Chapter 4
Mapping Conceptual Models to Database Schemas

David W. Embley and Wai Yin Mok

4.1 Introduction

The mapping of a conceptual-model instance to a database schema is fundamentally the same for all conceptual models. A conceptual-model instance describes the relationships and constraints among the various data items. Given the relationships and constraints, the mappings group data items together into flat relational schemas for relational databases and into nested relational schemas for object-based and XML storage structures.

Although we are particularly interested in the basic principles behind the mappings, we take the approach of first presenting them in Sections 4.2 and 4.3 in terms of mapping an ER model to a relational database. In these sections, for the ER constructs involved, we (1) provide examples of the constructs and say what they mean, (2) give rules for mapping the constructs to a relational schema and illustrate the rules with examples, and (3) state the underlying principles. We then take these principles and show in Section 4.4 how to apply them to UML. This section on UML also serves as a guide to applying the principles to other conceptual models. In Section 4.5 we ask and answer the question about the circumstances under which the mappings yield normalized relational database schemas. In Section 4.6 we extend the mappings beyond flat relations for relational databases and show how to map conceptual-model instances to object-based and XML storage structures. We provide pointers to additional readings in Section 4.7. Throughout, we use an application in which we assume that we are designing a database for a bed and breakfast service. To illustrate all conceptual-modeling features of

David W. Embley
Brigham Young University, Provo, Utah 84602, USA, e-mail: embley@cs.byu.edu

Wai Yin Mok
University of Alabama in Huntsville, Huntsville, Alabama 35816, USA, e-mail: mokw@uah.edu
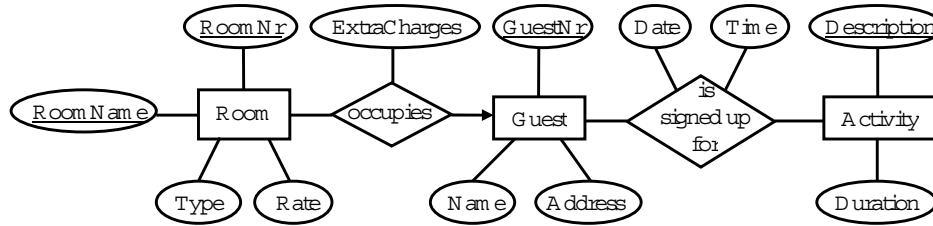
(a) ER Diagram.

Activity(Description, Duration)
Guest(GuestNr, Name, Address)
Room(RoomNr, RoomName, Type, Rate, GuestNr, ExtraCharges)
IsSignedUpFor(GuestNr, Description, Date, Time)

(b) Generated Schemas.

Fig. 4.1 Basic Mappings: ER Diagram and Generated Schemas.

interest, we take the liberty of poetic license in imagining what features might be of interest to the application.

This chapter assumes a solid understanding of several other chapters in this handbook: *The Entity-Relationship Model* (Chapter 3), *The Enhanced Entity-Relationship Model* (Chapter 4), *The Unified Modeling Language* (Chapter 5), and *Functional Dependencies and Normalization* (Chapter 6). We do not extensively discuss any of these topics. We do, however, add enough commentary about these topics to make this chapter reasonably self contained. This chapter also assumes a minimal understanding of Chapter 8 (*Relational Data Model*), Chapter 10 (*SQL*), Chapter 14 (*Object-Oriented Databases*), Chapter 15 (*Object-Relational Databases*), and Chapter 16 (*XML Databases*). We make no explanatory comments about these topics.

4.2 Entity-Relationship Model Mappings

4.2.1 Basic Mappings

We give an ER diagram in Figure 4.1a and the database schema generated from the ER diagram in Figure 4.1b. In our bed and breakfast application, as modeled in Figure 4.1, registered guests occupy rooms and are signed up for activities such as golf, tennis, and horseback riding. Although there may be other occupants of a room in a registered guest's party, in this initial example, we only capture each registered guest (presumably the ones who are paying the bill). Further, in this example we only allow a registered guest's party as a whole to sign up various activities.

Notationally, each box in Figure 4.1a represents an entity set, e.g., *Room*, *Guest*, and *Activity*. The diamonds with lines connected to entity sets represent relationship sets among the connected entity sets, e.g., *occupies* and *is signed up for*. The ovals represent attributes, e.g., *RoomNr*, *Date*, and *Duration*, which may be connected either with entity sets (boxes) or relationship sets (diamonds).

Cardinality constraints for binary relationship sets are one of

1. *many-many*, indicated by the absence of arrowheads on the lines connecting entity sets and relationship sets, e.g., *is signed up for*;

2. *many-one*, indicated by an arrowhead on the *one* side, e.g., *occupies* is *many-one* from *Room* to *Guest*, and thus there can be only one registered guest's party occupying a room, although the registered guest's party may occupy one or more rooms;

3. *one-many*, which is the same as *many-one* only in the opposite direction, e.g., *occupies* is *one-many* from *Guest* to *Room*; and

4. *one-one*, indicated by arrowheads on both sides, e.g., *occupies* would be *one-one* if the bed and breakfast had the policy that a guest's party could occupy at most one room.

Cardinality constraints for attributes are *many-one* from entity set to attribute or from relationship set to attribute. If an entity-set attribute is a key, however, as indicated by underlining the attribute name, then the cardinality is *one-one*. Thus, for example, $Type$ is *many-one* from *Room* to $Type$ so that many rooms can be of the same type (e.g., the bed and breakfast can have several single rooms, several double rooms, and several suites). $RoomNr$, on the other hand, is a key attribute for *Room*, and thus each room has one room number and each room number designates one room. $RoomName$ is also a key for *Room* (each room has a name such as the "Gold Room" or the "Penthouse Suite"). Although rare, relationship sets may also have keys designated by an underline (e.g., a guarantee number for a guest's reservation for a room). Relationship sets, of course, have keys, often a composite of the keys for its related entity sets (e.g., $\{GuestNr,$ $Description\}$, which is a composite key[1] for the relationship set *is signed up for*). The standard ER model, however, provides no way to directly designate composite keys for relationship sets.

ER Mapping Rule #1. An entity set E with n key attributes $A_1, ..., A_n$ and m non-key attributes $B_1, ..., B_m$ maps to the relational schema $E(\underline{A_1},$ $..., \underline{A_n}, B_1, ..., B_m)$. The underlines designate keys for the relational schema. If there is only one key, it is the primary key; if there are several keys, one is designated as the primary key. ER Mapping Rule #1 applies when E is a regular entity set (i.e., not a weak entity set) and has no entity set E'

---

[1] Here and throughout the chapter "composite key" always designates a minimal key, so that if any of the attributes of the key is removed, the remaining attribute(s) no longer provide the unique identification property of a key.

connected to E by a relationship set that is *one-one* or is *many-one* from E to E'.

ER Mapping Rule #1 applies to *Activity* in Figure 4.1a. *Activity* is a regular entity set (as are all entity sets in Figure 4.1a), and its only connected relationship set is *many-many*. When we apply ER Mapping Rule #1 to *Activity*, since *Description* is a key attribute and *Duration* is a non-key attribute, we obtain *Activity*(*Description*, *Duration*), which is the first relational schema in Figure 4.1b. ER Mapping Rule #1 also applies to *Guest*, yielding the second relational schema in Figure 4.1b. ER Mapping Rule #1 does not apply to *Room* because the connected relationship set *occupies* is *many-one* from *Room* to *Guest*.

ER Mapping Rule #2. Let E be a regular entity set with n key attributes A_1, ..., A_n, m non-key attributes B_1, ..., B_m, and p *many-one*-connected entity sets whose primary keys are C_1, ..., C_p and which have q attributes D_1, ..., D_q associated with the p *many-one* relationship sets. Assuming E has no *one-one*-connected entity sets, E maps to $E(\underline{A_1}, ..., \underline{A_n}, B_1, ..., B_m, C_1, ..., C_p, D_1, ..., D_q)$.

ER Mapping Rule #2 applies to *Room* in Figure 4.1a. *Room* has two key attributes (*RoomNr* and *RoomName*), two non-key attributes (*Type* and *Rate*), and one *many-one*-connected entity set with a primary key (*GuestNr*) and with an attribute (*ExtraCharges*) on its connecting relationship set (*occupies*). Thus, applying ER Mapping Rule #2 to *Room*, we obtain, *Room*(*RoomNr*, *RoomName*, *Type*, *Rate*, *GuestNr*, *ExtraCharges*), the third relational schema in Figure 4.1.[1]

ER Mapping Rule #3. Let E and E' be two regular entity sets connected by a single *one-one* relationship set R between them. Let E have n key attributes A_1, ..., A_n, m non-key attributes B_1, ..., B_m, and p *many-one*-connected entity sets whose primary keys are C_1, ..., C_p and which have q attributes D_1, ..., D_q associated with the p *many-one* relationship sets. Let E' have n' key attributes A'_1, ..., $A'_{n'}$, m' non-key attributes B'_1, ..., $B'_{m'}$, and p' *many-one*-connected entity sets whose primary keys are C'_1, ..., $C'_{p'}$ and which have q' attributes D'_1, ..., $D'_{q'}$ associated with the p' *many-one* relationship sets. And let R have r attributes R_1, ..., R_r. Then, E, E', and R together map to the single relational schema $R(\underline{A_1}, ..., \underline{A_n}, \underline{A'_1}, ..., \underline{A'_{n'}}, B_1, ..., B_m, B'_1, ..., B'_{m'}, C_1, ..., C_p, C'_1, ..., C'_{p'}, D_1, ..., D_q, D'_1, ..., D'_{q'}, R_1, ..., R_r)$.

ER Mapping Rule #3 does not apply to the ER model instance in Figure 4.1. It would apply if *occupies* were *one-one*, which would mean that a guest's party would occupy one room and could only occupy one room. If *occupies* were *one-one*, then we would map *Room*, *Guest*, and *occupies* together to *occupies*(*RoomNr*, *RoomName*, *GuestNr*, *Type*, *Rate*, *Name*, *Address*, *ExtraCharges*). Furthermore, there would be no separate schemas

---

[1] Unless otherwise explicitly stated, the first key listed in a relational schema is is the primary key — *RoomNr* in this example.

for *Room* and *Guest* schema since both would be entirely included in this *occupies* schema.

It becomes unwieldy to formally specify further generalizations of ER Mapping Rule #3. Furthermore, these generalizations seldom arise in practice. The generalizations involve adding more and more entity sets in a one-to-one correspondence with the entity sets already in a one-to-one correspondence. In principle, we just combine together into a single relational schema all the attributes of these entity sets, of their connecting one-one relationship sets, and of their connecting many-one relationship sets (but not their connecting one-many and many-many relationship sets), and all the primary-key attributes of their connecting many-one relationship sets. Unfortunately, however, we have to be careful. Basically the connected one-one relationship sets have to all have mandatory participation; and if there are cycles in the set of entity sets in the one-to-one correspondence, the one-one relationship sets have to all be semantically equivalent.[2]

ER Mapping Rule #4. Let R be a many-many binary relationship set with attributes A_1, ..., A_n. Let E and E' be the entity sets connected by R, and let P be the primary key attribute of E and P' be the primary key attribute of E'. Then, R maps to $R(\underline{P,\ P'}, A_1, ..., A_n)$.

ER Mapping Rule #4 applies to *is signed up for*, a many-many relationship set whose attributes are *Date* and *Time*. Its connected entity sets are *Guest* and *Activity*, whose primary keys are respectively *GuestNr* and *Description*. Thus, when we apply ER Mapping Rule #4, we obtain *IsSignedUpFor*(*GuestNr, Description*, *Date*, *Time*), which is the last relational schema in Figure 4.1b.

General Principle #1. In general, mappings of conceptual models to relational schemas are about finding key attributes and composite key attributes and grouping these attributes together into relational schemas along with attributes that directly depend on them. Finding key attributes and composite key attributes is about observing cardinality relationships among attributes (one-one, one-many, many-one, and many-many). Finding directly dependent attributes is about finding attributes that functionally depend on keys, but only on keys within the group of attributes mapped together into a relational schema (i.e., never on some non-key attribute or attribute group, never on a proper subset of a composite key, and never on a combination of a proper subset of a composite key and non-key attributes). Functional dependency arises from cardinality constraints — an attribute B functionally depends on another attribute A if there is a many-one (or one-one) relationship from A to B. More generally, an attribute B functionally depends on a set of attributes $A_1 A_2...A_n$ if there is a many-one (or one-one) relationship from the n-tuples in $A_1 A_2...A_n$ to B.

General Principle #2. Graphical instantiations of conceptual models dictate cardinality relationships among attributes. Sometimes the graphical

---

[2] We refer the interested reader to the additional readings in Section 4.7 for these esoteric mappings.

instantiations of conceptual models are insufficient to express all needed cardinality relationships. In this case, we express the missing cardinality constraints we need using a formal constraint language when one is defined for the conceptual model or notes in the absence of a defined formal constraint language.

General Principle #3. The following algorithm generally applies to all conceptual models.

Step 1   Group keys, which may be single-attribute keys or composite-attribute keys, into sets in which the keys in a set are all in a one-to-one correspondence with each other. (In practice, these key sets will often be singleton sets.)

Step 2   In each key set, designate one of the keys (or the only key) to be the primary key for the key set.

Step 3   To each key set, add all directly dependent non-key attributes, plus, from among other key sets, the attributes of all directly dependent primary keys.

Step 4   For each group of attributes formed in Step 3, select a name and form a relational schema. (Name selection is often obvious. Since keys are for entity sets or relationship sets, we typically use the entity-set name or the relationship-set name.)

If we apply General Principle #3 to the ER diagram in Figure 4.1a, Step 1 yields the set of key sets: $\{\{Description\}, \{GuestNr\}, \{RoomNr, RoomName\}, \{GuestNr\ Description\}\}$.[1] In Step 2 we designate $RoomNr$ as the primary key for the key set $\{RoomNr, RoomName\}$. All other key sets are singleton sets and thus each key in these singleton sets is a primary key. In Step 3 we group attributes, and in Step 4 we select names for these attribute groups and form relational schemas. For the key set $\{Description\}$, the only directly dependent attribute is $Duration$. Hence, we add it, yielding ($Description$, $Duration$). Based on the diagram in Figure 4.1a, the obvious name for this attribute group is $Activity$. Thus, $Activity(\underline{Description}, Duration)$ becomes the relational schema for the key set $\{Description\}$. This is the first relational schema in Figure 4.1b. The key set $\{GuestNr\}$ has two directly dependent attributes: $Name$ and $Address$. Thus, with the addition of the obvious schema name, $Guest(\underline{GuestNr}, Name, Address)$ becomes the relational schema for the key set $\{GuestNr\}$. This is the second relational schema in Figure 4.1b. The key set $\{RoomNr, RoomName\}$ has three directly dependent non-key attributes: $Type$ and $Rate$ from the entity set $Room$ and $ExtraCharges$ since it is an attribute of $occupies$, the many-one relationship set from $Room$ to $Guest$. From among the other key sets, $GuestNr$

---

[1] Here, we make use of the common set notation in the relational database literature that lets a sequence of attribute names designate a set. Thus, $\{GuestNr\ Description\}$ is a key set with a single composite key consisting of two attributes whereas $\{RoomNr, RoomName\}$ is a key set with two keys.

is the only primary key directly dependent on $RoomNr$.[1] Thus, with the addition of the obvious schema name, $Room(\underline{RoomNr},\ \underline{RoomName},\ Type,$ $Rate,\ GuestNr,\ ExtraCharges)$ becomes the relational schema. This is the third relational schema in Figure 4.1b. Finally, for $\{GuestNr\ Description\}$, which is the key set for the relationship set *is signed up for*, the only directly dependent attributes are $Date$ and $Time$.[2] Thus, with the addition of the obvious schema name, $IsSignedUpFor(\underline{GuestNr,\ Description},\ Date,$ $Time)$ becomes the relational schema.

Observe that ER Mapping Rules #1–#4 are simply special cases of the general principles written specifically for the ER model. ER Mapping Rule #1 gathers together key attributes and non-key attributes from a single regular entity set that has no connecting functionally dependent entity sets. ER Mapping Rule #2 gathers together key attributes and non-key attributes from a single regular entity set along with directly functionally dependent attributes that are not keys and the attributes of a primary key from each entity set functionally dependent on the single regular entity set for which we are constructing the relational schema. ER Mapping Rule #3 and its generalizations gather together all attributes from all entity sets in a one-to-one correspondence, plus all attributes connected to any of the relationship sets forming the one-to-one correspondence, plus all primary key attributes of all the entity sets connected by a many-one relationship set from any one of the entity sets in a one-to-one correspondence, plus all attributes of these many-one relationship sets. ER Mapping Rule #4 identifies the special case when the composite key for a relationship set consists of primary key attributes of its connecting entity sets. The attributes from these two primary keys along with any attributes connected to the relationship set form the relational schema.

4.2.2 Complex Key Attributes

Key identification is a central component of mapping conceptual models to database schemas. Conceptual models commonly provide a direct way to identify keys via diagramming conventions. For the ER model, it is common to underline key attributes for entity sets as Figure 4.1a shows. This convention along with cardinality constraints imposed over relationship sets provides

---

[1] Note that although $Name$ and $Address$ functionally depend on $RoomNr$ and also on $RoomName$, they do not directly functionally depend on either $RoomNr$ or $RoomName$ because they functionally depend on $GuestNr$, which is not a key attribute for the key set $\{RoomNr,\ RoomName\}$ for which we are building a relational schema.

[2] Note that $Duration$ functionally depends on $Description$, but since $Description$ is a proper subset of $GuestNr\ Description$, we exclude $Duration$. Similarly, we exclude $Name$ and $Address$ since $GuestNr$ is a proper subset of $GuestNr\ Description$.
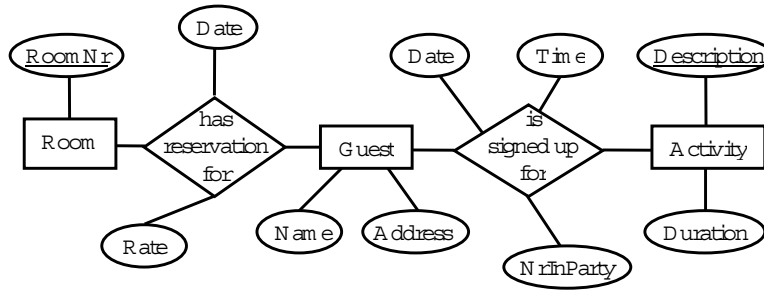
$$Name\,Address \rightarrow Guest$$
$$RoomNr\,Date \rightarrow Name\,Address\,Rate$$
$$Name\,Address\,Date\,Time \rightarrow Description\,NrInParty$$

(a) ER Diagram.

Room(<u>RoomNr</u>)
Guest(<u>Name, Address</u>)
Activity(<u>Description</u>, Duration)
HasReservationFor(<u>RoomNr, Date</u>, Name, Address, Rate)
IsSignedUpFor(<u>Name, Address, Date, Time</u>, Description, NrInParty)

(b) Generated Schemas.

Fig. 4.2 Mappings for Complex Key Attributes: ER Diagram and Generated Schemas.

sufficient information for identifying many keys for entity and relationship sets — indeed most keys in practice.

ER diagramming conventions, however, are not sufficient to allow us to identify all keys. Two common cases are (1) composite keys for entity sets and (2) keys for relationship sets not derivable from information about entity-set keys coupled with relationship-set cardinality constraints. The ER diagram in Figure 4.2a. gives examples of these two cases.

1.   The entity set *Guest* in Figure 4.2a has no key attribute. Neither *Name* alone nor *Address* alone uniquely identifies a guest. Many different people with the same name may make reservations, and we may wish to make it possible for different people with the same address to make reservations. *Name* and *Address* together, however, may uniquely identify *Guest*, and we may designate that this must hold in our database so that *Name Address* becomes a composite key. If no diagramming convention or other formal convention provides a way to designate composite keys for an entity set,[1] it is best to record this information in a note. In our notes here, we use standard functional dependency (FD) notation, allowing both attribute names and entity-set names to appear as components of left-hand

[1] There are many ER variants, and some have conventions to designate composite keys for entity sets (e.g., a connecting line among underlined attributes of an entity set).

and right-hand sides of FDs. To say that *Name Address* is a key for *Guest*, we write the FD *Name Address* → *Guest* as Figure 4.2a shows.

2.   Since only one guest's party can occupy a room on any given date, *RoomNr* and *Date* uniquely identify a reservation. Thus *RoomNr* and *Date* constitute the composite key for the many-many relationship set *has reservation for*. This is contrary to the usual convention, which would have the key consist of the primary keys from the connecting entity sets. It is easy to see, however, that *RoomNr Name Address* cannot be the key because it would allow multiple parties to occupy the same room on the same day. We designate the key for a relationship set by including all the primary-key attributes of all connecting entity sets and all the attributes of the relationship set in an FD. Thus, for example, we write *RoomNr Date* → *Name Address Rate* as Figure 4.2a shows. Similarly, we write *Name Address Date Time* → *Description NrInParty* to designate *Name Address Date Time* as a composite key for *is signed up for*, i.e., a guest's party can only sign up for one activity at a given time on a given date.

Figure 4.2b shows the relational schemas generated from Figure 4.2a. We obtain these relational schemas by following the algorithm in General Principle #3. In Step 1 we generate keys for each of the entity sets and each of the many-many relationship sets. This step yields the set of key sets {{*Name Address*}, {*Description*}, {*RoomNr Date*}, {*Name Address Date Time*}, {*RoomNr*}}. Since each key set has only one key, it becomes the primary key called for in Step 2. Next we add all directly dependent non-key attributes and needed directly dependent primary-key attributes as called for in Step 3 and select names as called for in Step 4. The result consists of the relational schemas in Figure 4.2b.[2]

ER Mapping Rule #1′, **ER Mapping Rule #2′**, and **ER Mapping Rule #3′**. The mapping rules for entity sets with possible composite keys are straightforward generalizations of ER Mapping Rules #1–#3. Their formal statement, however, is quite complex. Basically, instead of just simple single-attribute keys, we must also allow for composite keys. Thus, for example, for ER Mapping Rule #1′ (the generalization of ER Mapping Rule #1), we include all attributes that are keys, all attributes that are parts of composite keys, and all non-key attributes in the generated relational schema. It is possible, though rare, that composite keys overlap. For example, if we had kept *RoomName* from Figure 4.1a in our example for Figure 4.2a, we would also have the composite key *RoomName Date* grouped with the composite key *RoomNr Date*. In this case, the mapping rule should only generate three

---

[2] Whether we should keep a relation for *Room* here is an interesting question. Observe that its data may be completely recorded in the relation *HasReservationFor*. If so, we can discard it. In our application, however, it is possible (even likely) that there is no current reservation for some of the rooms. Thus, to preserve all room numbers in our database, we keep it.

attributes (not four) for these two composite keys. The single occurrence of *Date* in the relational schema would be part of both composite keys.

ER Mapping Rule #4′. For any relationship set, we gather together all attributes constituting primary keys of all related entity sets plus all attributes of the relationship set. We then determine, from among these attributes, which attributes and attribute sets are keys. This becomes the relational schema for the relationship set, except in two special cases having to do with ER Mapping Rules #2′ and #3′. If the primary key of one and only one of the connected entity sets E of the relationship set R is a key for R, then as part of Mapping Rule #2′, all the attributes of the relational schema for R become part of the relational schema for E; there is no separate relational schema for R. If the primary keys of two or more of the connected entity sets E_1, ..., E_n of the relationship set R is each a key for R, then as part of ER Mapping Rule #3′, all the attributes from the relational schemas for E_1, ..., E_n, and R are all combined to form a single relational schema for the database.

4.2.3 Recursive Relationship Sets & Roles

To be understood, recursive relationship sets require roles. Figure 4.3a shows an example. The role *Connecting Room* helps us understand that the relationship set denotes adjoining rooms with a doorway between them: a *Room* is connected with a *Connecting Room*.
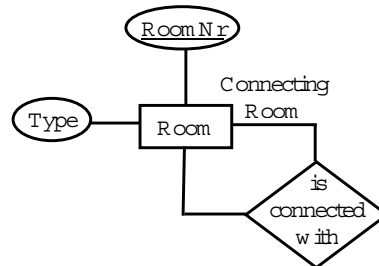
(a) ER Diagram.

    Room(<u>RoomNr</u>, Type)
    IsConnectedWith(<u>RoomNr, ConnectingRoomNr</u>)

(b) Generated Schemas.

Fig. 4.3 Mappings for Roles: ER Diagram and Generated Schemas.

Roles also help us choose attribute names for recursive relationship sets. We map a recursive relationship set to a relational schema in the same way we map regular relationship sets to a relational schema. One-one and many-one

recursive relationship sets become part of the relational schema for the entity set, and many-many recursive relationship sets become relational schemas by themselves. In all cases, however, there is one difference — we must rename one (or both) of the primary-key attributes. Because the regular mapping for a recursive relationship set would make the primary key of the entity set appear twice, and since a relational schema cannot have duplicate attribute names, we must rename one (or both) of them. The role helps because it usually gives a good clue about what one of the names should be. As Figure 4.3b shows, we map the many-many recursive relationship set to the relational schema with attribute names $RoomNr$ and $ConnectingRoomNr$.

We can also use roles in this same way even when a relationship set is not recursive. For example, we could have added a role $Occupied\ Room$ to the $Room$ side of $occupies$ in the ER diagram in Figure 4.1a. In this case we could have generated $Room(\underline{OccupiedRoomNr},\ \underline{OccupiedRoomName},$ $Type,\ Rate,\ GuestNr,\ ExtraCharges)$ in which $OccupiedRoomNr$ replaces $RoomNr$ and $OccupiedRoomName$ replaces $RoomName$ in the relational schema in Figure 4.1b. Using the role name in this way is optional, but may be useful for distinguishing the roles attributes play when we have more than one relationship set between the same two entity sets. For example, we could have also had $has\ reservation\ for$ in addition to $occupies$ as a relationship set between $Room$ and $Guest$ in Figure 4.1a.

4.2.4 Weak Entity Sets

A weak entity set is an entity set with no key among its directly associated attributes. If we augment our bed and breakfast example to a chain of bed and breakfast establishments as Figure 4.4a shows, the room number no longer uniquely identifies a room. Every bed and breakfast in the establishment likely has a Room #1 and a Room #2 and probably more. Thus, $Room$ has no key among its directly associated attributes, and it becomes a weak entity set.

In an ER diagram, we designate a weak entity set by enclosing the name of the weak entity set in a double box. In addition, we add a special relationship set connecting the weak entity set to the entity set on which the weak entity set depends for its key. We designate these special relationship sets by a double diamond and by adding an arrowhead on the side connecting to the entity set on which the weak entity set depends. Adding the arrowhead is appropriate since there is always a many-one relationship from the weak entity set to the entity set on which the weak entity set depends. The ER model does not, by the way, preclude the entity set on which a weak entity set depends from itself being a weak entity set.

Figure 4.4a is an ER diagram showing the three situations in which we normally find weak entity sets. Weak entity sets typically arise (1) when there is
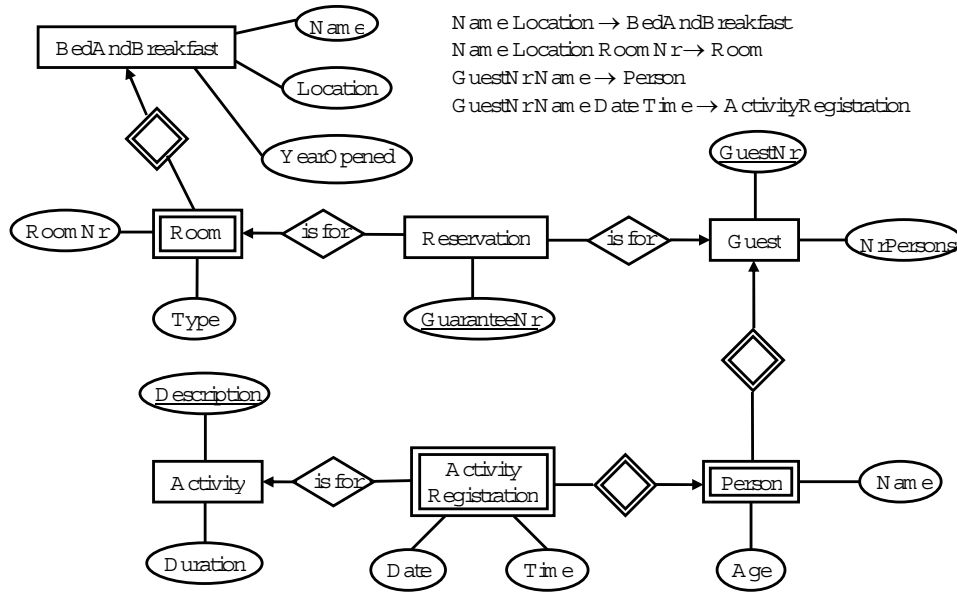
(a) ER Diagram.

BedAndBreakfast(Name, Location, YearOpened)
Room(Name, Location, RoomNr, Type)
Reservation(GuaranteeNr, Name, Location, RoomNr, GuestNr)
Guest(GuestNr, NrPersons)
Person(GuestNr, Name, Age)
Activity(Description, Duration)
ActivityRegistration(GuestNr, Name, Date, Time, Description)

(b) Generated Schemas.

Fig. 4.4 Mappings for Weak Entity Sets: ER Diagram and Generated Schemas.

an organizational subdivision (e.g., *Room* is an organizational division of the *BedAndBreakfast* chain), (2) when one entity depends on the existence of another (e.g., for the bed and breakfast database, *Person* depends on the existence of a registered *Guest*), and (3) when we wish to view relationship sets as entity sets (e.g., an *Activity Registration* rather than just a relationship set between *Person* and *Activity*).[1]

The general principles tell us how to map weak entity sets to relational schemas. We first identify keys. In every case in which we find a weak entity set, the identity of entities in the weak entity set depends on the key for some other one or more entity sets. The identity of a room depends on the bed and breakfast in which the room is located. The key for *BedAndBreakfast*,

---

[1] Note, by the way, that the entity set *Reservation* is not weak, even though it is certainly a relationship set we view as an entity set. When we turned it into an entity set, we gave it a key, *GuaranteeNr*, so that it did not become a weak entity set.

which for our example is the composite key *Name Location*,² plus a *RoomNr* uniquely identify a room. Thus, the key for *Room* is the composite of all three attributes, namely, *Name Location RoomNr*. For *Person*, names are not unique identifiers, but are usually unique within a family, which often constitutes a registered guest's party. In our example, we require unique names within a registered guest's party, and thus the key for the weak entity set *Person* is the composite of *GuestNr* and *Name* (of *Person*). A person can sign up for only one activity at a given time on a given date. Thus, to uniquely identify an *ActivityRegistration*, we need the key of *Person*, which is *GuestNr Name*, as well as *Date* and *Time* to all be part of the key. Thus, the composite key for *ActivityRegistration* is *GuestNr Name Date Time*.

After identifying keys for a weak entity set (and designating one to be the primary key in case there are several), we add all directly dependent non-key attributes and directly dependent primary-key attributes. We then choose a name — usually the name of the weak entity set — and form a relational schema. Figure 4.4b shows the result for the ER diagram in Figure 4.4a. For the weak entity set *Room*, the only directly dependent attribute is *Type*. Thus, since the key is the composite *Name Location RoomNr*, the relational schema is *Room*(<u>*Name*</u>, <u>*Location*</u>, <u>*RoomNr*</u>, *Type*), the second relational schema in Figure 4.4b. Similarly, for *Person*, since *Age* is the only directly dependent attribute, the relational schema for the weak entity set is *Person*(<u>*GuestNr*</u>, <u>*Name*</u>, *Age*). For the weak entity set *ActivityRegistration*, *Description* is a directly dependent primary-key attribute. Thus, since the composite key is *GuestNr Name Date Time*, we generate the last relational schema in Figure 4.4b.

ER Mapping Rule #5. Let W be a weak entity set, let E, entity set on which W depends (E may itself be weak), and let F_1, ..., F_m be the m other entity sets (if any) in a many-one relationship from W to F_i ($1 \leq i \leq m$). Form a relational schema called W with attributes consisting of all the attributes of W, the primary-key attribute(s) of E, all the primary-key attributes of F_1, ..., F_m, and all the attributes (if any) of the many-one relationship sets from W to each F_i ($1 \leq i \leq m$). From among the attributes, determine the keys for W and designate one as the primary key. Each key of W is formed from by adding one or more attributes of W to the primary key for E.³

---

² *Location* is meant to be a simple city or town or other designated place. Several bed and breakfast establishments can be in the same location (e.g., Boston), but each establishment in the same location must have a different name. Thus, *Name Location → BedAndBreakfast*.

³ Typically, as in our examples here, W has only one key. But, for example, if we also had *RoomName* for the weak entity set *Room*, we would have a second key for *Room*, namely *Name Location RoomName*.
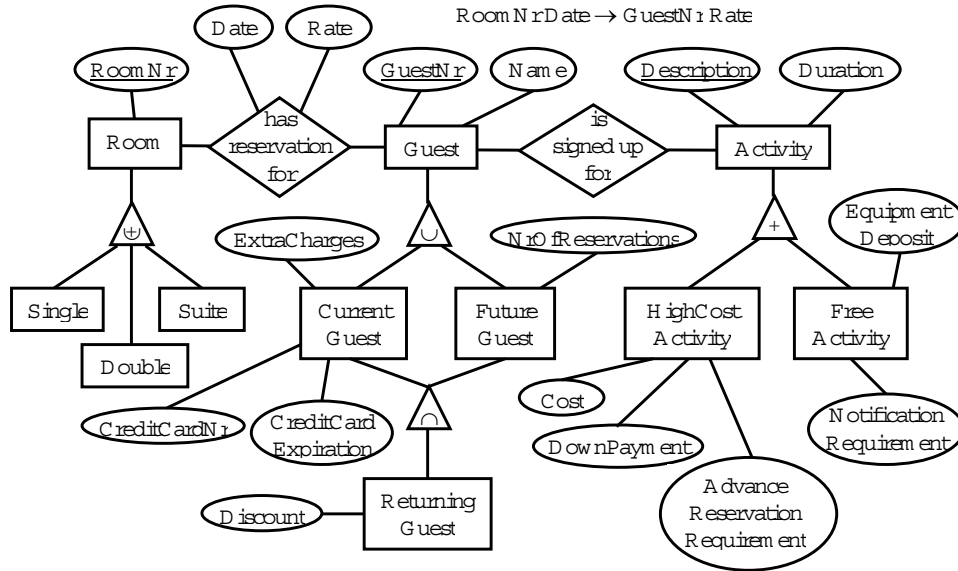
(a) ER Diagram.

    Room(RoomNr, RoomType)
    Guest(GuestNr, Name, ExtraCharges?, CreditCardNr?, CreditCardExpiration?,
        NrOfReservations?, Discount?)
    Activity(Description, Duration)
    HighCostActivity(Description, Cost, DownPayment, AdvanceReservationRequirement)
    FreeActivity(Description, EquipmentDeposit, NotificationRequirement)
    HasReservationFor(RoomNr, Date, GuestNr, Rate)
    IsSignedUpFor(GuestNr, Description)

(b) Generated Schemas.

Fig. 4.5 Mappings for ISA Hierarchies: ER Diagram and Generated Schemas.

4.3 Extended Entity-Relationship Model Mappings

Extended ER models include generalization/specialization or ISA hierarchies, with possible union, disjoint, overlapping, partition, and intersection constraints. They also include compound attributes, multi-valued attributes, computed attributes, and designations for mandatory and optional participation. In this section, we consider mappings for each of these extensions.

4.3.1 ISA Mappings

Figure 4.5a shows an ER diagram with several ISA constructs. Graphically, triangles denote ISA constructs, which fundamentally designate entity sets as subsets and supersets.[1] An entity in a subset entity set is also an entity in its superset entity set — thus the "ISA" designation. The apex of a triangle connects to superset entity sets, and the base connects to subset entity sets. In Figure 4.5a both *CurrentGuest* and *FutureGuest* are subsets of *Guest*, and *ReturningGuest* is a subset of both *CurrentGuest* and *FutureGuest*. We can constrain the subsets by placing symbols in the triangles: ∩ for intersection, ∪ for union, + for mutual-exclusion, and ⊎ for partition. An intersection constraint requires that the subset entity set be exactly the intersection of the superset entity sets (e.g., in Figure 4.5a *ReturningGuest* is exactly the intersection of *CurrentGuest* and *FutureGuest*). Without the intersection constraint (triangle with no special constraint inside it) the subset entity set could be a proper subset of the intersection. Union requires that the superset is exactly the union of the subsets (e.g., *Guest* is defined to be exactly the union of those guests who are currently at the bed and breakfast and those who will be future guests). Mutual-exclusion requires that the subsets pairwise have an empty intersection (e.g., *HighCostActivity* and *FreeActivity* have an empty intersection). Partition requires both union and mutual-exclusion (e.g., a room must be a single room, a double room, or a suite).

Figure 4.5b shows the relational schemas generated from the ER diagram in Figure 4.5a. For ISA hierarchies, there are three basic mappings, which we label as ER Mapping Rules #6.1–#6.3. (Combinations over multiple-level hierarchies are also possible.)

ER Mapping Rule #6.1. *Make a relational schema for all entity sets in the hierarchy.* Although not always best, this is the most straightforward mapping for ISA hierarchies. The mapping for an entity set in an ISA hierarchy that has no generalization is the same as the mapping for any entity set. The mapping for a specialization is also the same except that the primary-key attribute(s) of the generalization(s)[2] are also added to the relational schema. In general any key for the specialization can be the primary key. Normally, however, there will be only one key, the inherited primary key. In Figure 4.5

---

[1] In this context, we call subset entity sets "specialization entity sets" or just "specializations" and call superset entity sets "generalization entity sets" or just "generalizations."

[2] Since all specializations in an ISA hierarchy are subsets of their generalizations, entities in the specializations inherit their identity from their generalization(s). In most common cases there is only one generalization. When a specialization has more than one generalization, it inherits its identity from all generalizations. Often, however, all generalizations have the same identifying attribute inherited from some root generalization. *ReturningGuest* in Figure 4.5a inherits its identity from both *CurrentGuest* and *FutureGuest*, but these entity sets, in turn, both inherit their identity from *Guest*. Thus, *GuestNr* becomes the one and only key attribute that identifies returning guests.

we map the ISA hierarchy rooted at *Activity* in this way. The relational
schema for *Activity* in Figure 4.5b is the same as it would be without the
ISA hierarchy. *HighCostActivity* and *FreeActivity*, however, both inherit
the primary key *Description* from *Activity* and include it as the primary key
for their relational schemas along with all directly dependent attributes.

ER Mapping Rule #6.2. *Make a relational schema for only root entity
sets.* For this mapping, we collapse the entire ISA hierarchy to the entity set
of the root generalization,[3] so that all attributes of specializations become
attributes of the root and all relationship sets associated with specializations
become relationship sets associated with the root. We then map this single
entity set to a relational schema in the usual way. After doing the mapping,
we determine which attributes are nullable. All attributes that would not have
been in the mapping if we had not collapsed the ISA hierarchy are nullable.
In our relational schemas, we mark nullable attributes with a question mark.
When we transform generic relational schemas to SQL *create* statements for
implementation, we allow nullable attributes to have the value NULL; non-
nullable attributes may not have the value NULL. In Figure 4.5 we map the
ISA hierarchy rooted at *Guest* in this way. When collapsing the ISA hierarchy,
the attributes of the three specializations, all become nullable attributes of
Guest. As Figure 4.5b shows, these five attributes all have an appended
question mark.

We might wonder if this mapping causes us to lose track of which guests
are current guests, which are future guests, and which are returning guests.
For this example we do not lose the information. According to the seman-
tics of the ER model instance in Figure 4.5a only returning guests will have
Discount values whereas current and future guests who are not returning
guests will not have *Discount* values. Future guests will have a value for
NrOfReservations whereas current guests will not. Similarly, current guests
will have extra charges, credit card numbers, and credit card expirations
whereas future guests will not. Sometimes, however, it is not possible to know
the specialization categories based on attribute values, and even when it is
possible, we may wish to have a way to record the specialization categories.
The following two additions to Mapping Rule #6.2 show us how we can pro-
vide attributes in which we can record this information about specializations.

- **ER Mapping Rule #6.2a**. *Add a new attribute for each specialization.*
  When mapping the generalization entity set to a relational schema, gen-
  erate an additional attribute for every specialization entity set. Values
  for these attributes are Boolean, saying for each record of the relational
  schema whether the entity the record represents is or is not in the spe-
  cialization. If we were to add these additional attributes for *Guest*, the
  relational schema for *Guest* in Figure 4.5b would instead be:

    Guest(*GuestNr*, *Name*, *ExtraCharges*?, *CreditCardNr*?,

---

[3] Although rare, if there are multiple roots, we collapse the hierarchy to all roots. Any
entity set that is the specialization of multiple roots collapses to all of them.

$CreditCardExpiration?$, $NrOfReservations?$, $Discount?$,
$CurrentGuest$, $FutureGuest$, $ReturningGuest$)

If we wish, we could omit *ReturningGuest* and just compute its value for a record as *yes* if both values for *CurrentGuest* and *FutureGuest* are *yes* and as *no* otherwise.

- **ER Mapping Rule #6.2b**. *Add only one new attribute representing all specializations.* This mapping only applies when the specializations are mutually exclusive. If so, when mapping the generalization entity set to a relational schema, we only need to generate one additional attribute to represent all specializations. The specialization entity-set names can be the values for this new attribute. In Figure 4.5a, *Room* has three mutually exclusive specializations that designate the room type. We therefore generate a new attribute, *RoomType*, for the generalization entity set *Room*. The values for this attribute can be the names of the specialization entity set. The generated relational schema for *Room* in Figure 4.5b has the attributes *RoomNr* and *RoomType*. Values for *RoomType* would be "*Single*", "*Double*", and "*Suite*" or any other designating value to say that the room is a single room, a double room, or a suite.

ER Mapping Rule #6.3. *Make a relational schema for only the leaves in the hierarchy.* The leaf entity sets inherit all attributes and all relationship sets from parents along a path all the way back to the root.[3] This mapping only applies when union constraints are present all along all paths. If a union constraint were missing, there could be members of the entity sets in the hierarchy that would not appear in the leaf entity sets and thus would be lost in the implementation. Further, we usually only apply this mapping when mutual-exclusion is also present along all paths. If not, then members of the entity sets could appear in more than one leaf entity set and thus would appear as duplicates in the implementation, once for each leaf entity set in which a member appears. As an example, assume that there are only high-cost activities and free activities and thus that the constraint for the ISA hierarchy rooted at *Activity* in Figure 4.5a is a partition ($\uplus$) constraint rather than a mutual-exclusion ($+$) constraint. Applying the mapping in which we only represent the leaves of the ISA hierarchy, we would replace the three relational schemas *Activity*, *HighCostActivity*, and *FreeActivity* in Figure 4.5b by the following two relational schemas:

$HighCostActivity(\underline{Description}, Duration, Cost, DownPayment,$
    $AdvanceReservationRequirement)$
$FreeActivity(\underline{Description}, Duration, EquipmentDeposit,$
    $NotificationRequirement)$

Observe that both *HighCostActivity* and *FreeActivity* include the attribute *Duration* as well as *Description* and that the connection to the *IsSignedUpFor* relational schema is accounted for through the *Description*

---

[3] If there are multiple roots, the leaves inherit from all roots.

attributes. When we make a relational schema for only the leaves in an ISA hierarchy, we must account for all attributes and relationship-set connections of all ancestors of each leaf entity set.

General Principle #4. Map ISA hierarchies to relational schemas by choosing to make a relational schema for (1) all entity sets in an ISA hierarchy, (2) only root entity sets, or (3) only leaf entity sets. Although there are guidelines that typically indicate which of the three mappings to use, making the best choice is often application dependent. Deciding among the possibilities depends on the ISA constraints and the number of attributes involved. Designers use the following rule-of-thumb guidelines.

- Select (1) when the generalizations and specializations all have many attributes.
- Select (2) when the specializations collectively have few attributes.
- When the specializations have no attributes, select (2a) either for an ISA union constraint or for an ISA with no constraint.
- When the specializations have no attributes, select (2b) either for an ISA partition constraint or for an ISA mutual-exclusion constraint.
- Select (3) for an ISA partition constraint, especially when there are many attributes for the specializations and few for the generalizations.

Often there is no obvious best choice. In this case the developer must choose one. Furthermore, in some complex cases, especially with large hierarchies, it may be best to make a separate choice for each individual ISA configuration in the hierarchy. In the end the mappings must account for representing all possible entities in every entity set in the ISA hierarchy and all possible relationships and attributes of these entities.

4.3.2 Mappings for Complex Attributes

Extended ER models allow for several types of complex attributes. Figure 4.6a includes examples for each type.

- A *multi-valued attribute* is an attribute whose values are sets of values. In an extended ER diagram, we denote a multi-valued attribute by a double oval. In Figure 4.6, *ActivityInterest* is a multi-valued attribute. Guests may be interested in several activities — one guest may be interested in the set of activities {golf, horseback riding, canoeing} while another guest may be interested in the set of activities {chess, hiking, canoeing}. *View* is also a multi-valued attribute, whose sets might be {Ocean, CityOverlook} or just {Ocean} depending on what can be seen by looking out the window(s) of a room in the bed and breakfast establishment.
- A *compound attribute* is an attribute with component parts each of which is also an attribute. In an extended ER diagram, we denote compound attributes by attaching component attributes directly to the compound at-

(a) ER Diagram.
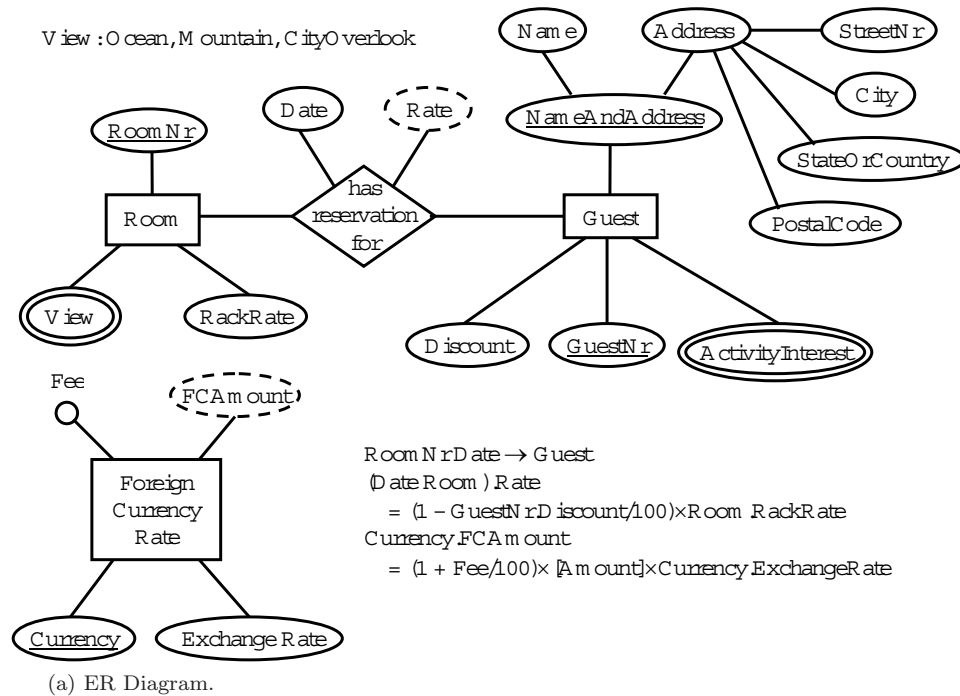
Guest(GuestNr, Name, StreetNr, City, StateOrCountry, PostalCode, Discount)
GuestActivityInterest(GuestNr, ActivityInterest)
Room(RoomNr, RackRate, Ocean, Mountain, CityOverlook)
HasReservationFor(RoomNr, Date, GuestNr, Rate)
ForeignCurrencyRate(Currency, ExchangeRate)
Fee(Fee)

(b) Generated Schemas.

Fig. 4.6 Mappings for Complex Attributes: ER Diagram and Generated Schemas.

tribute. In Figure 4.6a, $NameAndAddress$ is a compound attribute whose component attributes are $Name$ and $Address$. The component attribute $Address$ is also compound; its component attributes are $StreetNr$, $City$, $StateOrCountry$, and $PostalCode$.

- A *computed attribute* is an attribute whose value can be computed. In an extended ER diagram, we denote a computed attribute by a dashed oval. In Figure 4.6a, $FCAmount$ and $Rate$ are computed attributes.
- An *entity-set attribute*, called in other contexts a *class attribute*, is an attribute whose value is the same for all entities in the entity set and thus can be thought of as applying to the entire set of entities rather than each individual entity in the set. In an extended ER diagram, we denote an entity-set attribute by a small circle. In Figure 4.6a, Fee is an example.

It is a percentage and is meant to be the fee collected by the bed and breakfast establishment for accepting payment in a foreign currency.

Figure 4.6b shows how we map the various complex attributes in Figure 4.6a to relational schemas. Basically, the mappings are straightforward applications of the general principles. The cardinality constraints for multi-valued attributes make them have the properties of many-many relationship sets. The collapsing of compound-attribute trees make the leaves of these trees have the properties of directly dependent attributes. Computed attributes are the same as regular attributes except we may not need to store them. Although entity-set attributes could be treated as regular attributes, their singleton property makes them amenable to a different kind of mapping, making entity-set attributes an exception to the general principles.

ER Mapping Rule #7. Fundamentally, multi-valued attributes are in a many-many relationship with the entity set to which they are connected. Each entity in an entity set E with a multi-valued attribute A relates to n values v_1, ..., v_n of A, and each value of A relates to m entities e_1, ..., e_m in E. Thus, unless the number of values in A, $|A|$, is fixed and small, we treat A as if it were another entity set E' in a many-many relationship with E; E''s only attribute, and therefore its primary-key attribute, is A. When $|A|$ is fixed and small, it is possible to treat it as $|A|$ attributes v_1, ..., $v_{|A|}$ of E, whose values are Boolean stating whether an entity e relates to that value or does not relate to that value.

- **ER Mapping Rule #7a**. *If entity set E has a multi-valued attribute A, then if P is the primary key of E, generate the relational schema $N(\underline{P, A})$. If the primary key of E happens to be composite, P represents the attribute list for the composite primary key. N is a chosen name — often a concatenation of the name of E and the name of A.* The relational schema *GuestActivityInterest* in Figure 4.6b is an example. A guest can be interested in may different activities, and an activity can be of interest to many different guests. Thus, since *GuestNr* is the primary key of *Guest*, we generate *GuestNr* and *ActivityInterest* as the attributes and as the composite key for the relational schema.
- **ER Mapping Rule #7b**. *As an exception to ER Mapping Rule #7a, if entity set E has a multi-valued attribute A, n is the size of A, and n is fixed and small, then if $A = \{V_1, ..., V_n\}$, add V_1, ..., V_n as Boolean attributes to the relational schema formed for E.* The relational schema *Room* in Figure 4.6b is an example. As specified in a note in the diagram, a room can have only up to three views (*Ocean*, *Mountain*, or *CityOverlook*). Thus, for the multi-valued attribute *View* of *Room*, we add these three view names, *Ocean*, *Mountain*, and *City* as attributes to the relational schema. Values for these attributes are Boolean: if a front corner room has all three views, all three attribute values would have the value *yes*, and if a back center room looks out only on the mountains, the *Mountain* value would be *yes* and the *Ocean* and *cityOverlook* values would be *no*.

ER Mapping Rule #8. *Treat each leaf attribute of a compound attribute tree T of an entity set E as an attribute of E; then map E in the usual way. In addition, if any non-leaf node N of T is a key for E, form a composite key from the leaf attributes of the subtree rooted at N.* In Figure 4.6a *Guest* has a compound attribute *NameAndAddress*. Its leaf attributes are *Name*, *StreetNr*, *City*, *StateOrCountry*, and *PostalCode*. Thus, for *Guest* we form a relational schema with these attributes along with the regular attributes *GuestNr* and *Discount*. (Being multi-valued, *ActivityInterest* is not a regular attribute and is not included—neither are the non-leaf attributes of the compound attribute tree, *NameAndAddress* and *Address*.) Further, since the non-leaf attribute *NameAndAddress* is a key for *Guest*, we form a composite key from all its leaf attributes as Figure 4.6b shows.

ER Mapping Rule #9. *If a computed attribute is to be stored in the database, treat it as a regular attribute for the purpose of mapping it to a relational schema.* When values for attributes are computed, we may or may not want to store their values in the database. If computed values serve to initialize a field and the field value may later change, we store the values. In this case, there must be an attribute for it in the generated relational schema. On the other hand, if the value is computed from other existing values whenever we need it, we need not store it. In this case, we ignore it when generating relational schemas. In our example in Figure 4.6a, *Rate* is an initial value, which depends on a guest's discount and the room's rack rate but which can be set to another value. Thus, we generate an attribute for *Rate* in the relational schema for *HasReservationFor*, the place it would go if it were a regular attribute. *FCAmount*, however, is only computed when a guest wants to know how much to pay if the amount owed is to be paid in a foreign currency. Thus, we do not generate an attribute for *FCAmount* in any relational schema.

ER Mapping Rule #10. *For an entity-set attribute A, we either ignore it or map it to a single-attribute, single-valued relation A(A).* Values for entity-set attributes may be constants established in the program code, may be values accepted as input values when needed, or may be stored in the database and updated occasionally. In our example, we store the fee value as a percentage number in the database and thus need the relational schema *Fee(Fee)* as Figure 4.6b shows.

4.3.3 Mappings for Mandatory/Optional Participation

Figure 4.7a illustrates mandatory and optional participation in an ER diagram. We designate optional participation by placing a small "o" near a connection for attributes and relationship sets, and we designate mandatory participation by the absence of an "o." Optional participation for an attribute *A* means that an entity in an entity set need not have a value for *A*; manda-
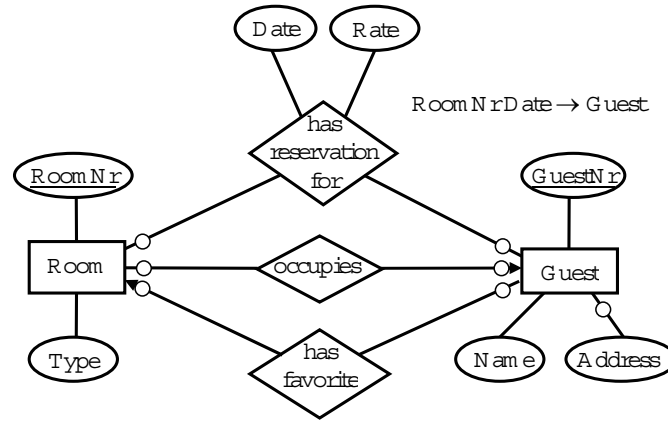
(a) ER Diagram.

RoomAndOccupant(RoomNr, Type, GuestNr?)
GuestAndFavoriteRoom(GuestNr, Name, Address?, RoomNr?)
HasReservationFor(RoomNr, Date, GuestNr, Rate)

(b) Generated Schemas.

Fig. 4.7 Mappings for Mandatory/Optional Constructs: ER Diagram and Generated Schemas.

tory participation means that an entity must have a value. In Figure 4.7a a *Guest* need not provide an *Address* when registering (i.e., the database system will allow the *Address* field in a record for a *Guest* to be null). Optional participation for a relationship set R means that an entity in an entity set need not participate in the relationship set; mandatory participation means that it must participate. For example, a *Room* in Figure 4.7a need not be occupied, need not be anyone's favorite, and need not be reserved by anyone. Similarly, someone recorded as a *Guest* in the database need not have a reservation, need not occupy a room, and need not have a favorite room. The database would allow, for example, a record to be kept for someone who had been a guest, but is not currently occupying a room, has no reservation, and has no particular favorite room.

Figure 4.7b shows how we consider optionality when we map to relational schemas. As before, the question mark means that an attribute is nullable. When attributes are optional, they are nullable; when they are mandatory, they are not nullable. Thus, for example, *Address* has an appended question mark in *GuestAndFavoriteRoom* whereas *Name* does not. When attributes of a relationship set plus the primary-key attributes of the associated entity sets are mapped into a relational schema for an entity set, if participation in the relationship set is optional, these imported attributes are all nullable. When participation is mandatory, these attributes are not nullable. Thus, for example, *GuestNr* is nullable in *RoomAndOccupant*. Be-

cause *occupies* is many-one from *Room* to *Guest*, the primary key for *Guest*, which is *GuestNr*, becomes one of the attributes of *RoomAndOccupant*, and because participation of a *Room* in the *occupies* relationship set is optional, *GuestNr* is imported as a nullable attribute. Similarly, *RoomNr* is nullable in *GuestAndFavoriteRoom* since the *has favorite* relationship set is many-one from *Guest* to *Room* and *Guest* optionally participates in *has favorite*. Observe that the relationship set *has reservation for* is many-many and is not imported into either *Room* or *Guest*. Thus, there is no special mapping based on the optionality of reservations for rooms and guests.

ER Mapping Rule #11. *Mark all nullable attributes in generated relational schemas. Attributes that are the primary key of a relational schema or that are in the composite primary key of a relational schema are never nullable. All other attributes may be nullable and should be so marked if their value for a record can be NULL.*

This rule is a little different from all other rules. It is written with respect to generated schemas. Thus, it does not say how to generate schemas, neither does it say exactly how to decide which attributes are nullable. Rather, it says which attributes are and are not potentially nullable, and it says that among those that are potentially nullable the database designer should decide which ones should allow null values. The reason for writing the rule this way is twofold:

1.   Many ER diagrams never specify mandatory/optional participation (sometimes because the notation does not allow it, and sometimes just because it is not commonly done). Thus the nullable properties of attributes in relational schemas are often not derivable from the ER diagram.

2.   When mandatory/optional participation can be specified in an ER diagram, even if someone specifies that an attribute that turns out to be part of the primary key of some relational schema should be nullable, it cannot be nullable. Relational database systems do not allow nulls in primary keys.

To illustrate Reason #1, we can consider the ER diagrams in Figures 4.1–4.6 in which no optional participation explicitly appears. One view we can take for all these diagrams is that there is no optional participation. In this case, all generated relational schemas remain as they are. This point of view, however, does say something about the semantics of the schemas. For example, in the *Room* schema in Figure 4.1b, we can only record room numbers, type, and rate for occupied rooms. If we want to store this information for unoccupied rooms, we would be forced to enter some bogus *GuestNr* (e.g., -1) and some bogus value for *ExtraCharges* (e.g., 0). It may be better to simply allow these attributes to be nullable. The same is true for address of a guest in the *Guest* schema in Figure 4.1b. Even if the guest is in the process of moving and has no address to give, or if the guest refuses to give an address, something (e.g., "none") must be entered.

To illustrate Reason #2, consider what it would mean if *RoomNr* for *RoomAndOccupant* in Figure 4.7b were marked as optional and thus allowed to be nullable. This means that some rooms would have no identifier — no room number. Suppose we try to store information about several no-number double rooms that currently have no occupants. Even if the database would let us store the room number as null, we would have trouble since we could not distinguish the rooms from one another. We would not even know how many unoccupied double rooms we have. This motivates the rule: *Attributes of primary keys in a relational schema may not be null.* Note that this rule does not say that key attributes cannot be null — only that primary-key attributes cannot be null. Suppose, for example, that a guest can have a guarantee number (*GuaranteeNr*) that uniquely identifies the guest in addition to a guest number (*GuestNr*). We could then add *GuaranteeNr* as a key attribute to the attributes already in *GuestAndFavoriteRoom* in Figure 4.7b and let it be nullable so that not all guests would have to have a guarantee number. Note also that this rule does not say that primary-key attributes imported into a relational schema cannot be null. Indeed, *GuestNr* is nullable in *RoomAndOccupant*, where it is not the primary key, even though it is the primary key for *GuestAndFavoriteRoom*.

General Principle #5. Make attributes nullable if they can have NULL as their value. Attributes of primary keys in a relational schema are not nullable.

4.4 UML Mappings

In this section, we explain how to generate relational schemas from UML class diagrams. We base our procedure on the general principles, and thus this explanation serves as an example of how to develop mapping rules to map any conceptual model to relational schemas. In general, we first need to understand the syntactic features of the conceptual model and determine which corresponding ER features or extended ER features they denote. We can then map them to relational schemas in the same way we map ER features to relational schemas.

We illustrate our UML mappings using the class diagram in Figure 4.8 as an example. We begin by pointing out several syntactic features of this class diagram and explain how they correspond to (extended) ER features. First, UML does not provide a graphical notation for specifying keys.[4] UML does, however, provide an *Object Constraint Language* (*OCL*), in which we can express FDs. Figure 4.9 shows an example of how to specify the FD *name location $\rightarrow$ BedAndBreakfast*. Thus, to derive keys of classes in a class di-

---

[4] UML does not use underlines to denote keys for classes; rather it uses underlines to denote static attributes — attributes that belong to classes, not attributes applicable to instances of classes.
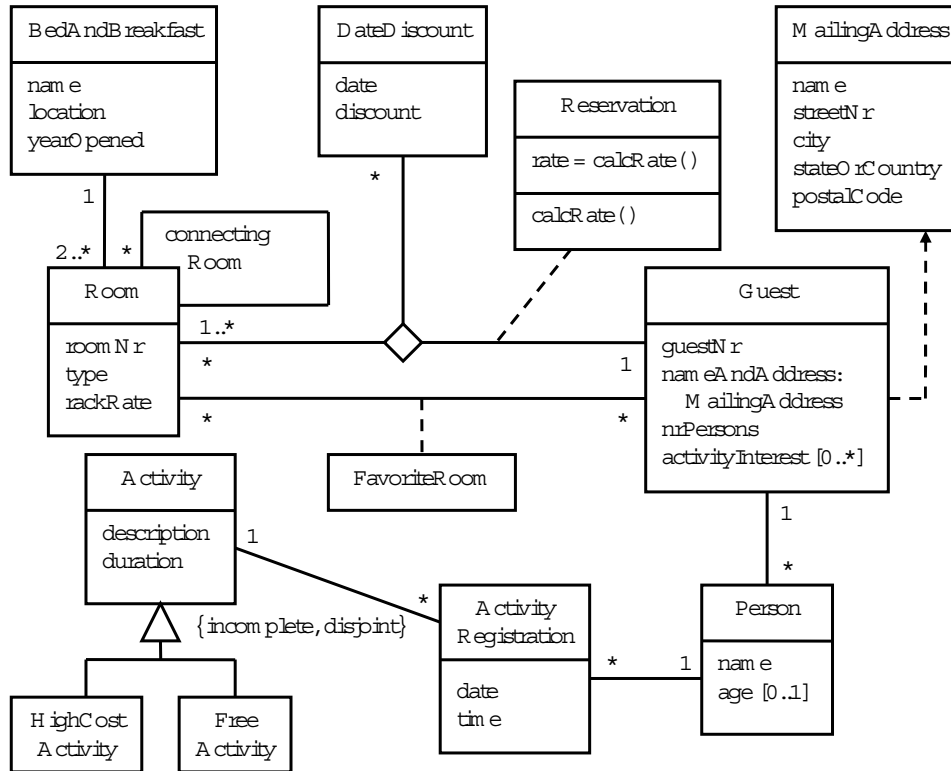
Fig. 4.8 UML Class Diagram

agram, we consult the OCL expressions associated with the class diagram. Second, we can use attribute multiplicity in class diagrams to specify optional attributes and multi-valued attributes. In Figure 4.8, *age* is an optional attribute and *activityInterest* is a multi-valued attribute. Third, UML allows the definition of a class to rely on the definition of another class. This allows us to specify a compound-attribute groups. For example, *MailingAddress* is a compound attribute group on which the class *Guest* depends. Fourth, UML allows operations to be defined in class diagrams. For example, *rate*, a computed attribute for the association *Reservation*, takes on the result of the operation *calcRate*() as its initial value.

Based on the general principles, we now present a high-level algorithm that generates relational schemas from a UML class diagram. As an example, we show how the UML diagram in Figure 4.8 along with the OCL constraints in Figure 4.9 map to the relational schemas in Figure 4.10.

Step 1   *Based on General Principles #1 and #2, identify keys for each class.*
    In our example, the UML OCL provides us with these keys. The left-hand-

context BedAndBreakfast
inv: BedAndBreakfast.allInstances()->forAll(b1, b2 |
    b1.name = b2.name and b1.location = b2.location implies b1 = b2)
    -- *name location → BedAndBreakfast*
...   -- *roomNr name location → Room*
...   -- *guestNr → Guest*
...   -- *date → DateDiscount*
...   -- *name guestNr → Person*
...   -- *description → Activity*
...   -- *name guestNr date time → ActivityRegistration*

Fig. 4.9 OCL for an FD Plus Other FDs Declared for Classes in Figure 4.8.

BedAndBreakfast(<u>name, location</u>, yearOpened)
Room(<u>roomNr, name, location</u>, type, rackRate)
ConnectingRoom(<u>roomNr, name, location, connectingRoomNr</u>)
Guest(<u>guestNr</u>, name, streetNr, city, stateOrCountry, postalCode, nrPersons)
GuestActivityInterest(<u>guestNr, activityInterest</u>)
FavoriteRoom(<u>roomNr, name, location, guestNr</u>)
DateDiscount(<u>date</u>, discount)
Reservation(<u>roomNr, name, location, date</u>, guestNr, rate)
Person(<u>name, guestNr</u>, age?)
Activity(<u>description</u>, duration, costLevel?)
ActivityRegistration(<u>name, guestNr, date, time</u>, description)

Fig. 4.10 Schemas Generated from Figures 4.8 and 4.9.

side of each FD in Figure 4.9 is the key for the class on the right-hand-side
of the FD.

Step 2   *Based on General Principles #1 and #2, identify keys for each as-
sociation.* Usually multiplicity constraints of associations along with keys
for classes determine these additional FDs. For example, if A is an n-ary
association that connects n classes C_1, ..., C_n whose primary keys are re-
spectively P_1, ..., P_n and the maximum value of the multiplicity for C_n in
this association is 1, then the FD $P_1...P_{n-1} → P_n$ holds for A. In our ex-
ample, we have *roomNr name location date → GuestNr* for the ternary
association *Reservation* in Figure 4.8. When an association has no max-1
multiplicity constraints, the key is the composite of the primary keys for
all associated classes. For example, *roomNr name location guestNr* is
the key for *FavoriteRoom*.

Step 3   *Based on General Principle #4, determine how generalization/spe-
cialization should be mapped.* Since the ISA constraints for the only gen-
eralization/specialization in Figure 4.8 are *incomplete* and *disjoint* and
since there are no associations for the specializations, we choose to intro-

duce one new attribute, *costLevel* representing all specializations as ER
Mapping Rule #6.2b suggests.

Step 4   *Based on General Principle #3, map classes to relational schemas.*
We generate a relational schema R for a class C as follows. R has the
following attributes:

- all the attributes of C except (1) those whose multiplicity is greater
  than one (e.g., *activityInterest* in Figure 4.8) and (2) those compound
  attributes that reference another class (e.g., *nameAndAddress* in Fig-
  ure 4.8);
- all leaf attributes of compound attributes (e.g., *name*, *streetNr*, *city*,
  stateOrCountry, and *postalCode* are all included in the relational
  schema for *Guest*);
- all attributes included in the primary key for C, if not already included
  (e.g., *name* and *location* from the class BedAndBreakfast are included
  in the relational schema for *Room*); and
- for each association A, if a key of A is also a key of R, then (1) all
  attributes of A, if any, and (2) for each class C' connected to C by A,
  the primary-key attributes of C' (e.g., *description*, the primary key of
  Activity, belongs in the relational schema for *ActivityReservation*).

Step 5   *Based on General Principle #3, map remaining associations to re-
lational schemas.* An association remains after Step 4 only if it is not
one-one and not one-many or many-one. We generate a relational schema
R for each remaining association A as follows. For each class C connected
by A, R includes the primary-key attributes of C. *FavoriteRoom* and
Reservation in Figure 4.10 are examples.

Step 6   *Based on General Principle #3, map multi-valued attributes to re-
lational schemas.* For each multi-valued attribute M in a class C, generate
a relational schema that includes M and the primary-key attributes of
C. The multi-valued attribute *activityInterest* in *Guest* in Figure 4.8 is
an example, which yields the relational schema *GuestActivityInterest* in
Figure 4.10.

Step 7   *Based on General Principle #5, identify nullable attributes.* In our
example *age* is nullable because it is a single-valued, optional attribute of
Guest, and *costLevel* is nullable because the ISA hierarchy in Figure 4.8
has an *incomplete* constraint.

Once we have relational schemas for the database, like the ones in Fig-
ure 4.10, we can derive SQL DDL for a relational database. We illustrate
here[5] how to turn generated relational schemas into SQL table creation state-
ments. Figure 4.11 shows our SQL DDL for the first three relational schemas
in Figure 4.10. The translation is straightforward.

---

[5] We could have illustrated the derivation of SQL DDL for all earlier generated schemas
as well as this one, but we only illustrate this derivation once.

1. Obtain the name and basic attribute structure for the tables to be created directly from the generated schemas. As Figure 4.11 shows, the *BedAndBreakfast* table has the attributes *name*, *location*, and *yearOpened*. We can rename attributes to make them more understandable. *BandBname* and *BandBlocation* are preferable to *name* and *location* in *Room* (otherwise most people would read *name* in *Room* as the name of the room and *location* as the location of the room).

2. Represent the constraints captured in the diagram and generated schemas. The primary-key constraints come directly from the relational schemas, as do other uniqueness constraints. The foreign-key constraints come indirectly from the relational schemas. An attribute or attribute group that is not a key in a relational schema R but is a key in a relational schema S is a foreign key for R that references S. *BandBname* and *BandBlocation*, for example, constitute a composite foreign key in *Room* that references the composite key *name* and *location* in *BedAndBreakfast*. As may be desirable, we also add *check* constraints, such as *roomNr* != *connectingRoomNr*. These constraints, however, are not derivable from the relational schemas we generate.

3. Add type declarations, which are usually only implicit in the conceptual-model instance. The types *varchar*, *number*, and *money* are examples.

4. Reflect the null properties of the relational schemas in the SQL DDL. Primary-key attributes are *not null* by default. All other attributes are nullable unless otherwise specified by a *not null* constraint.

4.5 Normal Form Guarantees

When mapping conceptual models to database schemas, the question of normalization naturally arises. Are generated relational schemas fully normalized? By "fully normalized," we mean they are in PJNF — Project-Join Normal Form — which also implies that they are in 4NF, BCNF, 3NF, 2NF, and 1NF. Interestingly, we can answer "yes" for conceptual-model diagrams that are *canonical*.

Although circular, the easiest way to define *canonical* is that when mapped according to the mapping rules or algorithms, every relational schema is fully normalized. In practice, many (if not most) diagrams are canonical.[6] Giving a general statement that characterizes canonical for all types of conceptual models (or even for one type of conceptual model) is difficult especially if the characterization is to be given in the least constraining way. Many conceptual model instances, however, satisfy stronger than necessary conditions, and it

---

[6] Many argue that if conceptual-model diagrams are not canonical, they are not good quality diagrams.

```
create table BedAndBreakfast(name varchar(20),
    location varchar(20),
    yearOpened number(4) not null,
    primary key (name, location));
create table Room(roomNr number,
    BandBname varchar(20),
    BandBlocation varchar(20),
    type varchar(10) not null,
    rackRate money not null,
    primary key (roomNr, BandBname, BandBlocation),
    foreign key (BandBname, BandBlocation)
        references BedAndBreakfast (name, location));
create table ConnectingRoom(roomNr number,
    BandBname varchar(20),
    BandBlocation varchar(20),
    connectingRoomNr number,
    primary key (roomNr, BandBname, BandBlocation, connectingRoomNr),
    foreign key (roomNr, BandBname, BandBlocation) references Room,
    foreign key (connectingRoomNr, BandBname, BandBlocation)
        references Room (roomNr, BandBname, BandBlocation)),
    check (roomNr != connectingRoomNr));
...
```

Fig. 4.11 SQL for Generated Schemas.

is easy to see that they are canonical. We can see, for example, that an ER model instance is canonical by checking the following criteria.

1.   Each attribute is atomic (i.e., not decomposable into component attributes we wish to access in the database).
2.   Each entity set has one or more keys (possibly inherited if the entity set is weak or in an ISA hierarchy), but has no other FDs among attributes with left-hand sides that are not keys.
3.   Each many-many relationship set has one or more keys, but no other FDs among attributes with left-hand sides that are not keys.
4.   Every n-ary relationship set ($n \geq 3$) is fully reduced (i.e., we can't losslessly decompose it into two or more relationship sets).
5.   There are no relationship-set cycles, or if there are cycles, then every path from one entity set to another is non-redundant in the sense that we cannot compute any relationship set as combinations of joins and projections of other relationship sets.

All the earlier ER diagrams in Sections 4.2 and 4.3 are canonical. Thus, all the generated relational schemas are fully normalized. Based on a similar set of criteria for UML, we can see that the UML diagram in Section 4.4 is also canonical and that its generated database schema is thus fully normalized.

To see that not all diagrams are canonical, consider Figure 4.12. This ER diagram violates the conditions listed above for being canonical. Supposing that we wish to access the first name and last name of the registered guest,
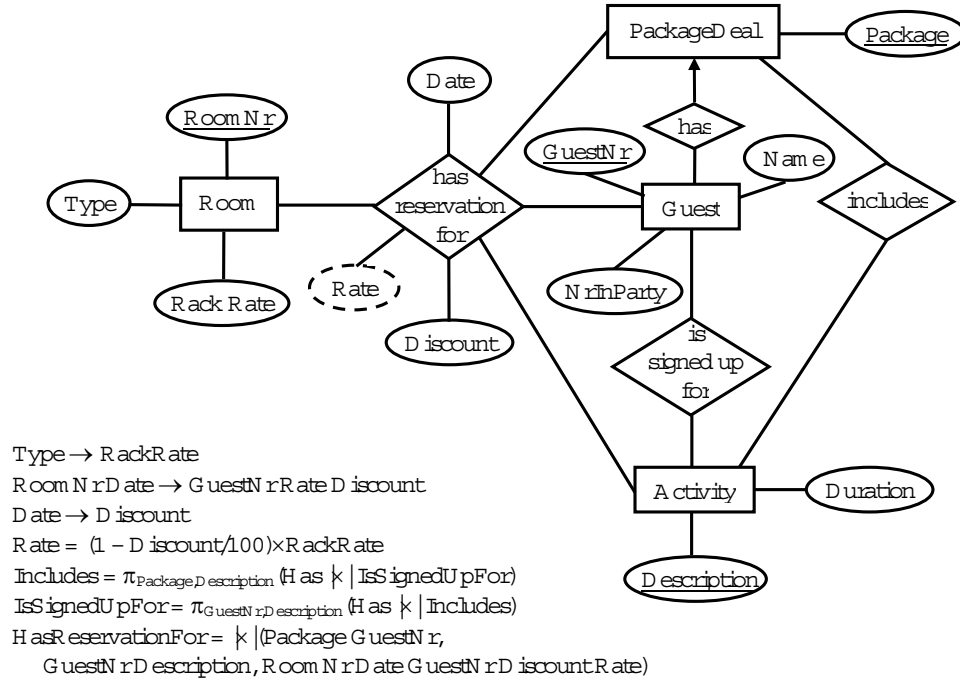
$Type \rightarrow RackRate$

$RoomNrDate \rightarrow GuestNrRate Discount$

$Date \rightarrow Discount$

$Rate = (1 - Discount/100) \times RackRate$

$Includes = \pi_{Package, Description}(Has \ltimes | IsSignedUpFor)$

$IsSignedUpFor = \pi_{GuestNr, Description}(Has \ltimes | Includes)$

$HasReservationFor = \ltimes |(Package GuestNr,$
$\quad GuestNr Description, RoomNrDate GuestNr Discount Rate)$

Fig. 4.12 ER Diagram with Constraints whose Standard Mapping will Yield Normal Form Violations.

Name is not atomic and thus is an example of a violation of Condition #1. The diagram violates Condition #2 because of the FD $Type \rightarrow RackRate$, whose left-hand side is not a key for *Room*. The diagram violates Condition #3 because the left-hand side of the FD $Date \rightarrow Discount$ is not a key for the relationship set *has reservation for*. (We are assuming for this example that the discount amount depends on the date — i.e., whether it is a weekend, weekday, holiday, during the off season, etc.) The diagram violates Condition #4 because we can losslessly decompose *has reservation for* into three relationship sets: one between *Package Deal* and *Guest* (which also happens to be equivalent to the *has* relationship set), one between *Guest* and *Activity* (which also happens to be equivalent to the *is signed up for* relationship set), and one between *Room* and *Guest*. (Perhaps this original quaternary relationship set in Figure 4.12 arose because a designer was told that when guests make reservations they always also sign up for a package deal which includes certain activities.) The diagram violates Condition #5 because of the cycle through the relationship sets *has*, *includes*, and *is signed up for*, in which *includes* and *is signed up for* are computable from the other two relationship sets in the cycle.

As we shall see in Section 4.5.1, if we use standard schema generation procedures, we would generate relational schemas that are not normalized.

We should then normalize them using standard normalization techniques discussed in Chapter 6. We will also see in Section 4.5.2, however, that we can rearrange the diagram so that it has the same semantics but is canonical. If we then use standard schema generation procedures, the resulting relational schemas will be fully normalized. We will thus see that there are two approaches to ensure that generated relational schemas are fully normalized. (1) We can first generate relational schemas and then normalize using standard normalization techniques. (2) We can first canonicalize a conceptual-model diagram and then generate relational schemas.

4.5.1 Map — Then Normalize

Room(RoomNr, Type, RackRate)
Guest(GuestNr, Name, NrInParty, Package)
Activity(Description, Duration)
HasReservationFor(RoomNr, Date, GuestNr, Rate, Discount, Package, Description)
IsSignedUpFor(GuestNr, Description)
Includes(Package, Description)

Fig. 4.13 Generated Relational Schemas — Not Normalized.

Figure 4.13 shows the relational schemas we would generate according to the standard ER mapping rules in Sections 4.2 and 4.3. We must now recognize the normal-form violations and fix them.

- *Guest* is not in 1NF because *Name* is not atomic. We replace *Name* by *FirstName* and *LastName* in *Guest*, which yields *Guest(GuestNr, FirstName, LastName, NrInParty, Package)*.
- *HasReservationFor* is not in 2NF because of *Date* → *Discount*. We decompose *HasReservationFor*, which yields *HasReservationFor(RoomNr, Date, GuestNr, Rate, Package, Description)* and a new relational schema *DateDiscount(Date, Discount)*.
- *Room* is not in 3NF because of *Type* → *RackRate*. We decompose *Room*, which yields *Room(RoomNr, Type)* and a new relational schema *RoomType(Type, RackRate)*.
- The relationship set *HasReservationFor* is not in 4NF/PJNF because of the join dependency ⋈(*Package GuestNr*, *GuestNr Description*, *RoomNr Date GuestNr Rate*). (Note that *Discount* is missing because we have already decomposed *HasReservationFor* in a step above.) We thus decompose *HasReservationFor* according to the join dependency, which yields three relational schemas: *HasReservationFor(RoomNr, Date, GuestNr,*

$Rate$), one whose schema is ($GuestNr, Description$), and one whose schema is ($GuestNr, Package$). Since ($GuestNr, Description$) is the same as $IsSignedUpFor$, we discard it, keeping only $IsSignedUpFor$, and since ($GuestNr, Package$) is embedded in $Guest$, we discard it, keeping only $Guest$.

- Finally, we observe that the schema $IsSignedUpFor$ is redundant because $IsSignedupFor = \pi_{GuestNr,Description}(Has \bowtie Includes)$. Note that although $Includes$ is also redundant because $Includes = \pi_{Package,Description}(Has \bowtie IsSignedUpFor)$, $IsSignedUpFor$ and $Includes$ mutually depend on each other. Thus, we can only remove one of them.

Normalizing the relational schemas as described results in the database schema in Figure 4.14.

Room(<u>RoomNr</u>, Type)
RoomType(<u>Type</u>, RackRate)
Guest(<u>GuestNr</u>, FirstName, LastName, NrInParty, Package)
Activity(<u>Description</u>, Duration)
HasReservationFor(<u>RoomNr</u>, <u>Date</u>, GuestNr, Rate)
DateDiscount(<u>Date</u>, Discount)
Includes(<u>Package</u>, <u>Description</u>)

Fig. 4.14 Normalized Relation Schemas.

4.5.2 Normalize — Then Map

In the alternative approach to normalization, we first make the conceptual-model diagram canonical. We then map it to relational schemas. Because the conceptual-model instance is canonical, the generated relational schemas will be normalized.

To make a conceptual-model instance canonical, we check for and ensure compliance with the basic criteria for characterizing canonical model instances presented earlier. We illustrate this process for ER model instances by making the diagram in Figure 4.12 canonical.

1.   *Non-atomic attributes*. Assuming we wish to have $FirstName$ and $LastName$ for $Guest$, $Name$ is not atomic. We add these attributes, making $Name$ a compound attribute as Figure 4.15 shows.

2.   *FDs whose left-hand sides are not keys*. Recognizing the FD $Type \rightarrow RackRate$ as an FD whose left-hand side is not a key, we create a new entity set, $RoomType$, as Figure 4.15 shows. $Type$ is a key attribute for $RoomType$, and $RackRate$ is a regular attribute. Further, as Figure 4.15
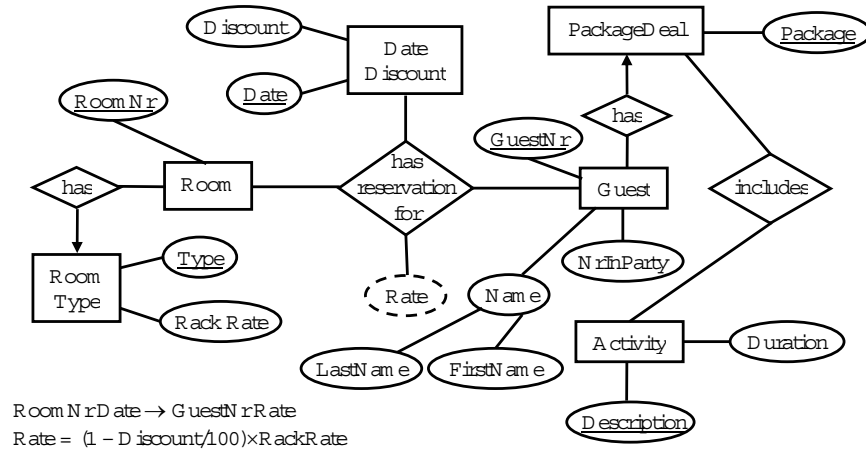
Fig. 4.15 ER Diagram Transformed to Generate Normalized Relation Schemas.

shows, because the FD $Date \rightarrow Discount$ is another FD whose left-hand side is not a key, we create another new entity set, *DateDiscount*. Its attributes are *Date* and *Discount*, with *Date* being a key attribute.

3.   *Reducible n-ary relationship sets.* We can losslessly decompose the relationship set *has reservation for*. After adding the new entity set *DateDiscount* to this relationship set, the relationship set *has reservation for* has become a 5-ary relationship set. We can decompose it losslessly into two binary relationship sets and one ternary relationship set. Since the two new binary relationship sets equate to the existing relationship sets *has* and *is signed up for*, we discard them. This leaves us with the ternary relationship set *has reservation for* in Figure 4.15.

4.   *Reducible cycles.* The cycle of relationship sets from *Guest* to *PackageDeal* to *Activity* and back to *Guest* is a reducible cycle. We can remove either *includes* or *is signed up for* because either is computable from the other two relationship sets. We cannot remove both, however, because we need each one to compute the other. We choose to remove *is signed up for* as Figure 4.15 shows.

Figure 4.15 shows the resulting canonical ER diagram, and Figure 4.14 shows the relational schemas generated from the canonical ER diagram. It should not be a surprise that we obtain the same results whether we first generate relational schemas and then normalize or we first canonicalize the diagram and then generate relational schemas.

An alternative way to do conceptual-model-diagram canonicalization is to (1) transform the conceptual-model diagram to a hypergraph whose nodes are attributes and whose edges are relationship sets,[7] (2) transform the hypergraph to a canonical hypergraph, and (3) map the canonical hypergraph

---

[7] Any ISA hierarchies remain intact without alteration.

to relational schemas. There are several advantages of this approach: (1) Transforming a conceptual-model diagram to a hypergraph lets us explicitly add all connections among attributes. In particular, it lets us add the troublesome FD connections among attributes that are only implicit in diagrams that force attributes to be attached only to entity sets (e.g., ER) or classes (e.g., UML).[2] (2) Since all the constraints needed for normalization appear directly in the diagram, the canonicalization process proceeds entirely with the diagram alone. (3) Finally, this approach leads to more straightforward mapping algorithms, and, as we shall see in Section 4.6, leads to mapping algorithms for object-based schemas and XML schemas.
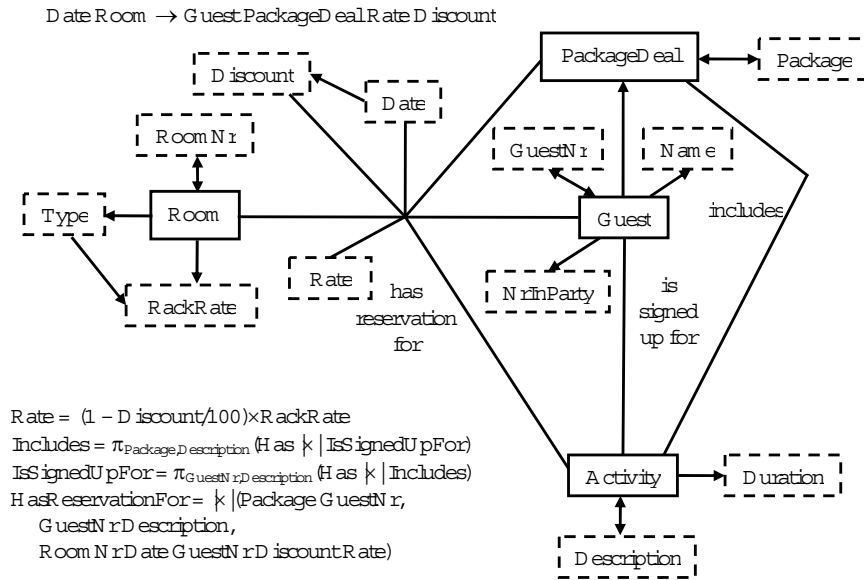

Fig. 4.16 Hypergraph Generated from the ER Diagram in Figure 4.12.

We illustrate this approach beginning with the non-canonical ER diagram in Figure 4.12, which we first convert to the non-canonical hypergraph in Figure 4.16. We convert an ER diagram to a hypergraph as follows.

1.   Make every attribute be a lexical node in the hypergraph. Lexical refers to readable/writable data; lexical nodes represent the data we store in a database. We represent lexical nodes by dashed boxes as Figure 4.16 shows.

[2] Some conceptual models (e.g., ORM [Hal95] and OSM [EKW92]) are directly based on hypergraphs and already include all connections among attributes. These conceptual models need no transformation to hypergraphs. For these hypergraph-based conceptual models, the canonicalization and mappings to relational schemas proceed as we describe here.

2.   Make every entity set be a non-lexical node in the hypergraph. Non-lexical refers to real-world objects that, if represented in a database, would be represented by object identifiers. We represent non-lexical nodes by solid boxes; *Room*, *Guest*, *PackageDeal*, and *Activity* are the non-lexical nodes in Figure 4.16.

3.   Make every relationship set be an edge in the hypergraph. The relationship set *has reservation for*, for example, connects the four non-lexical nodes that were originally entity sets and the three lexical nodes that were originally attributes as Figure 4.16 shows. If there is a functional relationship among all the nodes of the relationship set, we represent this functional relationship set by marking the functionally determined nodes with arrowheads. The *has* relationship set between *Guest* and *PackageDeal* is an example. If there are other functional relationships among some or all the nodes, we add an edge for each one. To keep the diagram from becoming too cluttered, we may visually add these additional edges as FDs (but they are nevertheless edges in the hypergraph). Figure 4.16 includes the FD *Date Room → Guest PackageDeal Rate Discount*, which is a functional edge relating all nodes of the 7-ary relationship set except *Activity*.

4.   Make every connection between an entity set and its attributes be an edge in the hypergraph. For multi-valued attributes, the edge is many-many (non-functional). For compound attributes, the edge is functional from entity set to leaf attribute, one edge for every leaf attribute. (Non-leaf attributes are discarded.) For all other attributes, the edge is functional from entity set to attribute. The functional edge from *Activity* to *Duration* in Figure 4.16 is an example. For singleton key attributes we also add a functional edge from attribute to entity. When we have a functional edge between an entity set and an attribute in both directions, we combine them as a single, bi-directional, one-one edge. In Figure 4.16, *RoomNr* for *Room*, *GuestNr* for *Guest*, *Package* for *PackageDeal*, and *Description* for *Activity* are all examples. For each composite key consisting of n attribute nodes, we add an $(n + 1)$-ary functional edge from the n attribute nodes constituting the composite key to the entity-set node. We also combine the $(n + 1)$-ary functional edge with the n functional edges from the entity set to these attribute nodes to form a single, one-one edge between the entity-set node and the n attribute nodes constituting the composite key.

5.   For every FD among lexical nodes, add an edge. For our example, we add the edges *Type → RackRate* and *Date → Discount* as Figure 4.16 shows.

We next make a non-canonical hypergraph canonical in three main steps.[8]
We illustrate these steps by converting the non-canonical hypergraph in Figure 4.16 to the canonical hypergraph in Figure 4.17.

1.   Decompose all hyperedges that can be losslessly decomposed. In Figure 4.16 we decompose the 7-ary edge along with the edge represented by the FD to a 4-ary functional edge *Room Date → Guest Rate* plus several other edges, all of which eventually turn out to be redundant. Figure 4.17 shows this 4-ary edge, but none of the redundant edges.
2.   Remove all redundant edges and all redundant hyperedge components. In addition to the redundant edges just generated and ignored, we also remove the *Room-RackRate* edge and the *Guest-Activity* edge.
3.   Merge each non-lexical node with its key (any one key, if there is more than one choice). For example, we merge *Room* with its primary key *RoomNr*, which, as Figure 4.16 shows, are in a one-one correspondence. The result of this merge is the lexical node *RoomNr*, which has been merged with *Room* as Figure 4.17 shows. Similarly, we merge *Package* with *PackageDeal*, *GuestNr* with *Guest*, and *Description* with *Activity*.
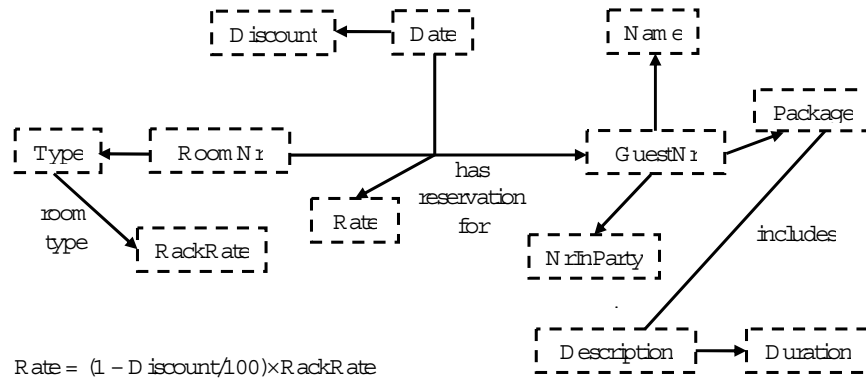

$$Rate = (1 - Discount/100) \times RackRate$$

Fig. 4.17 Canonical Hypergraph.

---

[8] Usually these three main steps are enough. Exceptions arise (1) when the hypergraph is cyclic after redundancies have been removed and (2) when optional participation interferes with our ability to capture all element values. An example of the first would be an additional edge between *Description* and *Duration* where one means the average duration for an activity and the other means the maximum allowable duration. In this case, we need to qualify *Duration* to be, say, *AveDuration* and *MaxDuration*, and generate the relational schema for *Activity* with three attributes, *Duration*, *AveDuration*, and *MaxDuration*. An example of the second would be optional participation for *Name* where we might want to keep all names of guests even if they have no guest number. In this case, we need a separate table for *Name* alone since we cannot capture all names in the *Guest* relational schema, which demands a *GuestNr* for every *Guest*. We can resolve both of these issues by adding roles. See [Emb98] for details.

Once we have a canonical hypergraph, the conversion to a relational database schema is straightforward. With one mapping rule, we can obtain a set of normalized relational schemas:

Every edge becomes a relational schema. If the edge is functional in only one direction, the tail attribute(s) constitute the key. If the edge is functional in both directions (from tail(s) to head(s) and from head(s) to tail(s)), the head attribute(s) constitute a key as well as the tail attribute(s). If the edge is not functional, all the connected attributes constitute the key.

For example, $RoomNr \rightarrow Type$ is a functional edge whose generated relational schema is ($\underline{RoomNr}$, $Type$), and $RoomNr\ Date \rightarrow GuestNr\ Rate$ is another functional edge whose generated relational schema is ($\underline{RoomNr, Date}$, $GuestNr$, $Rate$). The edge between $Package$ and $Description$ is a non-functional, and thus its generated relational schema is ($\underline{Package, Description}$). We have no edge that is functional in both directions. To illustrate a bidirectional edge, suppose, as in Figure 4.1, that in addition to room numbers, rooms in a bed and breakfast establishment also have identifying names. We would then have another attribute $RoomName$ in a one-to-one correspondence with $RoomNr$. In this case, we would generate the relational schema ($\underline{RoomNr}$, $\underline{RoomName}$) in which both $RoomNr$ is a key and $RoomName$ is a key.

Although this single rule generates relational schemas that are all normalized, it fragments the database into more relational schemas than necessary. Thus, to finalize the relational schemas for the database, we merge as many as we can without violating any normal form using the following simple rule:

Merge schemas that have a common key.

For example, we would merge ($\underline{GuestNr}$, $Name$), ($\underline{GuestNr}$, $Package$), and ($\underline{GuestNr}$, $NrInParty$), because they all have the key $GuestNr$. We also add a name as we merge so that we have a standard relational schema. We would thus obtain $Guest$($\underline{GuestNr}$, $Name$, $Package$, $NrInParty$). Note that we would *not* merge ($\underline{RoomNr}$, $Type$) and ($\underline{RoomNr, Date}$, $GuestNr$, $Rate$) because they do not have a common key — $\overline{RoomNr}$ is not the same as the composite key $RoomNr\ Date$. Forming the relational schemas from Figure 4.17 and then merging those that have a common key results in the relational schemas in Figure 4.14, except that $Name$ appears in place of $FirstName$ and $LastName$ since we started with $Name$ rather than $FirstName$ and $LastName$ in Figure 4.12.

Note that we have said nothing about ISA hierarchies. This is because there is no change to the way we map ISA hierarchies to relational schemas. There is a change, however, when we convert non-canonical hypergraphs to canonical hypergraphs. When we merge a non-lexical node with its key, we propagate this merge all the way down the hierarchy. Thus, for example, when we convert the ER diagram in Figure 4.5 to a canonical hypergraph,

every node in the ISA hierarchy rooted at *Guest* becomes a lexical node with the name $GuestNr$.[9] If we choose to have a relational schema for every node in the ISA hierarchy, we simply do not merge relational schemas in the ISA hierarchy that share a common key. If we choose to have a relational schema only for the root of the ISA hierarchy, we merge in the usual way. If we choose to have relational schemas only for the leaves of the ISA hierarchy, we merge along every path from the root to each of the leaves.

4.6 Mappings for Object-Based and XML Databases

In this section, we show how to generate generic hierarchical structures, called *scheme trees*, from a canonical hypergraph. Since scheme trees are generic hierarchical structures, it is straightforward to map scheme trees to database schemas that support hierarchical data. As an example, we show how to map scheme trees to object-relational database schema instances and XML schema instances. Since object-oriented databases are similar to object-relational databases, our example for object-relational databases serves for object-oriented databases as well.

The central idea of generating scheme trees is to ensure that each instance of a hyperedge in a canonical hypergraph only appears once in a scheme-tree instance. To do so, our scheme trees observe the many-one and one-one constraints in the hypergraph. We capture the essence of the idea in the following algorithm.[2]

> While there is an unmarked hyperedge, do:
>> Select a subset V of vertices in an unmarked hyperedge E
>>> to be the root node of a new scheme tree T.
>> While we can add an unmarked hyperedge E to T, do:
>>> (We can add an unmarked edge E if the following conditions hold: E must have a nonempty intersection S of vertices with T. A node N must exist in T such that the set S is contained in the union of the nodes above or equal to N and S functionally determines the union of the nodes above or equal to N.)
>> Add E to T as follows:
>>> If S functionally determines $E - S$
>>>> Add the vertices in $E - S$ to the node N.
>>> Else

---

[9] To keep the various nodes straight, we should add comments to the diagram. Except that these comments help us keep the nodes conceptually straight and may help us choose names for relational schemas, we can ignore these comments.

[2] The algorithm is a simplified version of the heuristic n-ary algorithm in [ME06], which generates scheme trees from a canonical hypergraph.

> Make a node consisting of the vertices in $E - S$ and
> add it to the tree as a child node of N.
Mark E as used.

If we select $Package$ (a subset of the vertices in the edge $GuestNr \rightarrow$
$Package$) as the root node, this algorithm generates the scheme tree in Figure 4.18a. Having selected $\{Package\}$ as the root for T and $E = \{Package,$
$GuestNr\}$ as the edge we are considering, we see that all conditions of the
while loop trivially hold (as they always do for the initial selection of a root
and an edge). Since $S = \{Package\}$ does *not* functionally determine $E - S$
$= \{GuestNr\}$ (it is actually the other way around), we add $\{GuestNr\}$ as
a child node of the node $\{Package\}$. Continuing, we next consider the edge
$GuestNr \rightarrow Name$. Here, the conditions of the while loop also hold with S
$= \{GuestNr\}$, $N = \{GuestNr\}$, and $GuestNr \rightarrow GuestNr\ Package$. Since
S functionally determines $E - S = \{GuestNr,\ Name\} - \{GuestNr\} =$
$\{Name\}$, we add $Name$ to the node containing $GuestNr$. Similarly, we add
$NrInParty$ to this node, thus completing the second node in Figure 4.18a.
We next consider the edge $RoomNr\ Date \rightarrow GuestNr\ Rate$. The intersection
S with T is $\{GuestNr\}$, which is contained in the node N we just created.
Further, $GuestNr \rightarrow GuestNr\ Name\ NrInParty\ Package$ so that it functionally determines the union of N and the node above N (the root). Since
S does not functionally determine $E - S$ (i.e., $\{GuestNr\}$ does not functionally determine $\{RoomNr,\ Date,\ Rate\}$), we make a new node consisting of
$\{RoomNr,\ Date,\ Rate\}$ and place it below N as Figure 4.18a shows. Continuing, we next consider the nonfunctional edge $\{Package,\ Description\}$.
The nonempty intersection S with the scheme tree T we have built so far is
$\{Package\}$. Since $\{Description\}$ does not functionally determine $\{Package\}$,
we form a new node consisting only of $\{Description\}$ and add it as another
child node of $\{Package\}$, resulting in the scheme tree T in Figure 4.18a. Of
the remaining four edges, $Type \rightarrow RackRate$ has an empty intersection with
T, and the rest ($Date \rightarrow Discount$, $RoomNr \rightarrow Type$, and $Description \rightarrow$
$Duration$) do not satisfy the functional-dependency condition of the while
loop for the scheme tree in Figure 4.18a.

Figure 4.18b shows the textual representation for the scheme tree in Figure 4.18a. In the textual representation, each node appears within parentheses with a star to indicate repetition; parenthesized nodes appear in line
according to their nested position in the scheme tree.

Choosing the best starting place to run the algorithm depends on the semantics of the application. In our example, starting with $Package$ is probably
not best. The main point of the application is to rent rooms to guests, not
about the activity packages they may choose. Thus, most of the processing
is likely to be about looking up guests. Hence, a likely better starting place
for our scheme-tree algorithm is to choose $GuestNr$ as the root. When we
start with $GuestNr$ as the root of the first tree and run the algorithm repeatedly until all edges are in some scheme tree, we obtain the scheme trees
in Figure 4.19.
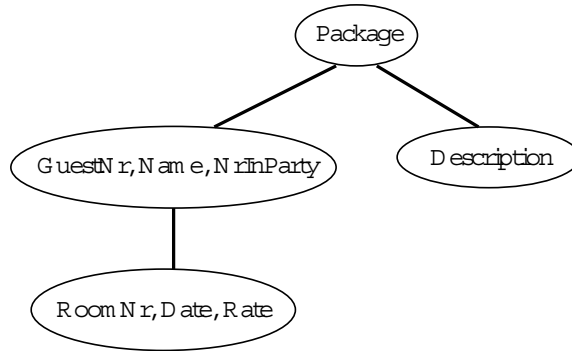
(a) Tree Representation.

  (Package, (GuestNr, Name, NrInParty, (RoomNr, Date, Rate)*)*, (Description)*)*

(b) Textual Representation.

Fig. 4.18 Scheme-Tree Representations.

In Figure 4.19 we have marked keys in the usual way. In our example, $GuestNr$, $Date$, $Type$, and $Description$ values must all be unique in the usual way we expect key values to be unique. Since $RoomNr\ Date \rightarrow GuestNr$, $RoomNr$-$Date$ value pairs must be unique in the nested relation in which they appear. Similarly, since $RoomNr \rightarrow Type$, $RoomNr$ values must also be unique in the nested relation in which they appear.

  (GuestNr, Name, NrInParty, Package, (RoomNr, Date, Rate)*)*
  (Date, Discount)*
  (Type, RackRate, (RoomNr)*)*
  (Description, Duration, (Package)*)*

Fig. 4.19 Generated Scheme-Tree Forest.

Figure 4.20 shows an object-relational database schema instance derived from the first scheme tree in Figure 4.19. The main idea of this derivation is to scan a scheme tree bottom up and generate object types and collection types as we go up the scheme tree. Specifically, consider a leaf node v whose parent is u. We generate an object type t_v for v such that t_v has all the attributes in v. Then, we create a collection type of t_v. Afterwards, we generate an object type t_u for u such that t_u has all the attributes in u and a field whose type is the collection type of t_v. We then continue this process up to the root node of the scheme tree. Thus, for example, for the first scheme tree in Figure 4.19, we create an object type called $reservation$ that includes three fields: $roomNr$, $reservationDate$, and $rate$. We then create a collection type called $collectionOfReservation$, which is a variable length array that stores

reservation objects. Finally, we create an object type called *guest* for the root node. Note that there is a field called *reservations* in *guest* whose type is *collectionOfReservation*. To store *guest* objects, we create *tableOfGuest*— a table of this type. Unfortunately, declared key constraints are not typically provided by object-relational databases. Thus, as for any constraint we wish to enforce that is not directly supported by the database, we must provide our own code to check and enforce it.

```
create type reservation as object (
     roomNr          integer,
     reservationDate date,
     rate            number,
     static function createReservation(roomNo integer, resDate date, rate number)
          return reservation,
     static function createReservation(roomNo integer, resDate date) return reservation);
create type body reservation is
     static function createReservation(roomNo integer, resDate date, rate number)
               return reservation is
          begin
               return reservation(roomNo, resDate, rate);
          end;
     static function createReservation(roomNo integer, resDate date) return reservation is
          begin
               return reservation(roomNo, resDate,
                         (1-discount.getDiscountRate(resDate)/100)
                         * roomType.getRackRate(roomNo));
          end;
end;
create type collectionOfReservation as
     varray(200) of reservation;
create type guest as object (
     guestNr        integer,
     name           varchar2(30),
     nrInParty      integer,
     package        varchar2(40),
     reservations   collectionOfReservation);
create table tableOfGuest of guest;
```

Fig. 4.20 Derived Object-Relational Database Schema Instance.

Figure 4.21 shows the first and last parts of a derived XML schema instance for the generated scheme-tree forest in Figure 4.19. Several derivations are possible; Figure 4.21 shows one derived as follows. For each nesting of a scheme tree we provide two names—one for describing the group and one describing an individual in the group. Thus, for the scheme tree (Type, RackRate, (RoomNr)*)* we introduce the name *RoomTypes* for the group, *RoomType* for the individuals in the group, *Rooms* for the nested group, and *Room* for the individuals in the nested group. We nest these names ap-

propriately as Figure 4.21 shows. We then put the attributes in their proper place — we nest $Type$ and $RackRate$ under $RoomType$ and $RoomNr$ under $Room$. The type for $Type$ is $xs{:}ID$ making it unique throughout the document. Since $Type$ appears nowhere else in the scheme-tree forest in Figure 4.19, this simple declaration is sufficient. For $RoomNr$, which does appear in one other scheme tree, we scope the extent of uniqueness to be within $RoomTypes$ and make a key declaration as Figure 4.21 shows. Further, when an attribute appears in more than one scheme tree, we use ref to reference a single declaration for its type. $RoomNr$ is an example; its type declaration along with the type declarations for $Date$ and $Package$ which also appear in more than one scheme tree appear globally at the end of the XML schema instance in Figure 4.21. In the first part of the XML schema instance in Figure 4.21, we have declarations for the roots of each of the scheme trees, $Guests$, $Activities$, $DateDiscounts$, and $RoomTypes$ all under the ultimate root, $Document$. We allow each of them to be optional except $RoomTypes$. (Although there may be no guests, activities, or date-discounts, if there are no rooms, there is no bed and breakfast establishment.)

4.7 Additional Readings

From the beginning, mappings from conceptual-model instances to database schemas have been an important part of the conceptual-modeling literature. Peter Chen's seminal article on the ER model includes an extensive discussion of mapping the ER model to various database models [Che76]. Ever since the appearance of this seminal article, discussions of mapping conceptual-model instances to database schemas have continued. Notable, along the way, are an *ACM Computing Surveys* article [TYF86] and a foundational book on conceptual database design [BCN92]. Most current database text books (e.g., [UW02, EN04, SKS06]) contain chapters on mapping conceptual models to database schemas.

Normalization concerns have also always been a part of mapping conceptual-model instances to database schemas. Chen's seminal article [Che76] addressed normalization, and his mappings did yield relations in 3NF under his assumptions. Along the way other researchers have added insights about normalization. Researchers have suggested both the map-then-normalize approach [TYF86] and the normalize-then-map approach [CNC81, Lin85]. In [BCN92] the authors take the point of view that ER diagrams are not of good quality unless they are canonical, and they talk about canonicalizing ER diagrams as one way to help create high-quality diagrams. The hypergraph approach to normalization appeared initially in [EL89]; full details appeared later in [Emb98].

Only recently have articles appeared that describe the process of mapping conceptual models for object-based, object-oriented, and XML databases.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="Document">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Guests" minOccurs="0"/>
        <xs:element ref="Activities" minOccurs="0"/>
        <xs:element ref="DateDiscounts" minOccurs="0"/>
        <xs:element ref="RoomTypes"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
...
  <xs:element name="RoomTypes">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="RoomType" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Rooms">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Room" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute ref="RoomNr"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="Type" type="xs:ID"/>
            <xs:attribute name="RackRate" type="xs:double"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="RoomNrKey">
      <xs:selector xpath="./RoomType/Rooms/Room"/>
      <xs:field xpath="@RoomNr"/>
    </xs:key>
  </xs:element>
  <xs:attribute name="RoomNr" type="xs:integer"/>
  <xs:attribute name="Date" type="xs:date"/>
  <xs:attribute name="Package" type="xs:string"/>
</xs:schema>
```

Fig. 4.21 Derived XML Schema Instance.

Some initial thoughts appeared in [Emb98, EM01]. Proofs that generated scheme trees have no redundancy appeared later [ME06].

References

[BCN92]   C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1992.

[Che76]   P.P. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[CNC81]   I. Chung, F. Nakamura, and P.P. Chen. A decomposition of relations using the entity-relationship approach. In *Proceedings of the 2nd International Conference on Entity-Relationship Approach to Information Modeling and Analysis (ER'81)*, pages 149–171, Washington D.C., October 1981.

[EKW92]   D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-oriented Systems Analysis: A Model-Driven Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[EL89]   D.W. Embley and T.W. Ling. Synergistic database design with an extended entity-relationship model. In *Proceedings of the 8th International Conference on Entity-Relationship Approach (ER'89)*, pages 118–135, Toronto, Canada, October 1989.

[EM01]   D.W. Embley and W.Y. Mok. Developing XML documents with guaranteed 'good' properties. In *Proceedings of the 20th International Conference on Conceptual Modeling (ER2001)*, pages 426–441, Yokohama, Japan, November 2001.

[Emb98]   D.W. Embley. *Object Database Development: Concepts and Principles*. Addison-Wesley, Reading, Massachusetts, 1998.

[EN04]   R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, Boston, Massachusetts, fourth edition edition, 2004.

[Hal95]   T. Halpin. *Conceptual Schema & Relational Database Design*. Prentice Hall of Australia Pty. Ltd., Sydney, Australia, second edition, 1995.

[Lin85]   T.W. Ling. A normal form for entity-relationship diagrams. In *Proceedings of the 4th International Conference on Entity-Relationship Approach (ER'85)*, pages 24–35, Chicago, Illinois, October 1985.

[ME06]   W.Y. Mok and D.W. Embley. Generating compact redundancy-free XML documents from concptual-model hypergraphs. *IEEE Transactions on Knowledge and Data Engineering*, 18(8):1082–1096, August 2006.

[SKS06]   A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, New York, New York, fifth edition, 2006.

[TYF86]   T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[UW02]   J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, Upper Saddle River, New Jersey, second edition edition, 2002.