics in RES was orginally defined because of activity handleStatistics in V2. In RES', activity pay=handleStatistics corresponds to handleStatistics in V2, but handleStatistics in RES would not correspond to pay=handleStatistics in RES' according to h.

Therefore, the criterion formulated above is quite intuitive and necessary as such object life-cycles like RES' are not considered as possible more general specializations of the object life-cycles in the views.

6.3.1.2 Integration: Correctness and Process

In this section, we define the correctness criterion for integration by specialization and give an overview of the integration process. We refer to the object life-cycles in the standardized view schemas that must be integrated as B_1 and B_2 and refer to the integrated object life-cycle as $\tilde{B}$.

Correctness criterion. We summarize the correctness criteria motivated by the previous example. If two object life-cycles $B_1 = (S_1, T_1, F_1, L_1, l_1)$ and $B_2 = (S_2, T_2, F_2, L_2, l_2)$ are integrated by specialization, resulting in an integrated object life-cycle $\tilde{B} = (\tilde{S}, \tilde{T}, \tilde{F}, \tilde{L}, \tilde{l})$, the following properties must hold:

Observation consistent specialization: There must be a function h_1 and a function h_2, where for $i \in \{1, 2\} : h_i : \tilde{S} \cup \tilde{T} \cup \tilde{L} \to S_i \cup T_i \cup L_i \cup \{\varepsilon\}$, such that $\tilde{B}$ is an observation consistent specialization of B_1 with specialization function h_1 and $\tilde{B}$ is an observation consistent specialization of B_2 with specialization function h_2.

Correspondence compliance: If an element $\tilde{e}$ in $\tilde{B}$ is related to an element e_1 in B_1 and an element e_2 in B_2, it must hold that e_1 and e_2 correspond to each other. This correspondence will be denoted by a correspondence relation Θ in our approach. An object life-cycle is a *correspondence-compliant* observation consistent specialization of B_1 and B_2 if it is an observation consistent specialization of B_1 and B_2 and it fulfills correspondence compliance.

Most general specialization: There is no object life-cycle $B' = (S', T', F', L', l')$, where $B' \neq \tilde{B}$, such that B' is an correspondence-compliant observation consistent specialization of B_1 and B_2, where the specialization between B' and B_1 (and B_2) is represented by specialization function h'_1 (and h'_2, respectively), $\tilde{B}$ is an observation consistent specialization of B' with specialization function h, and where elements defined in the views are specialized consistently in B and B', i.e., $\forall i \in \{1,2\}, e' \in S' \cup T' \cup L', \tilde{e} \in \tilde{S} \cup \tilde{T} \cup \tilde{L} : h(\tilde{e}) = e' \Rightarrow (h_i(\tilde{e}) = h'_i(e'))$.

Object life-cycle $\tilde{B}$ is defined as follows:

1. All common elements of B_1 and B_2 are integrated at the most detailed level of granularity. This rule follows from *observation consistent refinement,* where an element can be refined in the specialized object life-cycle.

2. All unique labels and all elements of B_1 and B_2 that are labeled with unique labels only, are taken into the integrated object life-cycle, which follows the rule of *observation consistent extension.*

3. View-specific alternatives of B_1 and B_2, i.e., all unique activities and states that are labeled with at least one common label, are omitted according to the rule of partial inheritance of *observation consistent extension.*

The integrated object life-cycle is defined in the following steps, which are illustrated in Figure 6.9.

1. *Decomposition:* Common elements and view-specific aspects are integrated in two separate steps. To determine the sub-views to be considered in these steps, each view B_i (for $i \in \{1,2\}$) is decomposed into (1) a *common sub-view* B_i^c, which contains the restriction of the view to all common labels and all activities, states, and arcs that are labeled with at least one common label, and (2) a *unique sub-view* B_i^u, which

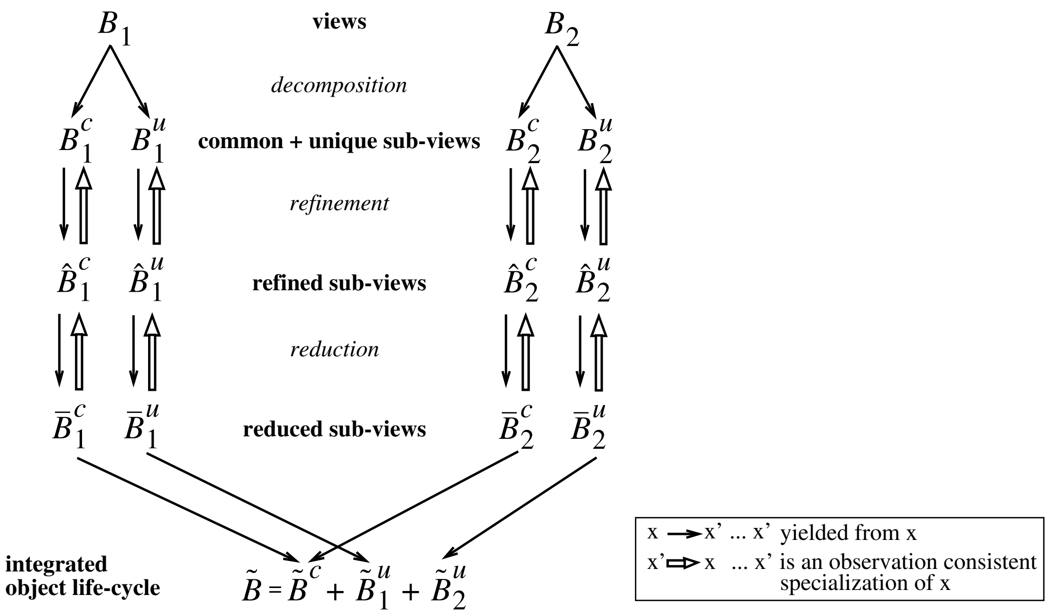


Figure 6.9: Integration by specialization

consists of all unique labels and all activities, states, and arcs that are labeled with at least one unique label. These sub-views may be overlapping, as an activity, state, or arc may be labeled with common *and* unique labels. Notice that by decomposing a view into a common and a unique sub-view no information is added or removed.

2. *Integration of common elements:* The common sub-views B_1^c and B_2^c are integrated to the *common view* $\tilde{B}^c$. To resolve granularity conflicts, the common sub-views B_1^c and B_2^c are refined to a common level of granularity according to the rules of observation consistent refinement yielding *common refined sub-views* $\hat{B}_1^c$ and $\hat{B}_2^c$. To resolve heterogeneities due to *view-specific alternatives*, alternatives with common labels in one view that are not modeled in the other view are omitted from $\hat{B}_1^c$ and $\hat{B}_2^c$, whereby the rules of observation consistent extension are obeyed because of the rule of *partial inheritance*. The resulting *reduced sub-views* $\bar{B}_1^c$ and $\bar{B}_2^c$ are identical up to renaming. To resolve naming conflicts, a common view $\tilde{B}^c$ that is isomorphic to $\bar{B}_1^c$ and $\bar{B}_2^c$ is defined, where the elements of $\tilde{B}^c$ have different identities than their counterpart elements of $\bar{B}_1^c$ and $\bar{B}_2^c$ and possibly different names.

3. *Integration of view-specific aspects:* For each *unique sub-view* B_i^u, activities and states that are included in the unique sub-view B_i^u as well

as in the common sub-view B_i^c and that are refined in $\hat{B}_i^c$, are refined in B_i^u, too, yielding a *refined unique sub-view* $\hat{B}_i^u$. Then for each B_i^u, view-specific alternatives that have been omitted in the reduced common subview $\bar{B}_i^c$ are omitted from the refined unique sub-view $\hat{B}_i^u$, too, yielding a *reduced unique sub-view* $\bar{B}_i^u$. *Naming conflicts* between $\bar{B}_1^u$ and $\bar{B}_2^u$ are resolved — as described above for the common sub-views — yielding $\tilde{B}_1^u$ and $\tilde{B}_2^u$. Overlaps between B_i^c and B_i^u are taken into account in the definition of the identities of $\bar{B}_i^u$ in that an element in $\tilde{B}^c$ and an element in $\tilde{B}_i^u$ that have the same counterpart in B_i receive the same identity.

Heterogeneities due to *view-specific aspects* are resolved in that the integrated business process $\tilde{B} = (\tilde{S}, \tilde{T}, \tilde{F}, \tilde{L}, \tilde{l})$ is defined as composition $\tilde{B} := \tilde{B}^c + \tilde{B}_1^u + \tilde{B}_2^u$ (cf. Definition 24).

Definition 24 (Composition of behavior diagrams) *The* composition *of object life-cycles $B_a = (S_a, T_a, F_a, L_a, l_a)$ and $B_b = (S_a, T_b, F_b, L_b, l_b)$ yields a behavior diagram $B_c = (S_c, T_c, F_c, L_c, l_c)$, where $S_c = S_a \cup S_b, T_c = T_a \cup T_b, F_c = F_a \cup F_b, L_c = L_a \cup L_b, l_c = \{(f, X) | f \in F_c \wedge X = l_a(f) \cup l_b(f)\}$.*

As the common sub-view B_i^c and the unique sub-view B_i^u are refined the same way and as $\hat{B}_i^c$ is an observation consistent refinement of B_i^c and $\hat{B}_i^u$ is an observation consistent refinement of B_i^u, it holds that $\hat{B}_i = \hat{B}_i^c + \hat{B}_i^u$ is an observation consistent refinement of B_i (cf. Figure 6.10). Since $\bar{B}_i^c$ is an observation consistent extension of $\hat{B}_i^c$, and since $\bar{B}_i^u$ is an observation consistent extension of $\hat{B}_i^u$, it holds that $\bar{B}_i = \bar{B}_i^c + \bar{B}_i^u$ is an observation consistent extension of $\hat{B}_i$. As the common sub-views $\bar{B}_1^c$ and $\bar{B}_2^c$ are isomorphic up to renaming and as the set of labels of the common sub-views are disjoint with the set of labels of each unique sub-view $\bar{B}_1^u$ and $\bar{B}_2^u$ and as the labels of the unique sub-views are disjoint, too, it holds that the integrated business process $\tilde{B} = \tilde{B}^c + \tilde{B}_1^u + \tilde{B}_2^u$ is an observation consistent specialization of B_1 and B_2.

6.3.2 Integration by Generalization

Integration by generalization means that an object type $\check{o}$ is defined in the integrated schema that is a supertype of two existing object types o_1 and o_2. The object life-cycle $\check{B}$ of $\check{o}$ has to be defined in a way that the object life-cycle B_1 of o_1 and the object life-cycle B_2 of o_2 are observation consistent specializations of $\check{B}$. As the rule of original definition of elements has to be fulfilled, object life-cycle $\check{B}$ has to contain *all* common elements of B_1 and B_2.

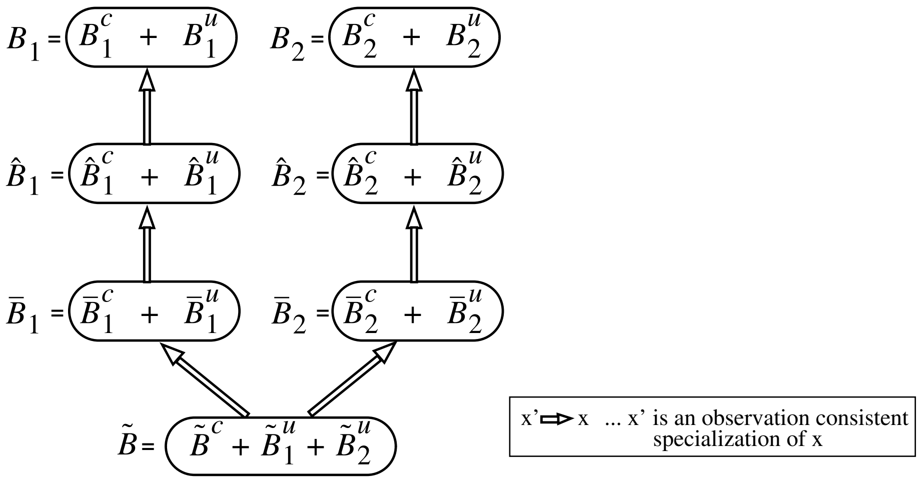


Figure 6.10: Relationships between object types during integration by specialization

Thus, we require that the integrated object life-cycle $\check{B}$ is the *most special* observation consistent generalization of B_1 and B_2.

There is a very important difference between integration by specialization and integration by generalization: In the former, only the object life-cycle that results from integration is taken into the conceptual schema, i.e., if B_1 and B_2 are defined in views and $\tilde{B}$ results from integration, only $\tilde{B}$ is defined in the conceptual schema. In the latter, the object life-cycles B_1 and B_2 as well as the common supertype with object life-cycle $\check{B}$ that results from integration are defined in the conceptual schema. Thus, B_1 and B_2 exist already in the conceptual schema at the time when object life-cycle $\check{B}$ is defined.

We will first present a motivating example for integration by generalization and then define the correctness criteria and the overall integration steps.

6.3.2.1 Motivating Example

Suppose that two similar object types R-STD and R-VIP are defined in the conceptual schema that represent structure and behavior of standard reservations and vip reservations, respectively (cf. Figure 6.11 and 6.12). In the conceptual schema, a common supertype RESVT must be defined, which includes the common elements of R-STD and R-VIP. Notice that the object life-cycle of R-STD is equal to the object life-cycle shown in Figure 6.4. During integration of n view schemas, this situation may often occur: An object

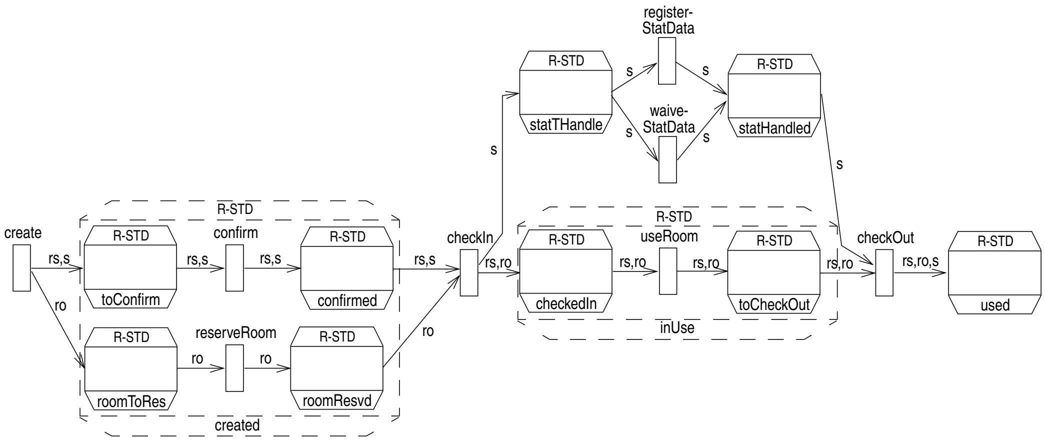


Figure 6.11: Object life-cycle of R-STD

life-cycle is defined during integration by specialization and is the basis for integration by generalization in a subsequent binary integration step.

Several heterogeneities may be detected, which are resolved as follows:

Granularity conflicts. Different object life-cycles may define information at different levels of detail (inclusion) or may refine some abstract activity, state, or label in different ways (set correspondence). The reason for inclusions may be that some activities are invoked on instances of one object type, whereas nothing happens with instances of the other object type. Consider the abstract state used in R-VIP: After check-out, the manager (or someone else) says goodbye to the vip guests as a preferential treatment. This service is not defined for standard guests, such that only one single state used is defined in R-STD. Further, the abstract state inUse is refined differently. State created and label r, which are defined in R-VIP without being further refined, are refined to a subnet and a set of labels rs and ro, respectively, in R-STD. The reason may be that the object life-cycle of R-STD considers more details than the object life-cycle of R-VIP.

View-specific aspects. View-specific aspects must be labeled with unique labels as they represent parallel extensions to the object life-cycle of the supertype (which is to be defined during integration). In the object life-cycle of R-VIP, a unique label v and some activities and states labeled with v are introduced to represent special vip services, which are only rendered with vip reservations.

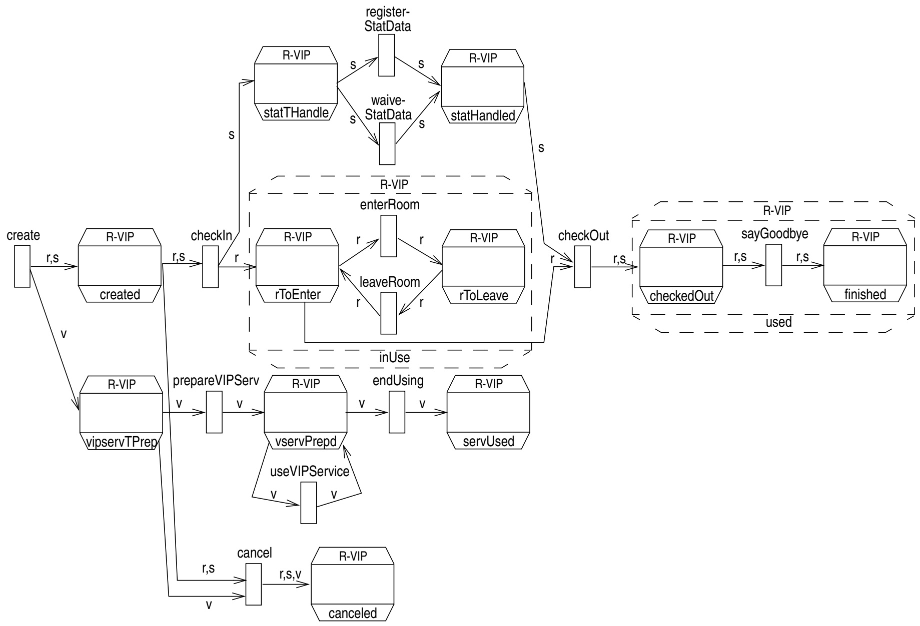


Figure 6.12: Object life-cycle of R-VIP

View-specific alternatives. Alternatives defined in one view may be missing in the other view. This situation arises if an alternative is defined in the supertype and inherited in one view, but omitted according to partial inhertiance in the other view. Consider the examples: vip reservations can be canceled (cf. activity cancel and state canceled), whereas standard reservations cannot be canceled.

The integrated object life-cycle. The integrated object life-cycle $\check{B}$ is defined with the common supertype of the object types whose object life-cycles B_1 and B_2 are integrated. Therefore, $\check{B}$ must be an *observation consistent generalization* of the given object life-cycles, where it must include all common elements as specialized as possible.

This means that elements that are defined in different detail in B_1 and B_2 are defined at the lower level of granularity in $\check{B}$, corresponding subnets and corresponding sets of labels are abstracted to single elements. View-specific extensions are omitted by observation consistent restriction, whereas view-specific alternatives are included. Object life-cycles B_1 and B_2 must be observation consistent specializations of $\check{B}$, where elements of $\check{B}$ can be refined in each B_i, view-specific aspects can be added, and view-specific alternatives can be removed. The object life-cycle of RESVT that is the common

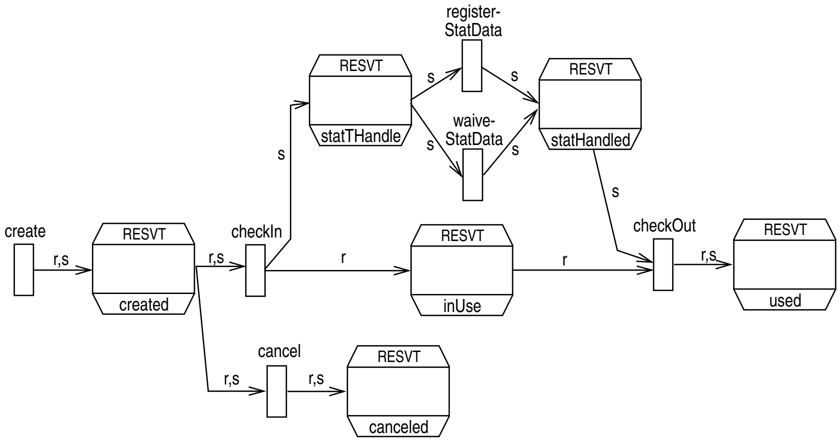


Figure 6.13: Object life-cycle of RESVT

supertype of R-STD and R-VIP is shown in Figure 6.13.

We require that any elements e_1 in B_1 and e_2 in B_2 are represented in $\check{B}$ by the same element e if and only if they correspond to each other. This requirement corresponds to the criterion of *correspondence compliance* defined for integration by specialization.

Similar as the most general specialization must be determined during integration by specialization, the *most special* generalization must be determined during integration by generalization. Thus, no unnecessary abstractions or restrictions are allowed and no additional alternatives can be inserted. Suppose that an object life-cycle is defined that deviates from the object life-cycle shown in Figure 6.13 in that label s and all activities and states that are only labeled with s are removed and elements checkIn, inUse, and checkOut are abstracted to one single activity use. This object life-cycle is no correct integration by generalization.

If an object life-cycle B' is supposed to be a more special observation consistent generalization of B_1 and B_2 than $\check{B}$, it must be based on the same set of correspondences between elements as $\check{B}$. This requirement has been defined for integration by specialization, too.

6.3.2.2 Integration: Correctness and Process

Correctness criterion. Object life-cycle $\check{B} = (\check{S}, \check{T}, \check{F}, \check{L}, \check{l})$ is correctly defined from B_1 and B_2 according to integration by generalization if the following properties hold:

Observation consistent generalization: There must be a function h_1 and a function h_2, where for $i \in \{1,2\} : S_i \cup T_i \cup L_i \rightarrow \check{S} \cup \check{T} \cup \check{L} \cup \{\varepsilon\}$, such that B_i is an observation consistent specialization of $\check{B}$ with specialization function h_i.

Correspondence compliance: We require that an element e_1 in B_1 and an element e_2 in B_2 are only represented by the same element e in B if e_1 and e_2 correspond to each other. An object life-cycle is a correspondence compliant observation consistent generalization of B_1 and B_2 if it is an observation consistent generalization of B_1 and B_2 and it fulfills the criterion of correspondence compliance.

Most special generalization: There is no object life-cycle $B' = (S', T', F', L', l')$, where $B' \neq \check{B}$, such that B' is a correspondence-compliant observation consistent generalization of B_1 and B_2 with specialization function h'_1 and h'_2, respectively, and B' is an observation consistent specialization of $\check{B}$, where $\forall i \in \{1,2\}, e \in S_i \cup T_i \cup L_i, e' \in S' \cup T' \cup L' : h'_i(e) = e' \Rightarrow h_i(e) = h(e')$.

Object life-cycle $\check{B}$ is defined as follows:

1. All common elements are represented in $\check{B}$ at the lower level of granularity, elements may be refined in each B_i according to *observation consistent refinement.*

2. All unique labels as well as activities and states that are only labeled with unique labels are removed in $\check{B}$. View-specific aspects in B_i, where a unique label is defined are allowed according to *observation consistent extension.*

3. All alternatives that are defined in only one B_i, i.e., unique activities and states that are labeled with at least one common label, are taken into $\check{B}$. This corresponds to the rule of *observation consistent extension,* which allows *partial inheritance.* That B_i that does not contain this alternative is an observation consistent extension of $\check{B}$ because of the rule of partial inheritance.

The integrated object life-cycle is defined in the following steps. These steps are summarized in Figure 6.14:

1. *Restriction:* Object life-cycle B_i is restricted to $\dot{B}_i$, which comprises all common labels and all elements of B_i that are labeled with at least one common label. Thereby, *view-specific aspects* are removed. This step is

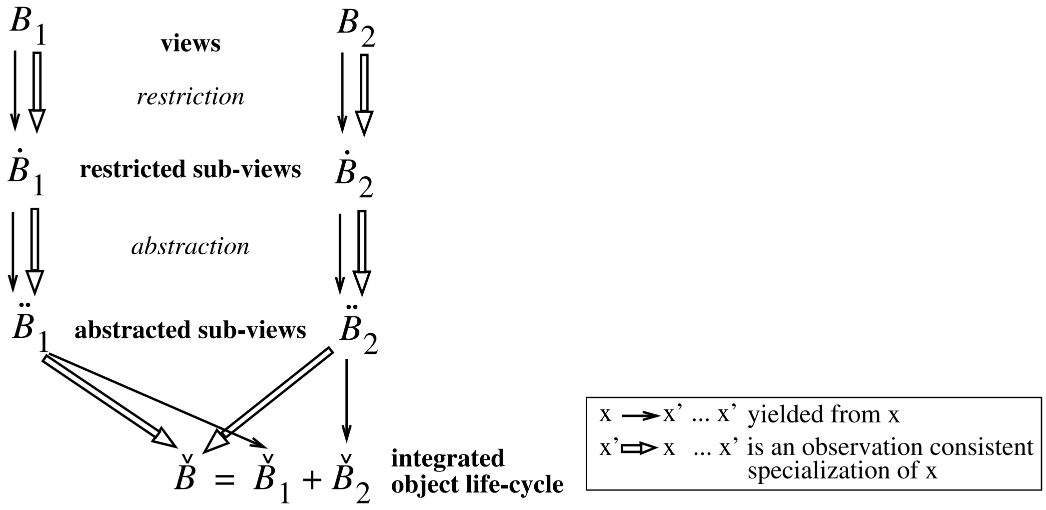


Figure 6.14: Integration by generalization

equal to the definition of the common sub-views during *decomposition* defined in integration by specialization (cf. Section 6.3.1).

2. *Abstraction:* The elements of $\dot{B}_i$ are abstracted to a common level of granularity. The resulting object life-cycles are refered to as $\ddot{B}_1$ and $\ddot{B}_2$. During that step, *granularity conflicts* are resolved as after abstraction, all elements that are defined with the object life-cycles either correspond to an element of the other object life-cycle by element equivalence or are unique elements.

3. *Composition:* Object life-cycle $\check{B}$ is defined by defining an element for each pair of equivalent elements and an element for each unique element. Unique elements can only exist because of view-specific alternatives as unique elements because of view-specific aspects have already been omitted during restriction.

   As all unique elements stem from view-specific alternatives and these elements are integrated in $\check{B}$, heterogeneities due to *view-specific alternatives* are resolved. *Naming conflicts* are resolved by defining an object life-cycle $\check{B}_i$ for each $\ddot{B}_i$, where the former is isomorphic to the latter. The only difference between $\check{B}_i$ and $\ddot{B}_i$ is that each activity, state, and label of $\ddot{B}_i$ gets a new "identity" in $\check{B}_i$. Correspondences between $\ddot{B}_1$ and $\ddot{B}_2$ are considered, as corresponding elements get the same identity. The integrated object life-cycle $\check{B}$ is defined as the composition of $\check{B}_1$ and $\check{B}_2$, denoted as $\check{B} = \check{B}_1 + \check{B}_2$ (cf. Definition 24). The correspondence of elements of $\check{B}$ with elements in each B_i is defined by the specialization function $h_i : S_i \cup T_i \cup L_i \to \check{S} \cup \check{T} \cup \check{L} \cup \{\varepsilon\}$, not by the names of the elements.

6.4 Integration Process

In this section, the steps of the integration process for object life-cycles are described in detail. We will define how the object life-cycles are transformed in each step and which heterogeneities are resolved. Although these steps are described sequentially, there may be several iterations between them as decisions in some step may be revised at a later point in the process.

Example: We will use views on object life-cycles of reservations for integration by specialization and by generalization. The respective object life-cycles will be shown later in this section.

6.4.1 Overview of the Process Steps

The core of the integration process, integration by generalization and specialization, has already been introduced before. In order to be able to define an integrated object life-cycle, we need to prepare the views of the object life-cycles.

The following steps are defined for integrating object life-cycles:

1. *Intra-type standardization:* The process step of intra-type standardization, which was introduced in Section 4.2 consists of several sub-steps. The result of intra-type standardization are *standardized view schemas*.

   (a) *Standardization of activity specifications:* Object life-cycles and activity specifications are standardized, where an activity that can be invoked in different variants is replaced by several alternative activities or vice versa; thereby, *activity–port conflicts* are resolved.

   (b) *Standardization of conditions:* Pre- and poststates defined with activities are replaced by textual pre- and postconditions or vice versa. Thus, *condition–state conflicts* are resolved in this sub-step. Notice that the decision whether a condition should be represented by a state or a textual condition must be taken by the system integrator. The main goal for this step is to define comparable kinds of conditions equally.

   (c) *Schema enrichment:* During schema enrichment, labels and redundant states are inserted and the assignment of labels may be defined or changed. Thereby, heterogeneities because of *inconsistent states and labels* are resolved.

2. *Determination of correspondences:* Correspondences between the views of object life-cycles are determined and represented in a correspondence relation.

3. *Integration:* The object life-cycles are integrated by generalization or specialization as discussed in Section 6.3. The remaining heterogeneities are resolved in this step. The result of this step is an *integrated schema.* As already introduced, two types of integration, i.e., integration by specialization and integration by generalization can be used:

   - *Integration by specialization:* This type of integration is used to define an object life-cycle in the integrated schema if a pair of *equivalent* object types is defined in the views.
   - *Integration by generalization:* This type of integration is used if a pair of *similar* object types is defined in the views. Then, both object types are taken into the integrated schema and a common supertype is defined by generalization.

   Notice that each object type in the integrated schema results either from integration by specialization or by integration by generalization or is taken into the conceptual schema without being integrated with another object type. Thus, for each object type, at most one kind of integration must be applied.

In this integration process, pairs of object types are integrated. Thereby, all object types in the integrated schema are defined. Yet, supertypes and subtypes in a type hierarchy in the integrated schema must be related to each other. This problem is treated in Section 6.5: (1) A specialization function h must be determined between an object type o' and its supertype o in the integrated schema. (2) As discussed in the introduction, elements that are defined in all subtypes of an object type in the integrated schema must be defined in the supertype, too (cf. *generalization of elements* discussed in the introduction).

The steps of intra-type standardization are independent of the integration step as they modify object types within one view. Intra-type standardization can be executed on unique object types, which are not integrated with any other object type, too, to improve the schema. Determination of correspondences is only applied to object types that have to be compared and integrated with another object type. There are slight differences between determination of correspondences for integration by specialization and integration by generalization.

As intra-type standardization is independent of integration and determination of correspondences for integration by specialization and for integration by generalization differs in small details only, these steps will not be discussed separately for each type of integration. Yet, the discussion of the integration step is divided to separate sections for integration by specialization and integration by generalization.

6.4.2 Intra-type Standardization

Intra-type standardization comprises the following steps: (1) standardization of activity specifications, (2) standardization of conditions, and (3) schema enrichment. Notice that intra-type standardization is performed for the object life-cycle of each object type within each view separately. Yet, it must be ensured that the view schemas stay correct in that the rules of observation consistent specialization are obeyed after intra-type restructuring. This means that if an object type is restructured, the object life-cycle of its supertype and subtypes must be restructured consistently.

6.4.2.1 Standardization of Activity Specifications

During normalization of activity specifications, one single activity a is replaced by a set of alternative activities A' if activity a can be invoked in different variants that are distinguished by different values for some input ports of a. Thereby, *activity–port conflicts* are resolved.

Activity–port conflicts arise if one view defines a set of alternative activities A', whereas the other view defines one single activity a that represents the same concept as activities A', but different alternatives in processing are defined in different ways: In one view, each alternative of processing is represented by a separate activity $a' \in A'$, whereas in the other view, there is one single activity a, where different processing is defined via one or more input ports (so-called *discriminating ports*) whose domains are small sets of values, where each possible combination of values assigned to these ports corresponds to one alternative.

Example: Consider the activity specifications shown in Figure 6.15 and suppose that the activity specification in part (a) of the figure is defined

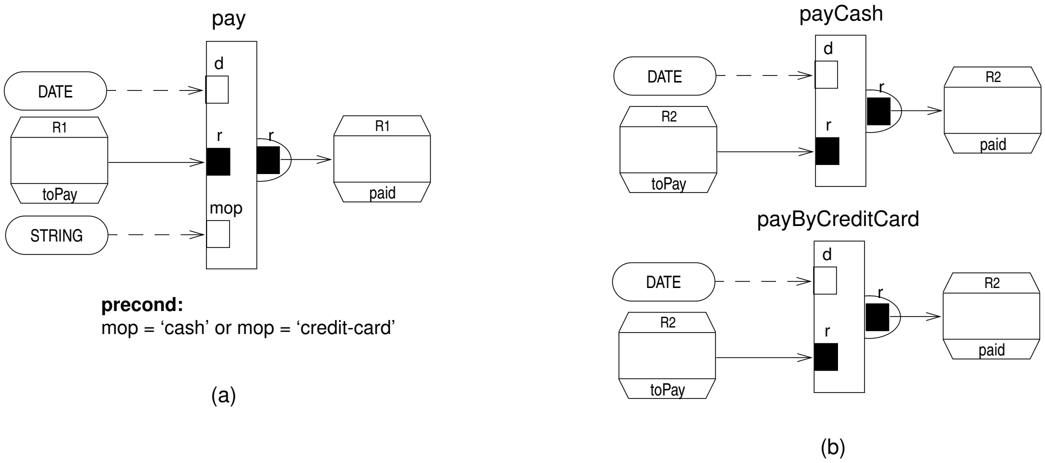


Figure 6.15: Example: activity–port conflicts

in one view, whereas the activity specifications in part (b) are defined in the other view. In case (a), the form of payment is defined by port mop, which gets the actual value cash or credit-card, whereas in case (b), different methods of payment are represented by alternative activities, such that the paramenter specifying the method of payment can be omitted.

Activity–port conflicts can be compared with a similar problem in the relational model, where similar information can either be represented in a separate table or distinguished by attributes within a table (cf. [KS91b], *table-versus-attributes conflicts*). Further, it is comparable with the problem treated with *subtyping* (cf. Section 5.2.3), where similar objects can either be represented as instances of different object types with a common supertype or as instances of the same type with a set of discriminating properties. During inter-type standardization, different object types were introduced instead of one object type with discriminating properties.

For activity–port conflicts, we use a similar approach as for subtyping: One activity with discriminating ports is replaced by a set of alternative activities. The new alternative activities have the same activity specification as the original activity, except that the discriminating ports are omitted.

Example: In our example shown in Figure 6.15, the single activity defined in part (a) of the figure will be replaced by two alternative activities as shown in part (b).

We can extend the problem to the case that a set of alternative activities A' in one view corresponds to a set of alternative activities A'' in the other view, but no $a' \in A'$ corresponds to a single activity $a'' \in A''$ or to a set of activities $\hat{A}'' \subseteq A''$. Further, a textual condition can be used for splitting of activities instead of the discriminating ports defined before. This means that in difference to the case before, where each value of the discriminating ports constitutes an alternative, an activity can be split according to a precondition of the activity.

Example: Consider the activities defined in two views, shown in part (a) and part (b), respectively, of Figure 6.16. The first view defines three alternatives of payment, by cash, credit card, or money transfer, where money transfers cannot be initiated on the day of check-out, but only later (cf. the precondition of activity payByMoneyTransfer, where r.to refers to the day of check-out).

The second view distinguishes two alternatives with payment at once (on the day of check-out) or later. If the customer pays at once, only cash and credit cards are accepted; if he/she pays later, credit cards and money transfer are accepted.

This more general case is resolved similar to the first case. Each activity can be split either by using discriminating ports or a precondition. If an activity is split according to a precondition, all ports are taken into the activities that result from splitting and the precondition that holds for each of these activities is added to the activity specification. The goal of this splitting is that *1 : 1* correspondences can be determined between the activities defined in the views.

Example: The activities are standardized as follows: activity payByCreditCard in the first view is split to payByCCAtOnce and payByCCLater; activity payAtOnce is split to payCashAtOnce and payByCCAtOnce, activity payLater is split to payByCCLater and payByMTLater. Then, activities payCash, payByCCAtOnce, payByCCLater, and payByMoneyTransfer in the first view correspond to activities payCashAtOnce, payByCCAtOnce, payByCCLater, and payByMTLater, respectively, in the second view. As activity payByCreditCard is split because of a precondition, no discriminating port can be omitted.

6.4.2.2 Standardization of Conditions

During standardization of conditions, *condition–state conflicts* are resolved by replacing textual pre- and postconditions by pre- and poststates, respectively, or vice versa. There are no unique rules in which cases a condition should be defined by states or textually.

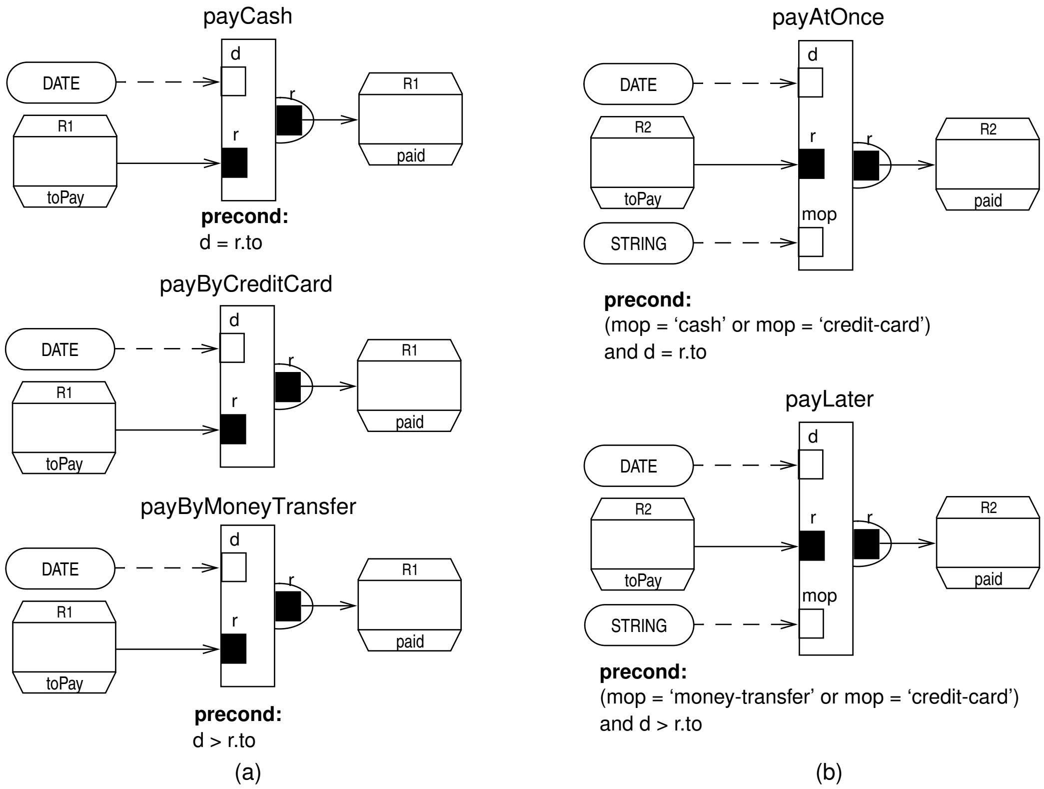


Figure 6.16: Example: activity–port conflicts with two sets of activities

Suppose that an object resides in a state from which several alternative activities consume. The decision which alternative has to be chosen may be free, i.e., all of these alternatives can be invoked on the object. Yet, preconditions may be defined with each alternative that refer to some properties of the considered object and that restrict the set of alternatives that can be invoked on the object. Further, the alternatives may have additional, different prestates. The set of allowed alternatives depends on which of these prestates are marked with the object. In the case of preconditions the set of allowed alternatives depends on the values of the object's properties that have been defined in previous activities; in the case of prestates the set of allowed alternatives depends on the question into which states previous activities produced the object. Thus, in both cases, the set of alternatives depends on the previous processing of the object.

Example: Consider the examples shown in part (a) and (b) of Figure 6.17, respectively, each one representing a part of the object life-cycle of reservations: In both cases, a deposit can be cashed by activity payDeposit or waived by execution of activity waiveDeposit. If the reservation is canceled (cf. activity cancel), the object moves into state canceled. There are two alternative activities refundDeposit and payNoRefund, where the former is invoked if a deposit has been cashed and the latter is invoked if no deposit has been cashed before. Thus, the decision whether refundDeposit or payNoRefund is invoked depends on whether activity payDeposit or waiveDeposit has been executed before.

In case (a), this dependency is defined by two states depPaid and depWaived. In case (b), pre- and postconditions are defined textually: A postcondition is defined with activity payDeposit that says that property deposit of the reservation object is set to a positive value, whereas the postcondition of waiveDeposit requires that the value of property deposit is zero. The value of property deposit is checked in the preconditions of refundDeposit and payNoRefund such that the same sequences of activity invocations are allowed as in example (a). Notice that the value of deposit must not be changed in activity cancel.

During standardization of conditions, the system integrator must decide which representation should be prefered. The decision whether prestates and preconditions are equivalent may in many cases be more difficult than in the example above, particularly, if pre- or postconditions are missing.

Suppose that there is a state s that is a poststate of activity a_1 and a prestate of activity a_2. If the system integrator decides to replace state s by textual conditions, he/she must add a postcondition to a_1 and a precondition

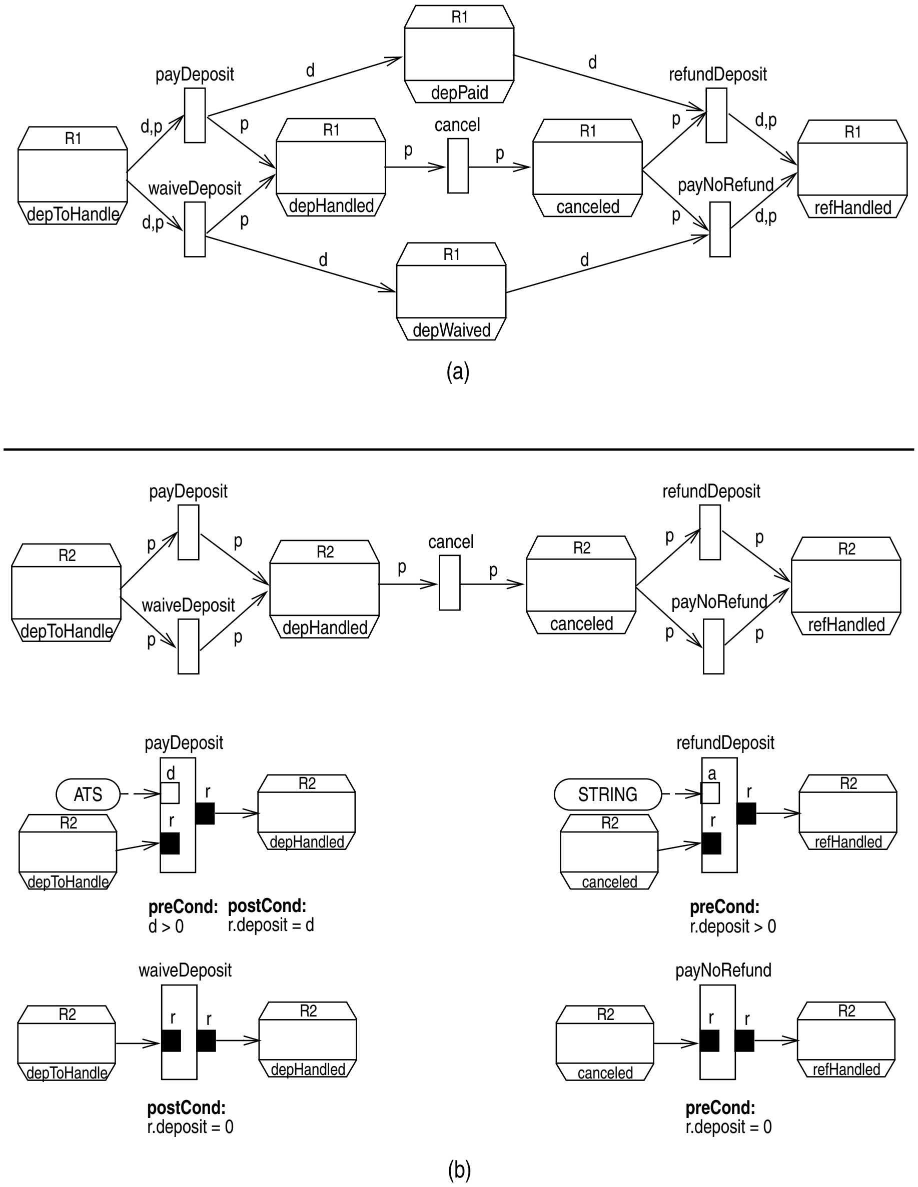


Figure 6.17: Example: condition–state conflicts

to a_2; the postcondition states that a certain condition holds after execution of a_1 whereas the precondition of a_1 requires a condition to hold that is only true if a_1 has been executed before. These conditions will probably refer to the properties of the object on which the activity is invoked.

Example: Consider once more Figure 6.17: If the object life-cycle as shown in part (a) of the figure is replaced by the object life-cycle shown in part (b), the omitted state depPaid is replaced by a postcondition in payDeposit and a precondition for refundDeposit.

If conditions are replaced by states, the system integrator has to examine whether an activity a_2 can only be invoked if a_1 has been executed at some time before as the precondition of a_2 is not fulfilled otherwise.

Example: Suppose that an object life-cycle shown in part (b) of Figure 6.17 is given. The precondition for payNoRefund can only be fulfilled if waiveDeposit has been executed before as no other alternative sets r.deposit to 0.

6.4.2.3 Schema Enrichment

Schema enrichment means that the object life-cycles to integrate are defined in a way that correspondences between the object life-cycles can be determined and the subsequent integration step can be invoked. During schema enrichment, heterogeneities because of *inconsistent states and labels* are detected and resolved.

The goal of schema enrichment is that corresponding labels are assigned to corresponding activities and states, i.e., if a label x in one view is assigned to a set of activities and states E, then the corresponding label x' (or a corresponding set of labels X') in the other view is assigned to a set of elements E', where the elements of E correspond to the elements of E'.

First, we give some guidelines how states and labels should be defined and second, we will introduce operations how an object life-cycle can be restructured to resolve inconsistent definitions of states and labels.

The following guidelines should be considered for schema enrichment:

1. Redundant states and labels should be avoided if possible. It is the best to define as few labels and states as possible because object life-cycles may become difficult to read if they are overloaded with redundant elements.

2. Unique activities and states in a view, i.e., activities and states that are not defined in the other view, neither at same level of granularity, nor refined or abstracted, and that do not constitute a view-specific alternative, should be labeled with unique labels to constitute a view-specific aspect.

3. A redundant state is used to define a processing state of a certain aspect of business objects that is represented by a certain label. In the analogy of paper forms, each label may be considered as a "part" of the business object. A label should only be assigned to a subnet of activities or states if it is considered in this subnet. It is inadequate to assign a label to a subnet of activities and states if this label is not considered there. Instead, it should be assigned to another parallel chunk or to a redundant state that is defined in parallel to the subnet.

4. The system integrator will have to find out which activities and states a label passes from creation to archivation. It is useful to define states and labels in a way that corresponding labels are assigned to corresponding activities and states in both views. For that, the operations introduced in the following can be used.

During schema enrichment, object life-cycles are restructured, where the set of sequences of possible activity invocations must not be changed. We will allow to change the object life-cycles by the following operations: (1) inserting and removing redundant states, (2) inserting and removing labels, and (3) changing assignment of labels to states and activities.

The system integrator has to restructure the object life-cycles following the analogy with paper forms (which has been discussed in Section 2.4). Of course, some inconsistencies in labeling will be detected when correspondences are determined, thus, some iterations between schema enrichment and the subsequent step of identification of correspondences may be necessary.

Inserting and removing redundant states: Redundant states are those states that do not restrict the possible sequences of activity invocations. Redundant states can be inserted in the object life-cycle or removed from the object life-cycle. If a redundant state is inserted, it has to be labeled with at least one label. Therefore, either a new label is introduced that is assigned to the new state or the assignment of an existing label is changed. If a state is removed, the assigned labels have to be removed, too (see below), or they are assigned to parallel states and activities.

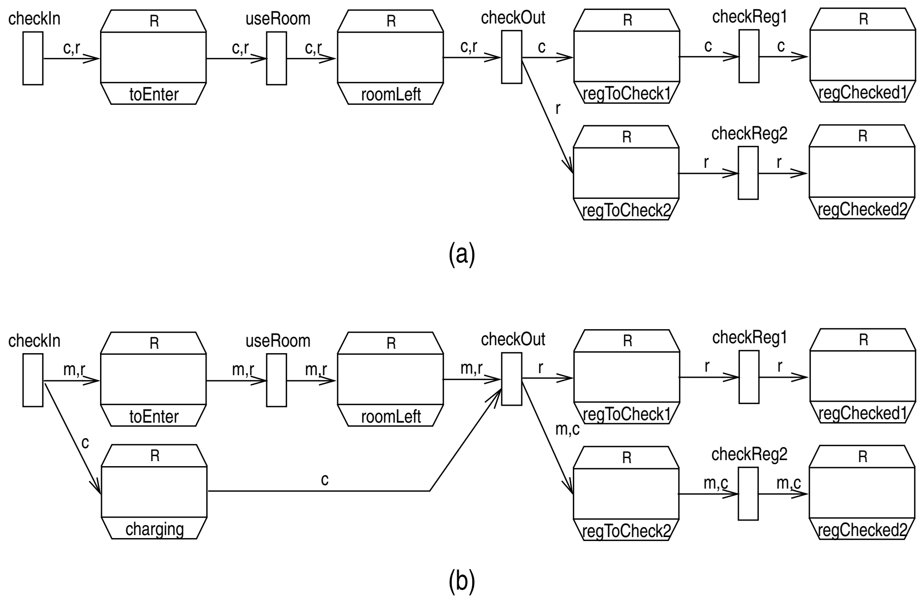


Figure 6.18: Changes of object life-cycles during schema enrichment

Example: Figure 6.18 (a) shows a part of an object life-cycle. The system integrator finds out that label c (representing the aspect of charging services in the object life-cycle) is not considered for the use of the room (i.e., for activity useRoom). Therefore, a redundant state charging is inserted and label c is assigned to the new state in the object life-cycle shown in part (b).

Inserting and removing labels: A label can be inserted into an object life-cycle or removed from an object life-cycle if the labeling properties are obeyed. As an object life-cycle has to be correctly labeled before a label is inserted, the new label is not necessary for correctness and thus not essential. Inserting a label may be useful as in a subsequent step, a redundant state is defined for which a label is needed; then, the new label can be reassigned to the new redundant state. The new label need not be assigned to the same arcs as one of the existing labels.

A label x must not be removed if there is at least one arc where label x is the only assigned label.

Example: In the object life-cycle shown in Figure 6.18 (b), an additional label m is inserted. Notice, that the assignment of this m to arcs need not correspond to the assignment of any existing label.

If label m existed before, it could be removed as the labeling properties were not violated. Label r could not be removed, as some arcs only carry label r.

Changing assignments of labels: The assignment of labels to arcs may be changed during restructuring if labeling properties are obeyed.

Example: Consider once more Figure 6.18 (a), particularly the labeling of arcs (checkOut, regToCheck1), (regToCheck1, checkReg1), (checkReg1, regChecked1), (checkOut, regToCheck2), (regToCheck2, checkReg2), and (checkReg2, regChecked2). In part (b), the assignment of labels c and r to these arcs is exchanged without violating labeling properties.

This new assignment of labels may be useful to provide for a consistent labeling with the other view to integrate: Suppose that one view assigns labels c and r as shown in Figure 6.18 (a), whereas the other view assigns labels as shown in Figure 6.18 (b). During the subsequent steps of integration, elements with the same names are considered equivalent. We will require later in this chapter that corresponding labels must be assigned to corresponding activities and states. If label c is assigned to checkReg1 and label r is assigned to checkReg2, whereas in the other view, labels are assigned the other way round, a conflict will arise.

In our previous works [PS98a, PS98b], we assumed that the given object life-cycles of the views are unlabeled, which may be useful as the system integrator can define labeling from scratch and need not restructure given label assignments. The system integrator has to define all labels and redundant states where we require that the sequences of activity invocations of the unlabeled and the labeled object life-cycle are the same.

More formally, the following conditions must hold: (1) Each sequence of starts and completions of activities in the unlabeled object life-cycle $\breve{B} = (\breve{S}, \breve{T}, \breve{F})$ must be a valid sequence of starts and completions of activities in the enriched object life-cycle $B = (S, T, F, L, l)$, too, and vice versa. (2) For each life-cycle occurrence $\breve{\gamma} = [\breve{\sigma}_1, \ldots, \breve{\sigma}_n]$ of $\breve{B}$ and each life-cycle occurrence $\gamma = [\sigma_1, \ldots, \sigma_n]$ of B, where $\breve{\gamma}$ and γ result from the same sequences of $n-1$ starts and completions of activities, it must hold that $\forall 1 \leq i \leq n : \breve{\sigma}_i = \{s \in S | \exists (s, x) \in \sigma_i\}$.

Example: As an example for integration by specialization, we use two object life-cycles that represent views on the business process of reservations. The reception defines object type R1 (shown in Figure 6.19),

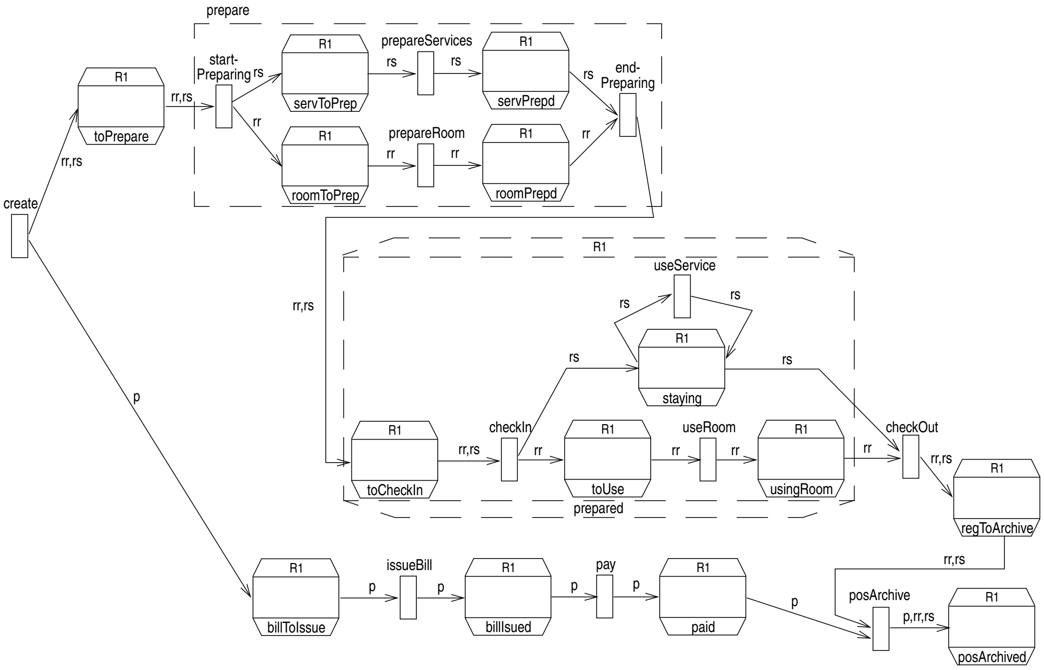


Figure 6.19: View schema of R1 (reception)

whereas the accounting department defines object type R2 (shown in Figure 6.20). Object types R1 and R2 are equivalent such that their object life-cycles must be integrated by specialization.

Consider the object life-cycle shown in Figure 6.20. It shows the view of reservations defined by the accounting department. Suppose that no labels are defined and the states depicted by dotted borders are missing. Further, ignore the state symbols with dashed borders and the brackets around label c. During schema enrichment, labels c, p, and r are introduced to represent the aspects of *c*harging, *p*ayment, and *r*egistration, respectively. To provide for a correctly labeled object life-cycle, redundant states regFinished and chgFinished are introduced. Intuitively, activities stopCharging and issueBill are synchronized only by the aspect of charging, but not by the aspect of registration. Therefore, only label c is assigned to state chargStopped, which is defined as a poststate of stopCharging and as a prestate of issueBill, whereas label r is assigned to the new state regFinished, which indicates that the registration aspect is not considered any more except in activity posArchive.

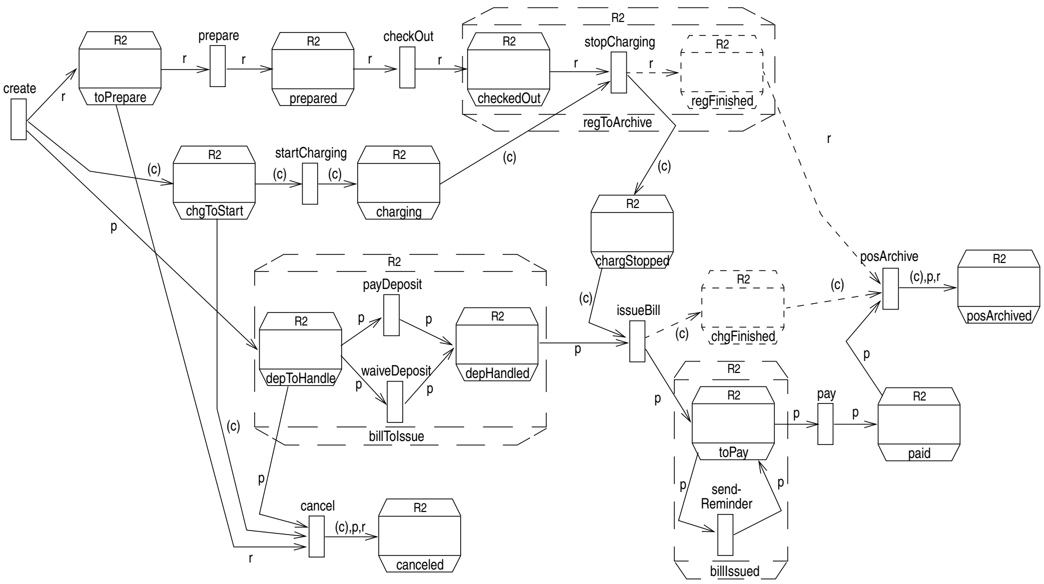


Figure 6.20: View schema of R2 (accounting department)

6.4.3 Identification of Correspondences

Correspondences between activities, states, and labels of the object life-cycles have to be determined. Correspondences are equivalences, inclusions, and set correspondences as defined in Section 6.2. Determination of correspondences is equal for integration by specialization and integration by generalization except that no set correspondences are allowed in the former.

We require that correspondences are not overlapping, i.e., that each element or subnet E_1 in a view corresponds to at most one element or one subnet E_2 in the other view. Overlapping correspondences may occur if activities and states within the given views have been insufficiently refined such that there are aspects that are defined at an abstract level in both views. In such a case, the views have to be refined such that for each element there is a most-refined representation in at least one view.

For the remainder of the integration process, we define the correspondence relation $\Theta := \Theta_S \cup \Theta_T \cup \Theta_L$ for the standardized views B_1 and B_2, where $B_i = (S_i, T_i, F_i, L_i, l_i)$, for $i \in \{1, 2\}$, and where Θ_S, Θ_T, and Θ_L represent correspondences between states, activities, and labels, respectively (cf. Table 6.1). For each of these correspondence relations, we distinguish *(1) equivalences* Θ^E (i.e., two single elements correspond to each other), *(2) left inclusions* Θ^{I_1} (i.e., a single element in B_1 corresponds to a subnet in B_2), *(3) right inclusions* Θ^{I_2} (i.e., a single element in B_2 corresponds to a subnet

$\Theta_S^E \subseteq$	$S_1 \times S_2$
$\Theta_S^{I_1} \subseteq$	$S_1 \times (2^{S_2 \cup T_2} \setminus \{\emptyset\})$
$\Theta_S^{I_2} \subseteq$	$(2^{S_1 \cup T_1} \setminus \{\emptyset\}) \times S_2$
$\Theta_S^S \subseteq$	$(2^{S_1 \cup T_1} \setminus \{\emptyset\}) \times (2^{S_2 \cup T_2} \setminus \{\emptyset\})$
$\Theta_S :=$	$\{(\{e_1\},\{e_2\}) \vert (e_1,e_2) \in \Theta_S^E\} \cup \{(\{e_1\},E_2) \vert (e_1,E_2) \in \Theta_S^{I_1}\} \cup$ $\{(E_1,\{e_2\}) \vert (E_1,e_2) \in \Theta_S^{I_2}\} \cup \Theta_S^S$
$\Theta_T^E \subseteq$	$T_1 \times T_2$
$\Theta_T^{I_1} \subseteq$	$T_1 \times (2^{S_2 \cup T_2} \setminus \{\emptyset\})$
$\Theta_T^{I_2} \subseteq$	$(2^{S_1 \cup T_1} \setminus \{\emptyset\}) \times T_2$
$\Theta_T^S \subseteq$	$(2^{S_1 \cup T_1} \setminus \{\emptyset\}) \times (2^{S_2 \cup T_2} \setminus \{\emptyset\})$
$\Theta_T :=$	$\{(\{e_1\},\{e_2\}) \vert (e_1,e_2) \in \Theta_T^E\} \cup \{(\{e_1\},E_2) \vert (e_1,E_2) \in \Theta_T^{I_1}\} \cup$ $\{(E_1,\{e_2\}) \vert (E_1,e_2) \in \Theta_T^{I_2}\} \cup \Theta_T^S$
$\Theta_L^E \subseteq$	$L_1 \times L_2$
$\Theta_L^{I_1} \subseteq$	$L_1 \times (2^{L_2} \setminus \{\emptyset\})$
$\Theta_L^{I_2} \subseteq$	$(2^{L_1} \setminus \{\emptyset\}) \times L_2$
$\Theta_L^S \subseteq$	$(2^{L_1} \setminus \{\emptyset\}) \times (2^{L_2} \setminus \{\emptyset\})$
$\Theta_L :=$	$\{(\{e_1\},\{e_2\}) \vert (e_1,e_2) \in \Theta_L^E\} \cup \{(\{e_1\},E_2) \vert (e_1,E_2) \in \Theta_L^{I_1}\} \cup$ $\{(E_1,\{e_2\}) \vert (E_1,e_2) \in \Theta_L^{I_2}\} \cup \Theta_L^S$

Table 6.1: Correspondence relations

in B_1). *(4) set correspondences* Θ^S (i.e., two sets of elements correspond to each other).

The correspondence relations defined in Table 6.1 result from the following considerations:

Correspondences between states: An equivalence between states defined by Θ_S^E exists if one state of B_1 is equivalent to one state in B_2. A left inclusion exists if a single state of B_1 corresponds to a subnet of activities and states in B_2. Notice that it must hold that this subnet can be abstracted to a single state and the object life-cycle that results from abstraction is an observation consistent abstraction of the object life-cycle before abstraction. This rules holds analogously for right inclusions. Set correspondences in Θ_S^S mean that two subnets correspond to each other, where both subnets can be abstracted to a single state.

Correspondences between activities: The same rules hold as for correspondences between states with the following remarks: The definition of Θ_S^T seems to be equal to Θ_S^S as in both cases, two subnets correspond to each other. Yet, a subnet may only be refered to in Θ_S^T if it can be abstracted to an activity.

Correspondences between labels: Correspondences between labels only need to consider labels within the correspondence relation (whereas the former had to consider activities *and* states).

The relations Θ_S, Θ_T, and Θ_L are defined to be able to handle equivalences, inclusions, and set correspondences, together.

As overlapping correspondences are not allowed, no element may appear in more than one pair of corresponding elements, i.e., $\forall(E_1', E_2') \in \Theta, (E_1'', E_2'') \in \Theta : (E_1', E_2') \neq (E_1'', E_2'') \Rightarrow (E_1' \cap E_1'' = \emptyset \wedge E_2' \cap E_2'' = \emptyset)$. If object life-cycles are integrated by specialization, no set correspondences are defined, i.e., $\Theta^S = \emptyset$.

Further, we require that all common activities and states are labeled with at least one common label, i.e., for $i \in \{1, 2\}$, it holds that $\forall(E_1, E_2) \in \Theta_S \cup \Theta_T, e \in E_i \; \exists(X_1, X_2) \in \Theta_L : \lambda_i(e) \cap X_i \neq \emptyset$.

In the remainder of this chapter, we will use a predicate $hasCorr_\Theta(e)$, which is true if and only if element e is included in a correspondence relation Θ and which is defined as follows:

$$\text{for } i \in \{1, 2\}, e \in S_i \cup T_i \cup L_i : hasCorr_\Theta(e) :\Leftrightarrow \exists(E_1, E_2) \in \Theta : e \in E_i$$

Example: In views R1 and R2 (cf. Figures 6.19 and 6.20), several activities, states, and labels are equivalent. For simplicity, they carry the same names, e.g., activity create, state toPrepare, and label p. Activity prepare in R2 corresponds to a subnet of activities, states, and arcs in R1, and label r in R2 corresponds to labels rr and rs of R1 (right inclusion). State billToIssue in R1 corresponds to the subnet consisting of depToHandle, payDeposit, waiveDeposit, and depHandled in R2 (left inclusion). The subnets involved in element inclusions are depicted within a symbol with dashed borders indicating the corresponding element of the other view. Set correspondences do not exist in this example as the object life-cycles are integrated by specialization.

Activity cancel and state canceled of R2 are unique because of a view-specific alternative, whereas activity startCharging, state charging, label c, and other elements are unique because of a view-specific aspect. In the figures, unique labels are put within brackets.

6.4.4 Integration

We distinguish two types of integration, which are discussed in separate sections now: integration by specialization and integration by generalization.

$B_i^c = (S_i^c, T_i^c, F_i^c, L_i^c, l_i^c)$, where $L_i^c = \{x \in L_i \| \exists (E_1, E_2) \in \Theta_L : x \in E_i\}$ $S_i^c = \{s \in S_i \| \lambda_i(s) \cap L_i^c \neq \emptyset\}$ $T_i^c = \{t \in T_i \| \lambda_i(t) \cap L_i^c \neq \emptyset\}$ $F_i^c = \{(e', e'') \in F_i \| e' \in S_i^c \cup T_i^c \wedge e'' \in S_i^c \cup T_i^c\}$ $l_i^c = \{(f, X) \| f \in F_i^c \wedge X = l_i(f) \cap L_i^c\}$
$B_i^u = (S_i^u, T_i^u, F_i^u, L_i^u, l_i^u)$, where $L_i^u = \{x \in L_i \| \nexists (E_1, E_2) \in \Theta_L : x \in E_i\}$ $S_i^u = \{s \in S_i \| \lambda_i(s) \cap L_i^u \neq \emptyset\}$ $T_i^u = \{t \in T_i \| \lambda_i(t) \cap L_i^u \neq \emptyset\}$ $F_i^u = \{(e', e'') \in F_i \| e' \in S_i^u \cup T_i^u \wedge e'' \in S_i^u \cup T_i^u\}$ $l_i^u = \{(f, X) \| f \in F_i^u \wedge X = l_i(f) \cap L_i^u\}$

Table 6.2: Definition of the common and unique sub-views

6.4.4.1 Integration by Specialization

The overall process of integration by specialization was introduced in Section 6.3.1. We will divide this process into three main parts: *(1) decomposition*, where common and unique sub-views are defined, *(2) integration of common sub-views,* where common labels as well as activities and states labeled with at least one common label are integrated and a temporary integrated object life-cycle is defined, and *(3) integration of unique sub-views,* where unique labels as well as activities and states labeled with unique labels are integrated into the integrated object life-cycle (cf. Figure 6.9 for splitting views into common and unique sub-views). In difference to the process shown in Figure 6.9, we will not define the intermediate refined and reduced sub-views, but derive the common view $\tilde{B}^c$ and the unique sub-views $\tilde{B}_1^u$ and $\tilde{B}_2^u$ directly from the common and unique sub-views B_1^c, B_2^c, B_1^u, and B_2^u.

Decomposition. The object life-cycles of the views are decomposed into two parts: (1) into a *common sub-view*, which includes all common labels and all activities and states that are labeled with at least one common label and (2) into a *unique sub-view*, which comprises unique labels as well as all activities and states that are labeled with at least one unique label.

Formally, the common sub-views B_i^c and the unique sub-views B_i^u are defined as shown in Table 6.2.

Example: Consider the object life-cycles of our example (cf. Figures 6.19 and 6.20): The object life-cycle of view schema R1 is equal to the common sub-view of R1 as no unique labels are defined. The common

$$
\begin{array}{|ll|}
\hline
\tilde{S}^c := & \{(e_1,e_2)|(e_1,e_2) \in \Theta_S^E\} \cup \\
& \{(e_1,e_2)|\exists (e_1,E_2) \in \Theta_S^{I_1} \cup \Theta_T^{I_1} : e_2 \in S_2^c \cap E_2\} \cup \\
& \{(e_1,e_2)|\exists (E_1,e_2) \in \Theta_S^{I_2} \cup \Theta_T^{I_2} : e_1 \in S_1^c \cap E_1\} \\
\tilde{T}^c := & \{(e_1,e_2)|(e_1,e_2) \in \Theta_T^E\} \cup \\
& \{(e_1,e_2)|\exists (e_1,E_2) \in \Theta_S^{I_1} \cup \Theta_T^{I_1} : e_2 \in T_2^c \cap E_2\} \cup \\
& \{(e_1,e_2)|\exists (E_1,e_2) \in \Theta_S^{I_2} \cup \Theta_T^{I_2} : e_1 \in T_1^c \cap E_1\} \\
\tilde{L}^c := & \{(e_1,e_2)|(e_1,e_2) \in \Theta_L^E\} \cup \\
& \{(e_1,e_2)|\exists (e_1,E_2) \in \Theta_L^{I_1} : e_2 \in E_2\} \cup \\
& \{(e_1,e_2)|\exists (E_1,e_2) \in \Theta_L^{I_2} : e_1 \in E_1\} \\
\hline
\multicolumn{2}{|l|}{h_1^c(e^c) := e_1^c \text{ and } h_2^c(e^c) := e_2^c, \text{ where } e^c = (e_1^c, e_2^c)} \\
\hline
\tilde{F}^c := & \{(e',e'')|e' \in \tilde{S}^c \cup \tilde{T}^c \wedge e'' \in \tilde{S}^c \cup \tilde{T}^c \wedge \\
& (((h_1^c(e'),h_1^c(e'')) \in F_1^c \wedge (h_2^c(e'),h_2^c(e'')) \in F_2^c) \vee \\
& ((h_1^c(e'),h_1^c(e'')) \in F_1^c \wedge h_2^c(e') = h_2^c(e'')) \vee \\
& (h_1^c(e') = h_1^c(e'') \wedge (h_2^c(e'),h_2^c(e'')) \in F_2^c))\} \\
\tilde{l}^c := & \{((e',e''),X)|(e',e'') \in \tilde{F}^c \wedge X = \{x \in \tilde{L}^c| \\
& ((h_1^c(e'),h_1^c(e'')) \in F_1^c \wedge h_1^c(x) \in l_1^c(h_1^c(e'),h_1^c(e'')) \wedge \\
& (h_2^c(e'),h_2^c(e'')) \in F_2^c \wedge h_2^c(x) \in l_2^c(h_2^c(e'),h_2^c(e''))) \vee \\
& ((h_1^c(e'),h_1^c(e'')) \in F_1^c \wedge h_1^c(x) \in l_1^c(h_1^c(e'),h_1^c(e'')) \wedge h_2^c(e') = h_2^c(e'')) \vee \\
& (h_1^c(e') = h_1^c(e'') \wedge (h_2^c(e'),h_2^c(e'')) \in F_2^c \wedge h_2^c(x) \in l_2^c(h_2^c(e'),h_2^c(e'')))\}\} \\
\hline
\end{array}
$$

Table 6.3: Definition of the common view

sub-view of R2 consists of labels p and r as well as all activities and states that are labeled with p, r, or both. The unique sub-view of R2 consists of labels c and all activities and states labeled with c. Thus, activity checkOut belongs to the common sub-view, activity startCharging belongs to the unique sub-view, and activity issueBill is included in both sub-views.

Integration of common sub-views. The common view $\tilde{B}^c$ is defined based on the *common sub-views* B_1^c and B_2^c and the determined correspondences. Table 6.3 shows the definition of $\tilde{B}^c$ and the refinement function $h_i^c : (\tilde{S}^c \cup \tilde{T}^c \cup \tilde{L}^c) \rightarrow (S_i^c \cup T_i^c \cup L_i^c)$ for each B_i^c.

$\tilde{B}^c$ is defined as follows:

1. Each activity, state, and label is represented by a pair of elements that shows to which elements of the common sub-views it corresponds. Notice that no set correspondences are allowed with this type of integration, thus Θ^S must be empty and need not be considered. Elements

that are related to a subnet by inclusion are refined during integration. Elements that belong to view-specific alternatives are omitted in this step as they are not included in the correspondence relation Θ.

2. An arc between two elements e' and e'' in $\tilde{B}^c$ is defined in several cases:
   (a) Each sub-view B_i^c contains two elements $\hat{e}'$ and $\hat{e}''$ that correspond to e' and e'', respectively, and that are connected by an arc.
   (b) Sub-view B_1^c contains two elements $\hat{e}'$ and $\hat{e}''$ that correspond to e' and e'', respectively, and that are connected by an arc, whereas in B_2^c, one abstract element $\hat{e}$ corresponds to both e' and e'' and thus, no arc can be defined there.
   (c) Rule (b) holds analogously if B_1^c and B_2^c are exchanged.
3. The labels for an arc $(e', e'') \in \tilde{F}^c$ are defined as follows:
   (a) An arc corresponding to (e', e'') is defined in both common sub-views (cf. case 2. (a)). Then, all labels that are defined in either one of the corresponding arcs in the sub-views are taken into the common view at the finest level of granularity.
   (b) An arc corresponding to (e', e'') is defined only in B_1^c (according to case 2. (b)). Then, all labels corresponding to the labels with this arc are assigned to (e', e'') at the finest level of granularity.
   (c) Rule (b) holds analogously if B_1^c and B_2^c are exchanged.

In this step, *granularity conflicts* are resolved by refining elements that are defined at different levels of granularity in the sub-views; *view-specific alternatives* are omitted as unique activities and states are not included in $\tilde{B}^c$; *naming conflicts* need not be resolved as each element in the integrated object life-cycle is represented by a pair that represents the correspondence to elements in the sub-views.

Example: Figure 6.21 shows the object life-cycle that results from integration of common elements if the labels within brackets and the shaded symbols are ignored. Element equivalences are represented by one element (e.g., state posArchived, activity checkOut, and label p). Element inclusions are represented at the most refined level (e.g., the refinement of state billToIssue, of activity prepare, and of label r).

The view-specific alternative in R2, which consists of activity cancel and state canceled is omitted as only activities and states that appear in the correspondence relation are taken into the common reduced sub-views.

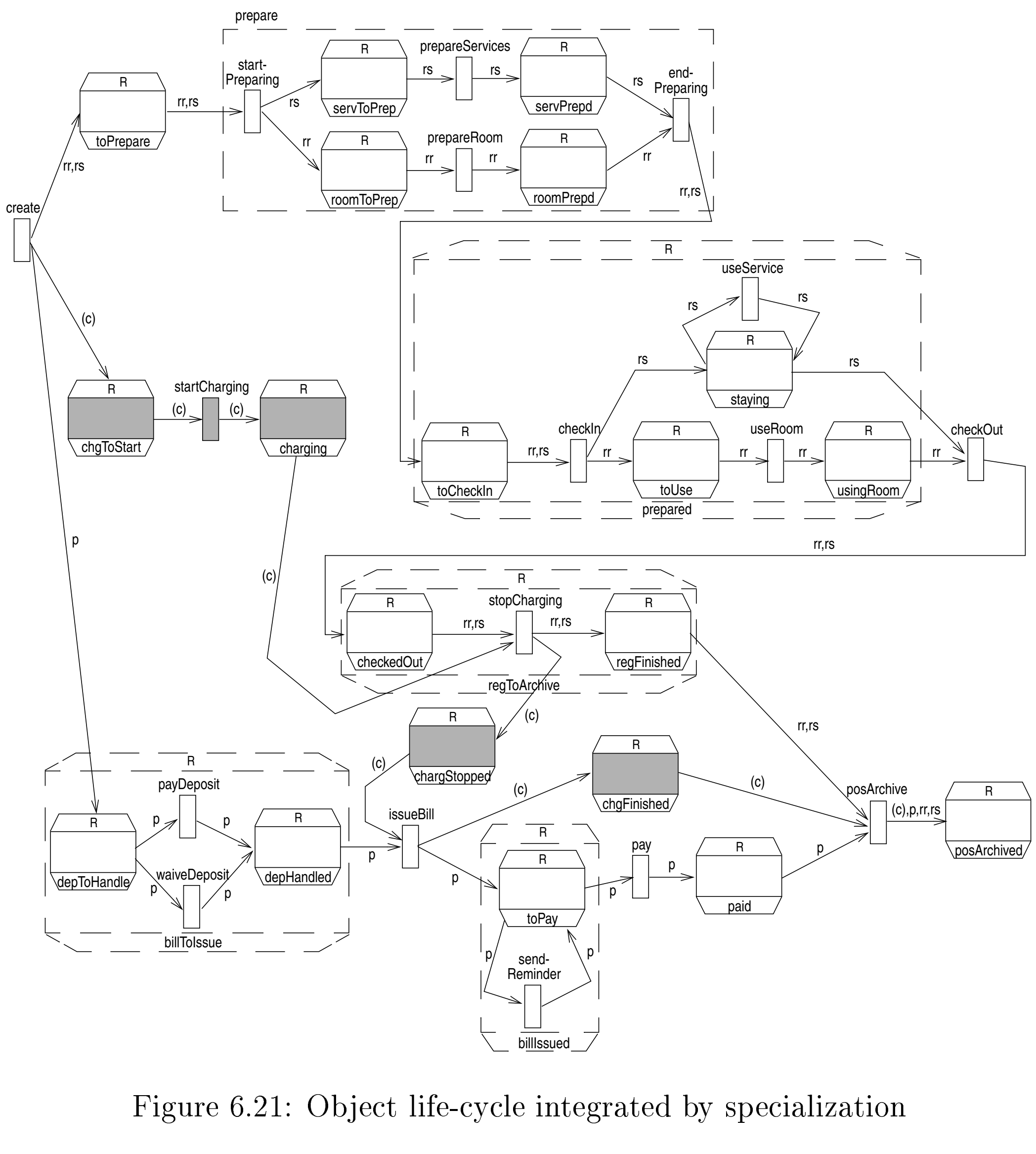


Figure 6.21: Object life-cycle integrated by specialization

Integration of unique sub-views. In the last step, the unique labels as well as activities and states that are labeled only with unique labels are treated in order to resolve heterogeneities due to *view-specific aspects*.

States and activities that are included in the common sub-view B_i^c as well as the unique sub-view B_i^u of a view B_i and that have been refined in the refined common sub-view $\hat{B}_i^c$ have to be refined in the unique sub-view in the same way (yielding the *refined unique sub-view* $\hat{B}_i^u$). Elements that belong to view-specific alternatives that are removed from the common view are removed from the unique sub-view, too, yielding the *reduced unique sub-view* $\bar{B}_i^u$. For all elements of the reduced unique sub-view, a new element is defined in the *unique sub-view* $\tilde{B}_i^u$.

The unique views $\tilde{B}_i^u$ may be defined directly from the unique sub-views B_i^u; $\tilde{B}_i^u$ has to be a correct labeled behavior diagram and an observation consistent specialization of B_i^u based on the specialization function $h_i^u : \tilde{S}_i^u \cup \tilde{T}_i^u \cup \tilde{L}_i^u \rightarrow S_i^u \cup T_i^u \cup L_i^u$.

The specialization of the unique sub-views must not contradict the specialization of the common sub-views. Further, no unnecessary specialization must be defined. Therefore, the following conditions must hold:

1. *Forbidden extension:* No new labels are introduced during integration, i.e., $\forall x' \in \tilde{L}_i^u : h_i^u(x') \in L_i^u$.

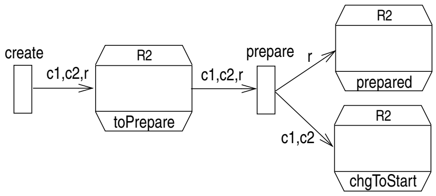
2. *Consistent refinement:* All elements that exist in the common sub-view as well as in the unique sub-view have to be refined in the same way, i.e., $\forall e \in (S_i^c \cup T_i^c) \cap (S_i^u \cup T_i^u), e' \in \tilde{S}^c \cup \tilde{T}^c \cup \tilde{S}_i^u \cup \tilde{T}_i^u : h_i^c(e') = e \Leftrightarrow h_i^u(e') = e$

3. *Consistent definition of arcs:* The same set of arcs is defined for elements that exist in the common view as well as in the unique view, i.e., $\forall e' \in (\tilde{S}^c \cup \tilde{T}^c) \cap (\tilde{S}_i^u \cup \tilde{T}_i^u), e'' \in (\tilde{S}^c \cup \tilde{T}^c) \cap (\tilde{S}_i^u \cup \tilde{T}_i^u) : (e', e'') \in \tilde{F}^c \Leftrightarrow (e', e'') \in \tilde{F}_i^u$.

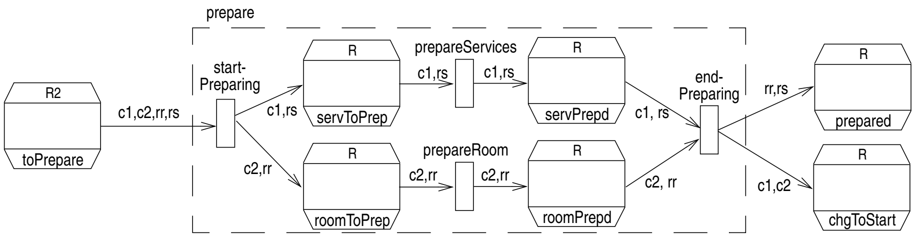
4. *Minimal refinement:* Unique activities, states, and labels in a unique sub-view B_i^u are not refined in $\tilde{B}_i^u$, i.e., $\forall e \in S_i^u \cup T_i^u \cup L_i^u, \tilde{e}_1 \in \tilde{S}_i^u \cup \tilde{T}_i^u \cup \tilde{L}_i^u, \tilde{e}_2 \in \tilde{S}_i^u \cup \tilde{T}_i^u \cup \tilde{L}_i^u : (\neg hasCorr_\Theta(e) \wedge h_i^u(\tilde{e}_1) = e \wedge h_i^u(\tilde{e}_2) = e) \Rightarrow (\tilde{e}_1 = \tilde{e}_2)$.

5. *Inclusion of alternatives with unique labels:* Alternative chunks that are only labeled with unique labels must not be removed, i.e., $\forall t \in T_i : \lambda(t) \subseteq L_i^u \Rightarrow (\exists t' \in \tilde{T}_i^u : h_i^u(t') = t)$.

The integrated business process $\tilde{B} = (\tilde{S}, \tilde{T}, \tilde{F}, \tilde{L}, \tilde{l})$ is a composition of $\tilde{B}^c, \tilde{B}_1^u$, and $\tilde{B}_2^u$. It will be an observation consistent specialization of B_1



(a) part of the modified object life-cycle R2



(b) resulting part of the integrated object life-cycle

Figure 6.22: Refinement during integration of unique sub-views

and B_2, if $\tilde{B}^c$ is an observation-consistent specialization of B_1^c and B_2^c, and for $i \in \{1,2\}$, $\tilde{B}_i^u$ is an observation-consistent specialization of B_i^u. The specialization function $\tilde{h}_i : \tilde{S} \cup \tilde{T} \cup \tilde{L} \rightarrow S_i \cup T_i \cup L_i \cup \{\varepsilon\}$, which represents the specialization of $\tilde{B}$ to B_i is defined as follows: $\tilde{h}_i(\tilde{e}) := h_i^c(\tilde{e})$ if $\tilde{e} \in \tilde{S}^c \cup \tilde{T}^c \cup \tilde{L}^c$, $\tilde{h}_i(\tilde{e}) := h_i^u(\tilde{e})$ if $\tilde{e} \in \tilde{S}_i^u \cup \tilde{T}_i^u \cup \tilde{L}_i^u$, and $\tilde{h}_i(\tilde{e}) := \varepsilon$ else.

Example: Elements that are included into R during integration of unique elements are shown shaded (activities and states) or in brackets (labels) (cf. Figure 6.21). In this case, the unique sub-view need not be refined as no element that was refined in the common view belongs to the unique sub-view.

Consider another case (cf. Figure 6.20): Suppose that state chgToStart in the object life-cycle of R2 is a poststate of prepare and label c is assigned to the arcs (create, toPrepare), (toPrepare, prepare), and (prepare, chgToStart). Then, the unique sub-view comprises activity prepare that must be refined in the unique refined sub-view as it is refined in the common view. Yet, as it is refined to a subnet with two parallel chunks, one label c is not sufficient. Therefore, two labels c1 and c2 must be introduced in the object life-cycle of R2 during schema enrichment (cf. part (a) of Figure 6.22) such that the refinement of prepare is possible during integration of unique sub-views (cf. part (b) of the figure).

Correctness. We defined correctness criteria for integration by specialization in Section 6.3.2.2. We will refer to these criteria once more and discuss how these criteria are fulfilled in the presented integration process.

Observation consistent specialization: Object life-cycle $\tilde{B}$ is defined in the process by integrating corresponding elements at the most refined level of granularity, by defining all unique labels and elements labeled with unique labels, and by excluding view-specific alternatives. The arcs between elements are assigned corresponding to the arcs defined in the views, the labeling function is defined corresponding to the labeling functions in the views. Yet, the integrated object life-cycle is (1) a correct behavior diagram according to the definitions in Section 2.4 and (2) a correct observation consistent specialization of the object life-cycles in the views only if correspondences between elements in the views have been determined consistently. Therefore, we require that after integration, it must be checked whether these criteria (1) and (2) are fulfilled. If they are not fulfilled, the system integrator will have determined inconsistent correspondences.

Remark: The system integrator has to check criteria (1) and (2). If criterion (1) is not fulfilled, criterion (2) need not be checked, of course. If criterion (1) is fulfilled, i.e., the integrated object life-cycle obeys the definitions of correct object life-cycles, criterion (2) remains to be checked. It will be subject to future work to determine in which cases criterion (1) is not fulfilled and in which cases criterion (1) is fulfilled but criterion (2) fails.

The check whether the integrated object life-cycle is a correct object life-cycle and it is an observation consistent specialization of the respective object life-cycles in the views is very simple by using the axioms defined in Table 2.1 and Table 2.2. Integration tools could automatically check whether the specialization criterion is fulfilled. As an extension of this approach, rules could be defined to check whether the correspondences are consistent; if these consistency rules are fulfilled, it is guaranteed that the resulting object life-cycle is correct. Then, inconsistent correspondences could be identified earlier, during identification of correspondences; e.g., (1) a_1 is equivalent to a_2, s_1 is equivalent to s_2, there is an arc between a_1 and s_1, but there is no arc between a_2 and s_2; (2) a subnet is considered corresponding to a single activity, but the activity cannot be refined to this subnet by observation consistent refinement. The mentioned rules to check correspondences will be a topic of further research in the future.

Correspondence compliance: This criterion requires that if an element in $\tilde{B}$ corresponds to an element e_1 of B_1 and an element e_2 of B_2, it must hold that e_1 and e_2 correspond to each other. This criterion is fulfilled as an element in $\tilde{B}$ is defined based on an entry in the correspondence relation Θ or based on a unique element of exactly one view.

Most general specialization: This rule requires that no unnecessary refinements and extensions are defined and no alternatives that are defined in both views are removed. Unnecessary refinements can only exist if elements of the views are further refined in the integrated schema. This requires that activities and states are defined that belong to this refinement. Further extensions can only be defined by introducing labels that do not exist in any view. Yet, in the integration process, all activities and states in $\tilde{B}^c$ correspond to an entry of the correspondence relation, no further activities or states are introduced. Thus, no further refinements can be defined. Additional extensions do not exist as all labels defined in $\tilde{B}^c$ are defined based on an element in the correspondence relation Θ_L. No common alternatives are omitted as each entry in the correspondence relation is represented in the common view. During integration of unique elements, we require that no new labels are introduced and no unique elements are further refined. Further, no elements that are only labeled with unique labels are removed during integration of unique sub-views such that alternatives within view-specific aspects cannot be removed (which would cause an unnecessary extension by partial inheritance).

Example: The integrated object life-cycle shown in Figure 6.21 fulfills these criteria: It is a correct object life-cycle and it is an observation consistent specialization of the object life-cycles in the views. Further, no additional refinements are defined, no further parallel extensions are defined, and no common alternatives are omitted. Each element is either defined based on a correspondence between elements in the views or based on unique elements from view-specific aspects.

Freeness from deadlocks. The object life-cycle that results from integration has to be deadlock-free. During integration by specialization, where all view-specific aspects are integrated, deadlocks may occur if view-specific aspects introduce ordering constraints between common activities. This situation is illustrated by a small example shown in Figure 6.23:

The object life-cycle of RES (depicted in the lower half of the figure) results from integrating the object life-cycles of R1 and R2 (depicted in the upper

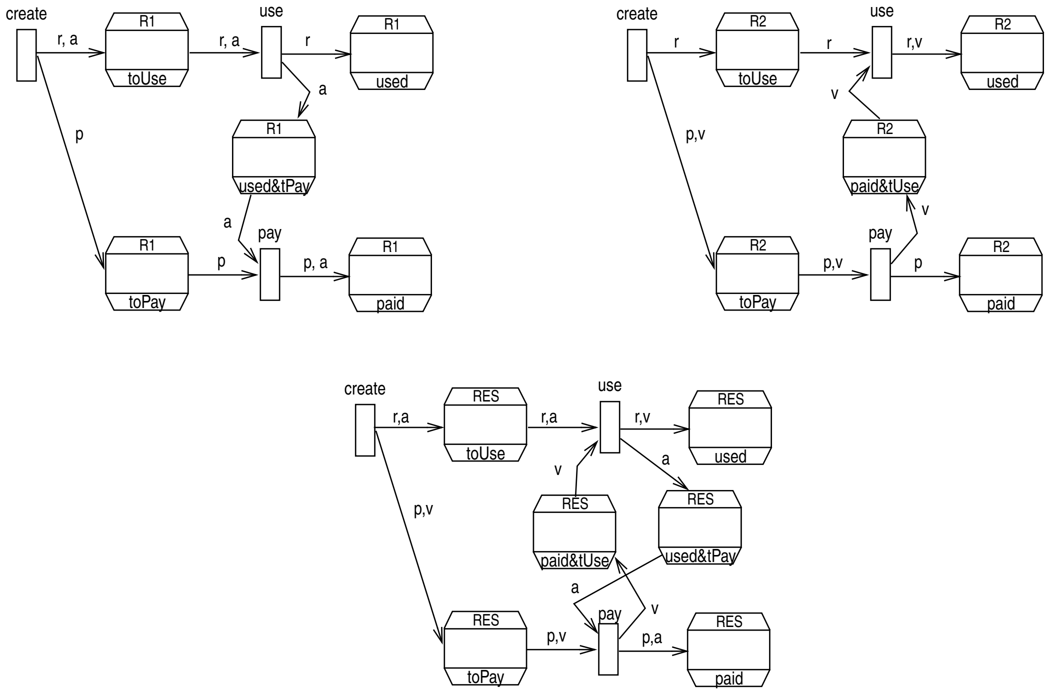


Figure 6.23: Deadlock in the integrated object life-cycle

half of the figure). One view requires that activity use is executed before pay, whereas the other view requires that pay is executed before use. Thus, both views introduce a unique label (a and v, respectively) that causes the intended order dependencies. In the integrated schema, a deadlock occurs.

If such a situation occurs, either ordering constraints due to workflow aspects are contained in at least one of the view schemas or object life-cycles of different object types have been integrated. Of course, no object can be treated according to an object life-cycle that causes deadlocks. If the views reflect processing of business objects according to everyday work in this organization, such deadlocks should not occur.

References from other object types. We refer to the object types for which B_1 and B_2 is defined in the views as o_1 and o_2, respectively, and to the integrated object type as o. There may be a specification of an activity in some object type that contains a port p that refers to state s of the object type o_1 or o_2 in its domain. This reference must be changed to object type o. If state s corresponds to a subnet of activities and states in the other view, state s will not any more appear as a basic state in the integrated object life-cycle $\tilde{B}$, as it is replaced by the subnet. This is no severe restriction at

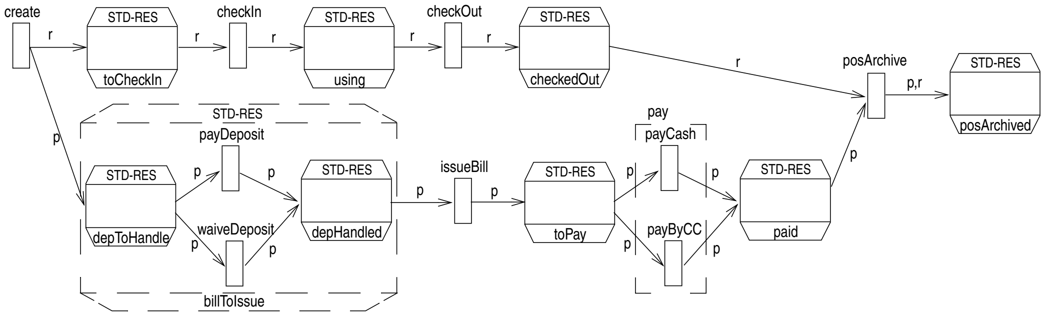


Figure 6.24: Object life-cycle of STD-RES

all, as a non-primary port can refer to an abstract state of an object life-cycle (cf. Section 2.2.2).

6.4.4.2 Integration by Generalization

The overall process of integration by generalization was introduced in Section 6.3.2. We will distinguish two main parts of integration by generalization: *(1) decomposition,* where the restricted sub-views, i.e., the sub-views with common labels, are separated from the unique labels, and *(2) integration,* where the restricted sub-views are integrated.

Example: As a running example for integration by generalization, suppose that object type STD-RES, which represents standard reservations, is defined in one view, whereas object type VIP-RES, which represents vip reservations is defined in the other view. The object life-cycles are shown in Figure 6.24 and 6.25, respectively.

These object types are considered similar such that a common supertype RES must be modeled, whose object life-cycle contains the common behavior of instances of STD-RES and VIP-RES. We will not discuss the steps of intra-type standardization and determination of correspondences once more as they are equal as in integration by specialization.

Correspondences between elements are again represented by elements with equal names and by symbols with dashed borders to represent inclusions. Notice, however, that set correspondences are allowed, e.g., the abstract activity pay is defined by different subnets in the views. Vip reservations can be canceled by the user such that a view-specific alternative is defined in the object life-cycle of VIP-RES (cf. e.g., activity cancel).

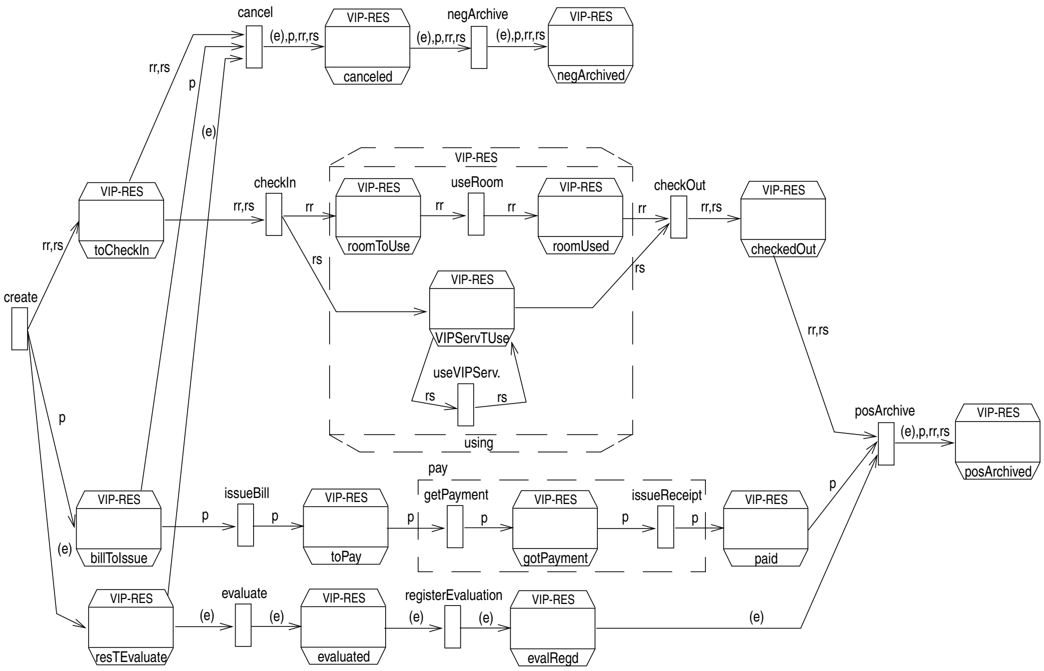


Figure 6.25: View schema of VIP-RES

Decomposition. During decomposition, all unique labels and activities and states that are labeled with unique labels only are removed, yielding the *restricted sub-views* $\dot{B}_1$ and $\dot{B}_2$. The definition of restricted sub-views is equal to the definition of common sub-views during integration by specialization (cf. the upper half of Table 6.2).

Example: The restricted sub-view of object type STD-RES is equal to the original view of STD-RES as no unique labels are defined there. The restricted sub-view of VIP-RES omits the unique label e and all activities and states that are only labeled with e.

Integration. An object life-cycle $\check{B} = (\check{S}, \check{T}, \check{F}, \check{L}, \check{l})$ is defined based on the restricted object life-cycles $\dot{B}_1$ and $\dot{B}_2$. $\check{B}$ has to be an observation consistent generalization of $\dot{B}_1$ and of $\dot{B}_2$ as inclusions and set correspondences are abstracted and view-specific alternatives are included; further, it has to be an observation consistent generalization of the object life-cycles before restriction, namely B_1 and B_2. The specialization function $h_i : S_i \cup T_i \cup L_i \rightarrow \check{S} \cup \check{T} \cup \check{L} \cup \{\varepsilon\}$ between B_i and $\check{B}$, for $i \in \{1, 2\}$ will be defined, too. Formally, the integrated object life-cycle is defined as shown in Table 6.4.

View-specific alternatives in a view v are taken into the integrated schema without change. But there may be a set of labels X in v that is abstracted during integration if X corresponds to a single label in the other view by inclusion or corresponds to a set of labels in the other view due to a set correspondence. But if two labels $x_1 \in X$ and $x_2 \in X$ are assigned to different elements of a view-specific alternative, the labeling properties in the integrated schema are violated if X is abstracted to a single label, but the view-specific alternative is defined in full detail. Therefore, we require that activities and states that belong to a view-specific alternative and that are labeled with a label that is abstracted during integration (because of an inclusion or of a set correspondence) are labeled with all labels that belong to this inclusion or set correspondence, i.e., for $i \in \{1, 2\} : \forall e \in (\dot{S}_i \cup \dot{T}_i), (X_1, X_2) \in \Theta_L, x' \in X_i, x'' \in X_i : (\neg hasCorr_\Theta(e) \wedge x' \in \lambda(e)) \Rightarrow x'' \in \lambda(e)$.

Example: The object life-cycle of VIP-RES contains labels rr and rs, which correspond to a single label r in STD-RES. Thus, they will be abstracted to a single label r in the integrated object life-cycle. The activities and states that belong to the view-specific alternatives in VIP-RES, i.e., activities cancel and negArchive as well as states canceled and negArchived carry both labels rr and rs. Thus, the above criterion is fulfilled.

The required property is no severe restriction: Suppose that one label x in B_1 corresponds to a set of labels X in B_2, where the above criterion does not hold in B_2, i.e., there are activities and states that belong to a view-specific alternative in B_2 but that do not carry all labels of X. Then, the object life-cycle of B_1 can be changed by replacing the single label x by a set of labels X' such that each label of X' is considered equivalent to one label of X; this operation is allowed according to *schema enrichment*.

Example: Suppose that the object life-cycle of VIP-RES contains a view-specific alternative as shown in Figure 6.26, i.e., a subnet of several activities, states, and arcs is defined instead of the single state canceled. There, labels rr and rs are assigned to different elements such that they cannot be abstracted to a single label r. This problem can be resolved by replacing label r in the object life-cycle of STD-RES by labels rr and rs such that label rr (and rs) of STD-RES corresponds to label rr (or rs, respectively) in VIP-RES by equivalence. Then, labels rr and rs are defined in the integrated object life-cycle, too, and the labels need not be abstracted.

The integrated object life-cycle $\check{B}$ is defined as follows:

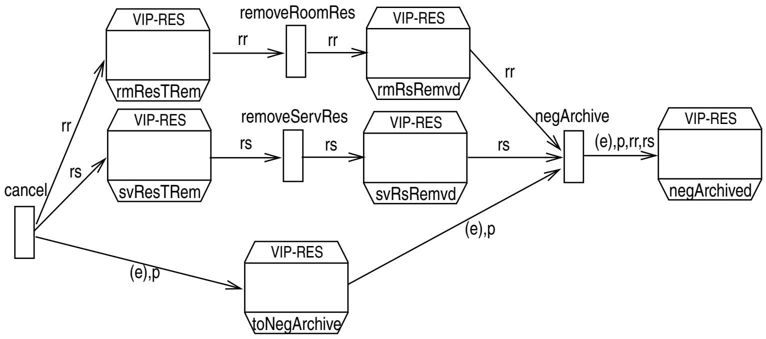


Figure 6.26: Modification of a view-specific alternative in VIP-RES

$$
\begin{array}{|ll|}
\hline
\check{B} = & (\check{S}, \check{T}, \check{F}, \check{L}, \check{l}), \text{ where} \\
\check{S} := & \{(E_1, E_2) | (E_1, E_2) \in \Theta_S\} \cup \{(\{e_1\}, \emptyset) | e_1 \in \dot{S}_1 \wedge \neg hasCorr_\Theta(e_1)\} \cup \\
 & \{(\emptyset, \{e_2\}) | e_2 \in \dot{S}_2 \wedge \neg hasCorr_\Theta(e_2)\} \\
\check{T} := & \{(E_1, E_2) | (E_1, E_2) \in \Theta_T\} \cup \{(\{e_1\}, \emptyset) | e_1 \in \dot{T}_1 \wedge \neg hasCorr_\Theta(e_1)\} \cup \\
 & \{(\emptyset, \{e_2\}) | e_2 \in \dot{T}_2 \wedge \neg hasCorr_\Theta(e_2)\} \\
\check{L} := & \{(E_1, E_2) | (E_1, E_2) \in \Theta_L\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l|}
\hline
\text{for } e \in S_1 \cup T_1 \cup L_1 : \\
h_1(e) := \begin{cases} (E_1, E_2) & \text{if } \exists (E_1, E_2) \in \check{S} \cup \check{T} \cup \check{L} : e \in E_1 \\ \varepsilon & \text{else} \end{cases} \\
\text{for } e \in S_2 \cup T_2 \cup L_2 : \\
h_2(e) := \begin{cases} (E_1, E_2) & \text{if } \exists (E_1, E_2) \in \check{S} \cup \check{T} \cup \check{L} : e \in E_2 \\ \varepsilon & \text{else} \end{cases} \\
\hline
\end{array}
$$

$$
\begin{array}{|ll|}
\hline
\check{F} := & \{(e', e'') \in (\check{S} \times \check{T} \cup \check{T} \times \check{S}) | \\
 & (\exists (\hat{e}', \hat{e}'') \in \dot{F}_1 : h_1(\hat{e}') = e' \wedge h_1(\hat{e}'') = e'') \vee \\
 & (\exists (\hat{e}', \hat{e}'') \in \dot{F}_2 : h_2(\hat{e}') = e' \wedge h_2(\hat{e}'') = e'')\} \\
\check{l}(e', e'') := & \{x \in \check{L} | \\
 & (\exists (\hat{e}', \hat{e}'') \in \dot{F}_1, \hat{x} \in \dot{L}_1 : \\
 & h_1(\hat{e}') = e' \wedge h_1(\hat{e}'') = e'' \wedge h_1(\hat{x}) = x \wedge \hat{x} \in \dot{l}_1(\hat{e}', \hat{e}'')) \vee \\
 & (\exists (\hat{e}', \hat{e}'') \in \dot{F}_2, \hat{x} \in \dot{L}_2 : \\
 & h_2(\hat{e}') = e' \wedge h_2(\hat{e}'') = e'' \wedge h_2(\hat{x}) = x \wedge \hat{x} \in \dot{l}_2(\hat{e}', \hat{e}''))\} \\
\hline
\end{array}
$$

Table 6.4: Definition of the integated object life-cycle in integration by generalization

1. First, states, activities, and labels of $\check{B}$ are determined: Each element in the integrated object life-cycle $\check{B}$ is defined as an ordered pair which defines to which elements of the restricted sub-views $\dot{B}_1$ and $\dot{B}_2$ the element corresponds. As view-specific alternatives are included in the integrated schema, elements in one view that do not correspond to elements in the other view are integrated, too.

2. Two elements in $\check{B}$ are connected by an arc if their "counterparts" in the views $\dot{B}_1$ or $\dot{B}_2$ are connected by an arc. Thus, an arc is defined between two elements e' and e'' in $\check{B}$ if there is a view where an arc is defined between elements that correspond to e' and e''.

3. An arc in $\check{B}$ receives the labels that are defined for the corresponding arc in a $\dot{B}_i$. Of course, if the labels that are assigned to the corresponding arc in the view are abstracted during integration, the *abstracted* labels are assigned to the arc in $\check{B}$.

Correctness. We defined correctness criteria for integration by generalization in Section 6.3.2.2. We will refer to these criteria once more and discuss how these criteria are fulfilled in the presented integration process.

Observation consistent generalization: The integrated object life-cycle $\check{B}$ will be an observation consistent generalization of the given object life-cycles B_1 and B_2 if correspondences have been determined consistently. Similar to integration by specialization, we require that this criterion is checked after integration; if observation consistent generalization is violated, the correspondence relation must be changed. Intuitively, the integration process represents all pairs of equivalent elements, inclusions, and set correspondences by one single element, omits all unique labels and elements labeled only with unique labels, and includes all view-specific alternatives. The arcs and the labels assigned to arcs in the integrated object life-cycle are defined like in the views.

Correspondence compliance: Each element in the integrated object life-cycle $\check{B}$ is defined either based on a correspondence between elements in the views or based on a unique element that belongs to a view-specific alternative. No element in $\check{B}$ corresponds to elements in the views that do not correspond to each other.

Most special generalization: This criterion requires that no common labels are omitted by observation consistent restriction, elements are not unnecessarily abstracted, and no additional alternatives are defined.

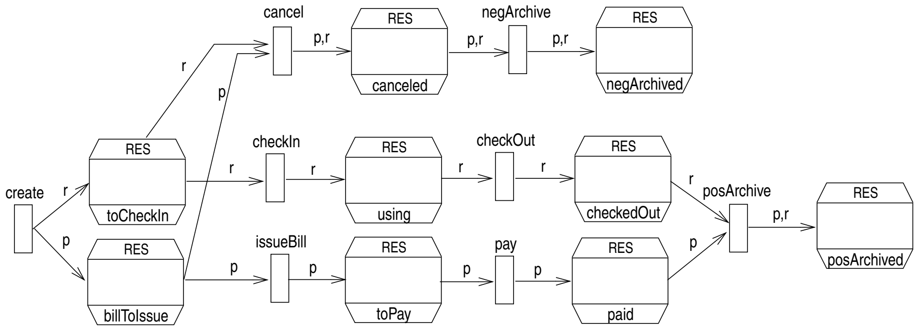


Figure 6.27: Object life-cycle RES, integrated by generalization

All common labels are defined in the integrated schema. All elements are only abstracted if an inclusion or a subnet correspence is defined. No additional alternatives are added as only those elements are defined in the integrated schema that exist in at least one view.

Example: Figure 6.27 shows the object life-cycle RES, which has been integrated by generalization. An element is defined for element equivalences, e.g., label p, state toCheckIn, and activity checkIn. Inclusions are integrated after abstraction, e.g., label r and state using. Set correspondences are abstracted, too, e.g., activity pay. The view-specific alternative, which is defined in VIP-RES and which consists of activities cancel and negArchive as well as states canceled and negArchived is defined in the integrated object life-cycle, too.

6.5 Consistency of Integrated Object Types

During integration by specialization, object types in the integrated schema are defined independently from each other. Yet, there are some issues on the consistency between object types in the integrated schema, which will be treated in this section:

1. If two object types in the integrated schema are related to each other by a subtype-relationship, then a specialization function h must be defined to represent the specialization of the object life-cycle of the subtype compared to the object life-cycle in the supertype.

2. Elements that are defined in more than one subtype of an object type must be defined with the supertype, too. This problem was motivated in the introduction as *generalization of elements.*

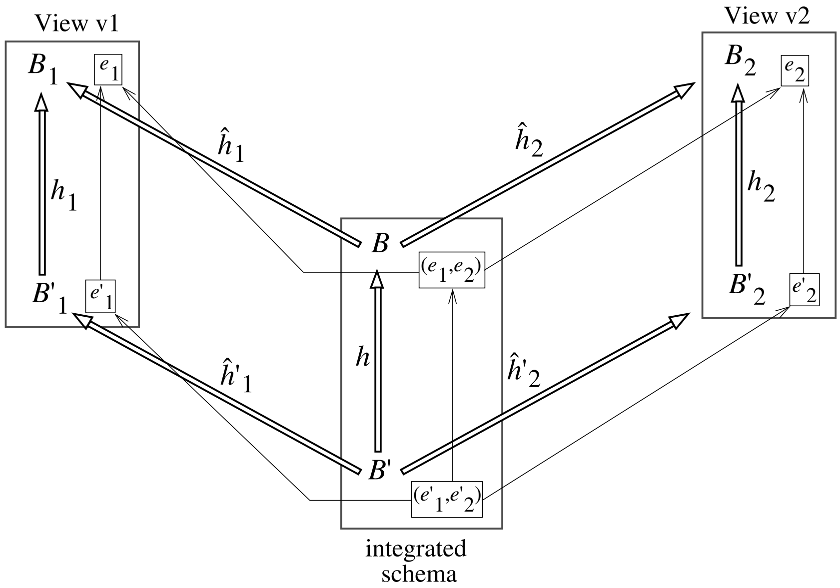


Figure 6.28: Specialization functions between object types in views and in the integrated schema

6.5.1 Determination of Specialization Functions

Object types in the integrated schema are defined independently from each other. If two object types in the integrated schema have to be related by a subtype relationship, a specialization function h must be defined.

Suppose that there are two views v_1 and v_2. In view v_1, two object types o_1 and o_1' are defined, where o_1' is a subtype of o_1; thus, the behavior diagram B_1' of o_1' is an observation consistent specialization of the behavior diagram B_1 of o_1 based on the specialization function h_1. Analogously, there are object type o_2 and o_2' in view v_2, where the latter is a subtype of the former, such that behavior diagram B_2' of o_2' is an observation consistent specialization of B_2 of o_2 based on the specialization function h_2. This information is defined in the views; cf. Figure 6.28 for an illustration.

Further, suppose that o_1 and o_2 are equivalent, such that they are integrated to object type o with behavior diagram B. Object type o_1' and o_2' are equivalent, too, they are integrated to object type o' with behavior diagram B'. The correspondences between elements of behavior diagram B and elements of B_1 and B_2 are defined during integration by specialization; there the specialization functions $\hat{h}_1$ and $\hat{h}_2$, respectively, are constructed. The correspondences between B' and B_1' and B_2' are represented by the specialization functions $\hat{h}_1'$ and $\hat{h}_2'$.

As object type o'_1 is a subtype of o_1 and object type o'_2 is a subtype of o_2, object type o' must be a subtype of o in the integrated schema. Thus, a specialization function h must be determined between the elements of behavior diagram B' and B.

Intuitively, we require that correspondences that are detected between elements of B_1 and B_2 must be consistent with the correspondences between elements of B'_1 and B'_2: Suppose that elements e_1, e_2, e'_1, and e'_2 are defined in B_1, B_2, B'_1, and B'_2, respectively, as illustrated in Figure 6.28. e'_1 and e'_2 are specializations of e_1 and e_2, respectively. Elements e_1 and e_2 have been considered equivalent in the correspondence relation, i.e., $(e_1, e_2) \in \Theta$. For consistent specialization, we require that it holds that $(e'_1, e'_2) \in \Theta$, too, i.e., if there is a correspondence between elements in the supertypes, then there must be a correspondence between their specialized elements in the subtypes, too.

Corresponding elements e_1 and e_2 are integrated to a single element (e_1, e_2) in B, where the specialization functions relate the element in B to the elements in B_1 and B_2, i.e., $\hat{h}_1(e_1, e_2) = e_1$ and $\hat{h}_2(e_1, e_2) = e_2$. For consistent correspondences, we require that e'_1 and e'_2, which are the specializations of e_1 and e_2, correspond to each other, too, such that they are represented in B' by one element (e'_1, e'_2). Then, element (e'_1, e'_2) in B' is the specialization of (e_1, e_2) in B, i.e., it must hold that $h(e'_1, e'_2) = (e_1, e_2)$.

Based on this consideration, we define the specialization function h between $B' = (S', T', F', L', l')$ and $B = (S, T, F, L, l)$. For the definition, we need a function $h^{-1} : S \cup T \cup L \to 2^{(S' \cup T' \cup L')}$, which is defined for a specialization function $h : S' \cup T' \cup L' \to S \cup T \cup L \cup \{\varepsilon\}$ as follows: $h^{-1}(e) = \{e' | h(e') = e\}$.

The specialization function h between B' and B of the integrated schema (refered to as the *h-function* in the following) is defined as follows: for $e' \in S' \cup T' \cup L', e \in S \cup T \cup L :$

$$h(e') := \begin{cases} e & \text{if } \hat{h}'_1(e') \neq \varepsilon \wedge h_1(\hat{h}'_1(e')) \neq \varepsilon \wedge \hat{h}_1^{-1}(h_1(\hat{h}'_1(e'))) = \{e\} \quad (1) \\ e & \text{if } \hat{h}'_2(e') \neq \varepsilon \wedge h_2(\hat{h}'_2(e')) \neq \varepsilon \wedge \hat{h}_2^{-1}(h_2(\hat{h}'_2(e'))) = \{e\} \quad (2) \\ \varepsilon & \text{else} \quad (3) \end{cases}$$

The $h-$function consists of three rules:

1. Rule (1) determines an element e from B that corresponds to e' of B' via the object life-cycles B_1 and B'_1. If element e' has a corresponding element e'_1 in B'_1 and e_1 in B_1 and there is exactly one element e in B that corresponds to e_1, then this e is returned.

2. Rule (2) is defined analogously to rule (1) for B_2 and B'_2.

3. Rule (3) sets the value of $h(e')$ to ε if none of the other rules applies, i.e., there is no element in B that corresponds to e'.

We distinguish the following cases during definition of the $h-$function:

Omitted elements: The simplest case are elements that are defined in B, but that are not defined in B' according to partial inheritance. Such elements need not be considered in the h-function as this function is only defined for elements that are defined in B'.

Definition in the views' subtypes: If an element e' of B' is based on corresponding elements in B'_1 or B'_2 or both, but there are no corresponding elements in the behavior diagrams B_1 or B_2 of the supertypes, then the condition $h_i(\hat{h}'_i(e')) \neq \varepsilon$ is false for $i \in \{1,2\}$, such that $h(e')$ is set to ε according to rule (3) of the $h-$function.

Definition in one view's supertype: Suppose that an element e_i (for $i = 1$ or $i = 2$) is defined in B_i, for which an element e is defined in B, i.e., $\hat{h}_i(e) = e_i$. Element e'_i in B'_i is the specialization of e_i, whereas e' in B' is defined for e'_i. Element e_i is unique, i.e., no corresponding element is defined in the other view. Thus, it must hold that $h(e') = e$. The term $h_i(\hat{h}'_i(e'))$ returns e_i. As e_i is unique and unique elements are defined in the integrated object life-cycle without change, function $\hat{h}_i^{-1}$ returns exactly one element, i.e., the term $\hat{h}_i^{-1}(h_i(\hat{h}'_i(e')))$ returns element e of B.

Definition in both views' supertypes: Finally, an element e' in B' corresponds to an element e in B if e results from integrating a pair of corresponding elements in B_1 and B_2. If an element e in B is defined because of an *equivalence* of elements e_1 in B_1 and e_2 in B_2, element $h(e') = e$ can be determined by rule (1) or, alternatively, by rule (2) of the $h-$function. Both rules will return the same element. The reason is that exactly one element in the integrated schema is defined for a pair of equivalent elements in the views, such that $\hat{h}_i^{-1}(h_i(\hat{h}'_i(e')))$ returns exactly one element for both $i = 1$ and $i = 2$.

If a single element e_2 in B_2 corresponds to a subnet with a set of elements E_1 in B_1 (*right inclusion*), there will be a set of elements E defined in B where one element $e \in E$ is defined for each element $e_1 \in E_1$. The correspondence between e' in B' and an element e in B must be determined by rule (1), as the term $\hat{h}_1^{-1}(h_1(\hat{h}'_1(e')))$ returns exactly one element e for e', whereas the term $\hat{h}_2^{-1}(h_2(\hat{h}'_2(e')))$ of rule (2) returns the whole set of elements E.

In case of a *left inclusion*, i.e., an element in B_1 corresponds to a set of elements in B_2, rule (2) is applied, analogously.

As a correctness criterion, we require that B' is an observation consistent specialization of B according to the specialization function h. If this criterion is not fulfilled, the system integrator has determined conflicting correspondences between elements of B_1 and B_2 and between B_1' and B_2', such that the correspondence relations between the object types have to be re-examined. The $h-$function can be determined and the criterion of observation consistent specialization can be checked automatically by an integration tool. Yet, similar to the discussion in Section 6.4.4.1, the consistency of correspondence relations of supertypes and their subtypes could be checked earlier, during determination of correspondences. Rules to check consistency of correspondences are — as mentioned before — subject to further research.

6.5.2 Generalization of Elements

Object types are integrated pair-by-pair. In the introduction we argued that an object type in the integrated schema should contain all elements that are defined in all of its subtypes. Yet, as object types are integrated separately, an element could be defined with each subtype of an object type, but not with the supertype. A motivating example for this case was given in Section 1.3.4.3.

Suppose that there is an object type o with subtypes $O' = \{o_1, \ldots, o_n\}$ defined in the integrated schema, where the object types have behavior diagrams $B, B_1, \ldots, B_n$, respectively. For a correct specialization hierarchy, we require that each element is originally defined in only one type (according to the rule of original definition of elements).

After integration of two views, the specialization hierarchy may not fulfill this correctness criterion but there may be elements defined in several or even all B_i, but not in B. We distinguish two cases: (1) If an element is defined in *all* B_i, but not in B, it must be included in B. (2) If an element is defined in a subset of the subtypes $O'' \subset O'$, the rule of original definition of elements is violated, but this element cannot be included in B as it is not defined in all B_i. Thus, a new intermediate object type o^* with behavior diagram B^* must be introduced that is a subtype of o and a supertype of all $o_i \in O''$. Then, the common elements of the object types in O'' can be originally defined in o^*, such that case (2) can be reduced to case (1).

In case (1), behavior diagram B may be changed to B' in the following ways (these rules can be applied to case (2), as well, after an intermediate object type has been inserted):

- Refinements that are defined in all B_i are defined in B, too, by observation consistent refinement.
- Extensions that are defined in all B_i are added in B, too, by observation consistent extension.
- Alternatives in B, that have been omitted in all B_i, are omitted in B, too, by partial inheritance, as they are needed in no subtype.

As only elements are refined or added in B', that are refined or added in *all* B_i, too, and only alternatives are removed that are removed in all B_i, it must hold that B' is an observation consistent generalization of each B_i. Further, B' must be the most special generalization of all B_i, such that the same correctness criteria must hold as defined for integration by generalization in Section 6.3.2.2. In fact, generalization of elements is similar to integration by generalization. During integration by generalizaton, a new object type o is introduced for a set of existing object types, where all common elements of the subtypes of o are integrated. During generalization of elements, the supertype has already been defined before, but it does not include *all* common elements of its subtypes. Thus, the results of these steps must fulfill the same correctness criteria.

Example: Consider the object life-cycles shown in Figure 6.29: Object type RES is the supertype of RES1 and RES2. The object life-cycle of RES' results from RES by generalization of elements: Label p and elements toPay, pay, and paid are defined in both subtypes such that they are inserted in the supertype, too. Activity use is refined to startUsing, inUse, and endUsing; the alternative of canceling consisting of activity cancel and state canceled is removed as it appears in no subtype.

Notice, that the criterion of original definition of elements cannot be fulfilled in every case if only single inheritance is used. Suppose that an object type o with three subtypes o_1, o_2, and o_3 exists. There are two elements e_1 and e_2, where e_1 is defined in o_1 and o_2 whereas e_2 is defined in o_2 and o_3. If an intermediate object type o^* is defined as a supertype of o_1 and o_2, which includes element e_1, then there is no object type in which element e_2 can be originally defined, such that the information that the same element e_2 is defined in two object types o_2 and o_3 is lost. For this case, multiple inheritance were necessary: object type o_1^* is a supertype of o_1 and o_2 with element e_1, whereas object type o_2^* is a supertype of o_2 and o_3 with element e_2.

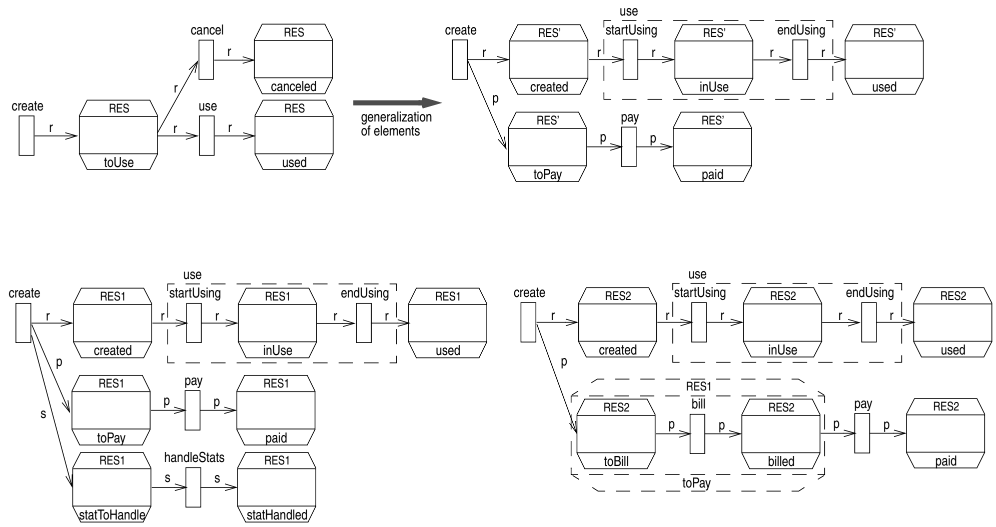


Figure 6.29: Generalization of elements

Chapter 7

Integration of Activity Specifications

Contents

Definition of behavior consists of object life-cycles and activity specifications. This chapter presents integration of activity specifications, which is heavily influenced by the results of integrating object life-cycles. In Chapter 6, the system integrator decided *which* activities are defined in the integrated object life-cycle. In this chapter, we have to deal with the question how the activity specifications are defined for these activities.

Corresponding to the integration approaches by *specialization* and *generalization* defined with object life-cycles (cf. Section 6.3), we will distinguish integration by specialization and by generalization for activity specifications, too.

This chapter is structured as follows: Section 7.1 discusses which problems of view integration have not yet been dealt with. In Section 7.2, we list heterogeneities between activity specifications. Section 7.3 gives an overview of the integration approach and the integration process; Section 7.4 describes the steps of the integration process in detail.

7.1 The Open Issues

Activity specification diagrams consist of the following parts: (1) definition of the primary port, (2) definition of non-primary ports, and (3) definition of pre- and postconditions. In this section, we will deal with the question which parts of the activity specifications have already been defined and which problems remain for this chapter.

For each invocable activity that is defined in an object life-cycle in the integrated schema, there must be an activity specification diagram. The definition of *primary ports* in activity specifications must be equal to the definition of pre- and poststates in object life-cycles. As object life-cycles have already completely been defined, the definition of primary ports in the activity specifications can be directly taken from the object life-cycles. Therefore, no work remains to do for primary ports. Therefore, we will not further deal with primary ports in this chapter and only treat *non-primary ports* and *pre- and postconditions.* This means that if we speak of *ports* in this chapter, we will mean *non-primary ports* unless stated differently.

7.1.1 Specification of Activities in Previous Steps

First, we summarize how activity specifications have been treated during the steps of integration discussed in the previous chapters of this thesis.

Extraction of business processes: In Section 5.1, specifications of business-process activities were extracted from the specification of worksteps.

Inter-type standardization: During inter-type standardization, activity specifications were changed as follows:

1. *Splitting object types:* If object types are split, activities are assigned to the object types that result from splitting. There, synchronization states are replaced by triggered synchronization constraints, hierarchical synchronization constraints, and domain restriction constraints. These constraints can be expressed by additional non-primary ports such that additional non-primary ports are included in the activity specifications. Further, activity specifications that refer to a state of the object type before splitting are changed to refer to the corresponding state after splitting.
2. *Explicit definition of relationships:* Activities of the original object types can be assigned to the object life-cycle of the new association type. Activity specifications are modified because of triggered synchronization constraints, hierarchical synchronization constraints, and domain restriction constraints.
3. *Subtyping:* During subtyping, new object types are introduced, where the object life-cycle of each of these object types results from omitting alternatives. Thus, no new activities are introduced and no new activity specifications are necessary. Activities that refer to a state of the original object type refer to one of the object types of the type hierarchy that results from subtyping.

Intra-type standardization of object life-cycles: Intra-type standardization of object life-cycles comprises the following steps:

1. *Standardization of activities:* One activity is replaced by a set of alternative activities, where discriminating ports may be omitted and textual preconditions may be defined.
2. *Standardization of conditions:* A prestate or poststate of an activity can be replaced by a pre- or postcondition and vice versa.
3. *Schema enrichment:* During schema enrichment, redundant states are inserted, which do not influence non-primary ports.

Integration of object life-cycles: If an object type o_1 of a view was integrated with object type o_2 of the other view, yielding object type o in the integrated schema, the domains of ports in all activity specifications in the views that refered to object type o_1 or o_2 were changed to o.

7.1.2 Specification of Activities During Integration

The remaining issues are activity specifications for activities that are defined in the integrated object life-cycle of an object type during integration by generalization or integration by specialization. Each activity in an integrated object life-cycle is defined based on activities in object life-cycles in the views.

We require as a correctness criterion (cf. Section 1.3.5) that all activities in the leaf nodes of the specialization hierarchy are specified, i.e., an activity specification diagram is specified for each of these activities. Further, the rule of original definition of ports (cf. Section 2.3.3) must be obeyed: if there are two object types o_1 and o_2 in the integrated schema, that have a common supertype o, and an activity a_1 in object type o_1 and a corresponding activity a_2 in object type o_2 are defined that have a common port p, then port p must be defined in activity a of o that is the generalization of a_1 and a_2, too.

We discuss in this section, in which cases activity specifications must be defined and how integration of activity specifications works.

7.1.2.1 Integration by Generalization

During integration by generalization, two object life-cycles B_1 and B_2 of similar object types are given, and an object life-cycle B for the common supertype is defined.

Elements in B are defined because of the following cases:

Equivalence: If an activity a_1 in B_1 is equivalent to an activity a_2 in B_2, one activity a in B is defined. If both a_1 and a_2 are specified, i.e., an activity specification is defined for both activities, an activity specification is defined for activity a that includes the common properties of a_1 and a_2 according to the rule of original definition of ports. We will present an approach to define the activity specification for a in this chapter. If at least one activity is unspecified, activity a in B is unspecified, too, i.e., no activity specification needs to be defined.

Inclusion: If one single activity in B_1 corresponds to a subnet of activities and states in B_2 (or vice versa), one single unspecified activity is defined in B.

Set correspondence: If a subnet in B_1 corresponds to a subnet in B_2, and both subnets can be abstracted to an activity, one single unspecified activity is defined in B.

No correspondence: For a unique activity a_i in one B_i (which stems from view-specific alternatives), one activity a is defined in B. The activity specification of a can be taken from a_i. Thus, no integration is necessary.

We see that the definition of activity specifications is trivial except in the case of equivalence if both equivalent activities are specified. For integrating specifications of equivalent specified activities, integration by generalization will be defined in this chapter.

7.1.2.2 Integration by Specialization

During integration by specialization, two object life-cycles B_1 and B_2 of equivalent object types are given, and an object life-cycle B is defined for the equivalent object types by specialization.

Equivalence: If activity a_1 in B_1 is equivalent to activity a_2 in B_2, we require that at least one of them is unspecified (cf. Section 1.3.5). If one activity is unspecified, whereas the other one is specified, the activity specification of the latter is taken into the integrated schema. If both activities are unspecified, the activity is unspecified in the integrated schema, too.

Inclusion: If one abstract (and thus, unspecified) activity in B_1 corresponds to a subnet in B_2 (or vice versa), the more refined definition (i.e., the subnet) is defined in B. The activity specifications can be taken from the respective view, too.

Set correspondence: Set correspondences are not allowed in integration by specialization.

No correspondence: Unique activities in a B_i (which belong to view-specific aspects) are represented in B, too. The activity specification can be taken from B_i.

We see that no integration of activity specifications is necessary if equivalent object types have been integrated by specialization.

Corresponding to integration of object life-cycles where elements that were defined in the subtypes of an existing object type o were defined in the object life-cycle of object type o, we will introduce *generalization of elements* for activity specifications, too, at the end of this chapter.

As we use a binary integration strategy, the integrated view schema that results from integrating two view schemas need not define an activity specification for each activity in an instantiable object type. Yet, the criterion must only be fulfilled after *all* view schemas have been integrated.

7.2 Heterogeneities

The specifications of two equivalent activities may differ from each other. These heterogeneities are particularly relevant for integration by generalization, as similar object types have been defined by different users and a common supertype must be defined for these object types. As object types o_1 and o_2 are originally defined in different views, information may be represented in different ways, such that the activity specifications must be restructured before correspondences between ports can be determined. Yet, we will see in this chapter that information-capacity preserving restructuring is not enough but that several heterogeneities require redefinition of the view schemas in order to gain a proper design.

We distinguish two different kinds of heterogeneities: (1) *port conflicts,* which mean that different ports are defined or corresponding ports have different characteristics (single-valued/multi-valued port, read/transfer port, input/output port, return/no return port), and (2) *domain conflicts*, which mean that the domains of two equivalent ports are different. Further there may be *different pre- and postconditions* defined textually with the activity specifications, which cannot be uniquely classified as port conflicts or domain conflicts.

7.2.1 Port Conflicts

Port conflicts are heterogeneities between definitions of ports not yet considering the domains of ports.

Naming conflicts: Corresponding ports may carry different names (*synonyms*), whereas different ports may carry the same name (*homonyms*).

Inadequate definition of ports: Information that should be split to several ports is defined by a single port or information that should be defined in a single port is distributed to a set of ports.

Multi-valued port vs. a set of ports: If an activity reads or returns a fixed number of objects of the same object type in the same state, it may either define a multi-valued port or define several single-valued ports.

View-specific extended ports: Activity a_1 includes ports that are not considered in activity a_2. View-specific extended ports occur if some aspects considered in an activity of one view are not considered in the corresponding activity in the other view.

View-specific implicit ports: Activity a_1 includes a port that is not explicitly defined in activity a_2, but exists there only implicitly. This means that the information that can be provided by this port is needed in a_2, but the port is not defined as its value is constant or derivable.

Multi-valued ports vs. single-valued ports: Port p_1 may correspond to port p_2 but one port is single-valued whereas the other port is multi-valued.

Read ports vs. transfer ports: Input port p_1 may be a defined as a read port whereas the corresponding input port p_2 is a transfer port.

Input ports vs. output ports: A port can be defined as an input port, an output port, or as an input/output port. Two corresponding ports p_1 and p_2 may be defined differently.

Return port conflicts: The definition of return ports is not equal with a_1 and a_2. This heterogeneity arises if a return port is defined with one activity, but no return port is defined with the other activity or different return ports are defined.

7.2.2 Domain Conflicts

Domain conflicts occur between two sets of corresponding ports p_1 and p_2 if the domains of the ports are not equal. Some of the following heterogeneities have already been defined in [KS91b] for attributes in the relational model.

Example: In the following list, we use the following example: One view defines object type VIP-RES and the other view defines STD-RES, which represent reservations for vip guests and standard guests, respectively. The object types are similar, i.e., both object types are taken into the integrated schema and a common supertype RES must be defined with the common activities of VIP-RES and STD-RES.

Value-type unit: p_1 and p_2 have similar value types as domains, but use different units of measurement. A value of one type can be mapped to a value of the other type and vice versa.

Example: There are two specificiations of activity pay, each one has a port amount. In one view, the domain type of this port is ATS, whereas in the other view, the domain type is EUR.

Value-type precision: p_1 and p_2 have similar domains but keep information at different levels of precision, e.g., one port requires an exact value, whereas the other one requires rounded values.

Example: Port checkInTime is defined for activity VIP-RES.checkIn as time (hours and minutes), whereas in STD-RES.checkIn, this port is defined with domain time in hours.

Different object types as domain: p_1 and p_2 may have different object types within the same type hierarchy or different states of the same object type as domain.

Example: The activity specifications of VIP-RES.create and STD-RES.create contain a port g, where the domain of the former is VIP-GUEST in state readyForRes, whereas the domain of the latter is STD-GUEST in state exists.

Limited domain: Port p_1 has a domain whose set of possible values is a subset of the domain of p_2. This heterogeneity is different from differences in *value-type precision*, as information is not considered with less precision, but fewer different values can be represented.

Example: VIP-RES.create and STD-RES.create both have port from, by which the start of the reservation period is defined. In one view, the domain of from is DATE, whereas in the other view, it is DAY_OF_YEAR (i.e., a number between 1 and 365 or 366), as only reservations within one year can be created for standard guests.

Object vs. related object: If there is a single-valued relationship defined in object type o_1 with object type o_2 as domain, the domain of port p_1 may be object type o_1, whereas the domain of port p_2 may be object type o_2. This situation will particularly occur if the single-valued relationship is a *hasRole-, hasComponent-, roleOf-,* or *componentOf* relationship.

Example: Suppose that there is an activity account with port b, that requires information about the bill that is issued for a reservation. In one view, BILL is defined as the domain type of the port, whereas in the other view, RESERVATION is defined as domain type, where a single-valued *hasComponent* relationship bill with domain BILL is defined with RESERVATION.

Further, suppose that there is a *multi-valued* relationship defined in o_1 that has domain o_2 and there is an activity a that reads or returns information about all instances of o_2 that are related to an instance of o_1. Then, a port will be defined with a that is single-valued and reads or returns a single instance of o_1 or that is multi-valued and reads or returns all related instances of o_2.

Example: Activity makeGuestStatistics may need all guest roles of a person. In one view, all instances of GUEST of that person are provided, whereas in the other view, the respective instance of PERSON is provided by a port.

Object type vs. value type: The domain of p_1 is an object type whereas the domain of port p_2 is a value type. This situation will particularly occur if one activity refers to an object, whereas the other activity refers to the key of the object.

Example: Suppose that activity create of RESERVATION requires the room that has to be reserved in input port room. In one view, an instance of ROOM in state available is provided, whereas in the other view, the room number, which is the key of instances of ROOM, is supplied.

The last two heterogeneities may arise involving *sets of ports*, too. There may be a set of ports P_2 that can be used to identify an instance of a certain type, which is used as domain type of p_1.

Example: Suppose that activity VIP-RES.create has two ports n and b that represent the name and the date of birth of a person, where name and birthDate are the key attributes of PERSON. In the other view, STD-RES.create with one port p is defined that requires an instance of PERSON.

Instead, suppose that activity create of VIP-RES in the one view has two ports with domain types PERSON and HOTEL to uniquely identify an instance of GUEST. The corresponding activity STD-RES.create defines one port with domain type GUEST.

7.3 Integration by Generalization: Basics and Process

In this section, we discuss the basics of integration by generalization and give an overview of the integration process.

7.3.1 Basics about Integration

As motivated before, the open issue is that activity specifications for two similar object types must be integrated to activity specifications in the common supertype. There, only pairs of equivalent specified activities must be considered. Corresponding to integration of object life-cycles where common elements were determined and integrated by generalizing them, we will determine the common ports of activities and take them into a corresponding activity in the integrated behavior diagram.

The goal of integration is to define an activity a by integrating activities a_1 and a_2, where all common ports of a_1 and a_2 are defined in a, such that a_1 and a_2 are *specializations* of a as defined in Section 2.3.3. As *all* common ports of a_1 and a_2 must be defined with a, we require that a is the most special generalization of a_1 and a_2.

We summarize the main rules of specialization of activity specifications, which were introduced in Section 2.3.3:

1. New ports can be added in the specialized activity specification.

2. Each port of an activity in the supertype must be inherited by the corresponding activity in the subtype. The following characteristics of ports must not be changed during specialization: single-valued/multi-valued port, read/transfer port, input/output/input+output port, return/no return port.

3. If the domain of a port is a value type, it must not be changed.

4. If the domain of a port is an object type o in a certain set of states, the domain can be specialized by using a subtype o' of o, adding states and refining states.

5. Pre- and postconditions can be defined more restrictive during specialization, i.e. if a condition c is defined with a, and a is specialized to a' with condition c', it must hold that $c' \Rightarrow c$.

From these rules, it follows that the following differences are allowed between activity specifications of a_1 and a_2:

View-specific extended ports: One a_i has a port that is not defined in the other activity. It is omitted in a, too, as it can be added if a_i is specialized from a.

Different object types as domain: Consider port p_1 in a_1 and p_2 in a_2, where p_1 is equivalent to p_2. The domain d_1 of p_1 and d_2 of p_2 are object types o_{d_1} and o_{d_2} in states S_{d_1} and S_{d_2}, respectively.

In the integrated schema, activity a is defined, where p_1 and p_2 are represented by port p with domain d, which is object type o_d in the set of states S_d:

- If $o_{d_1} = o_{d_2}$, then o_d is set to o_{d_1}. The set of states S_d must represent those pre- or postconditions that are defined with both S_{d_i}. If a state s is defined with S_{d_1} and S_{d_2}, it must be included in S_d, too. If a state is defined in only one S_{d_i}, it is not included in S_d, as the activity specification of a cannot include a state that is missing in the activity of a subtype. Thus, $S_d := S_{d_1} \cap S_{d_2}$. If $S_d = \emptyset$, i.e., S_{d_1} and S_{d_2} refer to disjoint states, an abstract state must be defined with the object life-cycle of o_d that comprises at least one state of S_{d_1} and one state of S_{d_2} and that is defined as S_d; this case will be discussed later.
- If $o_{d_1} \neq o_{d_2}$, but they have a common supertype o (where o can be equal to o_{d_1} or o_{d_2}), and there is no o' that is a subtype of o and that is a common supertype of o_{d_1} and o_{d_2}, then there must be a specialization function $h^+_{(B_i,B)}$ that indicates the relationship between elements of the object life-cycle B_i of o_{d_i} and B of o. o_d is set to this common supertype o. The set of states S_d is defined as $S_d := (\{h^+_{(B_1,B)}(s) | s \in S_{d_1}\} \cap \{h^+_{(B_2,B)}(s) | s \in S_{d_2}\}) \setminus \{\varepsilon\}$. This means that the states defined with each port are abstracted and restricted to the level of the common supertype (resulting in the generalized set of states), and the intersection of the generalized set of states is defined as the set of states for the domain of the port p in the integrated activity a. Possible ε values, which indicate that a state in o_{d_i} is not defined in o_d, are omitted. If $S_d = \emptyset$, an abstract state is defined as discussed later. The function $h^+_{(B',B)}$ is defined below. Notice that case (1) (where $o_{d_1} = o_{d_2}$) is a special case of case (2), if $h^+_{(B',B)}$ is the identity function for $B' = B$.
- If $o_{d_1} \neq o_{d_2}$ and they have no common supertype, then p_1 and p_2 are not equivalent.

The function $h^{+}_{(B',B)}$ is defined based on the specialization functions h, which defines how elements of the object life-cycle B of an object type have been specialized in B' of its direct subtype. For two object types o and o', where o' is a direct or indirect subtype of o, and the object life-cycles of o and o' that are refered to as $B = (S, T, F, L, l)$ and $B' = (S', T', F', L', l')$, respectively, function $h^{+}_{(B',B)} : S' \cup T' \rightarrow S \cup T \cup \{\varepsilon\}$ is defined as follows:

1. $h^{+}_{(B',B)}(\varepsilon) := \varepsilon$
2. $h^{+}_{(B',B)}(e) := h(e)$ for all $e \in S' \cup T'$ if h is given as a specialization function between B' and B. Function h must be given if B' results from observation consistent specialization of B.
3. $h^{+}_{(B',B)}(e) := h^{+}_{(B'',B)}(h(e))$, where B' results from observation consistent specialization of B'', where the specialization of the object life-cycle B' from B'' is defined by function h.

If S_d is the empty set, i.e., no common states are defined as pre- or postates of p_1 and p_2, a new abstract state s is defined for the object life-cycle of o_d that comprises a subnet of activities and states E of o_d and that is assigned to S_d. This state must be defined as follows: There must be at least one state $s_1 \in S_{d_1}$ and one state $s_2 \in S_{d_2}$ such that $e_1 := h^{+}_{(B_1,B)}(s_1)$ and $e_2 := h^{+}_{(B_2,B)}(s_2)$ are elements of E and there is no abstract state s' comprising a subnet E', where $E' \subset E$ that fulfills this property, too.

Example: Consider the object life-cycle shown in Figure 6.24 and 6.25, which are integrated by generalization to the object life-cycle shown in Figure 6.27: Suppose that there is a pair of equivalent activities a_1 and a_2 that must be integrated to a by generalization; activity a_1 comprises port p_1, activity a_2 comprises port p_2, where p_1 and p_2 are equivalent; thus, a port p must be defined with a. Port p_1 requires an instance of STD-RES in state checkedOut, whereas port p_2 requires an instance of VIP-RES in state VIPServTUse. Port p will have object type RES, which is the common supertype of STD-RES and VIP-RES, as domain type.

During determination of the set of states S_d, state VIPServTUse of VIP-RES is abstracted to state using; then, S_d is the empty set as state using is different from state checkedOut. Therefore, a new abstract state afterCheckIn is introduced with RES that comprises using, checkOut, and checkedOut and that is used for port p.

Different pre- and postconditions: If different preconditions (or postconditions) c_1 and c_2 are defined in a_1 and a_2, the precondition (or postcondition, respectively) c of activity a must be less restrictive, such that $c_1 \Rightarrow c$ and $c_2 \Rightarrow c$. The most general way is to define *true* as condition in a. Yet, the user should determine the most specific condition, instead. If each c_i can be represented as a conjunction $x_{i1} \wedge \ldots \wedge x_{ik_i} \wedge x_{i(k_i+1)} \wedge \ldots \wedge x_{in_i}$, where all x_{ij} with $1 \leq j \leq k_i$ refer only to ports that are defined both in a_i and a, then c can be defined as $c = (x_{11} \wedge \ldots \wedge x_{1k_1}) \vee (x_{21} \wedge \ldots \wedge x_{2k_2})$.

7.3.2 Overview of the Process Steps

In Section 7.2, we gave an overview of heterogeneities that may occur between two activities to integrate. In this section, we learned that only very few heterogeneities are allowed between a_1 and a_2. Thus, we will introduce a *correction step* during integration in which the specifications of a_1 and a_2 are changed to omit all unallowed heterogeneities. For this correction step, we cannot define general correctness criteria like information-capacity preservation; yet, this step is a *change* of the view schemas and does actually not belong to integration. Yet, we include it into our integration process to discuss reasons for some heterogeneities and to give guidelines how to resolve them.

Integration of activity specifications comprises the following steps. These steps are discussed in more detail in the subsequent sections of this chapter.

1. *Schema correction:* The user has to change the view schemas to resolve heterogeneities because of *view-specific implicit ports, read ports vs. transfer ports, input ports vs. output ports, return port conflicts, different value-type precisions, object vs. related object* and *object type vs. value type.*

2. *Intra-type standardization:* The activity specifications of each view can be improved by information-capacity preserving restructuring. Thereby the heterogeneities because of *different value-type units, inadequate definitions of ports, multi-valued port vs. a set of ports, multi-valued port vs. single-valued ports,* and *limited domains* are resolved.

3. *Integration:* The activity specification diagram of each specified activity in the integrated schema is determined. Heterogeneities due to *naming conflicts, view-specific extended ports, different object types as domains* and *different pre- and postconditions* are resolved.

7.4 Integration Process

In this section, we treat the integration steps in more detail and discuss generalization of activity specifications, which corresponds to generalization of elements in object life-cycles.

Example: As a running example, we assume that two similiar object types RES1 and RES2 are defined in the views. Each of these object types includes activities create, checkIn, and bill as shown in Figure 7.1.

7.4.1 Schema Correction

The specification of the activities is changed by the system integrator or a user. Schema correction is a change of the views, for which no general correctness criteria can be given. Instead, we will discuss reasons for these heterogeneities as well as possible solutions and give examples.

View-specific implicit ports. A port exists *implicitly* in an activity specification if its actual value is constant or derivable from some other information and therefore, the port is not explicitly defined. Implicit ports are explicitly included in the activity specification. The fact that the value of this port is constant or derivable is represented by a condition.

Example: In v2, input port dep is defined with activity create. In v1, this port is missing as no deposits are cashed there. Port dep is added in v1 and a precondition dep=0 is added.

Read ports vs. transfer ports. If one input port is defined as a read port whereas the equivalent input port is a transfer port, one activity changes an object whereas the other activity does not change the object.

There will be only one situation, where this mismatch can occur (cf. Figure 7.2 for an illustration): Suppose that two similar object types o_1 and o_2 are defined. There is an activity a_1 defined in o_1 and an activity a_2 defined in o_2, where a_1 and a_2 are considered equivalent; thus, they are integrated to a common activity a.

Suppose that port p_1 of activity a_1 is a read port, whereas port p_2 of activity a_2 is a transfer port. Port p_1 has object type $\hat{o}_1$ as domain, port p_2 has object type $\hat{o}_2$ as domain. Suppose that $\hat{o}_1$ and $\hat{o}_2$ have a common supertype $\hat{o}$. There is a state s defined with $\hat{o}$ that is inherited by $\hat{o}_1$ without change and refined in $\hat{o}_2$ to states s_{21}, activity a_{21}, and state s_{22}, where a_{21} consumes from s_{21} and produces to s_{22}. Activity a_2 consumes an instance i_2

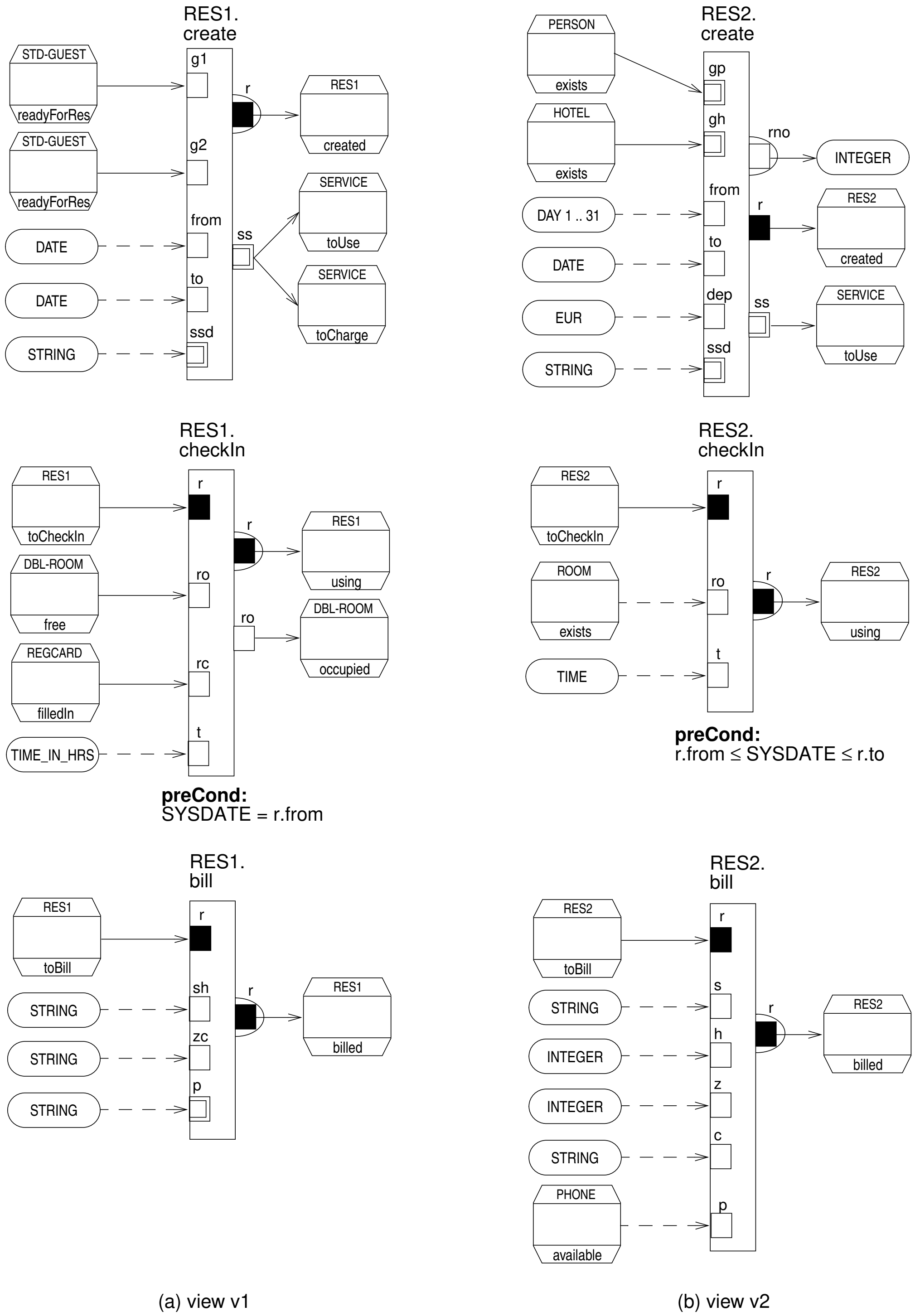


Figure 7.1: Example: activity specifications to integrate

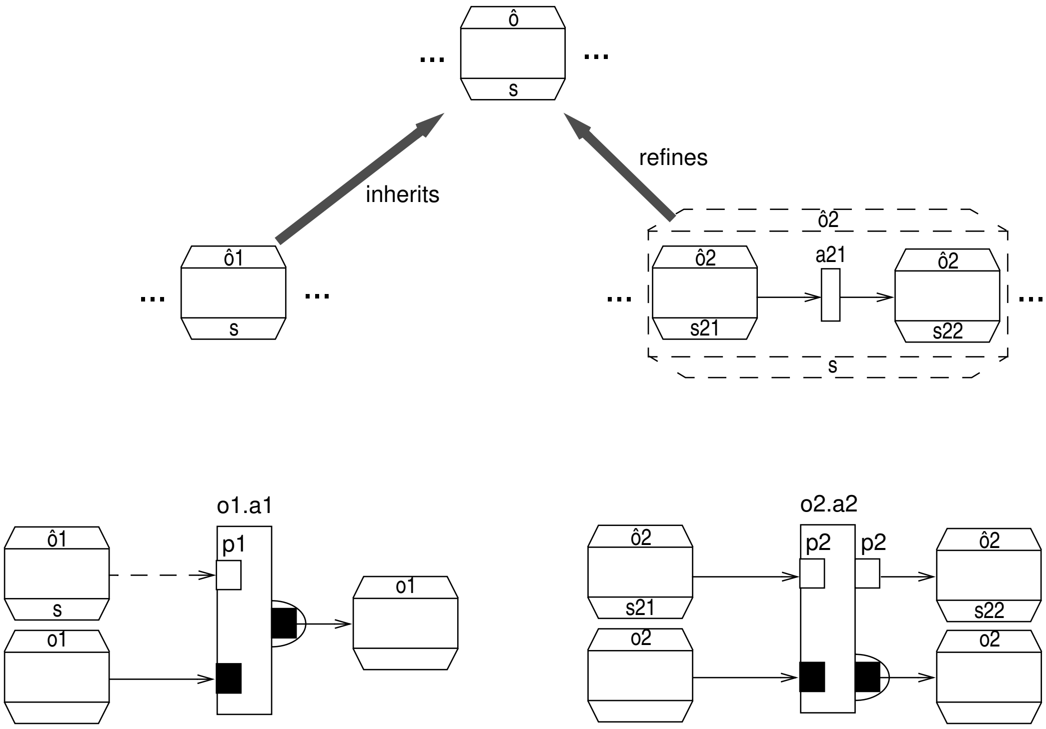


Figure 7.2: Read ports vs. transfer ports

of $\hat{o}_2$ in state s_{21}, invokes activity a_{21} on i_2 within a_2, such that i_2 is moved to state s_{22}. Activity a_1 reads an instance i_1 of $\hat{o}_1$ in state s, but does not change it, as it does not invoke an activity on i_1; thus, i_1 stays in state s.

In general, the read port will either be changed to a transfer port (as a transfer port says that the read object *can* be changed, but need not necessarily be changed) or the ports are not considered equivalent at all.

Example: In activity checkIn, both views define port ro, providing for information about the reserved room. Object type DBL-ROOM is a subtype of ROOM, where state exists has been refined to a subnet consisting of free, occupy, and occupied. In v1, activity checkIn invokes activity occupy, whereas in v2, an instance of ROOM is not changed as no activity occupy is defined there. The read port in view v2 is changed to a transfer port.

Input port vs. output port. We distinguish input ports, output ports, and input/output ports. If equivalent ports are specified differently in two activity specifications, the integrated activity specification has to define the port as input/output port. This situation will probably arise if a port is specified as input port in one view, whereas it is specified as input/output port in the other view. The modified object that is assigned to the input port is returned in the output port, again, in the latter, whereas it is not returned in the former. Further, we do not assume that an input port in one view is considered equivalent with an output port in the other view in practical use.

Example: Consider the specifications of activity checkIn: In view v1, port ro is defined as an input/outport port, whereas in view v2, it is defined as an input port. During schema correction, ro is defined as input/output port in v2, too.

Return port conflicts. If one activity specification defines a return port, whereas the other activity specification defines no return port, it has to be inserted in the latter. If both activities define different return ports, the system integrator has to change the specification of the return port in at least one view, such that a pair of equivalent ports are defined as return ports.

Example: The views define different return ports for activity create, namely output port r in v1 and output port rno in v2. View v2 is changed by specifying port r as the return port, too.

Different value-type precision. If equivalent ports have value types with different precision as domain, at least one of the respective domains must be changed such that both domains are equivalent.

Example: Both views define port t with activity checkIn, which defines the time of check-in. In v1, time is rounded to hours (cf. domain TIME_IN_HRS), whereas in v2, it is defined exactly (probably hours and minutes). To overcome this mismatch in precision, the domain in v1 is changed to TIME.

Object vs. related object. If an activity comprises a set of ports with domains $o_1, \ldots, o_n$ that are used to identify an instance of another object type o, the original ports should be replaced by a port with domain o.

Example: Activity create of RES2 has two ports gp with domain type PERSON and gh with domain type HOTEL. According to the object diagram shown in Figure 2.1, an instance of GUEST is identified by an instance of PERSON and an instance of HOTEL. If ports gp and gh are defined to identify a guest, the ports should be replaced by a single port g with domain type GUEST.

Object type vs. value type. If a set of ports is defined that represents the key attributes of an object, it should be replaced by a single port representing the object itself.

Example: Activity bill of in v1 has an input port p, which reads the phone number as a string value. If phone numbers are represented by a separate object type PHONE, which has attribute phoneNo as key, the domain of p should be changed to PHONE.

The activity specifications after schema correction are shown in Figure 7.3. Changes in the ports are depicted with boldface margins.

7.4.2 Intra-type Standardization

Information in the views may be defined inadequately. Therefore, the system integrator has to improve the activity specifications in the views.

Different value-type units. If different, but equivalent units are defined for two equivalent ports, the system integrator has to decide which one has to be used as domain of the port in the integrated schema. Usually, one of the domains used in the views will be chosen; but any other equivalent domain could be used, as well.

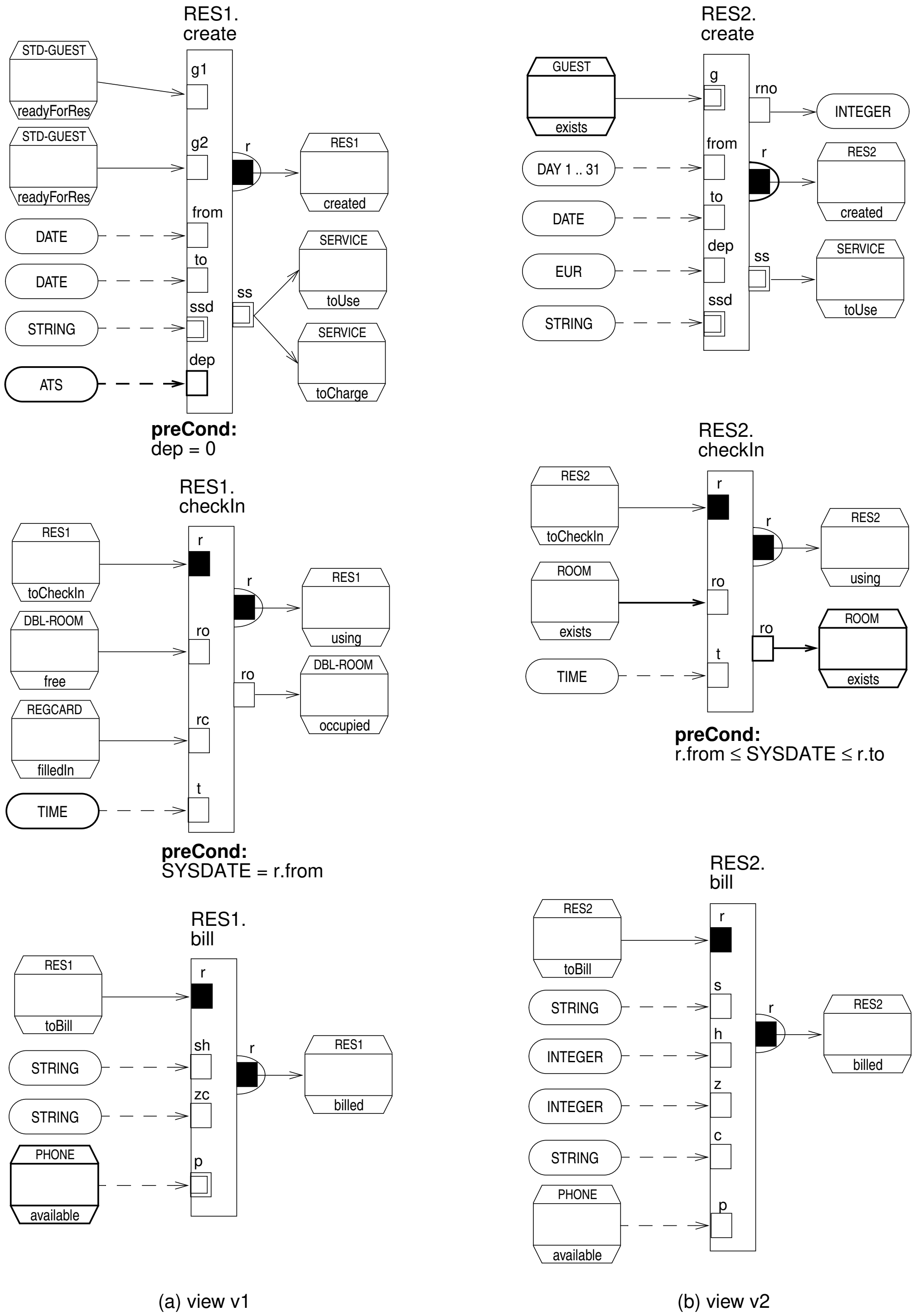


Figure 7.3: Example: activity specifications after schema correction

Example: Both views define port dep in activity create to define the deposit, which has to be paid for a reservation. In v1, the deposit is defined in Austrian Shillings (ATS), whereas in v2, it is defined in Euro (EUR). The system integrator decides to change the domain of dep in v1 to EUR.

Inadequate definition of ports. A port that represents several chunks of information should be split. A set of ports that represent together one chunk of information should be merged. The decision will be taken by the system integrator as there will not be a formal check whether some information should be split or merged.

Example: Activity bill has to be provided with information about the address where the bill should be sent to. In view v1, activity bill has two ports sh and zc, where the former takes information about street and house number whereas the latter takes information about the zip code and the city. The ports are split to s, h, z, and c for street, house number, zip code, and city, respectively.

Multi-valued port vs. a set of ports. If a set of single-valued ports with the same domains is defined, whereby a set of instances is provided to the activity for the same purpose, one multi-valued port should be defined instead. The number of provided instances is restricted in the original definition by the number of single-valued ports. In the new definition, this restriction must be expressed by a condition.

Example: Activity create in view v2 contains one multi-valued port g for the set of instances of GUEST, whereas create of v1 comprises two single-valued ports with domain GUEST. In v1, these ports are replaced by one multi-valued port g and the precondition $\mid \mathsf{g} \mid = 2$ is added.

Multi-valued vs. single-valued ports. A port may be defined single-valued in a view and multi-valued in the other view, if the set of assigned values is limited to one value in the former. Such single-valued ports are changed to multi-valued ports, the restriction to one object is represented by a condition. Notice that this case is a special case of *multi-valued port vs. a set of ports*.

Example: In activity bill of view v2, only one phone number can be assigned to port p, whereas in v1, a set of phone numbers can be assigned to the port. The port is defined multi-valued in v2, too, and the condition $\mid \mathsf{p} \mid = 1$ is added.

Limited domains. If the domain of a port is a value-type and the domain of the equivalent port in the other view is a more restricted value-type, the latter can be changed to the value-type of the former, where the restriction is represented by a condition.

Example: Activity create in v2 comprises two ports from and to to define the reservation period. The domain of port from in v2 is limited to DAY 1 .. 31, which means that only a date within the current month can be specified as actual value. The domain of from in v2 is changed to DATE and a precondition, from IN MONTH(SYSDATE), is added.

The activity specifications after intra-type standardization are shown in Figure 7.4.

7.4.3 Integration

In the final integration step, the remaining heterogeneities are resolved: *naming conflicts, view-specific extended ports, different object types as domains,* and *different pre- and postconditions*. The integration step has already been introduced in detail in Section 7.3, such that we will mainly give some examples in this section.

Naming conflicts. Naming conflicts between p_1 and p_2 can be resolved by determining a new name for p. The correspondence between ports in the integrated activity a and the original activities a_1 and a_2 can be represented by a function, similar to the specialization function h defined between object life-cycles. Alternatively, naming conflicts can be resolved by renaming ports in a_1 and a_2.

Example: For better readability, all equivalent ports carry the same names. Thus, no naming conflicts arise in the example.

View-specific extended ports. Extensions in an activity specification of a view are omitted during generalization.

Example: Activity create in v2 includes output port rno, which returns the reservation number and which is not defined in v1. Activity checkIn in v1 contains port rc, which provides information on the registration card that must be filled in for each reservation. During integration by generalization, these ports are omitted.

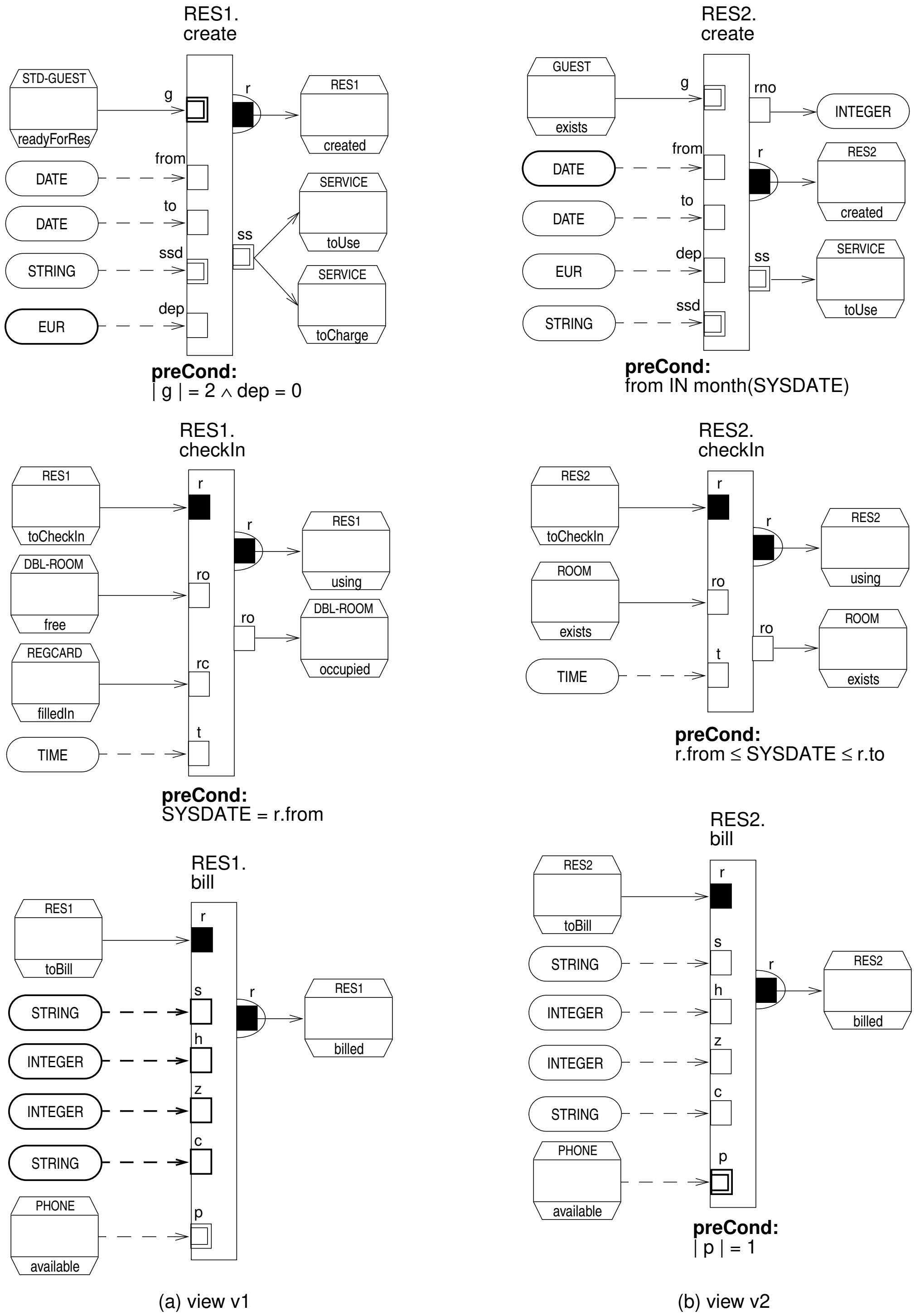


Figure 7.4: Example: activity specifications after intra-type standardization

Different object types as domains. If different domains are defined with equivalent ports, a more general domain is defined for the corresponding port in the integrated schema.

Example: Consider the specifications of activity create: Port g in v1 requires instances of STD-GUEST in state readyForRes, whereas port g in v2 requires instances of GUEST in state exists, where STD-GUEST is a subtype of GUEST and state readyForRes belongs to the refinement of exists. In the integrated schema, the domain of g is defined as in v2. Further, port ss produces instances of SERVICE in states toUse and toCharge in v1, whereas it produces instances of SERVICE in state toUse in v2. The more general specification, i.e., only state toUse is defined in the integrated schema.

Different pre- and postconditions. The pre- and postconditions in the new integrated object type must be equivalent to or less restrictive than the respective conditions in the subtypes.

Example: Different preconditions are defined with the activities in the views: The disjunction of the preconditions of RES1.create of v1 and RES2.create of v2 is defined with create in RES. View v1 requires that checkIn is only invoked on the day that is defined by property from of the respective instances of RES1, whereas v2 allows to invoke the activity at any day between from and to. The condition in the integrated schema is the disjunction of both conditions. As the condition in v1 implies the condition in v2, the part of the condition defined in v1 can be omitted. A precondition for bill is only defined in v2. As the precondition for bill in v1 is *true* by default, the disjunction is *true*, too, such that no precondition is explicitly defined for bill in RES.

The activity specifications in the integrated schema are shown in Figure 7.5.

7.4.4 Generalization of Elements

An activity specification is defined for an activity in at most one view. Similar to the problem of generalization of elements of object life-cycles, there may be the situation that activity specifications defined in the integrated schema must be modified after integration, too.

Consider the following case: View v_1 defines three object types o_1 with subtypes o'_{11} and o'_{12}. View v_2 defines object type o'_{21}, which is equivalent to o'_{11}. In all object types, there is an activity a; this activity is *unspecified* in

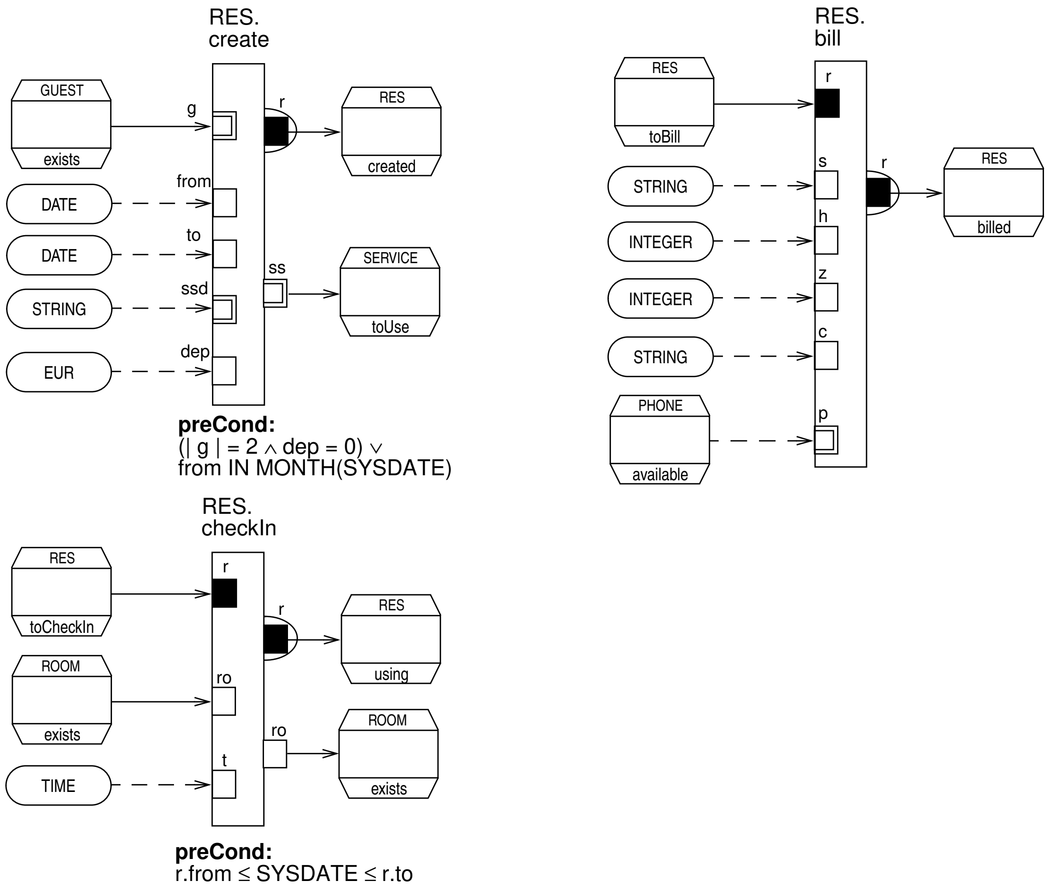


Figure 7.5: Integrated activity specifications

o_1 and o'_{11} and *specified* in o'_{12} and o'_{21}. The integrated schema will contain three object types: (1) object type o, which corresponds to o_1, (2) object type o'_1, which results from integrating o'_{11} and o'_{21} by specialization, and (3) object type o'_2, which corresponds to o'_{12}. As activity a is unspecified in o_1, it is unspecified in o, too; as a is specified in o'_{21} and unspecified in o'_{11}, it is specified in o'_1 (as it is taken from o'_{21}); finally, activity a is specified in o'_2 as it is specified in o'_{12}.

As a result, activity a is specified in the subtypes o'_1 and o'_2, but unspecified in the common supertype o. Thus, an activity specification for a in o is defined by generalization of the activity specifications of a in the subtypes, following the approach of integration by generalization.

In general, there may be an object type o with a set of n subtypes $o'_1, \ldots, o'_n$, where an activity specification for an activity a is defined in all o'_i, such that an activity specification for a must be defined in o, too. This activity specification must include all common ports and the pre- and postcondition must not be more restrictive than the pre- and postconditions defined with a in the subtypes. Then, the activity specification of a in each subtype o'_i is a correct specialization of the specification of a in o.

Chapter 8

Conclusion

Contents

This chapter summarizes this thesis, giving an overview of the contribution of this work and discussing further issues. Further, we will discuss future work in this area and in related fields of research. This chapter is structured as follows: We summarize this thesis in Section 8.1, we discuss some issues on practical use of the integration approach in Section 8.2, and give an outlook for future work in Section 8.3.

8.1 Summary

This thesis introduces a process to define object types and their behavior in an object-oriented conceptual schema by integrating object types defined in views. Integration of behavior is particularly considered useful in the area of design of business processes, where a business process shows the overall processing of business cases. We use *Object/Behavior diagrams* (OBD) to define the view schemas and the conceptual schema that results from integration.

Object/Behavior diagrams. Object/Behavior Diagrams are a graphical design notation for object-oriented systems. In Chapter 2, modeling structure and behavior of objects in OBD was introduced. We summarized specialization of object types and consistency rules for specialization, which were originally introduced in several publications (e.g., [KS96b, SS95, SS97]). Specialization was the main correctness criterion for integrating views. As we focussed on integration of behavior, the specialization of object life-cycles was discussed in detail.

Workflow transparency. Object life-cycles are used to present the overall processing of objects. As information systems are often used to model business processes, object types may represent structure and behavior of business objects, where the definition of structure (in Object Diagrams) represents the structure of business objects and object life-cycles (of Behavior Diagrams) represent the business process. In Chapter 3, we presented a two-schema architecture for modeling business processes and workflows, where the former are defined using object life-cycles of OBD and represent external, time-invariant business rules, whereas the latter are defined using a workflow model and represent internal, varying business rules.

Separating business processes from workflows is a very important task in view integration as the conceptual schema should only reflect business processes, but no workflows. Yet, the users who define views may perceive and model workflows, such that the overall business process must be extracted from the workflows before integration.

Integration of views. Chapters 4, 6, 5, and 7 comprise the core topics of this work, integration of behavior of object types. In Chapter 4, we presented an overview of view integration with the main integration steps. We gave a brief overview how the view schemas can deviate from the integrated schema and motivated a sequence of process steps to define the conceptual schema from the view schemas. We distinguished view preparation, where the view schemas were modified to be prepared for integration, and view integration, where the prepared view schemas were used to define an integrated conceptual schema.

We dealt with view preparation in Chapter 5. There we introduced extracting business processes from workflows to obtain views that consider only aspects of business processes and inter-type standardization to split object types that represent different entities or entities in different roles, to represent relationships by association types, and to define consistent specialization hierarchies. After these preparatory steps, object types could be integrated pair-by-pair.

In Chapter 6, the integration of object life-cycles was presented. We treated two main integration methods, integration by specialization to define the object life-cycle of an object type based on equivalent object types in the views, and integration by generalization to define the object life-cycle of a new root object type of a type hierarchy.

Chapter 7 presented definition of activity specifications in the integrated schema, where we argued that only integration by generalization is needed.

8.2 Practical Use of View Integration

The integration approach introduced in this work consists of a set of integration steps that are applied to the views sequentially. We point out here some issues on integration of views that should be considered by the system integrator.

Workflow aspects. The decision whether some aspect of the object life-cycle should be considered as a workflow aspect or a business process aspect may be difficult, as the question, whether some rule is external or internal cannot be answered definitely. Thus, during extraction of business processes from the workflow, internal and external aspects have to be separated carefully, maybe after consulting the users who modeled the respective view of the object life-cycle. The decision whether some constraint belongs to the business process or workflow can heavily influence the integrated schema. Suppose that two views comprise an object life-cycle for a certain object type. In one view, two activities (or subnets) a and b are defined in sequence, such that a has to be performed before b, whereas in the other view, a and b can be performed in parallel. Either the former object life-cycle defines the correct sequence of activities, which has been omitted in the latter (by view generalization) such that the integrated business process has to define a sequence, too; or the former considers some internal rule according to which a and b are performed sequentially, whereas the latter considers the pure business process; then the integrated schema has to define a and b in parallel.

Inter-type standardization. During inter-type standardization, object types that represent different entities or entities in different roles are split to define an object type for each kind of entity or role. The decision about which information should be considered as separate entities may be difficult if each view is considered separately. Suppose that object type HOTEL has an attribute city with domain STRING, where the location of each hotel is stored. If cities are not further considered in the view, no separate object type CITY will be defined. But, there may be another view that considers cities and their behavior in more detail and therefore defines a separate object type.

Thus, the decision about which information should be represented by object types of its own may be taken by comparing the views and examining which entities are represented by separate object types in one view, whereas they are only refered to by attributes in the other view.

During standardization of type hierarchies, object types have to be split to hierarchies if different kinds of objects should be distinguished. The decision about which object types should be split and according to which criterion they should be split will usually be taken by comparing the specialization hierarchies in all views.

At the end of inter-type standardization, equivalences and similarities between object types in the views must be determined. The decision about equivalences should be based on the properties defined with the object types in the views and the considered extension. But notice that no databases and thus, no extensions, exist. Instead, we assume that each modeler defines his/her view based on a set of actual business cases of everyday business such that the business objects considered during modeling of the views should be considered when the equivalence of object types is determined.

Intra-type standardization. Intra-type standardization consists of standardization of activity specifications, standardization of conditions, and schema improvement. Intra-type standardization changes the definition of the behavior of single object types. The decision about whether an activity should be replaced by a set of alternative activities or whether a state should be replaced by a textual condition (or vice versa) will probably depend on definitions in the other views to integrate, similar as the decision about inter-type standardization will be supported by information about the other views.

During schema improvement, labels and redundant states are defined. The goal of schema improvement is to identify those parts of the object life-cycle that consider a certain chunk of an object. The proper identification of labels is necessary to identify correspondences with labels in the other views and to integrate activities and states that are labeled with common labels.

Identification of correspondences. Correspondences are identified based on standardized view schemas. Several conflicts and contradictory information between the views may be determined in this step. The system integrator may find out that workflow aspects are still included in the schema, that schema standardization produced an inadequate result or that object types that supposed to be equivalent are not equivalent at all; e.g., view-specific alternatives may indicate that object types are not equivalent, but one object type is a subtype of the other one: Suppose that object type RES1 is defined in one view with two alternatives use and cancel. Object type RES1 is considered equivalent with object type RES2, which only includes use, but not cancel. This object types may either be equivalent, where object type RES1 includes an unnecessary alternative or RES1 is a supertype of RES2, where the latter omits the alternative of cancelation during specialization.

Integration. After determination of corresponding elements, the integrated object life-cycle is defined almost automatically. The system integrator or an integration tool will check whether the resulting object life-cycles fulfill the correctness criteria. If the results are wrong, conflicting correspondences will have been determined. Suppose that one view defines two activities a and b, where a must be executed before b. Another view defines activities c and d, where d must be executed before c. If the system integrator determines an equivalence between a and c and an equivalence between b and d, the resulting object life-cycle will be wrong, which is not surprising as the views enforce different sequences of execution.

Activity specifications. The definition of activity specifications is reduced to the problem of generalization. The rules of specialization of activity specifications in OBD allow only very restricted modifications of an activity in a subtype compared to the respective activity in the supertype. Therefore, several modifications may be necessary in the activity specifications of the views before they can be integrated. Therefore, our approach defines lots of heterogeneities that must be resolved before integration during schema correction.

8.3 Future Work

This thesis focusses on the integration of views of databases. In future, research can be continued in some directions, which are briefly discussed in this section. (1) There are further issues of view integration in the realm of workflow management. (2) Integration of views can be considered in the realm of federated database systems, where schemas of independent databases have to be integrated. (3) Tools should be developed that support the system integrator during the integration process.

8.3.1 Further Integration Issues for Workflow Management

Information systems are often used for modeling business processes and workflows. Business processes are defined by life-cycle diagrams of OBD, whereas workflows are defined by a model of its own (cf. Chapter 3). Bichler [Bic98] presented a very comprehensive approach of workflow modeling, which comprises modeling of data, processes, organizations, and authorizations, where business processes and workflows are distinguished in process modeling.

In this work, we treated integration of OBD schemas, which represent the objects' behavior according to the business process. Future work could discuss, how workflows, organization schemas, and authorization schemas have to be integrated.

Workflow systems are not only used to define how objects have to be processed, but they allow users to query the *history* of business objects [JB96], e.g.,
(1) "When was the value for property bill defined for a reservation?"
(2) "Was activity waiveDeposit performed on this object?"
(3) "When did a reservation move to state confirmed?"
(4) "Who started activity issueBill?"

The information system has to store information about processing objects in a way that such questions can be answered. The questions deal with changes of an object's properties (cf. (1)) as well as execution of activities (cf. (2)) and residence in states (cf. (3)). Question (4) requires information about the human agent who performed an activity.

This information may be required for queries about business cases or for further processing of business objects, e.g., the date of billing is important for maintaining reminders and for accounting; information about the agent who performs activity confirm is necessary to calculate the commission of employees.

Temporal and other workflow-specific information about processing objects can be explicitly defined in the schema by properties of objects and ports of activity specifications. Alternatively, this information can be maintained by the database system: Information about timestamps can be stored in a temporal database system [SA86, TCG$^+$93]; information about agents and other workflow-specific information can be defined by a meta model that requires that information about the agent is automatically recorded with each invocation of an activity.

How do these facts influence integration of schemas? One user may define a view, where a property is explicitly defined for some temporal or other workflow-specific information (e.g., date of billing for a reservation or information about the agent who invoked an activity), whereas another user may find that this information is provided by the information system and omits respective ports or properties in his/her view.

8.3.2 Integration of Database Schemas

A lot of research has been done so far in the area of schema integration (cf. the discussion in Chapter 1). As mentioned there, most approaches consider integration of structure only. Integration of object life-cycles, particularly in the realm of business processes should be investigated in future work.

Integration of object life-cycles in federated database systems is necessary for several reasons:

Complete specification of global schemas in federated databases: It is the continuation of the work in the area of database integration to be able to define integrated database schemas with complete definitions of structure and behavior. Integration started with integration of structure, continued with integration of activities, and now should be extended to object life-cycles.

Global view of local processing of objects: In a previous paper [PS98a], we assumed that enterprises or organizations may run business processes of their own. After a merger or an acquisition of such enterprises, business objects will be further processed according to the local business processes for a while, but an integrated business process is defined that comprises the common elements of the local business processes and that provides a common view of local processing of business objects according to the local business processes. Such a view may be useful for a joint management, which wants to *observe* local processing of business objects.

Cooperations in business: Enterprises cooperate in their everyday business more and more frequently. Consider a travel agency that is allowed to reserve rooms in hotels. The creation activity of a reservation object may be executed in the travel agency although the reservation object "belongs" to the hotel. Several approaches (as mentioned in the introduction) deal with cooperation between organizations. But these approaches usually are restricted to the exchange of information about business objects by some standardized data formats, like EDI.

In [SL90], several system architectures for federated database systems were discussed. One approach is to fully integrate all local databases into one global schema. This may result in a very large global schema. Yet, such global schemas are often not useful, as nobody needs all the information defined in the global schema. Another approach has been introduced where each component database defines export schemas with a part of the information of the database; several federated schemas can be defined that integrate some of the export schemas. Then, no single global schema is defined, but several small schemas with overlapping information. In loosely coupled federated database systems, parts of schemas may be exported by one database, refered to as export schemas, which can be imported by another database.

For any of these architectures, some kind of integration of two or more schemas is necessary. If object life-cycles are defined in the local schemas to represent the overall behavior of objects, they have to be integrated to an integrated schema.

The system integrator may decide to define one global integrated schema or to define several smaller schemas, each one comprising parts of the local business processes. For cooperation in business, an approach with import and export schemas may be useful: Cooperating enterprises will not integrate all information of their databases, but only integrate or exchange information in some strictly limited areas of business, e.g., a travel agency may only access information about availability of rooms and may create reservations, but it will not be allowed to read the payrolls of the hotel's staff.

8.3.3 Tool Support

The system integrator should not integrate views by hand. View definition and view integration can be supported by tools: An editor should be provided to the users who define a view and an integration tool should be provided to the system integrator, which supports the system integrator in comparing and integrating the view schemas.

Recently, an editor has been developed for *Active Object/Behavior Diagrams (AOBD)* [KT97, LOKT98]. This editor supports definition of Object/Behavior Diagrams with active extensions. If users define views using an editor, the integration of views will be simplified as the editor enforces compliance of the view schemas to the OBD model such that model conflicts can be avoided.

An integration tool supports the system integrator during the integration process. The tool will show the view schemas at different levels of detail, i.e., show only object types without their inner definition and zoom into the structure or behavior definition of a single object type if requested.

The user can compare the view schemas at the considered level of detail and determine correspondences by marking sets of elements with the mouse or keyboard and define them as corresponding by a mouse-click. Further, information can be restructured in the application window.

Correspondences between elements of the considered views are not determined independently from each other; yet, a detected correspondence will help to find out further correspondences. The integration tool can propose possible correspondences to the system integrator and check the correctness of correspondences, such that contradictory correspondences are detected as soon as possible.

We will illustrate some of these checks in the realm of integration of object life-cycles by generalization. The correctness criterion for correct integration by generalization is that the integrated object life-cycle is an observation consistent generalization of the given object life-cycles, which can be checked after complete determination of correspondences.

As conflicts between correspondences should not be detected at the end of determining correspondences, but as soon as a conflict occurs, the integration tool can check each newly determined correspondence and detect errors. In the following list, we consider integration of object life-cycles of similar object types defined in the view schemas v_1 and v_2 by generalization; elements of v_1 and v_2 will be refered to by index 1 and 2, respectively.

1. Correctness of equivalences of elements can be checked as a state (activity, or label) can only be equivalent to a state (activity, or label, respectively).

2. If activity a_1 is equivalent to activity a_2 and state s_1 is equivalent to state s_2, and an arc is defined between a_1 and s_1, then an arc has to exist between a_2 and s_2, too.

3. If an activity a_1 or a state s_1 is defined as corresponding to a_2 or state s_2, respectively, but the corresponding elements have no common labels, the tool will point this fact out to the system integrator. Determination of correspondences can only be completed after a common label that is assigned to both elements has been determined.

4. All arcs have to be labeled with the same set of corresponding labels. Suppose that state s_1, activity a_1, and label x_1 are equivalent to state s_2, activity a_2, and label x_2, respectively. If an arc is defined between a_1 and s_1 as well as between a_2 and s_2, then label x_1 is assigned to the arc between a_1 and s_1 if and only if label x_2 is assigned to the corresponding arc between a_2 and s_2.

5. If element inclusions or set correspondences are detected, the subnets have to be abstracted to a single element. The tool can check whether

this abstraction obeys the rules of observation consistency, i.e., if an activity a_1 or a state s_1 corresponds to a subnet of activities and states E_2, it must be possible to abstract E_2 to a single activity or state, respectively, without violating the refinement rules defined in Table 2.1.

6. If an activity is considered common then all pre- and poststates of this activity that carry at least one common label have to be common, too. The tool will advise the user to find correspondences for these states.

Tools should support backtracking steps in the integration process. As discussed earlier in this chapter, the integration process will possibly not be performed as straight-forward as introduced in this work, but several backtracking steps will be necessary as conflicts or wrong definitions that should be resolved earlier, are detected in a later phase of integration. Tools are needed to allow backtracking or even to hide the strict separation of integration steps from the user.

List of Figures

List of Tables

Bibliography

[AAH98] N. R. Adam, V. Atluri, and W.-K. Huang. Modeling and Analysis of Workflows Using Petri Nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998.

[AASY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In J. M. Peckman, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, Arizona*, volume 26 of *SIGMOD Record*, pages 417–427. ACM, June 1997.

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In J. Clifford and R. King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado*, SIGMOD Record, 1991.

[AS95] T. Allweyer and A.-W. Scheer. Modellierung und Gestaltung adaptiver Geschäftsprozesse. Technical Report Heft 115, Universität des Saarlandes, Institut für Wirtschaftsinformatik, May 1995.

[BCW91] M. Bouzeghoub and I. Comyn-Wattiau. View Integration by Semantic Unification and Transformation of Data Structures. In H. Kangassalo, editor, *Proceedings of the 9th International Conference on the Entity-Relationship Approach (ER '90), Lausanne, Switzerland*. Elsevier Science Publishers B.V. (North-Holland), 1991.

[BE96] O. Bukhres and A. Elmagarmid. *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*. Prentice-Hall, 1996.

[Ber92] E. Bertino. A View Mechanism for Object-Oriented Databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings of* *the 3rd International Conference on Extending Database Technology, Vienna, Austria*, volume 580 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 1992.

[BGV91] W. Brauer, R. Gold, and W. Vogler. A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 1–46. Springer, 1991.

[Bic98] P. Bichler. *Conceptual Design of Secure Workflow Systems: An Object-Oriented Approach to the Uniform Modeling of Workflows, Organizations, and Security.* PhD thesis, Universität Linz, 1998.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[BPS97a] P. Bichler, G. Preuner, and M. Schrefl. Workflow Transparency. Technical Report 97.02, University of Linz, Department of Business Information Systems, 1997.

[BPS97b] 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*, pages 423–436. Springer, June 1997.

[BS94] P. Bichler and M. Schrefl. Active Object-Oriented Database Design Using Active Object/Behavior Diagrams. In J. Widom and S. Chakravarthy, editors, *Proceedings of the 4th International Workshop on Research Issues in Data Engineering (RIDE'94), Houston, Texas*, pages 163–171, 1994.

[BS95] P. Bichler and M. Schrefl. Inheritance of Business Rules. In C. Eckert, H.-J. Klein, and T. Polle, editors, *Proceedings of the Seventh Workshop "Grundlagen von Datenbanken"*, Hildesheimer Informatik-Berichte, pages 20–24. Universität Hildesheim, Institut für Informatik, June 1995. (extended abstract).

[Cat95] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93 Release 1.2.* Morgan Kauffmann Publishers, 1995.

[CCPP96] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Semantic WorkFlow Interoperability. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Proceedings of the 5th International Conference on Extending Database Technology (EDBT), Avignon, France*, volume 1057 of *Lecture Notes in Computer Science*, pages 443–462. Springer, 1996.

[CE91] B. Czejdo and D. W. Embley. View Specification and Manipulation for a Semantic Data Model. *Information Systems*, 16(6):585–612, 1991.

[Coa96] The Workflow Management Coalition. Terminology & Glossary. The Workflow Management Coalition Specification — Document Number WFMC-TC-1011, June 1996.

[Con97] S. Conrad. *Föderierte Datenbanksysteme*. Springer, 1997. (in German).

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles & Systems*. McGraw-Hill International Editions, 1984.

[CST98] S. Conrad, I. Schmitt, and C. Türker. Considering Integrity Constraints During Federated Database Design. In S. M. Embury, N. J. Fiddian, W. A. Gray, and A. C. Jones, editors, *Proceedings of the 16th British National Conference on Databases (BNCOD), Cardiff, Wales, UK*, volume 1405 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 1998.

[DE94] M. Dobrovnik and J. Eder. Adding View Support to ODMG-93. In *Advances in Databases and Information Systems: ADBIS '94, Proceedings of the International Workshop of the Moscow ACM SIGMOD Chapter*, 1994.

[DO96] J. Desel and A. Oberweis. Petri-Netze in der Angewandten Informatik: Einführung, Grundlagen und Perspektiven. *Wirtschaftsinformatik*, 38(4):359–366, 1996.

[Dob95] M. Dobrovnik. *Externe Schemata in objektorientierten Datenbanksystemen: Logische Datenunabhängigkeit durch Änderungen über Sichten*. PhD thesis, Universität Klagenfurt, Fakultät für Wirtschaftswissenschaften und Informatik, 1995.

[EE94] J. Ebert and G. Engels. Observable or Invocable Behaviour — You Have to Choose. Technical report, Koblenz University, 1994.

[EJ96] L. Ekenberg and P. Johannesson. A Formal Basis for Dynamic Schema Integration. In B. Thalheim, editor, *Proceedings of the 15th International Conference on Conceptual Modeling (ER), Cottbus, Germany*, volume 1157 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 1996.

[EKW92] D. W. Embley, B. D. Kurtz, and S. N. Woodfield. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Prentice-Hall, Inc., 1992.

[Emb98] D. W. Embley. *Object Database Development: Concepts and Principles*. Addison-Wesley, 1998.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 2nd edition, 1994.

[FE97] H. Frank and J. Eder. Integration of Behaviour Models. In *Proceedings of the ER '97 Workshop on Behavioral Models and Design Transformations*, 1997.

[Fra96] H. Frank. *View Integration für objektorientierte Datenbanken*. PhD thesis, Institut für Informatik der Universität Klagenfurt, October 1996.

[FS97] M. Fowler and K. Scott. *UML distilled: applying the standard object modeling language*. Addison-Wesley, 1997.

[GK96] V. Gruhn and M. Kampmann. Modellierung unternehmensübergreifender Geschäftsprozesse mit FUNSOFT-Netzen. *Wirtschaftsinformatik*, 38(4):383–390, 1996.

[GLN92] W. Gotthard, P. C. Lockemann, and A. Neufeld. System-Guided View Integration for Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 4(1):1–22, February 1992.

[GPNS92] J. Geller, Y. Perl, E. Neuhold, and A. Sheth. Structural Schema Integration with Full and Partial Correspondence using the Dual Model. *Information Systems*, 17(6):443–464, 1992.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[GSCS95] M. García-Solaco, M. Castellanos, and F. Saltor. A Semantic-Discriminated Approach to Integration in Federated Databases. In S. Laufmann, S. Spaccapietra, and T. Yokoi, editors, *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS), Vienna, Austria*, pages 19–31, 1995.

[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.

[Hal96] U. Halter. Workflow-Integration im Kreditbereich. In H. Österle and P. Vogler, editors, *Praxis des Workflow-Managements: Grundlagen, Vorgehen, Beispiele*, chapter IX, pages 171–198. Vieweg, 1996.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science Of Computer Programming*, 8:231–274, 1987.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 10(2):201–260, 1987.

[HK95] H. Herbst and G. Knolmayer. Ansätze zur Klassifikation von Geschäftsregeln. *Wirtschaftsinformatik*, 37(2):149–159, 1995.

[HM85] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[HS93] A. Heuer and P. Sander. The LIVING IN A LATTICE rule language. *Data & Knowledge Engineering*, 9:249–286, 1993.

[Hür94] W. L. Hürsch. Should Superclasses be Abstract? In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP), Bologna, Italy*, volume 921 of *Lecture Notes in Computer Science*, pages 12–31. Springer, 1994.

[HZ90] S. Heiler and S. Zdonik. Object Views: Extending the Vision. In *Proceedings of the 6th International Conference onf Data Engineering, Los Angeles, USA*, pages 86–93, IEEE Computer Society Press, 1990.

[JB96] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation.* International Thomson Computer Press, 1996.

[Joh93] P. Johannesson. Schema Transformations as an Aid in View Integration. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Proceedings of the 5th International Conference on Advanced Information Systems Engineering (CAiSE), Paris, France*, volume 685 of *Lecture Notes in Computer Science*, pages 71–92. Springer, 1993.

[Joh94] P. Johannesson. Schema Standardization as an Aid in View Integration. *Information Systems*, 19(3):285–290, 1994.

[Joh97] P. Johannesson. Supporting schema integration by linguistic instruments. *Data & Knowledge Engineering*, 21:165–182, 1997.

[KBG89] W. Kim, E. Bertino, and J. F. Garza. Composite Objects Revisited. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, SIGMOD RECORD, pages 337–347. ACM Press, 1989.

[KFM+96] W. Klas, P. Fankhauser, P. Muth, T. C. Rakow, and E. J. Neuhold. Database Integration Using the Open Object-Oriented Database System VODAK. In O. A. Bukhres and A. K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*, chapter 14, pages 472–532. Prentice-Hall, 1996.

[KGF98] D. D. Karunaratna, W. A. Gray, and N. J. Fiddian. Establishing a Knowledge Base to Assist Integration of Heterogeneous Databases. In S. M. Embury, N. J. Fiddian, W. A. Gray, and A. C. Jones, editors, *Proceedings of the 16th British National Conference on Databases (BNCOD), Cardiff, Wales, UK*, volume 1405 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 1998.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, USA*, SIGMOD Record, pages 393–402. ACM Press, June 1992.

[KR95] H. A. Kuno and E. A. Rundensteiner. Materialized Object-Oriented View in *MultiView*. In O. A. Bukhres, M. Tamer Özsu, and M.-C. Shan, editors, *Proceedings RIDE-DOM '95, Fifth International Workshop on Research Issues in Data Engineering*, pages 78–85, 1995.

[KR96] H. A. Kuno and E. A. Rundensteiner. The *MultiView* OODB System: Design and Implementation. *Theory and Practice of Object Systems*, 2(3):202–225, 1996.

[Kri97] B. B. Kristensen. Subject Composition by Roles. In M. E. Orlowska and R. Zicari, editors, *Proceedings of the 1997 International Conference on Object Oriented Information Systems (OOIS), Brisbane, Australia*, pages 181–196. Springer, 1997.

[KRSR98] G. Kappel, S. Rausch-Schott, and W. Retschitzegger. Coordination in Workflow Management Systems — A Rule-Based Approach. In W. Conen and G. Neumann, editors, *Coordination Technology for Collaborative Applications — Organizations, Processes, and Agents*, volume 1364 of *Lecture Notes in Computer Science*, pages 99–120. Springer, 1998.

[KS86] C. H. Kung and A. Sølvberg. Activity Modeling and Behavior Modeling. In T. W. Olle, H. G. Sol, and A. A. Verrijn-Stuart, editors, *Proceedings of the IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies: Improving the Practice, Noordwijerhout, The Nederlands*, pages 145–171. North-Holland, 1986.

[KS91a] G. Kappel and M. Schrefl. Object/Behavior Diagrams. In *Proceedings of the 7th International Conference on Data Engineering (ICDE'91)*, pages 530–539, Kobe, Japan, April 1991.

[KS91b] W. Kim and J. Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, pages 12–18, December 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 the Entity-Relationship Approach (ER), Karlsruhe, Germany*, volume 645 of *Lecture Notes in Computer Science*, pages 41–61. Springer, 1992.

[KS94] G. Kappel and M. Schrefl. Inheritance of Object Behavior — Consistent Extensions of Object Life Cycles. In *Proceedings of the Second International East/West Database Workshop*. Springer, 1994.

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

[KT97] W. Kraus and T. Thalhammer. Design and Implementation of a Graphical Editor for Active Object/Behavior Diagrams. Master's thesis, University of Linz, Department of Business Information Systems, 1997.

[Lan97] P. Lang. *Conceptual Active Object-Oriented Database Design*. Dissertation, Universität Linz, September 1997.

[LC95] W.-S. Li and C. Clifton. Semint: A System Prototype for Semantic Integration in Heterogeneous Databases. In *Proceedings of the 1995 ACM SIGMOD Conference*, 1995.

[Lip91] S. B. Lippman. *C++ Primer*. Addison-Wesley, 2nd edition, 1991.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[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), Orlando, USA*, page 599. IEEE Computer Society, February 1998.

[LOS96] 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.

[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), Birmingham, UK*, April 1997.

[Mey92] B. Meyer. *Eiffel: The language.* Prentice Hall, 1992.

[MIR93] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In R. Agrawal, S. Baker, and D. Bell, editors, *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB), Dublin, Ireland*, pages 120–133, 1993.

[MIR94] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. Schema equivalence in heterogeneous systems: bridging theory and practice. *Information Systems*, 19(1):3–31, 1994.

[Mir97] I. Mirbel. Semantic integration of conceptual schemas. *Data & Knowledge Engineering*, 21:183–195, 1997.

[MM98] A. MacLean and P. Marqvardsen. Crossing the Border: Document Coordination and the Integration of Processes in a Distributed Organisation. In T. Wakayama, S. Kannapan, C. M. Khoong, S. Navathe, and J. Yates, editors, *Information and Process Integration in Enterprises: Rethinking Documents*, chapter 7, pages 109–124. Kluwer Academic Publishers, 1998.

[MP96] R. Motschnig-Pitrik. Requirements and Comparison of View Mechanisms for Object-Oriented Databases. *Information Systems*, 21(3):229–252, 1996.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, pages 541–579, April 1989.

[NEL86] S. Navathe, R. Elmasri, and J. Larson. Integrating User Views in Database Design. *IEEE Computer*, pages 50–62, January 1986.

[NS73] I. Nassi and B. Shneiderman. Flowchart Techniques for Structured Programming. *SIGPLAN Notices*, 8(8), 1973.

[Obe98] W. Obermair. *Active Object-Oriented Databases—From Conceptual Design to Logical Design.* Dissertation, Universität Linz, February 1998.

[OHM+88] T. W. Olle, J. Hagelstein, I. G. Macdonald, C. Rolland, H. G. Sol, F. J. M. Van Assche, and A. A. Verrijn-Stuart, editors. *Information Systems Methodologies: A Framework for Understanding.* Addison-Wesley, 2nd edition, 1988.

[OS96] A. Oberweis and P. Sander. Information System Behavior Specification by High-Level Petri Nets. *ACM Transactions on Information Systems*, 14(4):380–420, October 1996.

[Pet77] J. L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[Pit97] E. Pitoura. Providing Database Interoperability through Object-Oriented Language Constructs. *Journal of Systems Integration*, 7(2):99–126, August 1997.

[PRS92] L. Pomello, G. Rozenberg, and C. Simone. A Survey of Equivalence Notions for Net Based Systems. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 609 of *Lecture Notes in Computer Science*, pages 410–472. Springer, 1992.

[PS98a] G. Preuner and M. Schrefl. Observation Consistent Integration of Business Processes. In C. McDonald, editor, *Database Systems: Proceedings of the 9th Australasian Database Conference (ADC), Perth, 2–3 Feb. 1998*, volume 20 of *Springer Computer Science Communications*, pages 201–212. Springer, 1998.

[PS98b] G. Preuner and M. Schrefl. Observation Consistent Integration of Views of Object Life-Cycles. In S. M. Embury, N. J. Fiddian, W. A. Gray, and A. C. Jones, editors, *Proceedings of the 16th British National Conference on Databases, Cardiff, United Kingdom (BNCOD)*, volume 1405 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 1998.

[PSU98] L. Palopoli, D. Saccà, and D. Ursino. Semi-automatic, semantic discovery of properties from database schemes. In B. Eaglestone, B. C. Desai, and J. Shao, editors, *Proceedings of the 1998*

International Database Engineering & Applications Symposium (IDEAS), Cardiff, Wales, UK, pages 244–253. IEEE Computer Society, 1998.

[QFG97] M. A. Qutaishat, N. J. Fiddian, and W. A. Gray. Extending OMT to support bottom-up design modelling in a heterogeneous distributed database environment. *Data & Knowledge Engineering*, 22(2):191–205, April 1997.

[QW97] D. Quass and J. Widom. On-Line Warehouse View Maintenance. In J. M. Peckman, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, Arizona*, volume 26 of *SIGMOD Record*, pages 393–404. ACM, June 1997.

[RBPE91] J. Rumbaugh, M. Blaha, W. Premerlani, and F. Eddy. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[SA86] R. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–42, September 1986.

[SAD94] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In M. Jarke, J. A. Bubenko, and K. G. Jeffery, editors, *Proceedings of the 4th International Conference on Extending Database Technology (EDBT), Cambridge, UK*, volume 779 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 1994.

[San98] G. Santucci. Semantic schema refinements for multilevel schema integration. *Data & Knowledge Engineering*, 25:301–326, 1998.

[Sch91] M. Schrefl. Behavior Modeling by Stepwise Refining Behavior Diagrams. In H. Kangassalo, editor, *Proceedings of the 9th International Conference on the Entity-Relationship Approach (ER '90), Lausanne, Switzerland*, pages 119–134. North-Holland, 1991.

[Sch97] R. Scheckenbach. *Semantische Geschäftsprozeßintegration.* Gabler, Deutscher Universitätsverlag, 1997.

[Sch98] I. Schmitt. *Schemaintegration für den Entwurf Föderierter Datenbanken.* PhD thesis, Universität Magdeburg, 1998.

[SHJ+94] G. Saake, P. Hartel, R. Jungclaus, R. Wieringa, and R. Feenstra. Inheritance Conditions for Object Life Cycle Diagrams. In U. Lipeck and G. Vossen, editors, *Workshop Formale Grundlagen für den Entwurf von Informationssystemen*, pages 79–89. Technical Report University of Hannover, No. 03/94, 1994.

[SJB96] W. W. Song, P. Johannesson, and J. A. Bubenko jun. Semantic similarity relations and computation in schema integration. *Data & Knowledge Engineering*, 19(1):65–97, May 1996.

[SJHB96] H. Schuster, S. Jablonski, P. Heinl, and C. Bußler. A General Framework for the Execution of Heterogeneous Programs in Workflow Management Systems. In *Proceedings of the 1st IFCIS International Conference on Cooperative Information Systems (CoopIS), Brussels, Belgium*, pages 104–113. IEEE Computer Society, 1996.

[SK93] A. Sølvberg and D. C. Kung. *Information Systems Engineering — An Introduction*. Springer, 1993.

[SL90] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SLT91] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases (DOOD), Munich, Germany*, volume 566 of *Lecture Notes in Computer Science*, pages 189–207. Springer, 1991.

[SP94] S. Spaccapietra and C. Parent. View Integration: A Step Forward in Solving Structural Conflicts. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):258–274, April 1994.

[SS95] M. Schrefl and M. Stumptner. Behavior Consistent Extension of Object Life Cycles. In M. P. Papazoglou, editor, *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modelling (OO-ER), Gold Coast, Australia*, volume 1021 of *Lecture Notes in Computer Science*, pages 133–145. Springer, 1995.

[SS97] M. Schrefl and M. Stumptner. Behavior Consistent Refinement of Object Life Cycles. In D. W. Embley and R. C. Goldstein, editors, *Proceedings of the 16th International Conference on Conceptual Modeling (ER), Los Angeles, USA*, volume 1331 of *Lecture Notes in Computer Science*, pages 155–168. Springer, 1997.

[SSS98] J. Samos, F. Saltor, and J. Sistac. Definition of Derived Classes in OODBs. In B. Eaglestone, B. C. Desai, and J. Shao, editors, *Proceedings of the 1998 International Database Engineering & Applications Symposium (IDEAS), Cardiff, UK*, pages 150–158. IEEE Computer Society, 1998.

[SZ98] H.-P. Steiert and J. Zimmermann. JPMQ — An Advanced Persistent Message Queuing Service. In S. M. Embury, N. J. Fiddian, W. A. Gray, and A. C. Jones, editors, *Proceedings of the 16th British National Conference on Databases (BNCOD), Cardiff, Wales, UK*, volume 1405 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1998.

[TCG+93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[TK78] D. Tsichritzis and A. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems. *Information Systems*, 3:173–191, 1978.

[TS95] C. Thieme and A. Siebes. Guiding Schema Integration by Behavioural Information. *Information Systems*, 20(4):305–316, 1995.

[VA96] M. Vermeer and P. Apers. The Role of Integrity Constraints in Database Interoperation. In T. M. Vijayaraman, A. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB), Mumbai, India*, pages 425–435. Morgan Kaufmann Publishers, 1996.

[VA97] M. Vermeer and P. Apers. Behaviour specification in database interoperation. In A. Olivé and J. A. Pastor, editors, *Proceedings of the 9th International Conference on Advanced Information Systems Engineering (CAiSE), Barcelona, Spain*, volume 1250 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 1997.

[vdA96] W. M. P. van der Aalst. Three Good Reasons for Using a Petrinet-based Workflow Management System. In S. Navathe and

T. Wakayama, editors, *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC), Cambridge, Massachusetts, USA*, pages 179–201, 1996.

[vdA97] W. M. P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Proceedings of the18th International Conference on Application and Theory of Petri Nets (ICATPN), Toulouse, France*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 1997.

[vdAB97] W. M. P. van der Aalst and T. Basten. Life-Cycle Inheritance — A Petri-Net-Based Approach. In P. Azéma and G. Balbo, editors, *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN), Toulouse, France*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81. Springer, 1997.

[Ver96] F. B. Vernadat. *Enterprise Modeling and Integration: Principles and Applications.* Chapman & Hall, 1996.

[vGG90] R. J. van Glabbeek and U. Goltz. Refinement of Actions in Causality Based Models. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the 1989 REX Workshop on Stepwise Refinement of Distributes Systems: Models, Formalism, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 267–300. Springer, 1990.

[YOL97] L. L. Yan, M. T. Özsu, and L. Liu. Accessing Heterogeneous Data Through Homogenization and Integration Mediators. In *Proceedings of the 2nd IFCIS International Conference on Cooperative Information Systems, Kiawah Island, South Carolina*, 1997.

[YPS95] X. Ye, C. Parent, and S. Spaccapietra. On the Specification of Views in DOOD Systems. In T. Wang Ling, A. O. Mendelzon, and L. Vieille, editors, *Proceedings of International Conference on Deductive and Object-Oriented Databases (DOOD), Singapore*, volume 1013 of *Lecture Notes in Computer Science*, pages 539–556. Springer, 1995.

Curriculum Vitae

Personal Record

Name:	Dipl.-Ing. Günter Preuner
Date of Birth:	April 30, 1970
Place of Birth:	Bad Ischl
Citizenship:	Austria
Home Address:	A-4040 Linz, Altenberger Straße 5/33

University Education

1990–94: Study of computer science ("Informatik") at the Johannes Kepler University of Linz, Austria
graduation to "Dipl.-Ing." in 1994

1995–98: Doctoral study ("Doktoratsstudium der technischen Wissenschaften") at the Johannes Kepler University of Linz, Austria

Professional Experience

1991–94: Tutor for courses in programming languages at the Johannes Kepler University of Linz

1992–94: Instructor for courses in programming languages, algorithms, and systems engineering at the Johannes Kepler University of Linz

1994–95: World wide web administrator at the University of Linz, Department of Systems Engineering and Automation

1994–95: Participation in a project to survey the use of workflow systems in Austria, Germany, and Switzerland, carried out at the University of Linz, Department of Systems Engineering and Automation

since 1995: University assistant at the University of Linz, Department of Business Information Systems