View Integration of Object Life-cycles in Object-oriented Design

Günter Preuner
Institut für Wirtschaftsinformatik
Universität Linz, A-4040 Linz
E-Mail: preuner@dke.uni-linz.ac.at

Stefan Conrad
Institut für Informatik
Universität München, D-80538 München

Abstract

Object-oriented database schemas are often defined by several future users of the planned database, where each user defines a schema representing his/her view on the database; the complete database schema is defined by integrating these views in a subsequent step.

Behavior of objects is often defined at two levels of detail: at the level of methods and at the level of object life-cycles that represent the overall behavior of objects during their life time.

This paper presents an approach to integrate views of object life-cycles to a complete object life-cycle of an object type. We will particularly consider the problem that different users know about parts of the object life-cycle at different levels of detail and often do not consider exactly the same set of entities.

1 Introduction

Object-oriented database schemas are usually defined in cooperation with potential future users of a database. These users know the entities that should be represented by objects in the database from their everyday work and thus are able to define the structure and behavior of object types. Object-oriented schemas comprise the definition of structure and behavior, where behavior comprises the definition of methods and of object life-cycles. The former describe single activities performed on an object whereas the latter represent the overall behavior of objects during their life time from creation until archivation or the time when they are deleted. In this paper we will concentrate on the definition and integration of object life-cycles.

Users are usually able to define in detail those parts of behavior that they know from they everyday work whereas they know about activities that are performed by other users only roughly or even not at all. Correspondingly each view schema defines some information in detail, includes some behavior only at an abstract level, and misses some parts of behavior. The process of view integration comes up with incomplete object life-cycles in that it collects all user views and integrates them by including the most specific information from each view.

Consider, e.g., two views of an object type for room reservations in a hotel. One view is defined by the reservation department that handles requests for rooms, whereas the other view is defined by the reception. The first view contains information about the activities performed for reserving a room in detail, but knows about the use of the room only abstractly. The second view considers using of a room with check-in, check-out, and payment in detail, but does not know details about the preceding step of booking. Further, the reservation department has to consider requests that must be declined because the hotel is booked out for the time in question. Those requests are not visible to the reception. The reception, however, treats requests of guests that come to the reception and ask for a room without having reserved a room before.

In this paper we will identify two main differences between views of the object life-cycle of an object type: (1) heterogeneities that arise between two or more views because different users perceive activities at different levels of detail; (2) heterogeneities that arise because some users deal with entities that are not visible to some other users. Based on these heterogeneities we will introduce an approach for view integration.

We will use *Object/Behavior diagrams (OBD)* for defining object life-cycles. OBD was originally introduced in [14]; the behavior diagrams that are used to define object life-cycles are based on Petri nets. Specialization of object life-cycles from one object type to a subtype was treated in detail in [20, 21]. We will give a brief overview of specialization as far as we will use it for view integration. Although we present our approach using behavior diagrams, the main results can also be applied to models based on statecharts [13] (e.g., UML [3]), since specialization has been considered there, too (e.g., in [6, 12, 22]).

The problem of schema integration is well known from design methods that use view integration and from federated database systems (cf. e.g., [1, 23, 4]). Most approaches that have been published so far treat integration of *structure* of object types. Intensional

and extensional aspects are distinguished in [19]. Recently, *description logics* have been used for representing object-oriented schemas and for their integration (cf., e.g., [15] for a discussion). Some approaches consider methods (e.g., [24, 25]) and integrity constraints (e.g., [5]), but little work has been done so far on integration of object life-cycles. In [7], the authors introduced *conflictfreeness* to determine whether dynamic schemas can be integrated. Engels et al. [9] define integration of views based on graph transformations. The authors of [10, 11] use statecharts for view integration. They introduced five different ways to connect views of object life-cycles depending on how objects have to pass the object life-cycles of the views. View integration of behavior is briefly discussed for *state nets* in [8].

In earlier works [18, 16], we discussed integration of views of object life-cycles if the views are defined based on the same extension, which means that all users considered the same set of entities when defining their view schemas. In this work, we will extend this first approach to deal with the situation that different users define a view of an object life-cycle of the same object type but consider different, but possibly overlapping sets of objects. In the example given before, the views define behavior of overlapping sets of reservations as both views treat reservations, that are booked first and used later; yet, each view deals with objects that are not visible to the other view, e.g., requests that are refused by the reservation department and rooms that are used without having reserved them before. This extended approach may be better applicable in practice; yet, we will find out that the extended approach uses the basic concepts of the approach introduced in [18].

A different problem was handled in [17], where business processes of different enterprises were integrated to define a common process which contains all common parts of the original business processes. This problem deviates from view integration as *existing* processes have to be observed at a common reduced level of detail.

The remainder of this paper is structured as follows: In Section 2, we will introduce *Behavior diagrams* as far as necessary for this paper; Section 3 motivates the problem of integration of behavior diagrams and presents an overview of the integration process. The main topic treated in this paper is integration of views of object life-cycles with disjoint extensions; the correctness criteria and the integration steps are presented in Section 4. Finally, Section 5 concludes this work.

2 Object/Behavior Diagrams

In this section, we briefly introduce *Object/Behavior Diagrams (OBD)*, which have been originally presented as a graphical notation for the object-oriented design of databases [14] and have been later extended for the modeling of business processes [2]. We omit the description of object diagrams and concentrate on behavior diagrams with arc labels and their specialization. For more details, the reader is referred to [20, 21].

2.1 Labeled behavior diagrams

Behavior diagrams are based on Petri nets and consist of activities, states, and arcs. Activities correspond to transitions in Petri nets and represent work performed with business objects, states correspond to places in Petri nets and show where an object currently 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 are marked by this object. When the execution is completed the object is inserted into all poststates of the activity. In difference to Petri nets, we assume that activities may take some time. During the execution of an activity on some object, the object resides in an implicit activity state named after the activity.

Example: Fig. 1 shows a behavior diagram of object type RESVT. Activities are depicted by rectangles, and states are depicted by rectangles with a trapezium at the top and the bottom. Activity issue creates a new reservation object. After completion of this activity, the object resides in states toCheckIn and toPay, where activities use and payByCheque or payCash may be started. The virtual state α will be explained later in this section.

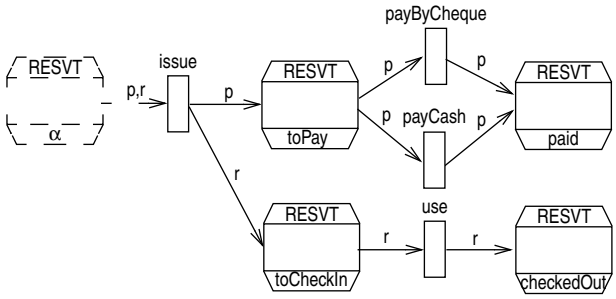

Figure 1: Behavior diagram of RESVT

Behavior diagrams may be *labeled*. The idea of labeling is taken from the processing of business objects 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. The labels of an arc (state, or activity) indicate which copies of a form flow along an arc, (reside in some state, or are processed by an activity, resp.). Notice, however, that *there exist no actual copies of a business object*, but several activities and states may refer at the same time to the same business object by its object identifier.

Labels have been introduced in [21] to facilitate the check whether a behavior diagram constitutes a consistent refinement of another behavior diagram. The labels are used to ensure that if an object leaves one state of a subdiagram (that constitutes the refinement of an abstract state), it leaves the subdiagram entirely.

Example: The business process of reservations shown in Fig. 1 uses two labels p and r corresponding to the payment and the registration copy of the reservation form.

Formally, *labeled behavior diagrams* are defined in Definition 1.

Definition 1 *A* labeled behavior diagram *(LBD)* $B = (S, T, F, L, l)$ *of a business-object type 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* $\forall s \in S : (\exists t \in T : (s,t) \in F) \vee (\exists t \in T : (t,s) \in F)$*.* $L \neq \emptyset$ *is a set of labels. The labeling function* $l : F \to 2^L \setminus \{\emptyset\}$ *(*2^X *denotes the power set of* X*) assigns a non-empty set of labels to each arc in* F*. States, activities, and labels are referred to as* elements*. There is exactly one* initial state $\alpha \in S$ *such that* $\nexists t \in T : (t, \alpha) \in F$ *and there exists a set of* final states $\Omega \subset S$ *such that* $\nexists (s,t) \in F : s \in \Omega$*.*

Labeled behavior diagrams must fulfill the following labeling properties: (1) *label preservation* (for each activity, the union of the labels of its incoming arcs is equal to the union of the labels of its outgoing arcs), (2) *unique label distribution* (all incoming arcs as well as all outgoing arcs of an activity have disjoint sets of labels), and (3) *common label distribution* (for each state, all its incident arcs carry the same labels).

States and activities are labeled, too, where the set of labels of a state or activity, denoted as $\lambda : S \cup T \to 2^L \setminus \{\emptyset\}$, is the union of the sets of labels of its incident arcs.

The initial state α is labeled with all labels, i.e., $\lambda(\alpha) = L$. The initial state represents the period of time when a business object has not yet been created. In the following, we will omit state α in the graphical representation. This means that an activity for which no prestates are shown consumes from state α.

At any time, an object resides in a non-empty set of states, its *life-cycle state.*

Definition 2 *A* life-cycle state *(LCS)* σ *of an object is a subset of* $(S \cup T) \times L$*. The initial LCS is* $\sigma_A = \{(\alpha, x) | x \in L\}$*. An LCS* σ *is* final *if* $\forall (e,x) \in \sigma : e \in \Omega$*.*

As the execution of activities may take some time and an object resides in an activity state during execution, we distinguish *start* and *completion* of an activity. An activity t can be *started* on an object if all of the activity's prestates are marked with the object; starting t yields a new life-cycle state which contains activity state t but no prestates of t. Activity t can be *completed* on an object if the object resides in activity state t; completion yields a new life-cycle state that contains all poststates of t, but does not include t any more.

A *life-cycle occurrence* is defined as the sequence of life-cycle states of a certain object:

Definition 3 *A* life-cycle occurrence (LCO) γ *of an object is a sequence of LCSs* $\gamma = [\sigma_1, \ldots, \sigma_n]$*, 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 will usually consist of a sequence of life-cycle states, where $\sigma_i \neq \sigma_{i+1}$. Yet, the condition that two subsequent LCSs may be equal will be needed for the check whether a behavior diagram is a correct specialization of another behavior diagram (cf. Section 2.2).

Example: Consider the behavior diagram shown in Fig. 1. A possible life-cycle occurrence of a reservation object is [{(α, p), (α, r)}, {(issue, p), (issue, r)}, {(toPay, p), (toCheckIn, r)}, {(toPay, p), (use, r)}, {(toPay, p), (checkedOut, r)}, {(payCash, p), (checkedOut, r)}, {(paid, p), (checkedOut, r)}].

2.2 Specialization of behavior diagrams

The behavior diagram of an object type may be specialized in a subtype in two ways: by *refinement*, i.e., by decomposing states and activities into subdiagrams and labels into sublabels, or by *extension*, i.e., by adding states, activities, and labels. *Observation consistency* as a correctness criterion for specialization guarantees that any life-cycle occurrence of a subtype is observable as a life-cycle occurrence of the supertype if extended elements are ignored and refined elements are considered unrefined. Observation consistency allows *"parallel extension"* but not *"alternative extension"*, i.e., an activity that is added in the behavior diagram of a subtype by extension may not consume from or produce into a state that the subtype inherits from the behavior diagram of the supertype. Observation consistency requires only *partial inheritance* of activities and states: "alternatives" modeled in the behavior diagram of a supertype may be omitted in the behavior diagram of a subtype (cf. [20][1]).

We use a total specialization function $h : S' \cup T' \cup L' \rightarrow S \cup T \cup L \cup \{\varepsilon\}$ to represent the correspondences between a more special behavior diagram $B' = (S', T', F', L', l')$ and a behavior diagram $B = (S, T, F, L, l)$. The inheritance function h can express four cases, (1) inheritance without change, (2) refinement, (3) extension, and (4) elimination, as follows: (1) If an element e is not changed, then $\exists e' \in S' \cup T' \cup L' : h(e') = e$ and $\forall e'' \in S' \cup T' \cup L', e'' \neq e' : h(e'') \neq h(e')$. (2) If an element e in B is refined to a set of elements E ($|E| > 1$), then $\forall e' \in E : h(e') = e$. (3) If a set of elements E is added in B', then $\forall e \in E : h(e) = \varepsilon$. (4) If a set of states and activities $E \subseteq S \cup T$ is removed in B', then $\forall e \in E \nexists e' \in S' \cup T' \cup L' : h(e') = e$.

For the definition of *observation consistent specialization*, we need to define the *generalization of a life-cycle state* and the *generalization of a life-cycle occurrence.*

Definition 4 *A* generalization of a life-cycle state σ' *of an LBD* B' *of object type* O' *to object type* O *with LBD* B*, denoted as* $\sigma'/_O$*, is defined as* $\sigma'/_O \subseteq (S \cup T) \times L$*, where* $\forall e \in S \cup T, x \in L : ((e, x) \in \sigma'/_O \Leftrightarrow \exists e' \in S' \cup T', x' \in L' : h(e') = e \wedge h(x') = x \wedge (e', x') \in \sigma')$.

Definition 5 *A* generalization of a life-cycle occurrence $\gamma' = [\sigma'_1, \ldots, \sigma'_n]$ *of an LBD* B' *of object type* O' *to object type* O *is defined as* $\gamma'/_O = [\sigma'_1/_O, \ldots, \sigma'_n/_O]$.

Definition 6 *An LBD* B' *of object type* O' *is an* observation consistent specialization *of an LBD* B *of object type* O *if and only if for any possible LCO* γ' *of* B' *holds:* $\gamma'/_O$ *is an LCO of* B.

[1] Omitting states and activities of a supertype in a subtype seems to be counter-intuitive at first. Yet, a more in depth analysis reveals that partial inheritance is coherent with a co-variant specialization of method preconditions followed in conceptual specification languages.

Example: The behavior diagram of object type RESVT' shown in Fig. 2 is an observation consistent specialization of the behavior diagram of object type RESVT in Fig. 1. Activity use has been refined to a subnet consisting of activities checkIn, useRoom, useGarage, and checkOut and several states between these activities. Label r has been decomposed into sublabels rr (registration for a room) and rg (registration for a parking lot in the garage). Activity accountVoucher, states vchrTAcc and vchrAcctd, and label v have been added by extension. Activity payCash has been omitted.

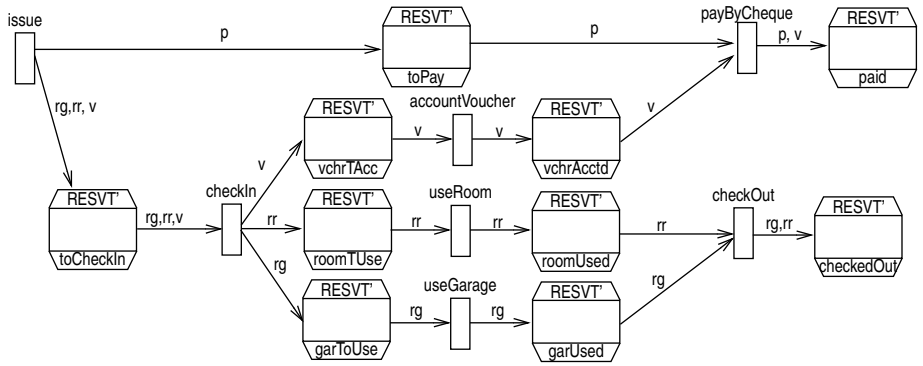

Figure 2: Business process of RESVT'

A set of sufficient and necessary rules has been introduced in [21] to check for *observation consistent extension* and for *observation consistent refinement* of LBDs. These rules can be applied to check for *observation consistent specialization* of LBDs, too, if specialization is considered as a concatenation of refinement and extension.

Definition 7 *An LBD B is an* observation consistent generalization (restriction, abstraction, respectively) *of B' if and only if B' is an observation consistent specialization (extension, refinement, respectively) of B.*

3 Overview of view integration

In this section, we will give an overview of the problems arising during view integration. To motivate the problem of integration and the heterogeneities that may arise, we discuss a small example in Section 3.1; based on this example, we will identify correspondences and heterogeneities between the object life-cycles to integrate in Section 3.2. In Section 3.3, we will present an approach on how to integrate object life-cycles such that the heterogeneities are resolved. The integration process is treated in more detail in Section 4.

3.1 Motivating example

We will motivate the problem of integrating object life-cycles using a small example. There are two views of an object life-cycle for an object type RES whose instances are room

reservations. The first view (indicated as V_1) is defined by the reservation group that treats requests for room reservations, whereas the second view (V_2) is defined by the reception, which has to deal with the use of a room (that may have been reserved before) as well as with requests of guests that ask for a free room although they have not reserved one before. The views are shown in Figures 3 and 4, respectively. Please ignore the symbols \ and ∩, which are depicted within the activity and state symbols for the moment.

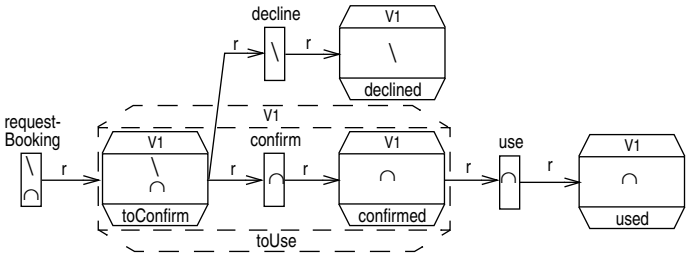

Figure 3: View V1 defined by the reservation department

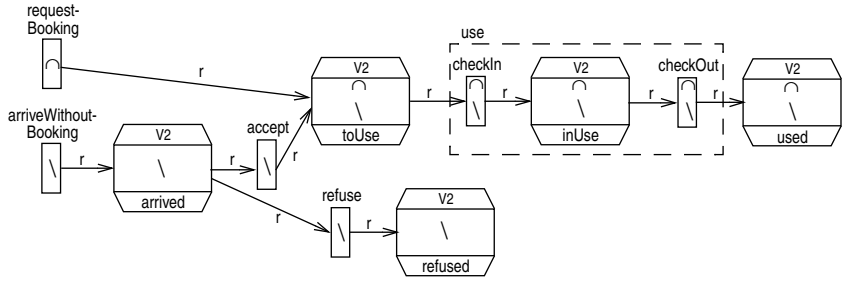

Figure 4: View V2 defined by the reception

There are two main kinds of differences between the views:

1. Some activities and states may be defined in detail in one view but may be defined only as abstract elements in the other view. The reason may be that every view schema is more detailed for those activities and states that represent the everyday work of the modeler of the respective view. In the example, confirmation of a request is defined in detail in V_1, whereas it is referred to only as an abstract state in V_2. Similarly, activity use is abstract in V_1 but defined in detail in V_2.

2. Some activities and states are defined in one view but not in the other one, neither at the same level of detail nor abstracted to an abstract state or activitiy. View V_1 comprises activity decline and state declined, which are not visible in V_2. Further, activities arriveWithoutBooking, accept, and refuse as well as states arrived and refused, which are defined in V_2 are not considered in V_1. The reason for missing activities and states is that objects for which these activities are invoked and that reside in those

states are not visible for the designer of the view since each view designer describes behavior of those entities that he/she is aware of. The reception does not consider requests that are declined by the reservation department; the reservation department cannot perceive reservations that do not pass this department.

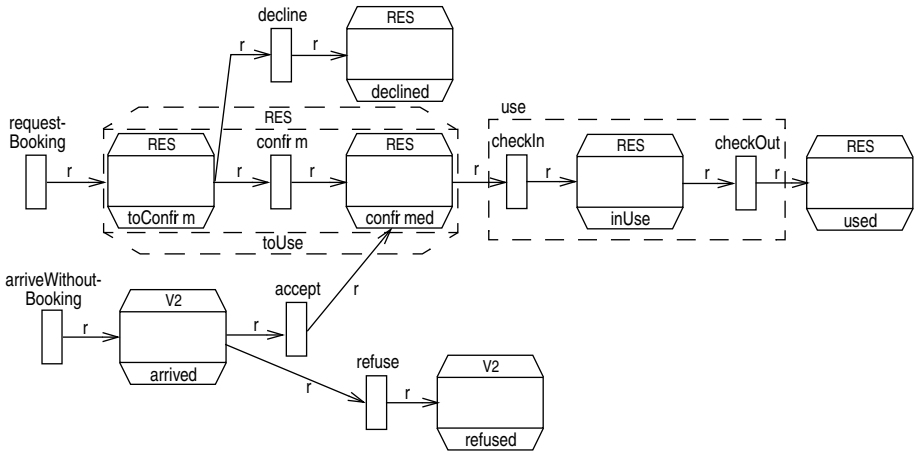

Figure 5: Integrated object life-cycle for RES

The integrated object life-cycle should define the behavior of all objects that have been considered in at least one of the views. Further, it has to contain all activities and states in full detail. In our example, the result of integration is shown in Figure 5.

3.2 Correspondences and heterogeneities

Integration of views is based on corresponding elements, i.e., activities, states, and labels that exist in both views. But as the views are defined by different users there are several heterogeneities to resolve. In this section, we will give an overview of the kinds of correspondences and heterogeneities that may arise between views. Some of them have already been motivated in the example before.

3.2.1 Correspondences

Elements in one view may correspond to elements in the other view in different ways:

1. *Equivalence:* A single activity, state, or label in one view is equivalent to a single activity, state, or label, respectively, in the other view if both elements have the same real-world semantics.

2. *Inclusion:* A single activity or state in one view corresponds to a subnet consisting of several activities, states, and arcs in the other view or one label corresponds to a set of labels.

3. *Subnet correspondence:* Two subnets consisting of activities, states, and arcs correspond to each other or two sets of labels correspond to each other.

4. *No correspondence:* An element in one view does not correspond to any element in the other view.

Example: In our motivating example, some activities, states, and labels are equivalent to each other (e.g., requestBooking, used, and label r). For simplicity, they carry the same name. Activity use in V_1 corresponds to the subnet consisting of checkIn, inUse, and checkOut in V_2 by inclusion. There are no subnet correspondences in this example. Some elements have no correspondence to elements in the other view, e.g., activity decline and state declined in V_1.

We will refer to elements that correspond to elements in the other view (by equivalence, inclusion, or subnet correspondence) as *common elements* and to all other elements as *unique elements.*

3.2.2 Heterogeneities

We distinguish two main kinds of heterogeneities: *(1) extensional heterogeneities* that result from the fact that the designers of two views consider different extensions, i.e., different sets of entities, when defining their view; *(2) intensional heterogeneities* that arise because different activities, states, and labels are defined in the object life-cycles, or elements are defined in different detail.

Extensional heterogeneities. Basically there is one extensional heterogeneity, i.e., *missing entities,* which means that one view considers entities that are not considered in the other view.

Intensional heterogeneities. We distinguish the following intensional heterogeneities:

1. *Naming conflicts:* Two equivalent elements have different names (synonyms) or different elements have the same name (homonyms).

2. *Granularity conflicts:* Corresponding elements are defined in different detail, which results in an inclusion or a subnet correspondence.

3. *View-specific alternatives:* An alternative branch is defined in only one view.

4. *View-specific extensions:* View-specific extensions are unique activities and states that are labeled with a label that exists in only one view. As discussed in Section 2, a label introduces a *parallel* extension, i.e., a branch that is in parallel to the other branches of the object life-cycles, which carry other labels.

Example: For better readability, there are no naming conflicts in our example. Granularity conflicts result from the detected inclusions (e.g., state toUse and activity use and their corresponding refinements). In the example view V_2 (cf. Figure 3), activities arriveWithoutBooking, accept, and refuse as well as states arrived and refused constitute a view-specific alternative.

There is no view-specific extension in our motivating example for simplicity. Yet, suppose that view V_2 contains another label p as well as activity pay with prestate toPay and poststate paid. The reason for this view-specific extension may be that only the reception considers payment because the receptionist must cash the money from the guest.

3.3 Overview of the integration process

The integration process has to treat both kinds of heterogeneities. Due to external heterogeneities, the extensions of the view schemas may be related to each other in one of the following ways. The views will be referred to as V_1 and V_2, the extension considered in the views will be referred to as E_1 and E_2, respectively.

Same extension. If both views consider the same extension, they are integrated by specialization, i.e., the integrated schema is an *observation consistent specialization* of V_1 and V_2. Then, the views can *observe* processing of objects in the integrated schema. There is no extensional heterogeneity to consider; the intensional heterogeneities are treated as follows: Granularity conflicts are resolved by integrating elements at the most detailed level (according to observation consistent refinement), view-specific extensions are integrated (by observation consistent extension), and view-specific alternatives are omitted (by partial inheritance in observation consistent extension). The reason for omitting view-specific alternatives is that an alternative that is defined in only one view can never be used if the other view does not include it and thus cannot observe processing according to this alternative. This case was treated in [18, 16] in detail.

Disjoint extension. The schemas V_1 and V_2 may define views of the same object type but consider disjoint extensions. They will, however, include corresponding elements, e.g., activities that are executed for all objects of E_1 and E_2. We require that the integrated object life-cycle V defines behavior for the extension $E_1 \cup E_2$. The correctness criteria that must hold for the integrated schema and the process how to integrate such views will be discussed in Section 4 in detail. Intuitively, all life-cycle occurrences that are allowed in at least one of the views must be reflected in the integrated object life-cycle. Thus, all different refinements of an abstract element, all view-specific alternatives, and all view-specific extensions must be included. As V defines behavior for $E_1 \cup E_2$, i.e., for a *superclass* of E_1 and of E_2, it must be an *observation consistent generalization* of V_1 and V_2.

Overlapping extension. In the most general case, it holds that E_1 and E_2 overlap. This case can be reduced to the cases of *same* and *disjoint* extensions (cf. Figure 6): During *separation*, the *disjoint sub-view* $V_i^{\setminus}$ and the *common sub-view* $V_i^{\cap}$ are defined from V_i. Sub-views $V_i^{\setminus}$ and $V_i^{\cap}$ define the behavior of those subsets of extension E_i that are only considered in V_i or in both views V_1 and V_2, respectively.

During separation, the system integrator has to determine for each sub-view which alternative branches apply to its extension. Alternative branches in V_i that must not be executed on the members of a sub-view's extension are omitted in this sub-view. Separation cannot be automated, but requires intervention by the system integrator and perhaps by the designer of the view schema.

The sub-views $V_i^{\setminus}$ and $V_i^{\cap}$ omit alternative branches from V_i, but they do not further change behavior. As a result, every sub-view $V_i^{\setminus}$ and $V_i^{\cap}$ is an *observation consistent specialization* of V_i, as alternatives can be omitted according to *partial inheritance* in *observation consistent extension.*

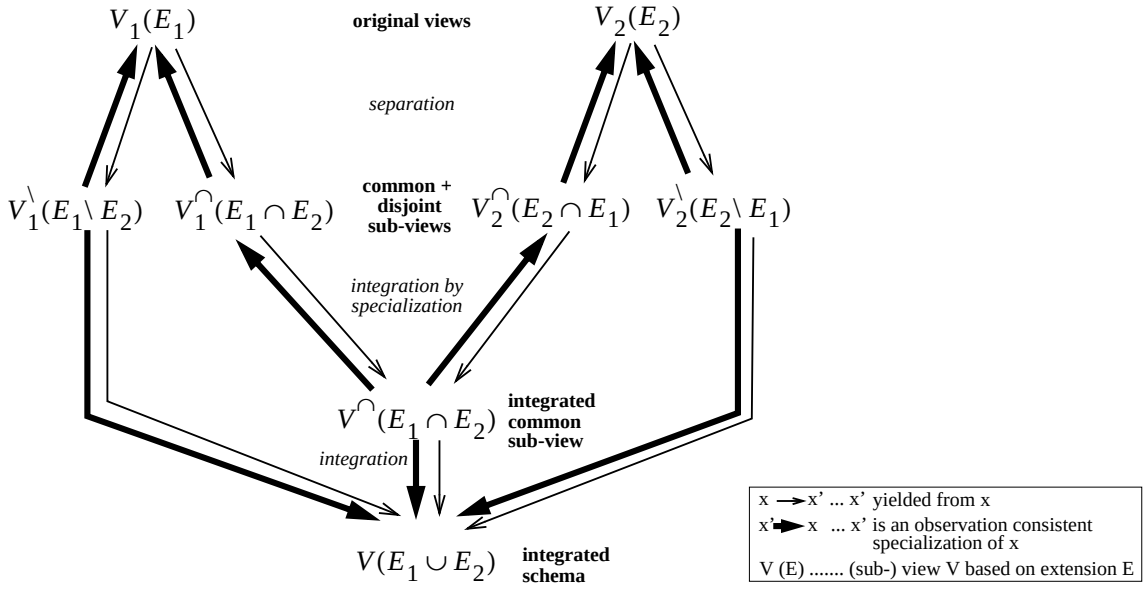

Figure 6: Overview of integration

Since the sub-views $V_1^{\cap}$ and $V_2^{\cap}$ are defined for the same extension, they can be *integrated by specialization* (see above), yielding the *integrated common sub-view* $V^{\cap}$. The sub-views $V^{\cap}, V_1^{\setminus}$, and $V_2^{\setminus}$ consider disjoint extensions such that they can be integrated according to the case *disjoint extension*; the result is the integrated schema V. In general, $2^n - 1$ sub-views must be defined for n given views.

Included extension. This is a special case of overlapping extension, where either $E_1 \setminus E_2$ or $E_2 \setminus E_1$ is the empty set, such that either the behavior diagram of $V_1^{\setminus}$ or $V_2^{\setminus}$ is empty.

Example: The views in the motivating example introduced in Section 3.1 consider overlapping extensions: Reservations that are first reserved and then used are treated in both views, whereas declined reservations are only considered in V_1, and arrivals without booking are only considered in V_2. In Figures 3 and 4, the elements that are defined in $V_i^{\setminus}$ and $V_i^{\cap}$ are indicated by the symbols $\setminus$ and $\cap$, respectively.

The specialization of views with overlapping extensions to sub-views is necessary to determine which alternatives apply to the common and to the disjoint subsets of the views' extensions. This influences the further integration process and the resulting integrated schema.

Example: The abstract activity use in V_1 is included in $V_1^{\cap}$, but not in $V_1^{\setminus}$. During integration by specialization, this abstract activity is refined to a subnet consisting of checkIn, inUse, and checkOut, which is defined in $V_2^{\cap}$. But suppose that activity use is defined in $V_1^{\setminus}$, too, which means that there may be reservations that are used but not treated by the reception (e.g., special treatment for special guests with express check-in and express check-out). Then the abstract activity use cannot be simply refined by the subnet in $V_2^{\cap}$, but there must be a third view V_3 that specifies the refinement of use for special guests. In the integrated schema, both refinements of use must be defined.

4 Integration process

We present the integration process for the integration of (sub-) views with disjoint extensions in more detail in this section. We will first motivate and define the correctness criteria for the integrated schema and then introduce the process of integration.

4.1 Correctness criteria

The integrated object life-cycle B must be defined from the views B_1 and B_2 by adhering to the following rules:

Observation consistent generalization. There must be a specialization function $h_i : S_i \cup T_i \cup L_i \rightarrow S \cup T \cup L \cup \{\varepsilon\}$ (for $i \in \{1, 2\}$) such that object life-cycle $B = (S, T, F, L, l)$ is an observation consistent generalization of $B_1 = (S_1, T_1, F_1, L_1, l_1)$ and $B_2 = (S_2, T_2, F_2, L_2, l_2)$ with specialization function h_1 and h_2, respectively.

No loss of information. Since all objects are treated according to the integrated schema, object life-cycle B must comprise all activities, states, and labels of B_1 and B_2, i.e., there must not be any abstraction or restriction of elements. Thus, for the specialization function h_i it must hold that $\forall e \in S_i \cup T_i \cup L_i : h_i(e) \neq \varepsilon$ (no restriction), $\forall e' \in S_i \cup T_i \cup L_i, e'' \in S_i \cup T_i \cup L_i : h(e') = h(e'') \Rightarrow e' = e''$ (no abstraction). The only possible generalization is adding alternative branches.

Correspondence compliance. We require that corresponding elements e_1 in B_1 and e_2 in B_2 are represented by the same element e in B, i.e., $e = h_1(e_1) = h_2(e_2)$. Since no abstractions are allowed in B, this constraint can only hold if e_1 and e_2 are equivalent (cf. Section 3.2.1). Inclusions or subnet correspondences are in fact different alternative implementations of an abstract activity or state such that they must be integrated as branches that are alternative to each other.

Correspondence of life-cycle occurrences. We require that the integrated object life-cycle B reflects exactly the union of all possible life-cycle occurrences in B_1 and B_2. Thus, we require that for any possible life-cycle occurrence γ in B, there is at least one B_i with a life-cycle occurrence γ', such that the generalization of γ' is equal to γ (where generalization corresponds to a $1:1$ mapping of elements). The inverse condition, i.e., any life-cycle occurrence in B_i must be valid in B, too, has been enforced by the criterion of observation consistent generalization.

4.2 Integration steps

We will introduce three integration steps that are ideally executed sequentially. Yet in practice, some iterations may be necessary until a suitable integration can be found.

The process involves the following steps: *(1) Schema conformation:* The view schemas have to be enriched yielding the *enriched view schemas* B_1 and B_2 which can be integrated in a subsequent step. *(2) Determination of correspondences:* The modeler must determine corresponding elements that will be integrated to the same element. *(3) Integration:* The integrated schema B is defined from the enriched schemas based on the determined correspondences.

Example: The integration process will be illustrated by an example, in which three views on object types RES have to be integrated (cf. Figure 7). Notice that we have extended our previous motivating example here to be able to illustrate some further issues of integration. Ignore label p and state declined in view V_1 for the moment.

4.2.1 Schema conformation

The original view schemas can be prepared for integration during schema conformation. In this step, the view schemas are not changed substantially but only restructured.

We assume that the original view schemas defined by the users are not completely labeled. The reason is that labels and the assignment of labels to arcs may only be determined by comparing the view schemas and thus figuring out the essential aspects considered in the views.

Adding labels is necessary to resolve the heterogeneity *view-specific extensions*, which means that a view schema comprises labels that are not defined in the other view schema. The integrated schema can only be correctly integrated if all views comprise the same set of labels. Otherwise labels that are defined in one view schema but are missing in the

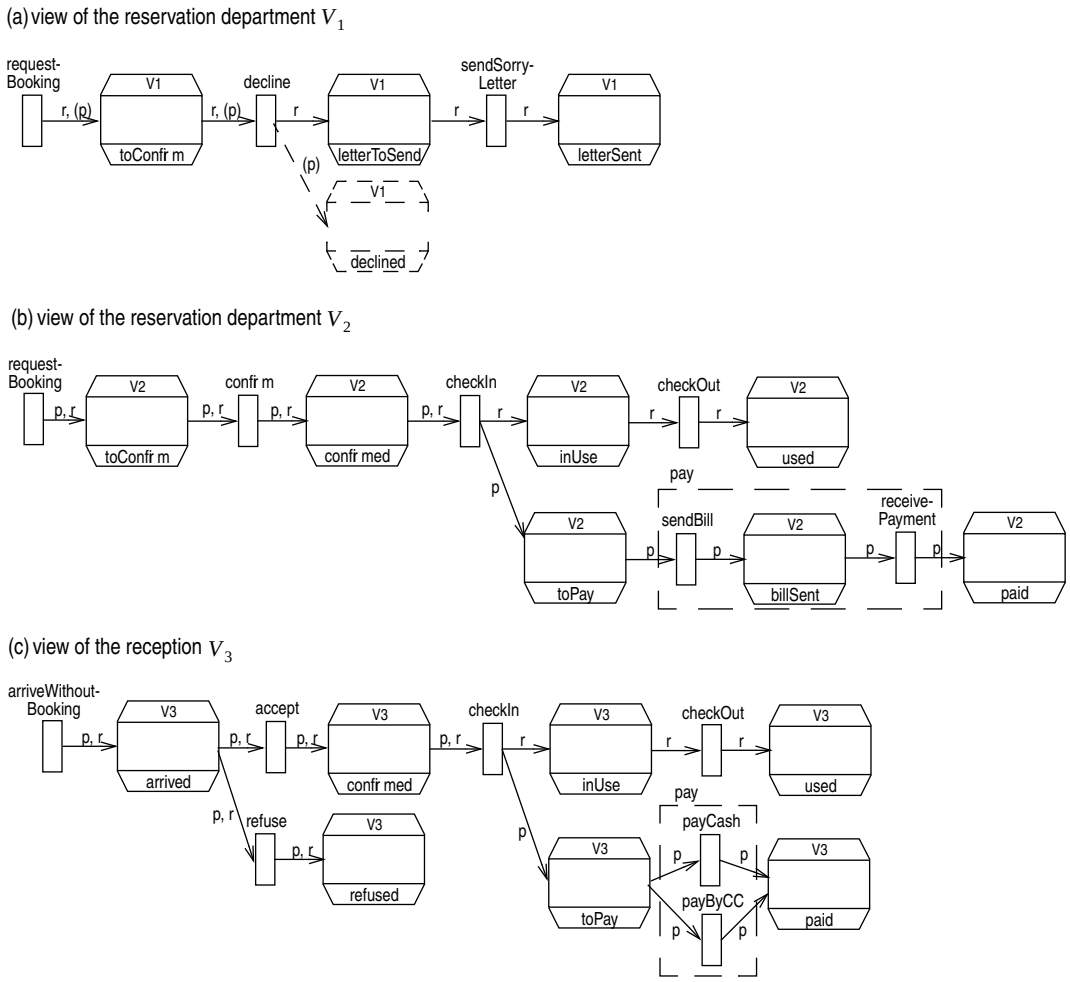

Figure 7: Views to integrate

other one would have to be omitted during integration to fulfill the condition of *observation consistent generalization.* Yet, omitting labels during integration would violate the rule *no loss of information.*

If labels are added to a view schema, it may be necessary to add *non-substantial* states, i.e., states that do not further restrict the possible sequences of activity invocations. Non-substantial states that are labeled with a set of labels X are useful to represent the fact that these labels reside in some "waiting state" during processing.

Example: In our example, label p (representing the aspect of payment) and state declined are introduced during schema conformation.

The possible sequences of activity invocations must be the same in the original (not-enriched) behavior diagram $\breve{B}$ and in the enriched behavior diagram B, such that the following condition must hold: For each life-cycle occurrence $\breve{\gamma} = [\breve{\sigma}_1, \ldots, \breve{\sigma}_n]$ in $\breve{B}$, there is a life-cycle occurrence $\gamma = [\sigma_1, \ldots, \sigma_n]$ in B such that it holds that $\forall 1 \leq i \leq n : \breve{\Sigma}_i \subseteq \Sigma_i \wedge$

$(\Sigma_i \setminus \breve{\Sigma}_i) \cap \breve{S} = \emptyset$, where $\breve{\Sigma}_i := \{s | \exists x \in \breve{L} : (s, x) \in \breve{\sigma}_i\}$ and $\Sigma_i := \{s | \exists x \in L : (s, x) \in \sigma_i\}$.

Intuitively, we may compare additional labels with additional copies of a paper form (cf. the analogy of paper forms introduced in Section 2) in paper work: If a user needs an additional copy of a paper form for some reason, this copy is added to the paper form. Another user will not define this copy unless he/she needs it, too. Yet during integration, that set of paper forms is introduced that fulfills the requirements of all users.

4.2.2 Determination of correspondences

The modeler has to determine correspondences between the view schemas. Thereby, the following restrictions apply:

1. *Correspondence of equivalent elements:* An activity (state, or label) in B_1 can be considered *equivalent* to another activity (state, or label, respectively) in B_2 only if both elements represent the view of the same element in the integrated schema (cf. the correctness criterion of *correspondence compliance*).

2. *Equivalence of labels:* Since no restriction or abstraction is allowed during integration, the set of labels in B_1 and B_2 must be equivalent, i.e., for each label in B_1, there must be an equivalent label in B_2 and vice versa. If this condition is violated, the modeler has to compare the labels in the views and to insert new labels during schema conformation.

3. *Inclusions and subnet correspondences:* Inclusions and subnet correspondences exist if the view schemas provide different refinements of an abstract activity or state.

4. *Disjointness of correspondences:* An element in one view is either unique (i.e., it does not correspond to any element in the other view) or it corresponds to exactly one element or subnet in the other view.

Example: Consider the example shown in Figure 7. For better readability, equivalent elements carry the same name (e.g., activity requestBooking in V_1 and V_2, state inUse in V_2 and V_3, label p in all views). View V_2 and V_3 define different subnets for an abstract activity pay. In this example, we assume that room reservations that have been reserved in the reservation department (cf. V_2) can be paid after a bill has been sent to the customer; reservations that have not been reserved in advance must be paid during the stay in the hotel by cash or credit card. These different kinds of payment correspond to each other by a subnet correspondence and are indicated in the figure by an activity symbol with dashed borders. There are unique elements, too (e.g., activity decline in V_1, state refused in V_3).

4.2.3 Integration

The integrated object life-cycle B is constructed from B_1 and B_2 based on the set of correspondences as follows:

1. *Equivalences:* One element is defined for a pair of equivalent elements in the views. *Naming conflicts* can be resolved by renaming; alternatively, differences in naming can remain as the relationship between an element in B and an element in B_i can be uniquely determined by the specialization function h (see below).

2. *Inclusions and set correspondences:* Elements that belong to inclusions or set correspondences are defined in B as they are integrated as branches that are alternative to each other. Thereby, *granularity conflicts* are resolved.

3. *Unique elements:* Unique elements belong to alternatives that are defined in only one view and thus are defined in the integrated schema.

4. *Arcs:* Each arc that is defined in a view is defined in the integrated schema, too.

5. *Assignment of labels:* A label is assigned to an arc if the corresponding label is assigned to this arc in a view schema.

In the following, we will write *e corresponds to e'* (or $e \simeq e'$), where e is an element in B and e' is an element in a B_i to state the fact that element e is defined in B because of element e' in B_i. The specialization function $h_i : S_i \cup T_i \cup L_i \rightarrow S \cup T \cup L \cup \{\varepsilon\}$ reflects these correspondences: $e = h_i(e') :\Leftrightarrow e \simeq e'$.

Example: The integrated object life-cycle for the view schemas is shown in Figure 8. The equivalent elements of the views are represented by one element in the integrated schema RES (e.g., requestBooking, inUse, p). The different kinds of payment are defined as alternative branches. All unique elements are defined in RES. The states and arcs depicted with dotted lines as well as label s will be explained later.

Each possible life-cycle occurrence in the integrated schema B must be reflected in at least one view. But since B comprises all alternative branches of the views, additional *synchronization states* must be introduced in B to enforce that alternatives can only be executed in the same way as defined in the views.

Example: Consider the example shown in Figures 7 and 8: According to view V_2, activity sendBill can only be executed if a room reservation has been confirmed before, i.e., activity confirm has been executed. Activities payCash and payByCC can only be executed according to V_3 if activity accept has been executed before. Therefore, a new synchronization label s and synchronization states are introduced to enforce the constraints defined in the views.

These additional labels and synchronization states can be defined in the view schemas B_1 or B_2 during *schema conformation* and integrated in B. A more practical approach will be that — as in the example — synchronization states and labels are introduced after integration if life-cycle occurrences are allowed in B that cannot be observed in any view. This step could be automated as follows: For each label x in B and each view schema i, an additional label l_x^i (the synchronization label) is defined in B. Further, one synchronization state s_x^i is defined in B for each label l_x^i. Then, an arc that is labeled with l_x^i is defined from an activity a to state s_x^i if and only if there is an activity a' in view B_i, where $a \simeq a'$

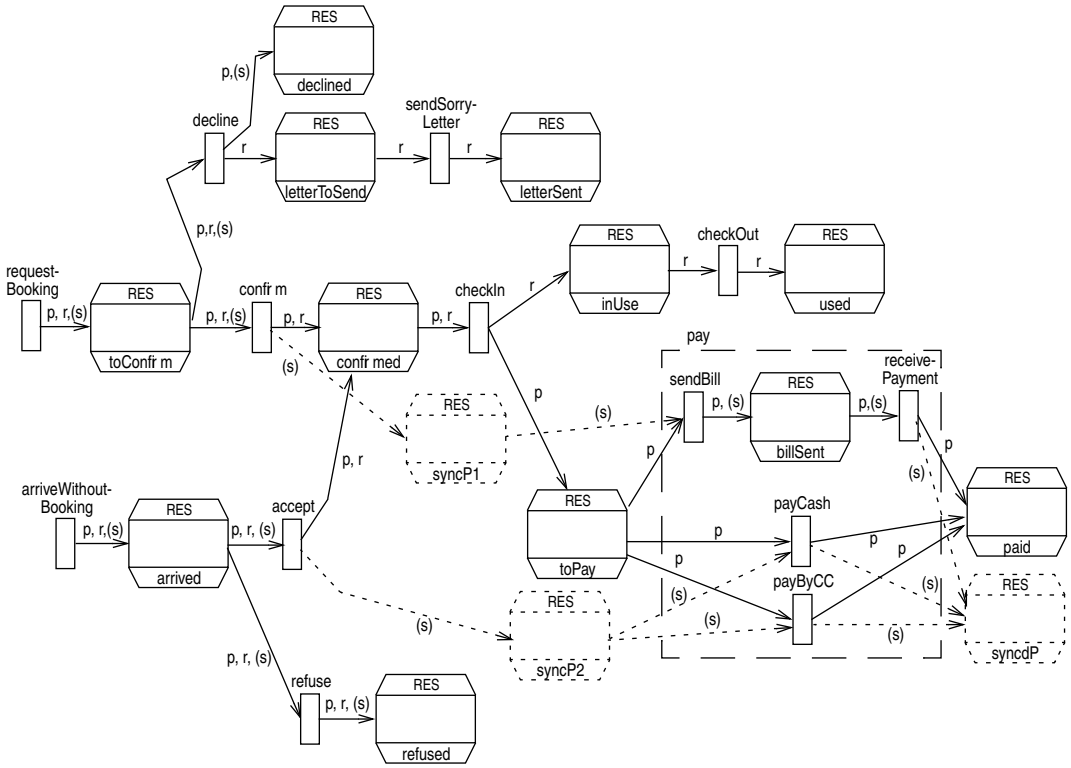

Figure 8: Integrated object life-cycle for object type RES

and a' produces label x', where $x \simeq x'$. An arc with label l_x^i is defined from s_x^i to activity a if and only if activity a' in a view B_i, where $a \simeq a'$, consumes label x', with $x \simeq x'$, and a does not consume from the virtual α state. If an activity consumes from state α, it consumes all labels (including the synchronization labels) from α.

The integrated object life-cycle is a correct integration of the object life-cycles in the views if the modeler has determined correspondences consistently. The check whether B is an observation consistent generalization of B_1 and of B_2 can be automated (cf. [20, 21]). Since all elements of the views are integrated in B, the criterion of *no loss of information* is fulfilled. *Correspondence compliance* is fulfilled as all elements that are equivalent to each other are integrated to the same element and all inclusions and subnet correspondences are defined as alternative branches. Further unique elements are *not* integrated with elements from another view. The *correspondence of life-cycle occurrences* is fulfilled as the set of possible life-cycle occurrences in B is restricted by synchronization states and synchronization labels.

Incorrect object life-cycles may result from contradictory information in the views or inadequate correspondences. Then the modeler must examine once more the set of correspondences, repeat the step of schema conformation, or must even change the view schemas, probably by consulting the modelers of the views.

5 Conclusion

We introduced an integration approach to define the object life-cycle of an object type based on incomplete views of the object life-cycle. View schemas can be incomplete with respect to the considered extension and intension. The integration process comprises two phases: First, the extensions considered in view schemas are separated to common and disjoint subclasses and the sub-views are defined for these subclasses. The behavior of common sub-views is defined using integration by specialization. In the second step, sub-views with disjoint extensions are integrated. The main idea of the second step is that the integrated schema includes all alternatives that are defined in any sub-view.

The decision whether views with disjoint extensions are integrated to one single object type or are defined as separate object types in the integrated schema (with a common supertype) is a design decision of the modeler in many cases; e.g., prereserved reservations and room reservations that are requested at the reception without booking could be represented by two object types BOOKED-RES and NOT-BOOKED-RES. Yet separate object types can only be defined if an object can be assigned to one of them at the time of creation, which is not possible, e.g., for two views handling accepted and declined reservations as the user cannot decide at the time of creation whether the reservation will be accepted or declined later in the process.

In future, tools will be implemented based on the theoretical results achieved so far to support the user in integrating view schemas as well as validating and verifying the resulting integrated schema.

References

[1] 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.

[2] 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.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* The Addison-Wesley Object Technology Series. Addison-Wesley, 1998.

[4] O. Bukhres and A. Elmagarmid. *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications.* Prentice-Hall, 1996.

[5] 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,*

Cardiff, UK (BNCOD 16), volume 1405 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 1998.

[6] J. Ebert and G. Engels. Observable or Invocable Behaviour — You Have to Choose. Technical report, Koblenz University, 1994.

[7] 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, Cottbus, Germany (ER '96)*, volume 1157 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 1996.

[8] D. W. Embley. *Object Database Development: Concepts and Principles.* Addison-Wesley, 1998.

[9] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A View-Oriented Approach to System Modelling Based on Graph Transformation. In M. Jazayeri and H. Schauer, editors, *Proceedings of the 6th European Software Engineering Conference (ESEC), Zurich, Switzerland*, volume 1301 of *Lecture Notes in Computer Science*, pages 327–343. Springer, 1997.

[10] H. Frank and J. Eder. Integration of Behaviour Models. In *Proceedings of the ER '97 Workshop on Behavioral Models and Design Tranformations*, 1997.

[11] H. Frank and J. Eder. Integration of Statecharts. In M. Halper, editor, *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, New York City, USA (CoopIS '98)*, pages 364–372. IEEE Computer Society, 1998.

[12] A. Le Grand. Specialization of Object Lifecycles. In C. Rolland and G. Grosz, editors, *Proceedings of the 1998 International Conference on Object Oriented Information Systems (OOIS), Paris, France*, pages 259–275. Springer, 1998.

[13] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science Of Computer Programming*, 8:231–274, 1987.

[14] G. Kappel and M. Schrefl. Object/Behavior Diagrams. In *Proceedings of the 7th International Conference on Data Engineering, Kobe, Japan (ICDE '91)*, pages 530–539, April 1991.

[15] M. Lenzerini. Description Logics and Their Relationships with Databases. In C. Beeri and P. Buneman, editors, *Proceedings of the 7th International Conference on Database Theory (ICDT '99), Jerusalem, Israel*, volume 1540 of *Lecture Notes in Computer Science*, pages 32–38. Springer, 1999.

[16] G. Preuner. *Definition of Behavior in Object-Oriented Databases by View Integration*, volume 53 of *PhD Theses on Database Systems and Information Systems (DISDBIS).* infix-Verlag, 1999.

[17] G. Preuner and M. Schrefl. Observation Consistent Integration of Business Processes. In C. McDonald, editor, *Proceedings of the 9th Australasian Database Conference (ADC '98), Perth, Australia*, volume 20 of *Australian Computer Science Communications*, pages 201–212. Springer, 1998.

[18] 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 (BNCOD 16), Cardiff, Wales, UK*, volume 1405 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 1998.

[19] I. Schmitt and G. Saake. Schema Integration and View Derivation by Resolving Intensional and Extensional Overlappings. In K. Yetongnon and S. Hariri, editors, *Proceedings of the 9th ICSA International Conference on Parallel and Distributed Computing Systems (PDCS '96), Dijon, France*, pages 751–758, September 1996.

[20] M. Schrefl and M. Stumptner. Behavior Consistent Extension of Object Life Cycles. In P. Papazoglou, editor, *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modeling, Gold Coast, Australia (OO-ER '95)*, volume 1021 of *Lecture Notes in Computer Science*, pages 133–145. Springer, 1995.

[21] 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, Los Angeles, USA (ER '97)*, volume 1331 of *Lecture Notes in Computer Science*, pages 155–168. Springer, 1997.

[22] M. Schrefl and M. Stumptner. On the Design of Behavior Consistent Specializations of Object Life Cycles in OBD and UML. In M. Papazoglou et al., editor, *Object-Oriented Data Modeling*. MIT Press, 1999. (to appear).

[23] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[24] C. Thieme and A. Siebes. Guiding Schema Integration by Behavioural Information. *Information Systems*, 20(4):305–316, 1995.

[25] 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, Barcelona, Spain (CAiSE '97)*, volume 1250 of *Lecture Notes in Computer Science*, pages 425–435. Springer, 1997.