Dissertation zur Erlangung des akademischen Grades
Dr. rer. soc. oec.
im Doktoratsstudium der Sozial- und Wirtschaftswissenschaften

Multi-Level Modeling with M-Objects and M-Relationships

(Online Version[1])

Angefertigt am

Institut für Wirtschaftsinformatik -
Data & Knowledge Engineering
Johannes Kepler Universität Linz

Eingereicht von

Mag. Bernd Neumayr

Betreut von

o. Univ.-Prof. Dipl.-Ing. Dr. Michael Schrefl
a. Univ.-Prof. Mag. Dr. Werner Retschitzegger

unter Mitwirkung von

Prof. Dr. Bernhard Thalheim

Linz, April 2010

Johannes Kepler Universität Linz, Altenberger Straße 69, 4040 Linz, Austria

[1]Some typos have been corrected in the online version.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht verwendet und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen deutlich als solche kenntlich gemacht habe.

Linz, April 2010

(Bernd Neumayr)

Danksagung

Zuerst danke ich Michael Schrefl für seinen unerschöpflichen Ideenreichtum, unsere endlosen Diskussionen, für sein Fachwissen, für seine Geduld und Beharrlichkeit und vor allem dafür, dass er mich in meiner wissenschaftlichen Entwicklung in allen Belangen unterstützt hat.

Weiters danke ich Bernhard Thalheim für seine Gastfreundschaft in Kiel und für die Rückmeldungen und Anregungen zu meiner Arbeit. Werner Retschitzegger danke ich für die Übernahme der Zweitbetreuung dieser Arbeit.

Großer Dank gilt meinen KollegInnen Christian Eichinger, Margit Brandl, Stefan Berger, Michael Karlinger, Katharina Grün, Christoph Schütz, Michael Huemer, Klaus Ettmayer und Georg Nitsche, die in den vergangenen Jahren für ein stets positives und angenehmes Arbeitsklima am Institut für Data & Knowledge Engineering gesorgt haben. Großer Dank gilt vor allem Stefan Berger für die Hilfsbereitschaft beim Korrekturlesen. Weiters danke ich meinen Diplomanden Christoph Schütz, Michael Huemer, Klaus Ettmayer, Karl Rieger, Alois Diwold, Joachim Huber und Christian Horner für Ihre Arbeit rund um den M-Object und M-Relationship-Ansatz.

Vor allem danke ich meinen Eltern, die mich immer in allen Belangen unterstützt und gefördert haben, sowie meinem Bruder für seine Geduld. Meinen Freunden danke ich für den nötigen Ausgleich. Der größte Dank gilt natürlich Andrea, die in den vergangenen Jahren auf viel gemeinsame Zeit verzichten musste und mich trotzdem immer liebevoll unterstützt hat.

Kurzfassung

Informationssysteme werden für immer größere und organisationsübergreifende Anwendungsbereiche entwickelt. Während die einzelnen Teilbereiche eines solchen Anwendungsbereichs einerseits gleichartig strukturiert sind, gibt es zwischen einzelnen Teilbereichen und Organisationen auch große Unterschiede, sowohl in Bezug auf den Anwendungsbereich selbst, als auch in Bezug auf die Informationsbedürfnisse der Anwender. Eine Überbetonung der Homogenität von Teilbereichen führt zu unflexiblen Systemen, die nur schlecht die Realität abbilden können. Andererseits führt eine Überbetonung der Heterogenität zu Insellösungen, die nicht mehr gemeinsam genutzt werden können.

Konzeptuelle Modellierung versucht, Anwendungsbereiche möglichst der Realität entsprechend und unabhängig von konkreten Implementierungen in Modellen abzubilden. Dabei bedient sie sich der klassischen Abstraktionsprinzipien —Klassifizierung, Generalisierung, Aggregation— um einzelne Objekte und deren Beziehungen zu abstrakteren Objekten und Beziehungen zusammenzufassen. Um die Homogenität und gleichzeitige Heterogenität von großen Anwendungsbereichen abzubilden, müssen diese Prinzipien kombiniert werden. Dies führt zu einer mehrfachen Abbildung einzelner Objekte in unterschiedlichen Abstraktionshierarchien, was wiederum zu schwer verständlichen und schwer wartbaren Modellen und Informationssystemen führt.

Mehr-Ebenen-Objekte (M-Objects) und Mehr-Ebenen-Beziehungen (M-Relationships) ermöglichen die kompakte Abbildung von Objekten und Beziehungen auf höheren Abstraktionsebenen gemeinsam mit der zusammenfassenden Beschreibung von konkreteren Objekten und Beziehungen auf verschiedenen darunter liegenden Abstraktionsebenen. Durch die Kombination und gleichzeitige Differenzierung von Klassifizierungs-, Generalisierungs- und Aggregations-Hierarchien in einer Konkretisierungshierarchie, ermöglichen M-Objects und M-Relationships eine schrittweise Modellierung von großen Anwendungsbereichen. Einerseits wird die Gleichartigkeit aller Teilbereiche auf unterschiedlichen Abstraktionsebenen sichergestellt und andererseits können Unterschiede in Bezug auf Anwendungsbereich und Informationsbedürfnis abgebildet werden.

In dieser Arbeit werden M-Objects und M-Relationships und deren konsistente Konkretisierung formal definiert und an Beispielen erklärt. Der M-Object und M-Relationship Ansatz wird durch formal definierte Operationen zum Anlegen, Manipulieren, und Abfragen von Mehr-Ebenen-Modellen komplettiert.

Die Anwendungsmöglichkeiten von M-Objects und M-Relationships sind vielfältig. In dieser Arbeit beschränken wir uns auf Ontologiemodellierung im Web und auf Data Warehousing. Im Data Warehousing können Konkretisierungshierarchien von M-Objects eingesetzt werden, um Heterogenitäten in Dimensionen abzubilden und darüber hinaus, um Heterogenitäten in den zu analysierenden Fakten zu ermöglichen. Diese Anwendung von M-Objects und M-Relationships wurde bereits prototypisch als Erweiterungspaket für Oracle DB implementiert.

Abstract

Information Systems maintain a representation of the state of a domain in order to provide information about that domain. They are often employed in a cross-organizational setting and have to represent different sub-domains. These sub-domains and different organizations are homogeneous as well as heterogeneous with regard to the domain itself, as well as with regard to information needs. Emphasis on homogeneity results in inflexible information systems. On the other hand, emphasis on heterogeneity leads to 'information islands', which are difficult to integrate.

As a prerequisite for information system development, conceptual modeling provides a high-level representation of a domain, independent of a specific implementation. For this purpose, conceptual modeling makes use of the classical abstraction principles: classification, generalization, and aggregation. In order to represent complex domains, being both homogeneous as well as heterogeneous, these principles need to be combined. Using traditional modeling techniques, this results in multiple representation of single objects in different abstraction hierarchies. This, in turn, results in models and information systems that are hard to comprehend and to maintain.

Multi-Level Objects (M-Objects) and Multi-Level Relationships (M-Relationships) allow to represent objects and relationships at higher abstraction levels, which encapsulate descriptions of sets of objects and relationships at multiple lower levels of abstraction. A single concretization hierarchy of m-objects (or m-relationships) combines and differentiates multiple classification-, generalization-, and aggregation-hierarchies. In this manner, m-objects and m-relationships facilitate stepwise modeling of complex domains at multiple abstraction levels. Concretization hierarchies guarantee a certain level of homogeneity between sub-hierarchies and, on the other hand, they also allow for heterogeneities between different sub-hierarchies.

In this thesis, we provide formal definitions and examples of m-objects and m-relationships. The m-object and m-relationship approach is complemented by formally defined operations for creating, manipulating and querying multi-level models.

Applications of m-objects and m-relationships are manifold, we especially discuss web ontologies and data warehousing. Applied to ontology engineering with OWL DL, m-objects and m-relationships provide a more powerful alternative to current multi-level modeling techniques, such as punning. In data warehousing, m-objects and m-relationships provide a solution to the well discussed problem of modeling heterogeneous dimension hierarchies and also allow heterogeneities within cubes. This solution has been implemented as an extension package for Oracle DB.

Contents

Chapter 1

Introduction

Contents

In this chapter we first briefly discuss some problems and challenges of conceptual modeling of information systems, and sketch our vision of how to meet these challenges. This is the subject of Section 1.1. In Section 1.2, we identify and exemplify a specific problem that lies in the center of our vision and that is to be solved in this thesis. In Section 1.3, we discuss our research approach with regard to frameworks and guidelines for design science. We then, in Section 1.5, give an overview of our solution, and, in Section 1.6, of its contributions. Finally, in Section 1.7, we outline the overall structure of this thesis.

1.1 The Big Picture

Information systems play a crucial role in industry and public administration. An information system, as discussed herein, is a 'designed system that collects, stores, processes, and distributes information about the state of a domain' (universe of discourse, subject domain) [94]. Its main functions are '*Memory*: to maintain a representation of the state of a domain', '*Informative*: to provide information about the state of a domain' – we call this business intelligence – and '*Active*: to perform actions that change the state of a domain' [94].

The representation of the state of a domain is based on how we perceive this domain. Typically, we make the *ontological assumption* that the world, as we perceive it, consists of entities (individuals, objects), e.g, *Jim* and *Mary*, and relationships between them, e.g., *Jim loves Mary*. In our mind, we classify these entities and relationships to *entity types* (classes), e.g., *Person*, and *relationship types*, e.g., *love*. These *concepts* form our *conceptualization* of a domain and guide the whole life-cycle of an information system. We use the notion of *concept* to refer to something that is an abstraction of something more concrete (an entity, a relationship, a property of an entity or relationship, or a more concrete concept). To avoid misunderstandings and facilitate communication, these concepts should not only be present in our minds but should also be formally represented in a *conceptual schema of a domain (ontology).* Based on an ontology one can define the *conceptual schema of an information system*, which defines, disregarding implementation issues, which information should be stored (*structural schema*), together with changes, actions, and processes represented in the information system (*behavioral schema*). The conceptual schema of an information system then serves as blueprint for its implementation. Ideally, the conceptual schema of an information system already serves as its implementation or can automatically be transformed into an implementation.

Conceptual modeling is the whole process from conceptualization to (implementable) conceptual schema of an information system. It is a matured discipline in academia and industry, within information systems engineering, data and knowledge engineering, and software engineering. Recently, in software engineering, the goal of using conceptual schemas as core of implementations has received increased attention under the name of *model-driven engineering* (see [61, 101] for introductions to model-driven engineering).

While conceptual modeling is well established and matured for relatively small subject domains, modeling of very large domains was widely neglected in academia. In industry, however, it is necessary to represent very large subject domains, leading to database schemas with 10000 to 100000 tables (like in Enterprise Resource Planning). Having no matured conceptual modeling tools but rather ad-hoc solutions for dealing with such huge schemas, they are immensely hard to comprehend, to maintain, to adapt and to integrate.

Conceptual modeling in the large, as envisioned here, tries to tackle at least the following areas: constructing, comprehending, and maintaining very large conceptual schemas, providing different views on data, integration and adaption of conceptual schemas, and blurring between schema and instance level. The term 'conceptual modeling in the large' was coined by Olivé [93] as a research direction that deals with very large conceptual schemas and was the topic of a workshop [56] at the International Conference on Conceptual Modeling (ER) 2009.

To illustrate the problem of comprehending and maintaining *very large schemas*, consider a typical mid-sized company. In such a company it is, first, very difficult to find employees who have a sufficiently broad conceptualization of their company (information structure, processes, business rules) to guide information systems engineering and, second, it is difficult to elicit these conceptualizations in terms of conceptual schemas, since only a few employees are able to use conceptual modeling or ontology languages in order to represent their conceptualizations. Thus, companies have to rely on best practices implemented in software solutions (such as SAP ERP). Furthermore, a whole industry of business and technical consultants is necessary to explain the conceptual schemas implemented in these software solutions and to adapt, more or less, these best practices to the needs of organizations, and to maintain them in order to reflect changes in law and in the business environment.

Furthermore, different users or different parts of an organization have varying information needs concerning the same or similar real-world entities. Thus, an information system has to provide *different views on data* at different abstraction levels. This is well acknowledged in data warehousing, where aggregated data is provided at multiple levels of granularity.

The great success of large, best-practice software solutions in Enterprise Resource Planning and similar domains is mainly due to *global, homogeneous schemas* that facilitate integration of all parts of an organization and allow minor customizations to specific needs. A global, homogeneous schema facilitates to write queries, methods or processes that cover the whole organization.

As organizations grow and merge, maintaining a homogeneous schema is very expensive. This is especially true when organizations are integrated across different countries with different laws and cultures or across different industrial sectors with different products and business processes. In order to support their non-uniform business processes, different branches of organizations often retain their local information systems with *local schemas.* These 'information islands', however, makes business intelligence very expensive. To facilitate at least some advantages of a global homogeneous schema within a world of local schemas, enterprise data integration and business process integration have become large sectors of the software and consulting industry.

State-of-the-art, both in academia and industry, does not provide appropriate modeling techniques for coping with the heterogeneity in organizations. Such modeling techniques have to retain the advantages of homogeneous schemas and at the same time have to be flexible enough to support specific information needs in specific contexts, e.g., for different countries, industrial sectors, product categories, companies, departments, and so forth. Such techniques go beyond classical customization and specialization.

Another property of very large schemas is the *blurring between instance and schema level.* While traditional modeling techniques maintain a strong distinction between instance level and schema level, the borderline between classes and instances is often application dependent. Instance data in one application might serve as schema data in another application. Since a conceptual schema often has to serve different applications, maintaining such a distinction is disadvantageous. Two notable examples are, in the context of Enterprise Resource Planning, product types vs. product items and routings (workflows that define, for each product type, the necessary production steps) vs. production orders (workflow instances according to routings, one for each product item). Thus, conceptual modeling in the large always has to address aspects of *metamodeling.*

The *vision* that guides our research is a concise formalism that allows to comprehensibly represent and efficiently implement very large conceptual schemas, that are easy to maintain and that provide for different views at different levels of abstraction. Furthermore such a formalism has to provide for the adaption of large schemas to specific information needs of (parts of) organizations and for the integration of different but similar schemas.

For all these needs, (orthogonal) abstraction hierarchies play a crucial role. Schemas may be specialized along such hierarchies at different levels by different users. Data may be aggregated along such hierarchies at different levels of abstraction to fit the information needs of different users. Like traditional schemas define which data is stored for which entity types, *aggregation* *schemas* define how and where aggregated data is provided, and *higher-level schemas* define how and where schemas may be specialized.

1.2 Specific Problem

Finding an overall solution for the vision discussed above is beyond the scope of one thesis. Thus we will focus only on the core constructs of conceptual modeling, namely entities/entity types and their attributes and relationships and how they can be represented in so-called hetero-homogeneous hierarchies at multiple levels of abstraction. In the remainder of this section we will outline and exemplify this problem and describe some comparison criteria to evaluate the fitness of multi-level modeling techniques for representing hetero-homogeneous hierarchies.

Objects are frequently organized in abstraction hierarchies consisting of multiple levels. Examples are product hierarchies, dimension hierarchies in data warehouses, and taxonomies in general. For example, a *product catalog* may consist of three levels, *category*, *model*, and *physical entity*. Sample instances of these levels could be *car*, *porsche911CarreraS*, and *myPorsche911CarreraS*, respectively.

Conceptual modeling of such hierarchies is straightforward if objects at each level are *uniform*, i.e. have the same structure. As information systems grow in size or previously independent systems are integrated across related domains, the need to handle levels of similar, yet non-uniform objects arises. Objects at the same level in different subhierarchies may differ from each other. For example, *product model*s have a *listprice*, but *car model*s (i.e., objects at level *model* belonging to *car*, an object at level *category*) have an additional attribute *maxSpeed*.

Heterogeneity may go beyond having differently structured objects at the same level. Sub-hierarchies beneath two objects at the same level may differ from each other in that they introduce different additional levels. For example, our product catalog may contain a product category *car*, which is described by an additional level *brand* (with objects like *porsche911*) between levels *model* and *physical entity*.

Such hierarchies with differently structured sub-hierarchies can also be referred to by the oxymoron term *hetero-homogeneous hierarchies*. A hetero-homogeneous hierarchy is a hierarchy that is (1) homogeneous in regard to a minimal common schema shared by all sub-hierarchies, and (2) heterogeneous in regard to the specialized schemas of sub-hierarchies.

Hetero-homogeneous hierarchies of objects at multiple levels of abstraction combine the three classical abstraction principles, namely classification/instantiation, generalization/specialization, and whole/part. When modeled with a conceptual modeling language, like the UML, that supports these abstraction principles, but not all of them in combination, entities and concepts have to be represented multiple times in different abstraction hierarchies (see Figure 1.1).

While, from a static perspective, the UML representation of hetero-homogeneous hierarchies does not involve unnecessary or *accidental complexity*, the accidental complexity becomes apparent when introducing new domain concepts. Introducing a single domain concept (such as new product category *car*) requires – besides introducing an instance *Car:CarCategory* – the addition of multiple classes, (in our case *CarCategory*, *CarModel*, *CarPhysicalEntity*) as well as associated aggregation, generalization, and classification relationships. Moreover, similar classes need to be added for each new product category entered into our product catalog. The redundancy and fragmentation caused by following such a modeling approach complicates maintenance and extension of the schema.

Several modeling techniques have been proposed in recent years to reduce accidental complexity in modeling domain entities and concepts at multiple levels of abstraction. The prevailing techniques simplify multi-level modeling by combining at least two of the classical abstraction principles:

- *Materialization* [30, 98, 23], at first sight, is akin to whole/part, but also provides a powerful *attribute propagation* mechanism that introduces aspects of classification/instantiation and generalization/specialization.
- *Powertypes* [91, 44, 32] allow to structure generalization/specialization hierarchies by classifying subclasses of a particular class to a powertype associated with that class.
- *Ontological metamodeling with potency-based deep instantiation* [7] starts with a multi-level classification hierarchy and allows to factor out common properties of instances of multiple classes to their metaclass, thus extending multi-level classification with some aspects of generalization/specialization.

However none of these multi-level techniques fully supports modeling hetero-homogeneous hierarchies. Chapter 3 discusses in detail their suitability for multi-level modeling regarding hetero-homogeneous hierarchies.

Furthermore, in order to provide a feasible multi-level modeling approach, not only objects, but also relationships need to be modeled at multiple levels of abstraction. When we started our research, none of the existing multi-level modeling approaches provided for that. In the meantime, parallel to our research, Atkinson et al. [42, 6] also identified this necessity and analyzed how relationships can be modeled at multiple levels, however only in the restricted context of ontological metamodeling with potency-based deep instantiation.

1.2.1 Running Example

In this section we introduce our running example that is used throughout the thesis.

Example 1.1 (Sample Problem)**.** The product catalog of an online-store is described at three levels of abstraction: *physical entity*, *model*, and *category*. Each *product category* has associated a *tax rate*, each *product model* has a *list price*. Book editions, i.e. objects at level *model* that belong to *product category Book*, additionally have an *author*. Our company keeps track of *physical entities* of *products* (e.g. copies of book *HarryPotter4*), which are identified by their *serial number*. In addition to *books*, our company starts to sell *cars*, i.e. it introduces *Car* as a new *product category*. Cars differ from books in that they are described at an additional level, namely *brand*, and in that they have additional attributes: *maxSpeed* at level *product model* and *mileage* at level *physical entity*. As our sample online-store specializes on selling cars of *brand Porsche 911*, it wants to be able to register physical entities of this *car brand* at the *Porsche 911 club*.

Our company further keeps track of *companies* that produce these products. Our product catalog describes the companies in which products are produced at three levels as well: (*factory*, *enterprise*, and *industrial sector*). Relationship *producedBy* between levels *physical entity* and *factory* can be abstracted to *product model* and *enterprise* and further to *product category* and *industrial sector*. The following dependencies exist between these abstraction levels of relationship *producedBy*: (1) A *physical entity* can only be produced at a *factory* that belongs to an *enterprise* that produces the corresponding *product model*. (2) A *product model* can only be produced at an *enterprise* that belongs to an *industrial sector* that produces the corresponding *product category*. Specifically, product category *Car* is produced by industrial sector *CarManufacturer*, each model of car brand *Porsche911* is produced by *Porsche Ltd* and physical entity *myPorsche911CarreraS* is produced by factory *Porsche Zuffenhausen*.

1.2.2 UML Representation

In this section we show how our sample problem can be modeled using a standard modeling language, like the UML, by representing domain entities and concepts by multiple objects/classes and organizing them in aggregation-, generalization-, and classification-hierarchies. While the resulting UML model should make intuitively apparent the problems with such a *multiple representation approach*, a more detailed evaluation is provided below.

In our reference UML model (see Figure 1.1) the three-level *product catalog* is modeled by arranging classes (*ProductCategory*, *ProductModel*, *ProductPhysicalEntity*) and their instances ({*Car, Book*}, {*Porsche911CarreraS, Porsche911GT3, HarryPotter4*}, {*myPorsche911CarreraS, myCopyOfHP4*}, respectively) in an aggregation hierarchy, where each aggregation level represents one level of abstraction (*product-category*, *product-model*, and *physical-entity*).

Objects at the same level in different subhierarchies may differ from each other. For example, *product model*s have a *listprice*, but *car model*s (i.e., objects at level *model* belonging to *car*, an object at level *category*) have an additional attribute *maxSpeed*. Furthermore, *physical entities* of *car* additionally have a *mileage*. In our reference UML model (see Figure 1.1) this non-uniformity between objects at the same level is modeled by specializing classes that are part of the aggregation hierarchy below *ProductCatalog*, i.e. *ProductCategory*, *ProductModel*, *PhysicalEntity*, to corresponding classes, *CarCategory*, *CarModel*, *CarPhysicalEntity*, that describe common properties of objects belonging to product category *Car* at the respective level. Note that singleton classes like *CarCategory* are used to refine aggregation relationships, e.g., to define that each instance of *CarModel* belongs to product category *Car*.

Multi-level modeling of relationships, as desired, is not possible in plain UML. A partial solution is to model each level of a relationship as an association and, when instantiating the association that represents the most abstract level, to specialize the others. Figure 1.2 exemplifies this approach by modeling relationship *producedBy* as associations *producedBy1*, *producedBy2*, and *producedBy3*. When introducing a *producedBy*-link between product category *Car* and industrial sector *CarManufacturer* one has to introduce

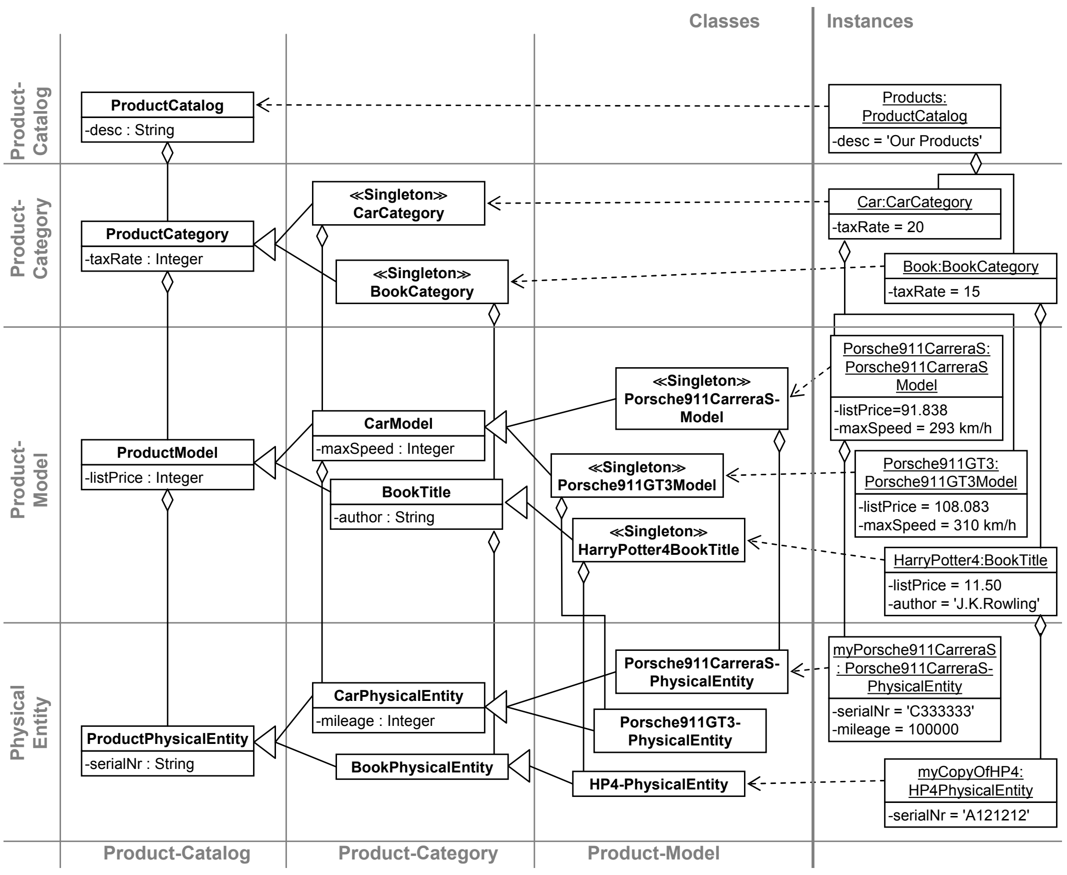


Figure 1.1: Product catalog modeled in plain UML, using generalization, instantiation and aggregation. Simplified: relationships and level *brand* are omitted

specialized associations between *CarModel* and *CarManufacturerEnterprise*, as well as between *CarPhysicalEntity* and *CarManufacturerFactory*. Another partial solution is to use OCL to express extensional constraints that are imposed from a higher to a lower level of abstraction of the *producedBy* relationship. Figure 1.3 exemplifies this approach by modeling relationship *producedBy* as three associations, between *ProductCategory* and *IndustrialSector* (in the following referred to as *A*1), between *ProductModel* and *Enterprise* (*A*2), and between *ProductPhysicalEntity* and *Factory* (*A*3). The constraint between *A*2 and *A*1 ensures that a *product model* can only be produced at an *enterprise* that belongs to an *industrial sector* that produces the corresponding *product category*. The link between product model

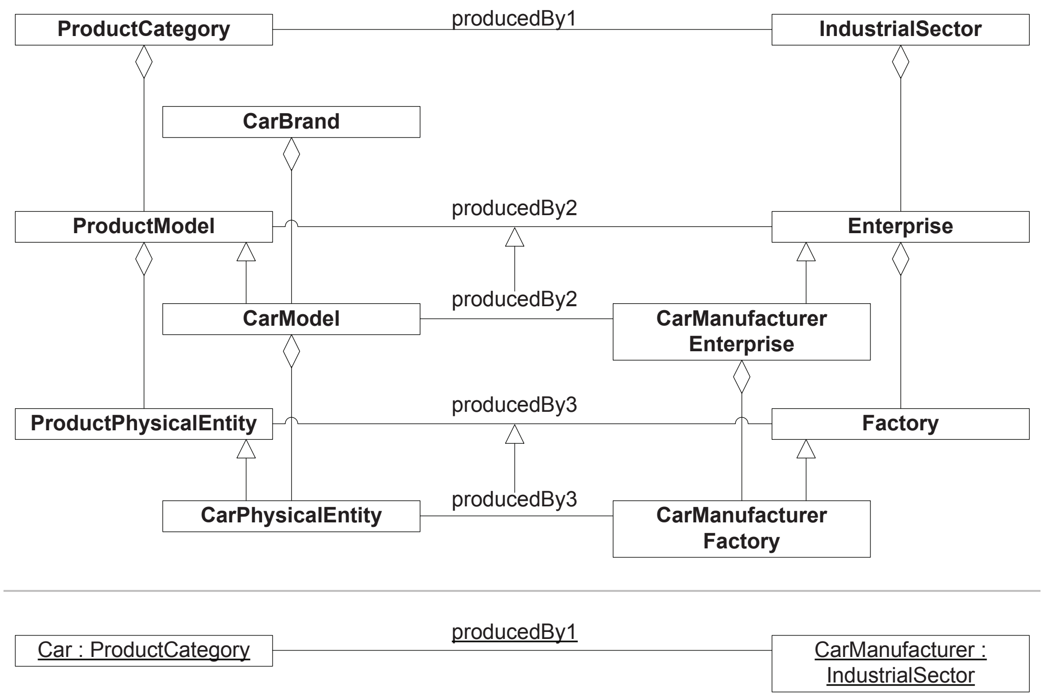
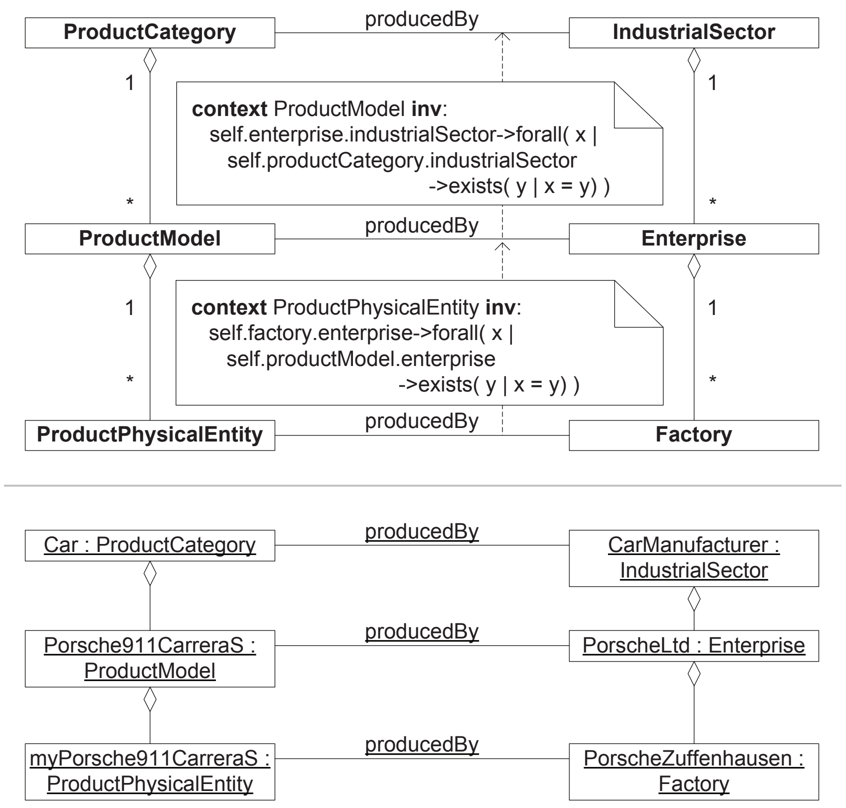


Figure 1.2: Modeling relationships at multiple levels of abstraction by specializing and instantiating associations.

P911CarreraS and enterprise *PorscheLtd*, which is an instance of *A*2, satisfies this constraint, since there is a corresponding link between product category *Car* and industrial sector *CarManufacturer*, which is an instance of *A*1. Likewise, the constraint between *A*3 and *A*2 ensures that a *physical entity* can only be produced at a *factory* that belongs to an *enterprise* that produces the corresponding *product model*. The link between physical entity *myPorsche911CarreraS* and factory *PorscheZuffenhausen*, which is an instance of *A*3, satisfies this constraint since there is a corresponding link between product model *P911CarreraS* and enterprise *PorscheLtd*, which is an instance of *A*2.

For an evaluation of these two partial solutions we refer to comparison criteria *multiple relationship-abstractions* in the following section.



```
context ProductModel inv:
  self.enterprise.industrialSector->forall( x |
    self.productCategory.industrialSector
                          ->exists( y | x = y) )
```

```
context ProductPhysicalEntity inv:
  self.factory.enterprise->forall( x |
    self.productModel.enterprise
                          ->exists( y | x = y) )
```

Figure 1.3: Relationships *producedBy* at different levels of abstraction modeled with UML and OCL

1.2.3 Comparison Criteria

The suitability of a multi-level modeling language that should be capable of modeling hetero-homogeneous hierarchies may be benchmarked against the following requirements. We exemplify these requirements by evaluating our UML solution.

Compactness: A modeling solution is compact if it is modular and redundancy-free. *Modularity* means that all model elements that represent aspects of the same domain concept are tied together and can be used and updated as a self-contained part of the model without interfering with the rest of the model. *Hierarchical Modularization* means that sub-hierarchies (e.g. the classes *CarCategory*, *CarModel*, *CarPhysicalEntity* together with their respective subclasses and instances) can be treated as self-contained models (sub-systems or federated systems with their own schema and own data). A model contains *redundancy* if information is represented more than once.

Our reference UML model (see Figure 1.1) is neither modular nor redundancy-free: Domain concept *car* is represented by four separate model elements, namely by object *Car:CarCategory*, and by classes *CarCategory*,*CarModel*, and *CarPhysicalEntity* which are not tied together to a self-contained part of the model. Concerning hierarchical modularization, subclasses and instances of *CarCategory*, *CarModel*, and *CarPhysicalEntity* cannot be regarded as self-contained part of the model. Furthermore, the hierarchic relation between *product* and *car* is represented redundantly as aggregation between *Car:CarCategory* and *Products:ProductCatalog* and as generalizations between *CarCategory* and *ProductCategory*, between *CarModel* and *ProductModel*, and between *CarPhysicalEntity* and *ProductPhysicalEntity*.

Query flexibility: A modeling solution supports query flexibility if it provides several pre-defined entry points for querying, such as class names or qualified identifiers to refer to various sets of objects. In the context of multi-level abstraction we check whether one may easily refer to the set of objects at one abstraction level that belong to a certain domain concept. To test this criterion, we query (1) all objects belonging to product-category *car* at level *product-model*, (2) all objects belonging to product-category *car* at level *physical-entity*, and (3) all objects belonging to level *product-physical-entity*. In our reference UML model these entry points for queries are available through the classes (1) *CarModel*, (2) *CarPhysicalEntity*, and (3) *ProductPhysicalEntity*.

Hetero-Homogeneous Level-Hierarchies: A modeling solution supports *hetero-homogenous level-hierarchies* if an additional abstraction level can be introduced without the need to change existing model elements. To evaluate this criterion, we will introduce an additional abstraction level *car-brand* for product-category *car* between levels *product-category* and *product-model* and describe one car-brand, namely *Porsche 911*. If this extension can be made without affecting the existing model, i.e, definitions of other product-categories, e.g., *book*, or generally of products, the modeling solution is considered to support hetero-homogenous level-hierarchies. Concerning our reference UML model the respective extension could be made by introducing a class *CarBrand* together with introducing as component of class *CarCategory* and as aggregate class of *CarModel*.

Multiple relationship-abstractions: A modeling technique supports multiple *relationship-abstractions* if it does not only support modeling of domain objects at multiple levels of abstraction, but equally supports multi-level abstraction of relationships. For an in-depth discussion of the three classical abstraction principles and their application to relationships we refer to Chapter 2.

Relationship occurences at higher levels of abstraction (implicitly) play roles of relationship classes and metaclasses. These relationship classes and metaclasses (implicitly) specialize relationship classes and metaclasses that are played by higher level relationships. We also refer to these (implicitly given) relationship classes and metaclasses as *class-facets* and *metaclass-facets of a relationship.*

For example, relationship occurrence *producedBy* between product category *Car* and industrial sector *Car manufacturer* can be regarded (1) as an instance of relationship class *producedBy* between *product category* and *industrial sector*, (2) as a relationship class, having relationships between *car physical entities* and *car manufacturer factories* as instances (this relationship class specializes the class of relationships between *product physical entities* and *factories*), (3) as a relationship class, having relationships between *car models* and *car manufacturer enterprises* as instances, (4) as a relationship metaclass, having such relationship classes as instances.

Support for multiple relationship-abstraction can be further characterized by:

(a) Provide for such (implicit) relationship classes and metaclasses and their specializations (*(Meta-)Classification of Relationships*).

(b) Convenient mechanisms to refer to these (implicitly given) relationship classes and metaclasses and to their extensions (*Adressability/Queryability*).

(c) Mechanisms to specialize domain and range of relationship classes, based on higher-level relationships, either by specialized classes (as just described) or by explicit constraints (*Specialization/Instantiation of Domain and Range*).

(d) If support for relationship abstraction is provided, we also evaluate compactness (see above). We assume that a modeling solution is impracticable, if information about relationships and its class- and metaclass-facets needs to be represented redundantly (*Compactness of Relationship Abstraction*).

The first partial solution (see Figure 1.2) explicitly represents class-facets of relationships as associations. One relationship (for example *producedBy* between *Car* and *CarManufacturer*) has to be represented redundantly: (1) as link between object *Car* and object *CarManufacturer* (not depicted), (2) as association *producedBy2* between *CarModel* and *CarManufacturerEnterprise*, and (3) as association *producedBy3* between *CarPhysicalEntity* and *CarManufacturerFactory*, together with specialization relationships. This solution is, thus, far from being (d) *compact* and, for that reason, impracticable.

The second partial solution (see Figure 1.3) employs OCL to express extensional constraints that are imposed from a higher to a lower level of abstraction of the *producedBy* relationship. This covers criterion (c) *Specialization of Domain and Range*. It does not provide (a) *(Meta-)Classification of Relationships*. Thus, regarding criterion (b) *addressability/queryability* it does also not provide for convenient mechanisms to refer to these (implicitly given) relationship classes and to their extensions.

1.3 Research Approach

This thesis contributes to conceptual modeling, especially to the vision of conceptually modeling in the large, as outlined in Section 1.1. Since the realization of this vision is beyond the scope of one thesis, we focus on the very core of the problem, namely multi-level modeling of entities and relationships, as exemplified in Section 1.2. The resulting formalism can serve as the central building block of further work on realizing our vision. Furthermore, already the core approach on its own is powerful enough for several interesting applications, such as modeling of domains where the borderline between classes and individuals is not clear cut as well as hetero-homogeneous hierarchies in data warehousing.

Our research follows the *design-science* research paradigm, like most of the research in data and knowledge engineering. While natural science is about how things are, design science is about how things ought to be [113, Chapter 5: The Science of Design].

March and Smith [76] propose a research framework for design sciences in IT research, which consists of two dimensions: research outputs and research activities. *Research outputs* are four kind of artifacts: constructs, models, methods, and instantiations. *Constructs* are the concepts which are used to describe problems within a domain. *Models* are based on constructs and describe how they are related to each other. *Methods* are algorithms or guidelines, based on constructs and models, that prescribe how certain tasks are to be fulfilled. *Instantiations* are implementations of constructs, models, and methods and show the feasibility of models and methods. *Research activities* in design science are build and evaluate, where *Build* is the activity of designing constructs, models, methods, and instantiations and *Evaluate* is the activity of developing a set of criteria and of assessing the performance or quality of artifacts against these criteria. Design science research typically falls into one or more of the eight resulting quadrants of the cross-product of *research outputs* and *research activities.*

We now apply this research framework to the domain of conceptual modeling. *Constructs* of conceptual modeling are used to describe concepts within subject domains, and *models* describe how conceptual modeling constructs are related to each other and can be used in combination to represent conceptualizations of subject domains, that is what is often referred to as meta-models. *Methods*, in this context, are either guidelines how to use modeling constructs and models or operations that can be applied on conceptual schemas. *Instantiations* in conceptual modeling are typically concerned with (1) (graphical) modeling environments that facilitate conceptual modeling based on specific constructs and models, (2) automating various tasks of conceptual modeling such as consistency checking, and (3) transforming conceptual schemas to logical schemas (implementations).

This thesis is concerned with building language constructs for multi-level modeling (*build-construct*) and with defining how they are related (*build-model*) and used for multi-level modeling. These language constructs are m-objects and m-relationships. We evaluate these constructs and model together with related modeling techniques against a set of comparison criteria (*evaluate-construct*, *evaluate-model*). Furthermore, we define operations for manipulating and querying multi-level models (*build-method*), and implement a proof-of-concept prototype (*build-instantiation*).

Hevner et al. [46] build on the above cited framework and provide more detailed guidelines for information systems research. According to them, design science research requires:

1. *Design as an artifact:* the output of design science is a purposeful artifact (construct, model, method, or instantiation).
2. *Problem relevance:* the design artifact must provide a technical solution to important problems in organizations.
3. *Design evaluation:* the design artifact must be evaluated, concerning its utility and quality.
4. *Research contributions:* design science is innovative, it has to solve unsolved problems or provide a better solutions to previously solved problems. These contributions have to be made clear and verifiable.
5. *Research rigor:* design science has to rely on rigorous methods, often in the form of mathematical formalism.
6. *Design as a search process:* design is an iterative process that can be described as a generate/test cycle. (Parts of) design alternatives are generated and tested until a viable solution is found.
7. *Communication of Research:* research output has to be presented to different audiences (technical and managerial).

Table 1.1 shows how we applied these guidelines in our research.

1.4 Scope

In this thesis we are concerned about structural (conceptual) models at multiple levels of abstraction. We concentrate on the core constructs of conceptual

Research guidelines	**as applied in this thesis**
Design as an artifact	Research outputs of this thesis are artifacts: language constructs, (meta)model, methods, and a proof-of-concept-prototype [111]
Problem relevance	Relevance of these artifacts is within realizing the vision of conceptual modeling in the large, and, on their own, in solving problems in data warehousing
Design evaluation	M-objects and m-relationships are evaluated using descriptive evaluation methods
Research contributions	Contributions are manifold and outlined in Section 1.6
Research rigor	Designed constructs, model, and methods are defined using mathematical formalisms such as set-theory
Design as a search process	Solution candidates where tested against a set of criteria (see Section 1.2.3)
Communication of Research	Key results where presented at international conferences and workshops.[86, 87, 88]

Table 1.1: Research guidelines for design science [46] as applied for this thesis

modeling, namely entities (individuals, objects) and relationships (links, associations, connectors) between them, as well as their abstraction at multiple levels.

We do not discuss modeling of passive and active behavior. We believe, however, that the provided multi-level modeling approach can also be extended and applied for modeling of passive behavior (methods, business processes) and active behavior (trigger, business rules).

The provided approach is first and foremost a conceptual modeling approach. But it is also meant to be easily implementable as extension of database management systems, object-oriented programming languages, and modeling tools. The provided approach is therefore also influenced by the goal to have good computational properties.

1.5 Solution Idea

The core of this thesis is dedicated to the structural and operational model of the m-object and m-relationship approach, originally introduced in [86]. *Multi-level objects* (m-objects) and *multi-level relationships* (m-relationships) provide for the *concretization* of objects and relationships along multiple levels of abstraction. The basic ideas of this approach, which builds on current approaches (powertypes, deep instantiation, and materialization), are (i) to encapsulate the different levels of abstraction that relate to a single domain concept (e.g., the level descriptions, *catalog*, *category*, *model*, *physicalEntity* that relate to the single domain concept, *product catalog*) into a single m-object (see *Product*), and (ii) to integrate aspects of the different abstraction principles (aggregation, generalization, and classification) into a single *concretization* hierarchy.

An m-object encapsulates and arranges abstraction levels in a linear order from the most abstract to the most concrete one. Thereby, it describes itself and, for each level of the concretization hierarchy beneath itself, the common properties of descendant objects. An m-object that concretizes another m-object, its parent, inherits all levels except for the top-level of the parent. It may also specialize the inherited levels or even introduce new levels. An m-object specifies concrete values for the properties of the top-level. This top-level has a special role in that it describes the m-object itself. All other levels describe common properties of m-objects beneath itself.

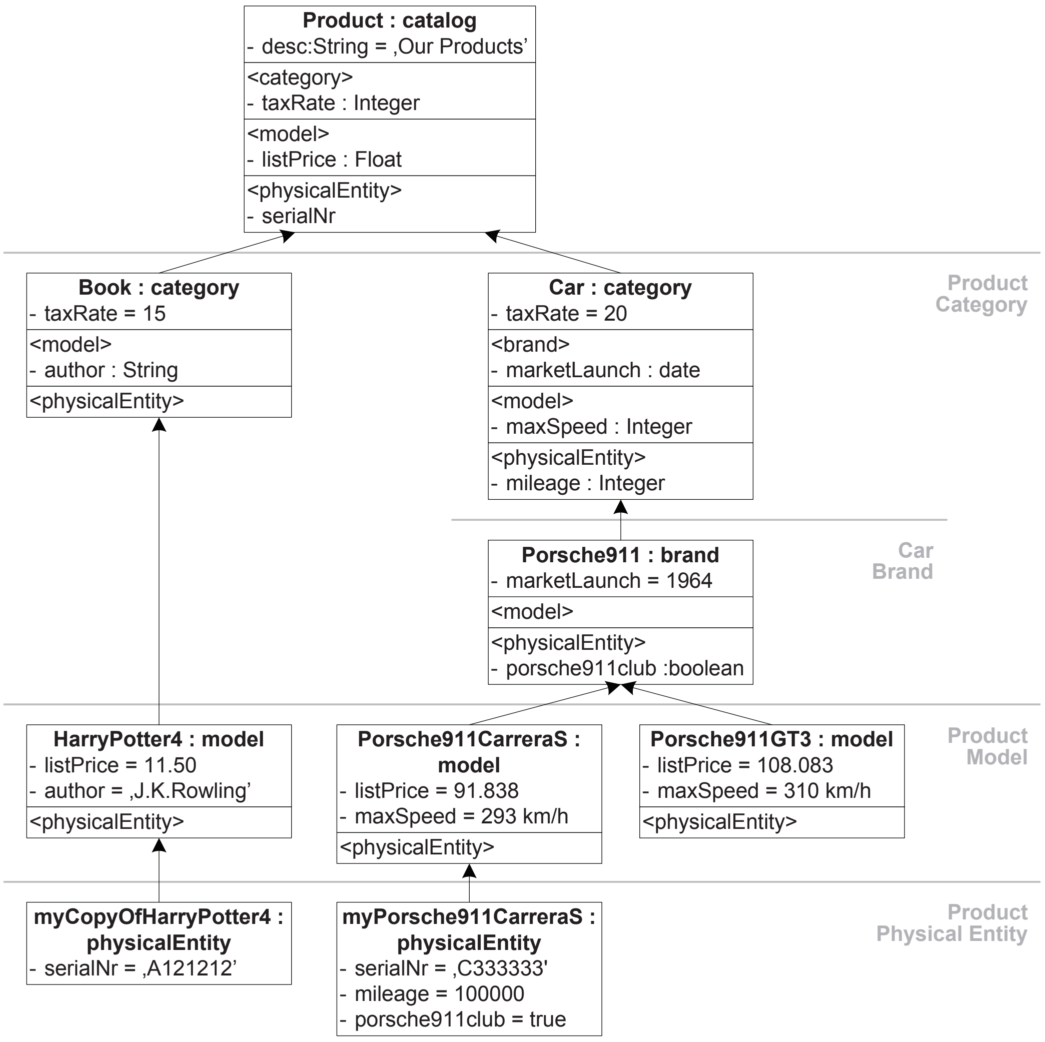


Figure 1.4: Product catalog modeled with m-objects

Consider m-object *Product* of Figure 1.4, which consists of four levels with one attribute each. Level *category* is the parent level of level *model*. Attribute *taxRate* is associated with level *category*, has domain *Integer* and does not define a value. The m-object resides at level *catalog*, which is its top-level. The top-level defines attribute *desc* of domain *String* and specifies value *'Our Products'* for this attribute.

An m-object can *concretize* another m-object, whereby the latter is referred to as its parent. Such a concretization relationship combines aspects of classification, generalization, and aggregation as follows:

- *Classification – Instantiation:* Each m-object can be regarded as instance of its parent m-object. It must adopt all levels from its parent except for the parent's top-level, and it can specify values for its attributes.
- *Generalization – Specialization:* The level descriptions of an m-object correspond to subclasses of the corresponding levels of its parent. The m-object can define new levels or add attributes to levels.
- *Aggregation – Decomposition:* The hierarchy between m-objects of different levels expresses the aggregation hierarchy. Thereby, an m-object must not change the relative order of levels and attributes which it inherits from its parent.

A concretization relationship between two m-objects does not reflect that one m-object is an *instance of*, *part of*, and *subclass of* another m-object as a whole. Rather, a concretization relationship between two m-objects, such as *Car* and *Product* in Figure 1.4, is to be interpreted in a multi-faceted way. Each m-object plays multiple roles with regard to the classical semantic abstraction hierarchies (also referred to as *facets* of an m-object). M-object *Car* concretizes m-object *Product* in Figure 1.4. The concretization relationship expresses classification: m-object *Car* is instance of level *category* of m-object *Product* because level *category*, which is the first non-top-level of m-object

Product, is its top-level. It also specifies a value for its attribute *taxRate*. M-object *Car* specializes m-object *Product* by introducing a new level *brand* and adding attribute *maxSpeed* to level *model* and attribute *mileage* to level *physical entity*. Level *model* of m-object *Car* can be regarded as a subclass of level *model* of m-object *Product*. Cars are further categorized into brands, models and physical entities. These levels define the aggregation hierarchy.

M-relationships are analogous to m-objects in that they describe relationships between m-objects at multiple levels of abstraction. M-relationships are bi-directional. To facilitate references to objects involved in binary relationships, we take, however, a (potentially) arbitrary directional perspective by considering one object in the *source* role and the other object in the *target* role of the relationship. Each m-relationship links a source m-object with a target m-object. Additionally, it connects one or more pairs of levels of the source and the target. These connections between source- and target-levels constitute the abstraction levels of an m-relationship, its *connection levels*. They define that m-objects at source-level are related with m-objects at target-level. We point out that the generic roles *source* and *target*, which we introduced here for simplicity, may be easily replaced by relationship-specific role names.

To represent that car models have a designer, one may introduce m-relationship *designedBy* between source m-object *Car* and target m-object *Person* (see Figure 1.5). More precisely, it links source-level *model* of m-object *Car* to target-level *individual* of m-object *Person*. While relationship *designedBy* only links one source-level with one target-level, relationship *producedBy* between *Product* and *Company* specifies multiple connection-levels. The relationship expresses that product categories are produced by industrial sectors, that product models are produced by enterprises, and that physical entities are produced by factories.

Like m-objects, m-relationships can be concretized. Concretizing an m-relationship means to substitute its source or its target for a direct or indirect

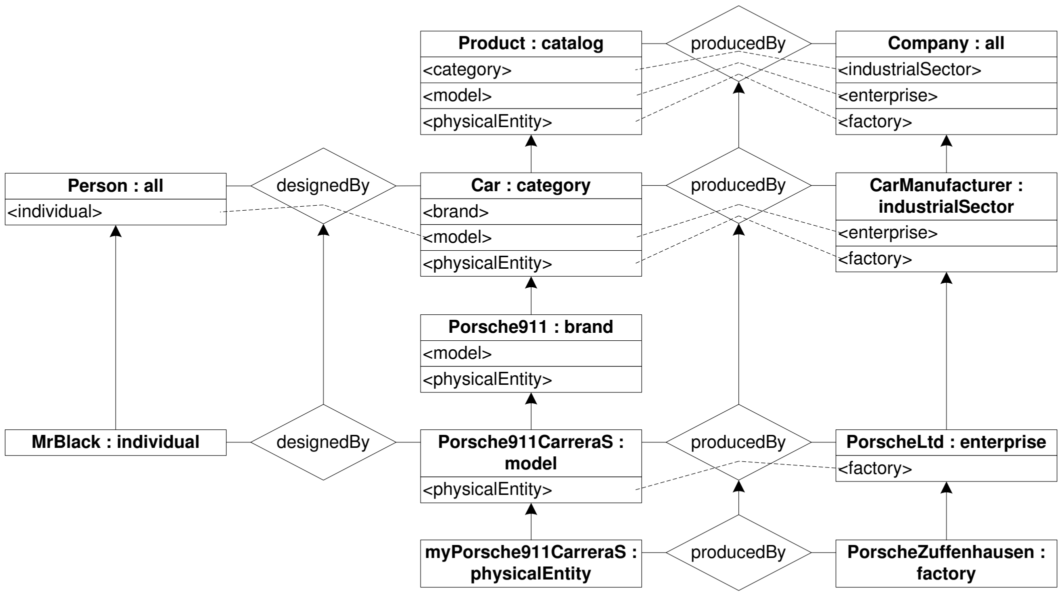


Figure 1.5: Product catalog modeled with m-objects and their m-relationships [86]

concretization. The concretizes-relationship between two m-relationships expresses instantiation and/or specialization.

Consider m-relationship *designedBy* between m-objects *Car* and *Person* in Figure 1.5. M-relationship *designedBy* between m-objects *Porsche911CarreraS* and *Mr.Black* concretizes the former m-relationship. In this case, the concretization solely represents an instantiation, i.e. no further concretization is possible. Now look at m-relationship *producedBy* between m-objects *Product* and *Company*. M-relationship *producedBy* between *Car* and *CarManufacturer* concretizes the former m-relationship, expressing that cars can only be produced by car manufacturers. This concretization further defines that each car model must be produced by enterprises that belong to the car manufacturer sector. Similarly, each physical entity of cars must be produced by a factory of the car manufacturer sector.

Dependent on the levels which an m-relationship connects, it can play various roles:

- *Relationship occurrence:* Each m-relationship links the top-level of its source m-object with the top-level of its target m-object and can be regarded as a relationship occurrence between these m-objects.

- *Relationship class:* An m-relationship that links a non-top-level of the source m-object with a non-top-level of the target m-object can be regarded as a relationship class. It defines that m-objects at the source-level may or must – depending on whether the relationship is optional or mandatory, a distinction which we do not discuss in this thesis – have a relationship with m-objects at the target-level, which are concretizations of the target m-object. An m-relationship that links n pairs of non-top-levels, with $n \geq 2$, can further be seen as a meta^{n-1} relationship class because these links constrain how this relationship has to be further concretized.

In Figure 1.5, the m-relationship *designedBy* between source m-object *Car* and target m-object *Person* can be regarded as a relationship occurrence between these two m-objects and as a relationship class, defining that each car model may or must (see above) have a relationship with an individual person. In our example, such a relationship exists between the car model *Porsche911CarreraS* and the individual person *Mr.Black*. This relationship represents a relationship occurrence. Similarly, m-relationship *producedBy* between *Product* and *Company* can be regarded as a relationship occurrence and as multiple relationship classes, namely at connection-levels *(category, industrialSector)*, *(model, enterprise)*, and *(physicalEntity, factory)*. M-relationship *producedBy* between m-objects *Car* and *CarManufacturer* is a relationship occurence and as such a member of the extension of m-relationship *producedBy* at connection-level *(category, industrialSector)*. It further refines relationship classes at connection-levels *(model, enterprise)* and *(physicalEntity, factory)*, by constraining producers of *Car models* to enterprises of industrial sector *CarManufacturer* and by constraining producers of *Car physical entities* to factories of industrial sector *CarManufacturer*.

1.6 Contributions and Limitations

In this section we outline the contributions and limitations of the m-object/m-relationship approach. A more detailed evaluation is provided in Section 9 which concludes the thesis.

To the fields of *conceptual modeling*, *data engineering*, and *software engineering* we contribute a multi-level modeling technique with a formally defined structural and operational model. Its novel features are especially

- heterogeneous level hierarchies
- multi-level modeling of relationships (parallel to our research, Atkinson et al. [42, 6] also identified the problem and discuss a somehow similar, yet less powerful and less versatile approach)
- a solution to the problem of 'strict-metamodeling vs. loose-metamodeling'[8, 65, 2, 29], by providing an abstraction relationship that combines classification and generalization, but without blurring them.

We apply m-objects and m-relationships in the realm of data warehousing. For this purpose, we extend the basic m-object/m-relationship approach with multiple concretization and n-ary m-relationships described by measures. In this way, we contribute to *data warehousing* as follows:

- We provide a solution to the problem of heterogeneous dimension hierarchies
- We provide an approach to work with heterogeneous base facts
- We provide a proof-of-concept-prototype as extension package for Oracle DB. This package was implemented under our supervision as part of a master's thesis.

To *ontology engineering* within the semantic web, we contribute a mapping from m-objects/m-relationships to the decideable fragment of the web ontology language OWL. We thus provide a more powerful alternative to *punning*, which is the standard way to ontological metamodeling in OWL.

Since the m-object/m-relationship approach, as presented in this thesis, is only the first step towards realizing our vision of conceptual modeling in the large, there are a couple of *limitations*. Most notably, this basic approach does not consider orthogonal abstraction hierarchies, i.e., we do not discuss generalization, classification, and aggregation orthogonal to our concretization hierarchies. We also neglect cardinality constraints and other custom integrity constraints which are necessary to unleash the full expressivity of m-objects and m-relationships. Besides, we discuss m-objects and m-relationships only as a structuring mechanism and do not consider their real-world semantics, that is, m-objects and m-relationships are, like object-orientation in general, at the epistemological level and not at the ontological level (see [37] for this distinction).

The outlook provided in Chapter 9 sketches how these limitations can be overcome without changing the basic m-object/m-relationship approach and thus how m-objects and m-relationships are at the very core of the realization of our vision of conceptual modeling in the large.

1.7 Outline

The remainder of this thesis is organized as follows.

Chapter 2 – Abstraction Principles clarifies our use of basic notions of conceptual modeling and reviews classical and more recent work on the basic abstraction principles, classification, generalization, and aggregation. It also discusses our general idea of abstraction hierarchies.

Chapter 3 – Comparison of Multi-Level Modeling Techniques evaluates multi-level modeling techniques, potency-based deep instantiation, materialization, and powertypes, with regard to comparison criteria introduced above.

Chapter 4 – M-Objects and M-Relationships introduces m-objects and binary m-relationships as well as their organization in concretization hierarchies. It provides set theoretic definitions of their basic structure and of global consistency criteria.

Chapter 5 – Creating and Manipulating Multi-Level Models gives formal definitions, in terms of pre- and postconditions, of operations for defining and manipulating multi-level models based on m-objects and m-relationships.

Chapter 6 – Querying Multi-Level Models provides a query algebra for m-objects and m-relationships. It also discusses query support for the task of top-down multi-level modeling.

Chapter 7 – Multi-Level Modeling and OWL shows how m-objects and m-relationships can be mapped to the de-facto standard for web ontologies.

Chapter 8 – Hetero-Homogeneous Hierarchies in Data Warehouses extends the basic m-object/m-relationship approach for its application in data warehouses to allow for concretization in dimension hierarchies and cubes. Hierarchies of m-objects serve as hetero-homogeneous dimension hierarchies, and hierarchies of n-ary m-relationships serve as multi-level cubes.

Chapter 9 – Summary and Outlook summarizes and evaluates the m-object/m-relationship approach and gives an overview of future work, especially discussing how m-objects and m-relationships are the first step towards realizing our vision of conceptual modeling in the large.

Chapter 2

Abstraction Principles

Contents

In this section we will first clarify how we use some basic notions of conceptual modeling, especially discussing the notion of *abstraction hierarchy*. Subsequently we will discuss the classical abstraction principles, classification, generalization, and aggregation in (conceptual) modeling and exemplify them with regard to our sample problem, followed by a short discussion of the notion of metaclassification. We will complement our discussion of abstraction in conceptual modeling by a short review of related work.

Note that the informal definitions of basic notions provided in this chapter are not meant to be valid for conceptual modeling in general. Our goal is rather to provide the conceptual framework for the discussion of modeling with m-objects and m-relationships. Since we assume that most of this chapter is common knowledge, we refrain from giving extensive bibliographic references. We will back up some of our basic assumptions by referring to Guizzardi's work on ontological foundations for structural conceptual models. Further references to related literature can be found in the related work section.

The discussion of basic notions and abstraction principles in this section is quite general, the relation to modeling hetero-homogeneous abstraction hierarchies will be mainly provided by the given examples. In order to visualize the given examples we use a simple graphical notation, which is loosely based on entity/relationship-diagrams [20] and on UML class diagrams [47]. We will explain this notation together with the examples.

2.1 Basic Notions

Modeling, in general, results in models. Following Stachowiak [117] as cited in [64], a *model* is based on an *original*, reflects a selection of the original's properties and is usable in place of the original with respect to some purpose. We also say a model represents an original. A model consists of elements and each *element* of the model (*model element*) represents one or more elements or properties of the original. Guizzardi [39] distinguishes between *model* and *model specification*, where a model only exists in the mind of one or more users and a model specification is a representation of such a model and captured in some concrete artifact. Most of the time we do not make this distinction between model and model specification, but rather use the notion of *model* in the sense of a model specification. Note that we do *not* use the notion of *model* as, for example, in *relational data model*, where it is rather used to refer to the set of constructs of a data modeling language. We also do not use the notion of *model* in its mathematical, model-theoretic sense.

A *structural model* is a model that represents the static structure of an original, disregarding its behavior. Such a *structural model* consists of two types of elements, namely of *objects* and of *relationships* between them. Objects connected by a relationship play different *roles* in that relationship. Elements in a structural model are further described by *attribute values*. Relationships are sometimes also referred to as *links* or *connectors*.

An *abstraction* combines a (possibly empty or singleton) set of elements (of the domain or of the model) by omitting some details. We especially consider the three classical abstraction principles, classification, generalization, and aggregation, as described below. We also use the notion of *concept* to refer to something that is an abstraction of something more concrete (entities, relationships, more concrete concepts).

In conceptual modeling the to-be represented original is a domain of interest as we perceive it, typically some part of the world. Following Chen [20] and many others, we make the ontological assumption that the world, or another domain of interest, as we perceive it, consists of *entities* and of *relationships* between them. Further information about entities and relationships can be obtained by observation or measurement. Each entity connected by a relationship plays a specific *role* in this relationship.

A *structural model of a domain* is a structural model that represents such a domain of interest, as we perceive it. Each element of a structural model of a domain represents an entity of that domain, a relationship of that domain, or an abstraction of entities or relationships. Further information about entities, relationships, and their abstractions are represented by attribute values.

The *semantics* (meaning) of a model is given by the represented original (*semantic domain*) and a mapping from model elements to elements of the represented original (*semantic mapping*) [43]. We use the notion of *real-world semantics* if we want to emphasize that the represented original is a part of the world, as we perceive it.

A *hierarchy* is a partially-ordered set of elements. The elements directly above an element are called its *parents*, the elements directly below an element are called its *children*. The order of elements in a hierarchy is given by a hierarchy relation, consisting of hierarchy-relationships which are pairs of child and parent elements. The elements directly or indirectly above an element are called its *ancestors*, the elements directly or indirectly below an element are called its *descendants*. An element without parent is called *root*. The *sub-hierarchy* of a given element in a hierarchy consists of this element and all its descendants, together with their order.

We distinguish between different kinds of hierarchies: A *total order* or list of elements is a hierarchy where for any two elements one of these elements is either descendant or ancestor of the other. A *tree* is a hierarchy with a single root, where each element has at most one parent. A *forest* is a set of trees. A *lattice* is a hierarchy where for any two elements (1) there is an element that is an ancestor of these two elements or where one of the two elements is an ancestor of the other, and (2) there is an element that is a descendant of these two elements or where one of the two elements is a descendant of the other.

Hierarchies may be structured by *levels*. Informally, the level an element belongs to is its vertical position in the hierarchy. The set of levels in a hierarchy is again a hierarchy. Very often, the set of levels in a hierarchy is a total order.

An *abstraction hierarchy* is a (part of a) structural model that arranges its elements in a hierarchy. An element in such a hierarchy is an abstraction of its children. A *transitive abstraction hierarchy* is a hierarchy where each element can be regarded as abstraction of all its descendants. An *anti-transitive abstraction hierarchy* is an abstraction hierarchy where each element is only an abstraction of its children, but not of descendants of its children. An *abstraction level* is a level in an abstraction hierarchy. The *schema of an abstraction level* defines the common structure of elements at this level. The *schema of an abstraction hierarchy* defines the order of its abstraction levels as well as abstraction level schemas, one for each abstraction level.

A *hetero-homogeneous abstraction hierarchy* is a transitive abstraction hierarchy in which each element may specialize the schema of its sub-hierarchy. *Specialization* of a schema of an abstraction hierarchy refers to introducing additional abstraction levels and to specializing schemas of abstraction levels.

Our notion of *multi-level modeling* refers to constructing such hetero-homogeneous abstraction hierarchies with multiple levels. A *multi-level model* (*m-model*) is a model with elements arranged in such hierarchies.

If we want to emphasize that an abstraction hierarchy is modeled top-down, that is from abstract to concrete, we write *concretization hierarchy*. The children of an element represent *concretizations* of this element or concretizations of what this element represents. Analogous *concretization principles* to classification, generalization, and aggregation are *instantiation*, *specialization*, and *decomposition*, respectively.

2.2 Classification and Instantiation

Classification is an abstraction principle according to which a set of similar elements (*instances*) is combined into one element (*class*) defining the structure of its instances (see [33]). *Instantiation* is a concretization principle according to which a new instance of a class is created using the structure defined in a class as a template. Instances are often also referred to as *individuals* or *objects*. Classes are often also referred to as *universals*.

We assume, that every class (implicitly) provides a *principle of identity* [39]. This is the premise for its instances being countable and identifiable. For example, class Person has another principle of identity than class Organization. If an individual is instance of more than one class, than these classes must carry the same principle of identity. We do not consider terms such as *Red* or *Driveable* as classes, since they only provide *attribution* [39] and do not carry a principle of identity. We further assume that *role types*, such as *Student*, carry a (dependent) principle of identity and can be represented as classes. For a brief discussion of these basic assumptions we refer to the paragraphs on *ontological foundations of conceptual modeling* and on *Role/Player* in Section 2.6.

A *conceptual schema* is a structural model of a domain that consists of object classes and relationship classes. We also say such models are on the *schema level.* Models that consist of individual elements which represent single entities and single relationships are on the *instance level.* A model on the schema level is also sometimes referred to as *type model.* A model on the instance level is also referred to as *token model* (see [64]).

A class represents first the common structure – in terms of attributes – of its instances (often referred to as *intension* or *type*) and second the set of its instances (often referred to as *extension* or *members* of a class). A class with exactly one instance is referred to as *singleton class.*

We often distinguish between classification of relationships and classification of objects. The former connects a *relationship instance* with a *relationship class.* The latter connects an object with an *object class.* In the Entity-Relationship Model (ERM) [20, 115] object classes are referred to as *entity sets* (or *entity types*) and relationship classes are referred to as *relationship sets* (or *relationship types*). In the Unified Modeling Language (UML) [16, 99, 96] relationship instances are called *links* and relationship classes are called *associations.*

The *structure* of an element in a model is given by its set of attributes (sometimes also referred to as slots [90]). We also use the notion of *own-type* to refer to the structure of an element. Each attribute is associated with a *value set* (or a *data type*). An *instantiated attribute* is additionally associated with a value from its value set. *Multi-valued attributes* may be associated with more than one value. A class defines attributes that are shared by all its instances. These attributes are the class's *member-attributes* (sometimes also referred to as *template-slots* [90]) and constitute the class's *member-type*. An object's own-attributes resemble the member-attributes of its class. Each instance may (or must) instantiate its *own-attributes*. A class may instantiate its member-attributes. In this case we distinguish between *shared values* and *default values*, a shared value is valid for all instances of the class, a default value is only valid for instances that do not define another value for the respective attribute.

Classification provides for *inheritance* from class to instance. Member-attribute definitions, including associated value sets and, if defined, shared or default values are propagated from the class's set of member-attributes to the set of own-attributes of its instances. This relieves the modeler of redundant structure definitions. Default attribute values can be *overwritten*, that is, they can be replaced by another value.

Example 2.1 (Classification)**.** Consider our sample problem (see Example 1.1). We perceive our domain of interest (which are products sold by an imaginary company) as consisting of elements at different levels of abstraction, *physical entities*, *product models*, and *product categories*. Using traditional modeling techniques, the set of *product categories* can be represented as class *ProductCategory* with instances *Car* and *Book*, that is, {*Car*,*Book*} is the extension of class *ProductCategory*. In our example, product categories are only described by the tax rate associated with them. This is represented by member-attribute *taxRate* of class *ProductCategory*, which is instantiated at *Car* and *Book*. This example is depicted in Figure 2.1 using a simplified graphical notation: classes and objects are depicted as rectangles. Attributes of classes and of objects are depicted within these rectangles and, in order to distinguish between member-attributes and own-attributes, member-attributes are depicted below a horizontal line. A classification is depicted as dashed arrow from instance to class.

Note that we do not make a strong distinction between concrete domain entities, such as *a physical entity of Car*, and abstract domain entities, such

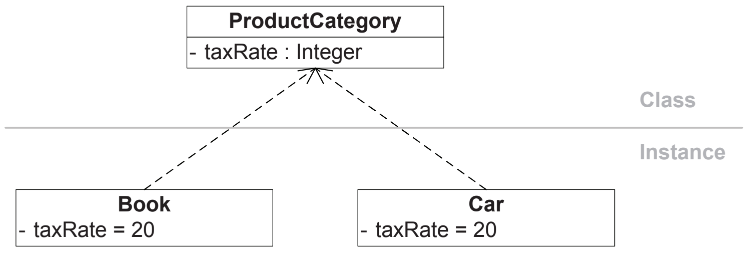


Figure 2.1: Classification

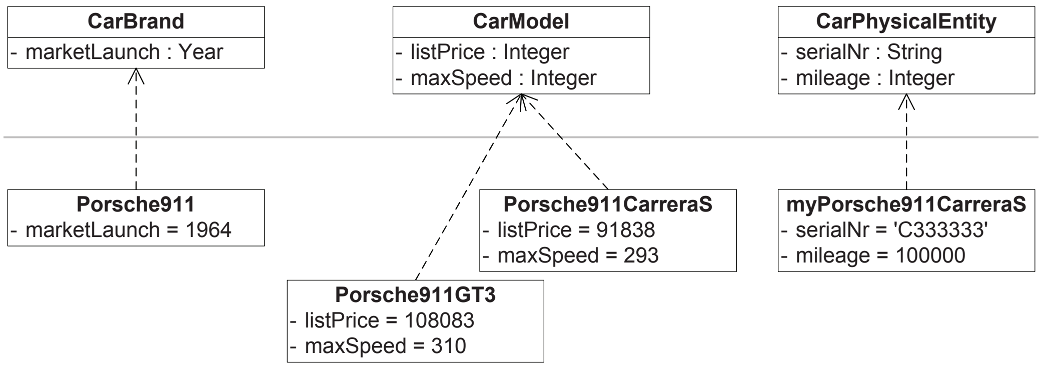


Figure 2.2: Class facets of product category *Car*

as *Car*. Concrete domain entities as well as abstract domain entities are first represented as objects that reside on the instance level (but having different principles of identity). Abstract domain entities, such as *Car*, may additionally play multiple *class roles*, such as *car model* or *car physical entity*. We also refer to these (implicit) classes as *class facets* on an individual.

Example 2.2 (Class Facets)**.** Consider again our sample problem (see Example 1.1). We perceive abstract entities, like product category *Car*, as entities in their own right, having different class facets, like *car brand*, *car model*, and *car physical entity*. Using traditional modeling techniques these different facets can be represented by classes. In Figure 2.2, the different class facets of product category *Car* are represented by classes *CarBrand*, *CarModel*, and *CarPhysicalEntity* having {*Porsche911*}, {*Porsche911CarreraS*, *Porsche911GT3*}, and {*myPorsche911CarreraS*}, respectively, as their extensions.

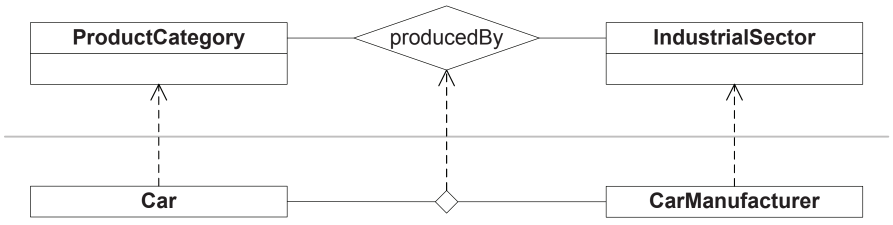


Figure 2.3: Classification of Relationships

The structure of a relationship is additionally defined by the *roles* played by the connected objects. These roles are typically defined by a relationship class. Roles may have names or be simply identified by the order of connected objects. Each role is associated with an object class. When instantiating a relationship class, the resulting relationship instance associates each role with an instance of the role's object class.

Example 2.3 (Classification of Relationships)**.** Consider again our sample problem (see Example 1.1). We are now interested in the producers of our products. When focusing on product categories we are interested in the industrial sectors in which the respective products are produced. This class of associations between product categories and industrial sectors can be represented by a relationship class, labelled *producedBy*, between object classes *ProductCategory* and *IndustrialSector* (see Figure 2.3). We do not explicitly model its roles, *product* and *producer*, but rather assume two generic roles *source* and *target* (not depicted). The relationship instance *producedBy* between objects *Car* and *CarManufacturer* is one of its instances. In order to be a valid relationship instance, the objects associated with its source role and its target role, *Car* and *CarManufacturer*, respectively, need to be instances of the object types associated with these roles, *ProductCategory* and *IndustrialSector*, respectively, which is given. In our simple graphical notation, relationships and relationship classes are depicted as labelled diamonds, where labels may be omitted for more compact representation.

When modeling relationships we do not distinguish between concrete and abstract relationships. We rather assume that relationships have different

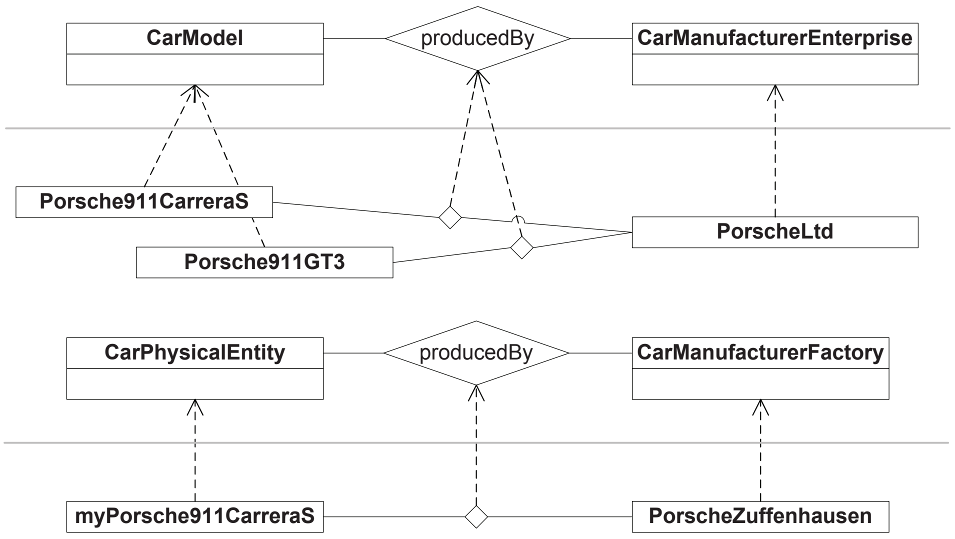


Figure 2.4: Relationship class facets of relationship *producedBy* between *Car* and *CarManufacturer*

facets, including relationship class facets. Using traditional modeling techniques, these relationship class facets may be again represented as relationship classes.

Example 2.4 (Relationship Class Facets)**.** Consider relationship *producedBy* between product category *Car* and industrial sector *CarManufacturer* which we introduced in Example 2.4. When looking at this relationship at more detail we may be interested in the connections between specific *car models* and their producing *enterprises* as well as in the *factories* that produce specific *car physical entities*. In Figure 2.4 these relationship class facets are modeled as relationship class between object classes *CarModel* and *CarManufacturerEnterprise*, and as relationship class between object classes *CarPhysicalEntity* and *CarManufacturerFactory*, respectively.

2.3 Generalization and Specialization

Generalization is an abstraction principle that allows to combine a set of more specific classes (subclasses) to a more generic class (superclass). A *generalization-relationship*, also referred to as *isA* or *subclassOf*, connects a subclass (child) with its superclass (parent). By applying generalization repeatedly, classes are organized in generalization hierarchies. If we want to emphasize top-down modeling of generalization hierarchies, we write *specialization*, which is the inverse to generalization.

Generalization is transitive, that is, all ancestors of a class in a generalization hierarchy are superclasses of this class. Thus, a generalization hierarchy is a transitive abstraction hierarchy. A class is *direct superclass* of another class if it is its parent in a generalization hierarchy. A class is *indirect superclass* of another class if it is its ancestor in a generalization hierarchy but not its direct superclass. *Single generalization* means that each class in a generalization hierarchy has at most one parent. *Multiple generalization* means that one class may have more than one direct superclass.

An instance of a class is also instance of this class's superclasses. An object is a *direct instance* of a class if it is not an instance of any of this class's subclasses. An object is *indirect instance* of a class if it is an instance of that class but not a direct instance of that class. We talk of *single classification* if each object is direct instance of at most one class and we talk of *multiple classification* if an object may be direct instance of more than one class.

A class and its superclass must *share their principle of identity*. We assume that classes having the same principle of identity are arranged in a single generalization hierarchy with a single root class. Root classes cannot be further generalized. Our abstraction levels (such as *physical entity*, *product category*, and *product model*) are such root classes. Each individual is (indirect) instance of exactly one such root class. A short discussion of these basic assumptions is provided in the paragraph on *ontological foundations of conceptual modeling* in Section 2.6.

Since each class has two facets, namely intension and extension, also generalization has an intensional as well as an extensional aspect. Intensionally, concerning the structure definitions in terms of member-attributes, a sub-

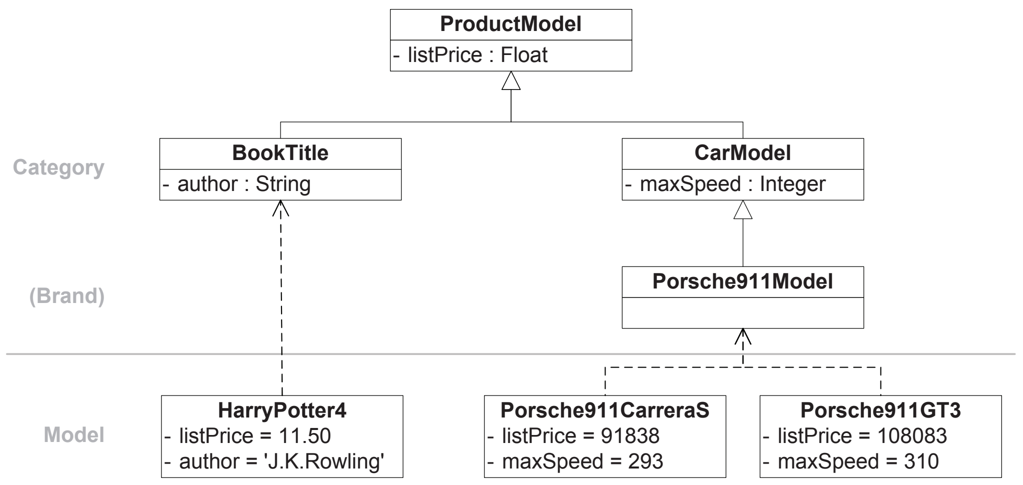


Figure 2.5: Generalization

class may introduce additional member-attributes and may refine existing member-attributes. Extensionally, the set of instances of a subclass is a subset of the set of instances of its superclass.

Generalization provides for *inheritance* from superclass to subclass. Member-attributes of a class are also member-attributes of its subclasses.

A *generalization schema* defines the structure of a generalization hierarchy. This is typically not explicitly modeled. Such a generalization schema divides the set of subclasses of a class into subsets at different *levels* (*taxonomic ranks*) or *partitions*. Such partitions of or levels of generalization hierarchies can be regarded as *metaclasses*, having classes as instances (see next section). *Powertypes*, which we will discuss in detail in Chapter 3, are one, however restricted, way to define such partitions. Well-known examples of taxonomies with explicitly modeled levels are biological taxonomies (see [77, 110]), where classes (also referred to as *taxa*, for example *Canis* and *Mammal*) are assigned to taxonomic ranks (for example *Genus* and *Class*, respectively).

Example 2.5 (Generalization)**.** Consider our sample product catalog (see Example 1.1) consisting of elements at different levels of abstraction: *physical entities*, *product models*, and *product categories*, and elements at some additional levels, such as *brand*, which exist only in certain sub hierarchies. One facet of this abstraction hierarchy is the set of elements at level *model* and their common structure, where this structure may differ between *product categories*. When focusing on this aspect of the product hierarchy, we consider abstraction level *product model* as instance level and are interested in classes of *product models* at higher levels of abstraction. Figure 2.5 depicts this generalization hierarchy of product model classes and their instances. The set of all product models is represented by class *ProductModel* (see Figure 2.5), it has as extension the set of product models in our product catalog and defines their common structure in terms of member-attribute *listPrice*. At abstraction level (taxonomic rank) *category*, class *ProductModel* is specialized to classes *BookTitle* (representing the class facet for level model of category *Book*) and *CarModel* (representing the class facet for level model of category *Car*). Since all book titles have an author, class *BookTitle* adds attribute *author* to its set of member-attributes. Since all *car models* are described in terms of their *maximum speed* it adds attribute *maxSpeed* to its set of member-attributes. The *Car* sub-hierarchy has an additional taxonomic rank *Brand*. At this level class *Porsche911model* (representing the class facet for level model of brand *Porsche911*) does not add further structure definitions. *HarryPotter4* is a direct instance of *BookTitle* and thus an indirect instance of *ProductModel*, and instantiates the inherited attributes. *Porsche911Model* has two direct instances, namely *Porsche911CarreraS* and *Porsche911GT3* which are indirect instances of *CarModel* and *ProductModel*.

Similarly, relationship classes can be generalized to more generic relationship classes. A relationship superclass connects the same classes or superclasses of the classes that are connected by the relationship-subclass. The extension (set of relationship instances) of a relationship class is a subset of the extension of its superclass's extension. Its intension, in terms of object classes assigned to its roles, is a refinement of its superclasses' intension.

Example 2.6 (Generalization of Relationships)**.** Consider again a fragment of our sample problem (see Example 1.1), namely *producedBy* relationships that connect physical entities with factories. When focusing on this aspect of

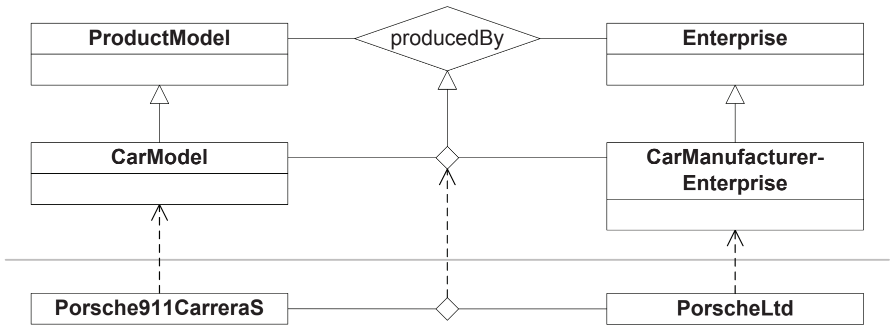


Figure 2.6: Generalization of Relationship Classes

produced by relationships we consider relationships that connect physical entities with factories as relationship instances and their abstraction at higher levels as relationship classes. The relationship class *producedBy* between *ProductModel* and *Enterprise* (*ProductModel* is the object type associated with role *source* and *Enterprise* is the object type associated with role *target*) has as extension all *producedBy* relationships between product models and enterprises, such as between product model *Porsche911CarreraS* and factory *PorscheLtd*. This relationship class can be specialized to a *producedBy* relationship class between *CarModel* and *CarManufacturerEnterprise*.

2.4 Aggregation and Decomposition

Aggregation is an abstraction principle that allows to combine a set of *parts* to a *whole*. Aggregation is often also discussed under different names such as *part/whole*, *weak entity*, *composition*, or *meronymic* relation. On the instance-level, *partOf* relates an object (part, component object) to another object (whole, composite object) of which it is, in some regard, a part. On the schema-level, *partOf* relates an object class (component class) to another object class (composite class) where instances of the part class are part of instances of the whole class. We also use the notion of *aggregation schema* to refer to a hierarchy of component and composite classes.

The precise meaning of aggregation in conceptual modeling is subject to a still ongoing discussion (see Section 2.6). In this thesis we use the notion of aggregation in a very generic form with regard to the different kinds of parthood, such as for example *component/functional complex* or *member/collection*. Note that the relationship between an instance and its class's extension can also be regarded as a kind of *member/collection* relationship. We employ a precise formal meaning by using the notion of aggregation synonymously with a *transitive*, *antisymmetric*, and *irreflexive* relation having the following further properties:

- *Functional dependency:* A partOf relationship between a component class and a composite class means that each instance of the component class is part of exactly one instance of the composite class.
- *Existential dependency:* A partOf relationship between a component object and a composite object means that the component object cannot exist without its composite object. That is, we only consider *inseparable parts* (as defined by Guizzardi [39]).
- *A whole is more than its parts:* That is we do not employ an *Extensional Mereology*, where a whole is considered as being equal to the set of its parts.
- *Aggregation preserves context:* As Guizzardi [39] states: "*. . . contexts demarcate the scopes in which transitivity can be guaranteed to hold*". Since we assume transitivity of aggregation, we assume that every element in an aggregation hierarchy shares a context with each of its descendants.

Concerning context in aggregation hierarchies we assume that when something is modeled as part of a whole then it has to be interpreted in at least one *context* of the whole (we assume that model elements can have more than one context). To illustrate this point, consider class *Person* modeled as part of *Organization*. This seems obviously wrong since a person can be part of multiple organizations and since the lifespan of a person is not restricted by its membership to an organization. However, assuming that the modeler is a sensible person, this partOf relationship helps us to understand what the modeler actually wanted to represent by class *Person*, namely persons in the context of an organization, that is in their *roles* as members of specific organizations. While such a *person role* is existentially dependent on the *organization* it belongs to, the person that plays this role is not. A *role*, in this sense, refers to a facet of an individual that is of interest only in a specific context (see Section 2.6 for interesting literature on roles).

We further assume that every aggregation hierarchy has a single root element. This root element provides a context that is shared by all its direct and indirect parts and thus transitivity holds. Consider, for example, the city of Linz is part of Austria and Austria is part of the European Union. While we cannot say, in general (i.e., in every context), that Linz is part of the European Union. If we assume *Geographic Location and Year 2010* as the context of the hierarchy, then it is obvious that Linz is also part of the European Union.

Example 2.7 (Aggregation)**.** Consider again our sample company with its three-level product hierarchy (see Example 1.1) consisting of product descriptions at least at three levels of abstraction, namely at levels *category*, *model*, and *physical entity*, where each *product phyiscal entity* belongs to one *product model* and each *product model* belongs to one *product category*. To represent this abstraction hierarchy we use aggregation as defined above and model class *ProductPhysicalEntity* as component class of composite class *ProductModel*, and *ProductModel* as component class of composite class *ProductCategory* (see left part of Figure 2.7). Our simplified graphical notation denotes aggregation relationships (as defined above, not in the UML sense of weak aggregation) by an edge with a diamond at the end

Note that our notion of aggregation does not exclude component classes from having multiple composite classes. In this case, each instance (component object) of the component class has to be part of multiple composite

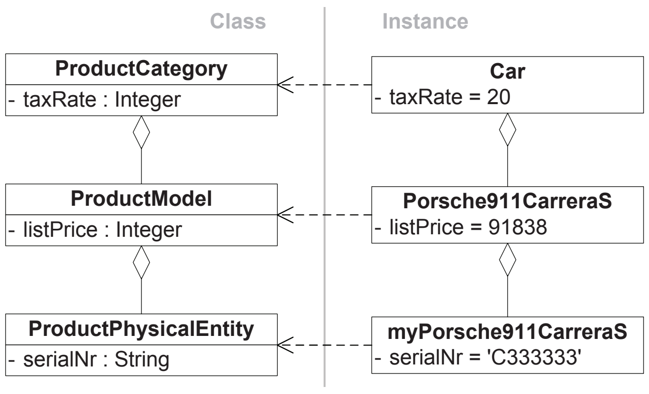


Figure 2.7: Product hierarchy modeled as aggregation hierarchy

objects. With regard to a specific composite class, each component object exclusively belongs to one composite object. This form of *exclusive part-whole relation* is also promoted by Guizzardi [39].

Aggregation can be applied, with the same properties, to relationship instances and relationship classes. A relationship may be part of another relationship or be part of an object. This means that the component relationship is existentially dependent on the composite relationship or the composite object, respectively. A relationship class may be a component of another relationship class or an object class, meaning that each of its relationship instances is part of an instance of the component relationship class or of an instance of the component object class, respectively. This typically gives rise to the need of local referential integrity, as discussed by Kappel and Schrefl [60]. In this thesis we discuss an extended form of aggregation between component relationships and their composite relationships. In the case of aggregation between relationships it must hold – in addition to the dependencies implied by a partOf relationship between objects – that the two relationships have the same roles and that for each role the associated object of the component relationship is a component of the object associated with that role at the composite relationship.

Example 2.8 (Aggregation of Relationships)**.** Consider again our sample problem (see Example 1.1). We are interested in the producers of our prod-

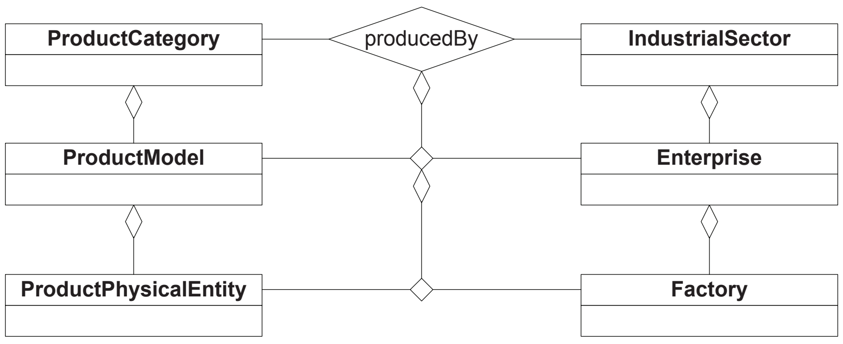


Figure 2.8: Aggregation of Relationships

ucts at different levels of abstraction. We have producedBy relationships between product physical entities and factories, each of these relationships has to be part of exactly one producedBy relationship which connects a product model, of which the physical entity is part of, with an enterprise of which the factory is part of. The same dependencies hold to the next higher level of abstraction where producedBy relationships connect product categories with industrial sectors. This can be modeled by aggregation between relationship classes as just described. Relationship class *producedBy* between *ProductCategory* and *IndustrialSector* has relationship class *producedBy* between *ProductModel* and *Enterprise* as component which in turn has relationship class *producedBy* between *ProductPhysicalEntity* and *Factory* as component.

2.5 Meta-Classification

Classification can not only be applied to objects and relationships but also to object classes and relationship classes and results in object metaclasses and in relationship metaclasses, respectively. Models that consist of metaclasses and relationship metaclasses are referred to as *metamodels*. Classification is *antitransitive*, that is, an object (at the instance level) is never instance of a metaclass (this is one of the properties of strict metamodeling [8]). When modeling with multiple classification levels, these levels are typically predefined as *instance-level*, *class-level* (schema-level), and *metaclass-level* (metamodel, metaschema). The instances of metaclass are its *member classes.*

Remember that each class has two aspects, namely its set of instances (*extension*) and the definition of the common structure of its instances (*type*, intension). Also consider that —when modeling with metaclasses— a class is also often regarded as an object on its own, having own-attributes that describe the class itself and not its instances. Thus, the notion of *metaclass* has various aspects, which are outlined in the following:

- A *metaclass* as an *object* describes itself by its own-attributes.
- The *extension* of a metaclass is the set of its member classes.
- The *meta-extension* of a metaclass is the set of extensions of its member classes and, thus, a set of sets of individuals.
- The set of *member-members* of a metaclass are given by the union of the sets in its meta-extension.
- The *member-type* of a metaclass defines the common structure of its member classes in terms of member-attributes, which are, in turn, instantiated by own-attributes of its member classes.
- The *member-member-type of a metaclass* defines the common structure of its member-members, in terms of *member-member-attributes*, .
- The *member-metatype of a metaclass* defines the structure of the structure definitions (types) of its member classes, typically in terms of available language constructs and in terms of constraints on the permissible use of these language constructs. Note that the member-metatype indirectly also influences the structure of its member-members.

Approaches to modeling and programming with metaclasses emphasize one or more of these different aspects. In this thesis we are mainly concerned with the extensional aspects of metaclasses. In comparison with modeling without metaclasses, considering extensional aspects of metaclasses provides for more expressiveness regarding query and constraint formulation.

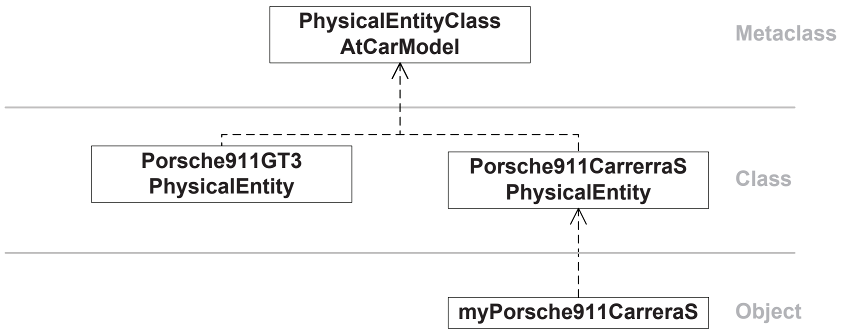

PhysicalEntityClass AtCarModel
Metaclass
Porsche911GT3 PhysicalEntity
Porsche911CarrerraS PhysicalEntity
Class
myPorsche911CarreraS
Object

Figure 2.9: Meta-Classification

Example 2.9 (Meta-Classification)**.** Consider again our sample product catalog (see Example 1.1). Product category *Car* has two product models, namely *Porsche911CarreraS* and *Porsche911GT3*. They have class-facets which have as extension the set of physical entities of *Porsche911CarreraS* and of *Porsche911GT3*, respectively. These class-facets may be explicitly represented by classes *Porsche911CarreraSPhysicalEntity* and *Porsche911GT3PhysicalEntity*. These two class-facets of descendant elements of *Car* which have physical entities as their instances and which belong to car models may now be classified to a metaclass-facet of product category *Car*. This metaclass-facet may be explicitly represented by metaclass *PhysicalEntityClassAtCarModel*. (see Figure 2.9)

Analogously to object classes and metaclasses, relationship classes may be further classified to relationship metaclasses. When classifying a relationship class, the resulting relationship metaclass associates with each role a metaclass of the object class which is associated with this role. And vice versa, when instantiating a relationship metaclass, the resulting relationship class associates with each role a member class of the metaclass associated with this role.

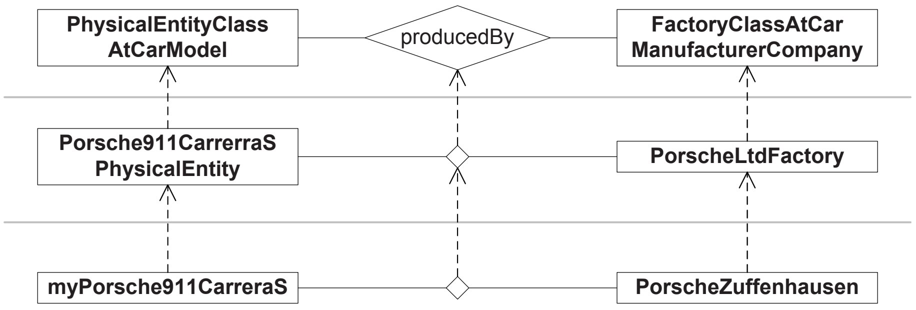

PhysicalEntityClass AtCarModel
producedBy
FactoryClassAtCar ManufacturerCompany
Porsche911CarrerraS PhysicalEntity
PorscheLtdFactory
myPorsche911CarreraS
PorscheZuffenhausen

Figure 2.10: Meta-Classification of Relationships

Example 2.10 (Meta-Classification of Relationships)**.** Consider again our sample product catalog (see Example 1.1). Relationship *producedBy* between between product category *Car* and industrial sector *CarManufacturer* has one descendant *producedBy* relationship at level *Model* to *Enterprise*, namely between *Porsche911Carreras* and *PorscheLtd*. This relationship has a relationship-class-facet which has descendant relationships at level *physical entity* to *factory* as its instances. This relationship-class-facet can be further classified to a relationship-metaclass-facet of relationship *producedBy* between *Car* and *CarManufacturer*. It is explicitly modeled as a relationship metaclass between object metaclasses *PhysicalEntityClassAtCarModel* and *FactoryClassAtCarManufacturerCompany*.

2.6 Related Work

In this section we give a brief overview of further aspects of and related work on abstraction in conceptual modeling. A detailed comparison of three multi-level modeling techniques, namely *powertypes*, *potency-based deep instantiation*, and *materialization*, is provided in Chapter 3. For an overview of the classical work on the basic abstraction principles in semantic data modeling see [109, 54, 97].

Ontological Foundations of Conceptual Modeling Guarino [36] introduced the *ontological level* to characterize modeling languages "*taking into account the intended meaning of its primitives*". This is in contrast to languages on the logical as well as on the epistemological level. Logical languages, such as *first-order predicate logic* or the *relational data model*, are neutral with regard to what is represented by their language constructs (they are *ontologically neutral*). Languages on the epistemological level, such as object-oriented, frame-based, as well as description-logic-based languages, provide additional structuring mechanisms. While these structuring mechanisms are often used with a certain meaning (real-world semantics) in mind, this meaning remains vague and unclear.

Work on *formal ontology* tries to overcome these shortcomings and provides "*highly general ontological notions drawn from philosophical ontology, especially what is now called analytic metaphysics.*" [38] that can be used to better describe and categorize a domain of interest. In their work on *OntoClean*, Guarino and Welty [38] give a concise account of some of these basic ontological notions (*essence*, *rigidity*, *identity*, and *unity*) that can be used to evaluate modeling decisions. Ontological analysis, such as provided by Guizzardi [39], not only draws from philosophy, but also from psychology and cognitive science to analyze modeling language constructs with regard to their adequacy for representing real-world phenomena (ontological adequacy).

In this chapter we referred to Guizzardi's work on ontological foundations of structural conceptual models [39] in order to underpin some of our basic assumptions. We especially used the notion of *principle of identity* to clarify our approach. We only considered classes which carry a *principle of identity*. Such classes are also referred to as *Sortals* [39]. According to Guizzardi [39], it is uncontested in the philosophy of language that "*the identity of an individiual can only be traced in connection with a Sortal Universe, which provides a* principle of individuation *and* identity *to the particulars it collects*". Thus, we assumed that every individual has to be an instance of a class which represents such a Sortal. In contrast, *Non-Sortals*, such as *Red* or *Young*, only provide attribution but no principle of identity. We assumed that *non-sortals* are not represented as classes but rather by other modeling constructs, such as attributes. We further assumed that all classes that have the same principle of identity are organized in a single generalization hierarchy with a single root class, also referred to as *Substance Sortal* [39]. Such a substance sortal cannot be further generalized. In line with Guizzardi [39], we also assumed that each individual in a conceptual model must be instance of exactly one such substance sortal.

Part/Whole There has been a lot of research concerning different kinds of aggregation, especially concerning the question whether aggregation is transitive. Wilson et al. [131], as cited in [39], proposed six types of parthood: component-functional complex (e.g, pedal-bike), member-collection (e.g, ship-fleet), portion-mass (e.g., slice-pie), stuff-object (e.g., steel-bike), feature-activity (e.g., paying-shopping), and place-area (e.g., Everglades-Florida).

More recently, Guizzardi [39] analyzed the ontological foundations of parthood and reduced the set of kinds of aggregation to: (1) *subQuantityOf* connecting two quantities, (2) *subCollectionOf* connecting two collections of objects, (3) *memberOf* connecting an object with a collection of objects, and (4) *componentOf* connecting a component with a functional complex. Examples for these kinds of part/whole are provided in Table 2.1. Guizzardi [39, 41] further analyzes transitiveness of these four kinds of part/whole relationships and comes up with the following results: *subQuantityOf* is always transitive, *subCollectionOf* is always transitive, and *memberOf* is antitransitive (never transitive). *componentOf*, arguably the most-interesting kind of aggregation relationship, is *sometimes* transitive, that is, some *componentOf* relations are transitive and others are not. A combination of *memberOf* and *subCollectionOf* is again always transitive.

	subQuantityOf	memberOf	subCollectionOf	componentOf
Roles in Relationship	Quantity – Subquantity	Collective – Member	Collective – Subcollective	Functional Complex – Component
Instance Example	the cappuccino i just had – the portion of milk in this cappuccino	Manchester United – Wayne Rooney	Black Forest – north part Of Black Forest	me – my left arm – my left hand
Schema Example	cappuccino – milk	soccer team – soccer player	forest – part of forest	person – arm– hand

Table 2.1: Different kinds of part/whole [39]

Note that memberOf is often considered as an abstraction principle on its own and is also known as *grouping* [82] or *association* [19].

The UML distinguishes between whole/part relationships with shareable parts (*aggregation in UML*) and with non-shareable parts (*composition in UML*). In line with Guizzardi [39] we think that these two kinds of aggregation are not suitable for a lot of modeling tasks, since aggregation in UML is too weak and composition in UML is too strict.

Linguistic vs. Ontological Metamodeling Atkinson and Kühne [9] introduced the distinction between ontological metamodeling and linguistic modeling. *Ontological metamodeling* refers to describing complex domains at multiple classification levels, especially in domains where the borderline between individuals and classes is not clear cut. *Linguistic metamodeling* refers to representing modeling language constructs in one or more higher levels, or meta*-schemas, thereby defining the syntax of a modeling language. To illustrate the difference between ontological and linguistic metaclassification consider a class *Dog* with instance *Lassie*. *Dog* can be regarded as ontological instance of ontological metaclass *Species* as well as linguistic instance of linguistic metaclass *Class* (*Class* as a UML language construct).

In the context of ontological metamodeling, Atkinson and Kühne [9] do not —in contrast to our conceptual framework— distinguish between abstract domain concepts, such as *Dog*, and their class facets, such as *DogIndividual*, but rather regard them as identical. We regard this as a shortcoming, since it hinders modeling of abstract domain concepts having multiple class facets, such as *Dog* having class facet *DogBreed* as well as class facet *DogIndividual*. We will further discuss this issue in the context of *ranked taxonomies* (see below).

Linguistic metamodeling is prevalent in model-driven engineering. Constructs of modeling languages are defined in terms of linguistic metaclasses in a linguistic metamodel. Most approaches to linguistic metamodeling focus on one aspect of metaclasses, namely on their role as *member-metatypes* (see Section 2.5). They do typically not consider other typing aspects of metaclasses, such as *member-type* and *member-member-type*, nor considering extensional aspects of metaclasses. The most prominent framework for linguistic metamodeling is the Meta-Object Facility (MOF) [95]. It has been used to define modeling languages such as the Common Warehouse Model (CWM).

Atkinson et al. [10, 6] argue for the *orthogonal classification architecture* which not only allows for linguistic but also for ontological metamodeling, as a basis for model-driven engineering. Henderson-Sellers and colleagues [32] argue for extensive use of powertypes (a meta-level construct we will look at in Chapter 3) in model-driven engineering.

Ranked Taxonomies A ranked taxonomy is a generalization hierarchy, where each class (also referred to as *taxa*), is assigned to a specific level (also referred to as *taxonomic rank*). Well-known examples of ranked taxonomies are biological taxonomies (see [77, 110]), often also referred to as *Linnaean taxonomy*. The elements in a biological taxonomy (e.g., *Dog*, *Animal*) are assigned to taxonomic ranks (e.g., *Species*, *Kingdom*).

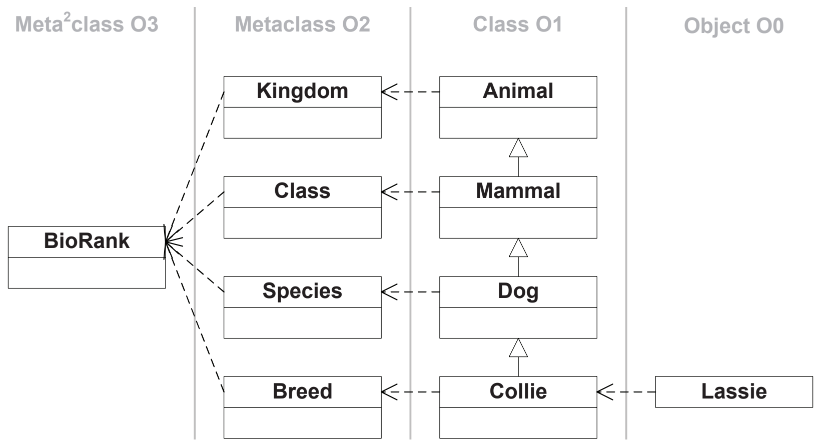


Figure 2.11: Modeling Ranked Taxonomies with Ontological Metaclasses

Atkinson and Kühne [9] promote the use of ontological metaclasses in conjunction with generalization for modeling ranked taxonomies. Taxonomic ranks, as in a biological taxonomy, are represented as ontological metaclasses (see Figure 2.11 for a simplified version).

We argue that ontological metaclasses are not adequate for the task of modeling ranked taxonomies. This is, first, because ontological metaclasses, as such, do not allow for explicitly modeling of hierarchies of levels. Second, this pattern assumes an agreed-on instance level (which in Figure 2.11 is assumed to be given by *individual creatures*), which hampers re-use (see below for a short discussion of the class vs. instance issue). The depicted specialization hierarchy (*Collie subclassOf Dog subclassOf Animal*) only allows to specialize the structure of instances (like *Lassie*) at level *individual creature*. This pattern does not, as such, provide for specializing the structure of objects at level Breed belonging to species *Dog*, in order to introduce an additional attribute *watchdog ability* for all *dog breeds*. Such a specialization of *Breed* to *DogBreed* needs to be introduced explicitly (and redundantly with the specialization at level *individual creature*).

Furthermore, when applying *simple metaclass compatiblity rules*, as Kühne does in his work on DeepJava [66], one ends up with a taxonomy as depicted in Figure 2.12. This is definitely not what we intended, since

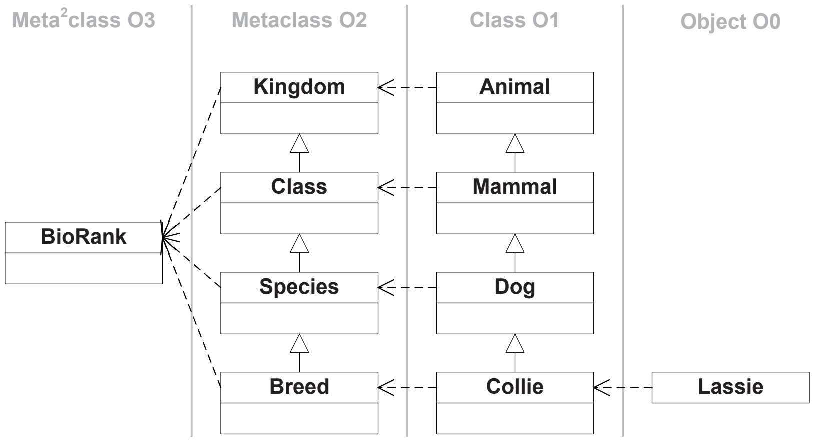


Figure 2.12: Modeling Ranked Taxonomies with Ontological Metaclasses obeying simple metaclass compatibility rules.

this would imply that class *Collie* is not only instance of metaclass *Breed*, but also instance of metaclass *Species*, of metaclass *Class*, as well as of metaclass *Kingdom*.

Class vs. Instance In (conceptual) modeling and ontology engineering one often has to decide whether a particular element is to be represented as class or as instance. Concerning this issue, Noy and McGuinness [89] state "*Deciding whether a particular concept is a class in an ontology or an individual instance depends on what the potential applications of the ontology are*". Consequently, partitioning a set of domain concepts into instances (such as *Lassie*) and classes (such as *Collie*) hampers reuse of structural models for different applications.

It is sometimes assumed that an absolute instance level (a level of elements that cannot be further concretized) is constituted by physical objects. We do not make this assumption. We rather assume that nearly every physical object can be further concretized, for example by decomposing an individual into smaller parts, by looking at it at different points in time (its *phases*[39] or states), or by looking at different *roles* it plays. We illustrate this subtle point by the following example given by Hofstadter [48, p.357].

'It is interesting to wonder if the cows on a farm perceive the invariant individual underneath all the manifestations of the jolly farmer who feeds them hay.'

Strict Metamodeling Atkinson and Kühne argue for *strict metamodeling* [8], and oppose it to loose metamodeling. In strict metamodeling, only instanceOf relationships cross classification level boundaries and every instanceOf relationship crosses exactly one classification level boundary. Other relationships must not cross classification levels.

We fully agree with Atkinson and Kühne and regard strictness as an important property of multi-level models. We think, however, that strictness is hard to maintain when modeling with ontological metaclasses, as promoted by Atkinson and Kühne. For this reason, we argue that each domain entity and domain concept should be represented as object at the instance level and additionally may have multiple class- and metaclass-facets, which reside on the class- and the metaclass-level, respectively (as exemplified in this thesis). Relationships may also have such multiple facets. Class-facets of objects may only be connected by class-facets of relationships and metaclass-facets of objects may only be connected by metaclass-facets of relationships.

Role/Player Another important abstraction principle not discussed in detail in this thesis is *Role/Player* which allows to decompose an individual into the different roles it plays in different contexts. A roleOf relationship connects a *role* with its *player*. A *role*, in this sense, is an *adjunct instance* that is existentially-dependent on its player, which can be another role or an individual. *roleOf* is transitive, antisymmetric, and irreflexive. Each role is – directly or indirectly – played by exactly one individual. A role carries information about its player in a specific context. It is typically assumed that information which is carried by the player is also available at its roles. A role is instance of a *role class* which is associated with a *player class* which is either another *role class* or an *object class*. We assume that a *role class* carries a *principle of identity*, given by the principle of identity of its player class combined with the context of the role. We assume that a single object may play multiple instances of a single role class, e.g., a person may play multiple student roles, one for each university it is enrolled to. Since we assume that a role class has its own principle of identity (dependent on the principle of identity of the player class) it is should be easy to incorporate roles into our conceptual framework.

By the means of roles one can include objects in aggregation hierarchies (with functional and existential dependency) without the object being dependent on the hierarchy. For example, *a person* can play various *student roles*, each such *student role* is part of a specific *university* (which provides the context of the role). While the *student role* is existentially dependent on the *university* it is enrolled to, the *person* itself is not.

Our notion of *roleOf* is based on the work of Gottlob et al. [33], which has been implemented in Smalltalk and in Java [108]. More recently, we applied it to the Web of Data, as a syntactical extension [53] of the Ressource Description Framework, as well as to Web Engineering as an extension [26] of *Ruby on Rails*. Note that there are many different notions of roleOf in the conceptual modeling literature. Steimann [118] provides an insightful overview.

Abstraction Hierarchies Our intuition of abstraction hierarchies is best illustrated by Example 2.11, originally introduced by Hofstadter [48]. This example highlights two important properties of abstraction hierarchies as discussed in this thesis. First, one element in an abstraction hierarchy may play multiple roles with regard to different abstraction principles, and, second, the distinction between concrete and abstract domain entities and concepts is blurred.

Example 2.11 (Abstraction Hierarchy)**.** A newspaper may be described at different levels of abstraction:

1. Publication
2. Newspaper
3. The San Francisco Chronicle
4. the May 18 edition of the Chronicle
5. my copy of the May 18 edition of the Chronicle
6. my copy of the May 18 edition of the Chronicle as it was when I first picked it up (as contrasted with my copy as it was a few days later: in my fireplace, burning)

In this example *newspaper* may be regarded as the class of all newspaper titles, the class of all newspaper editions, and the class of all newspaper copies. *The San Francisco Chronicle* is first an instance of *newspaper* and, concerning newspaper editions and newspaper copies, it represents subclasses of the respective newspaper classes. While it is often assumed that the absolute bottom level in an abstraction hierarchy is the physical entity level, this is not absolute, as illustrated by lines 5 and 6 of the newspaper example.

Combining abstraction principles In conceptual modeling, some approaches combined aggregation with aspects of specialization and instantiation. Kappel and Schrefl [60] proposed to use local classes, to specialize component classes of instances of composite classes. They also provided for object classes with component relationship classes, especially discussing referential integrity of component relationships. Troyer and Janssen [128] embed schemas into Schema Object Types and discuss its implication on subtyping.

Telos and ConceptBase Early work on metamodeling has been done by Mylopoulos and colleagues, especially their work on *Taxis* [83, 85], its successors *Telos* [84] and *ConceptBase* [57, 58] strongly influenced our work.

Recently, Jeusfeld [59] gave an overview of metamodeling and method engineering with ConceptBase. Concerning the formalization of Telos, the most important principle is: every model element is both an *object* (having an OID) and a *proposition* (having *source*, *target*, and *label*). Figure 2.13 shows the different kinds of objects/propositions. In this approach, classes (and meta$^+$-classes) are also objects. Thus, they can also have own-attributes which describe the class (as an object) and not the structure of its instances. To address this duality, Atkinson [5] later coined the term *Clabject* (short for class-object) to denote objects that both have member-attributes as well as own-attributes.

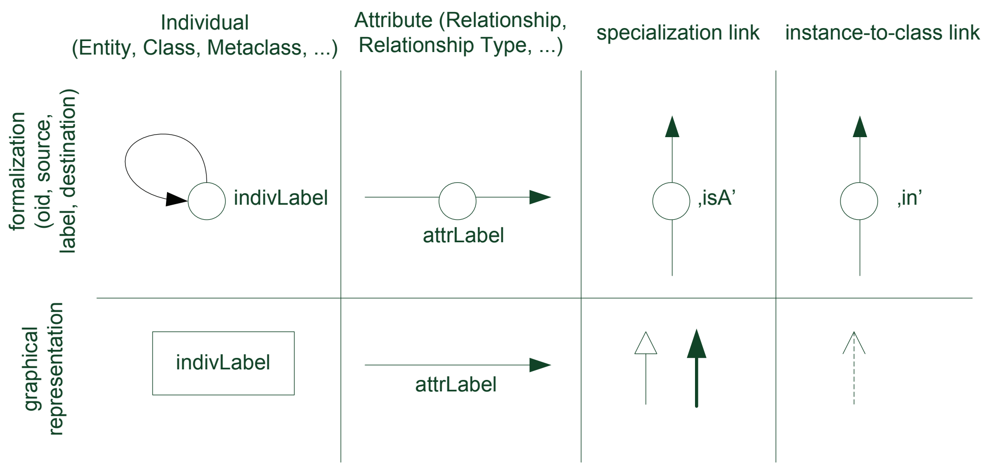


Figure 2.13: Telos: different types of objects/propositions (from left to right): individual, attribute, isA, instanceOf

Deep Instantiation Metamodeling through meta classes that provide for deep instantiation has been first put forward in the area of databases by the VODAK system [63]. VODAK associated several types with an object, the own-type describing itself (own-attributes), the instance-type describing objects at one classification level below (member-attributes), and the instance-instance-type describing objects at two classification levels below (member-member-attributes). In this way, meta classes in VODAK allow to describe not only common properties of instances but also common properties of instances of instances. The idea of deep instantiation has been later elaborated by Atkinson and Kühne [7, 11]. Rather than using different types for each level as in VODAK, attributes definitions are annotated with a potency, where this potency denotes the instance level. This approach is discussed in detail in Chapter 3.

Aschauer et al. discuss how multi-level models that employ the Orthogonal Classification Architecture (OCA) and potency-based deep instantiation can be queried efficiently [3]. They apply OCA in the domain of testbed automation systems and discuss how the system architecture of Model Driven Engineering tools is influenced by OCA [4]. They adjust and extend 'classical' OCA by relaxing the strictness criterion, and also identify the need to represent relationships at multiple levels and to instantiate and specialize them [2]. Their reluctance to adhere to the strictness criterion of OCA highlights a general problem of ontological metamodeling with potency-based deep instantiation. By combining a superclass and a metaclass in one metaclass, potency based deep instantiation combines metaclassification and generalization, leading to problems as described in [3]. These problems are due to representing object- and class-facets at the metaclass level, which is, at least in our opinion, not coherent with strict metamodeling.

The Higher-Order Entity-Relationship Model The higher-order entity-relationship model (HERM) [122] extends the classical entity-relationship model by complex attributes (type), relationship types of higher order and cluster types. Relationship types may have only one component and represent in this case a specialization of its component.

The Component Model The component model [126] (also see [100, 125, 123]) enhances the extended ER model by explicit introduction of an encapsulation based on component schemata, views defined for component schemata with or without update propagation through these views, and ports to components that allow an exchange of data among components based on the view assigned to the port. Components may consist itself of components. Views of inner components may be visible also to outer components. The abstraction principle provided by the component model is orthogonal to our work.

2.7 Summary

In this section we clarified how we use basic notions and abstraction principles of (conceptual) modeling. We provided a concise account of the classical abstraction principles – classification, generalization, and aggregation – and how we understand and apply them. An overview is given in Table 2.2. We especially discussed generalization, aggregation, and meta-classification of relationships. Summarizing, in this thesis we make the following basic assumptions:

1. Every class provides a *principle of identity*, which is typically not explicitly modeled. A subclass and its superclass have the same principle of identity. A component class often has another principle of identity as its composite class.

2. Generalization hierarchies as well as aggregation hierarchies are *transitive abstraction hierarchies.* Classification hierarchies are *anti-transitive abstraction hierarchies.*

3. We strictly distinguish between *individuals* (single elements), *classes* (sets of elements and their common structure) and *metaclasses* (sets of classes, sets of sets).

4. *Strict metamodeling*: Every classification relationship crosses exactly one classification level boundary. All other relationships (generalization, aggregation, as well as custom relationships) may only connect elements at the same classification level.

	Classification Instantiation	Generalization Specialization	Aggregation Decomposition
Generic Relationship	instanceOf	subclassOf	partOf
Roles in Relationship	class instance	superclass subclass	whole part
Example	ModelClassOf-ProductCategory CarModel Porsche911CarreraS	ProductPhysicalEntity CarPhysicalEntity Porsche911-PhysicalEntity	Car Porsche911CarreraS myPorsche911CarreraS
Schema	predefined	taxonomic rank	aggregation hierarchy at schema level
Schema Example	metaclass class instance	Root Category Model	ProductCategory ProductModel ProductPhysicalEntity
Transitivity	Anti-Transitive	Transitive	Transitive

Table 2.2: Abstraction Principles

5. *Aggregation* is a *transitive*, *antisymmetric*, and *irreflexive* relation that connects components with composites, where the component is *existentially dependent* on the composite. On the schema level, aggregation also represents a *functional dependency* between component class and composite class.

6. Every aggregation hierarchy has one root element that (implicitly) provides a *context* that is shared by all elements in the hierarchy.

7. *Relationships* can be likewise abstracted to relationship classes, relationship superclasses, relationship metaclasses, and composite relationships.

Concerning representation of hetero-homogeneous abstraction hierarchies we make the following conclusions. These conclusions guide our research on multi-level objects and multi-level relationships, as presented in the remainder of this thesis.

1. An *abstraction level* provides a unique principle of identity. Two abstraction levels in an abstraction hierarchy must not share their principle of identity.

2. Since classification is anti-transitive, we do not regard classification levels as levels in hetero-homogeneous abstraction hierarchies (which are transitive), but rather regard classification levels as being orthogonal to abstraction hierarchies.

3. Abstraction levels can be represented by classes arranged in aggregation hierarchies, where each class has a unique principle of identity. The order of abstraction levels can be represented by partOf relationships between classes.

4. Elements at different levels of abstraction are, at first, individuals. Additionally they may have multiple *class-* and *metaclass facets* which can be, using classical abstraction principles, represented as classes and metaclasses.

5. We exemplified how hetero-homogeneous hierarchies can be modeled using the classical abstraction principles alone. While this is possible, it is *impracticable* due to the sheer amount of resulting classes and their redundant arrangement in generalization hierarchies.

Instead of partitioning the set of elements of a multi-level model into instances, classes (sets), metaclasses (sets of sets), and so forth, we regard elements in a multi-level model as *multi-faceted constructs*. Every model element has an *instance-facet* describing itself (and the part of the original it represents) and may have multiple *class-facets*, defining common structure of descendants at a given level (intension) and providing entry points for querying them (extension). Model elements also have *meta*$^+$*-class-facets* that have sets of sets (of sets . . .) as extension.

The remainder of this thesis is dedicated to multi-level objects and multi-level relationships. A modeling technique which combines the discussed abstraction principles in a concise formalism that allows for compact representation of hetero-homogeneous abstraction hierarchies. The main idea of this approach is to represent an object (such as product category *Car*) together with its class-facets (such as *CarBrand*, *CarModel*, *CarPhysicalEntity*) by a single m-object (such as m-object *Car*) and to arrange these m-objects in a so-called concretization hierarchy. A concretization hierarchy resembles a classical aggregation hierarchy but additionally comprises multiple generalization hierarchies (concerning class facets of the represented elements). This also incorporates aspects of metaclassification. The same principle applies to relationships.

Chapter 3

Comparison of Multi-Level Modeling Techniques

Contents

In this chapter we compare several modeling techniques and discuss their suitability for multi-level abstraction. We cover Powertypes, Potency-based Deep Instantiation, and Materialization. We demonstrate each technique by our example and analyze it according to the requirements introduced in Chapter 1. We give an overview of further related work and conclude this chapter with an overview of the evaluation.

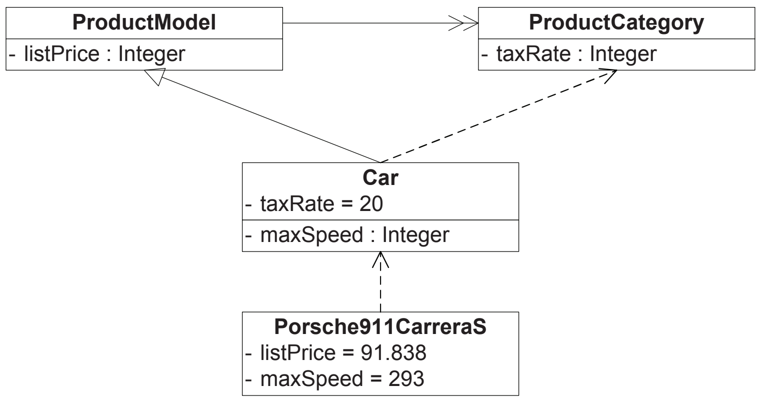


Figure 3.1: Powertype *ProductCategory*

3.1 Powertypes

A *powertype relationship* connects a class (called *partioned class*) with another class (called *powertype*). Each instance of the powertype is a subclass of the partitioned class. Thereby, powertypes can be regarded as metaclasses that additionally define a common superclass for their instances, thereby they can prescribe common structure for their instances' instances. Powertypes were introduced by Odell [91]. We will first give a simple example of the use of powertypes and will then try to model our sample problem.

Example 3.1 (Powertype)**.** Consider the following fragment of our sample problem: Every *product model* has a *listPrice* and belongs to a *product category* which defines a *taxRate*. For each *product category* the structure of the representation of *product models* may be specialized. For example, product models belonging to category *Car* are additionally described by their *maximum speed*.

Such a simple abstraction hierarchy can easily be represented by two classes, *ProductModel* and *ProductCategory* (see Figure 3.1) where *ProductCategory* is powertype of partitioned type *ProductModel* (we depict a powertype relationship by a two-headed arrow from partitioned type to powertype). *Car* is an instance of *ProductCategory*, for that reason it is also a subclass of *ProductModel*. Apart from instantiating attribute *taxRate* with value *20*, it introduces an additional member-attribute *maxSpeed*. *Porsche911CarreraS* instantiates *ProductModel* and instantiates attribute *listPrice* which was introduced at *ProductModel* as well as attribute *maxSpeed* which was introduced at *Car*.

To model a multi-level hetero-homogenous hierarchy, each abstraction level is represented by a class. Each such class is connected by a powertype relationship with the class that represents the next higher abstraction level. Consequently, an instance of a class, which represents some level, is also a subclass of the class that represents the next lower level. However, this only allows to specialize the structure of elements at the next lower abstraction level. Modeling of hetero-homogeneous hierarchies, as discussed herein, also requires to be able to specialize the structure of descendant elements below the next lower level of abstraction. To do so, one has to manually introduce subclasses of classes that represent lower abstraction levels. Additionally one has to connect these manually introduced subclasses by powertype relationships. An additional abstraction level in some sub-hierarchy can be modeled by introducing a new class that does not specialize an existing class and represents the additional abstraction level in this sub-hierarchy. The new class is included in the level hierarchy of the sub-hierarchy by introducing, first, a powertype relationship from the class representing the next lower abstraction level in this sub-hierarchy and, second, a powertype relationship to the class representing the next higher abstraction level this sub-hierarchy.

Example 3.2 (Sample Problem modeled with Powertypes)**.** To model a *product catalog* (see Example 1.1) consisting of abstraction levels *product category*, *product model*, and *physical entity* one introduces three classes, *ProductCategory*, *ProductModel*, and *ProductPE* that represent these levels. To represent the level hierarchy, powertype relationships connect *ProductPE* with *ProductModel* and *ProductModel* with *ProductCategory* (see Figure 3.2).

Next, one wants to introduce a new product category *Car* with tax rate *15*, an additional level *Brand*, and additional attributes *mileage* at level *phys-*

ical entity, *maxSpeed* at level *model* and *marketLaunch* at level *Brand*. This is modeled by introducing *Car* as an instance of *ProductCategory*. Member-attribute *taxRate* of *ProductCategory* becomes an own-attribute of *Car* and is instantiated with value *15*. Since *ProductCategory* is powertype of *ProductModel*, *Car* is also a subclass of *ProductModel* and introduces *maxSpeed* as additional member-attribute. An additional class *CarPE* is introduced manually to represent the physical entity level of the sub-hierarchy of *Car*. *CarPE* is connected with *Car* by a powertype relationship. Additional level *brand* is represented by class *CarBrand*. To model that level *brand* is above level *model*, *Car* is connected to *CarBrand* by a powertype relationship.

Further product categories, product models, and physical entities are introduced likewise. (see Figure 3.2).

Evaluation The powertype relationship is an important extension to modeling with metaclasses as discussed in the previous section. However, modeling of multi-level hetero-homogeneous hierarchies is not directly supported. We will now look in detail at how well the comparison criteria introduced in Section 1.2.1 are supported by powertypes.

1. *Compactness:* Multi-level hetero-homogeneous modeled with powertypes are *not modular*. Using powertypes, one can only tie together one instance-facet and one class-facet relating to one domain-concept. But to describe domain-concepts with more than two abstraction levels, like *Car*, more class-facets would be necessary. Representing domain concept *Car* requires to define three separate model elements, namely class *CarPE*, clabject *Car*, and class *CarBrand*. The abstraction relationship between domain concepts *Car* and *Product* has to be modeled *redundantly* by generalization between *CarPE* and *ProductPE*, as well as classification between *Car* and *ProductCategory* (with generalization between classes *Car* and *ProductModel* being implicitly given by the powertype relationship between *ProductModel* and *ProductCategory*).

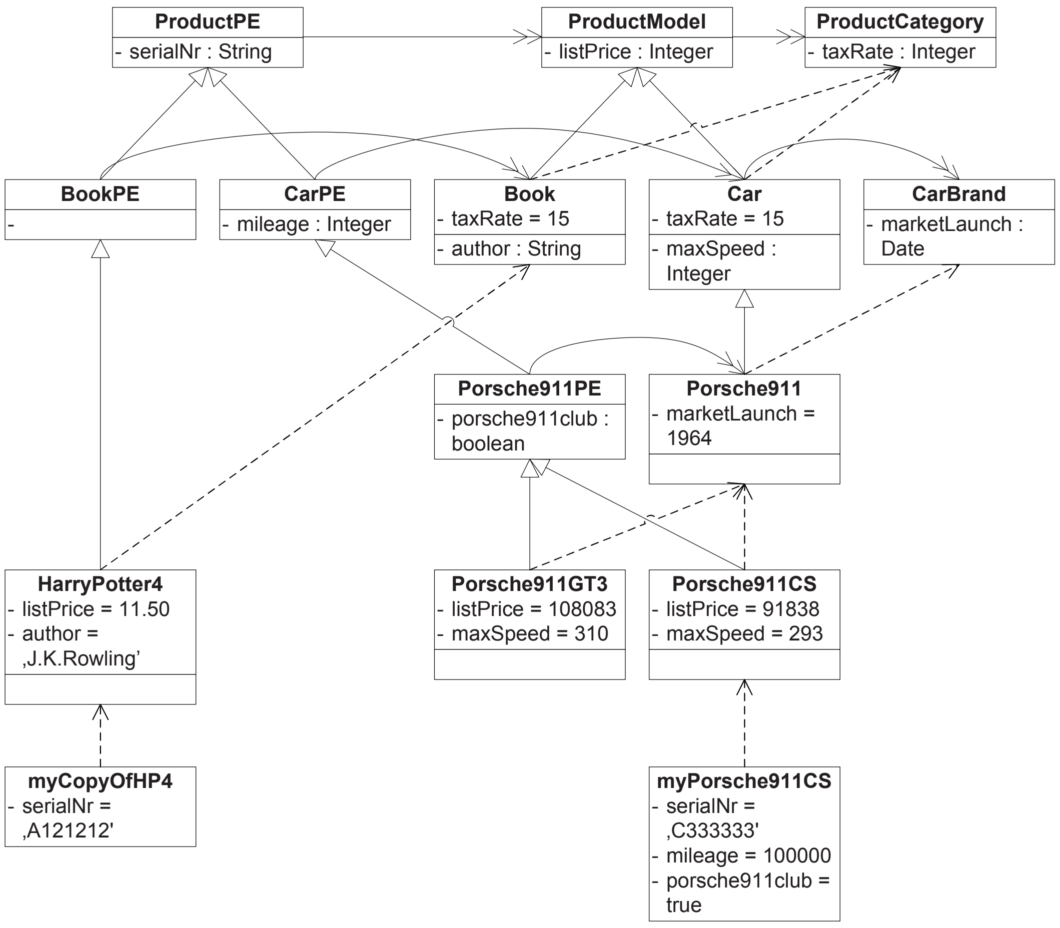


Figure 3.2: Cascaded powertypes - extended pattern

2. *Query flexibility:* Querying powertypes is not explicitly treated in the literature we are aware of. But pre-defined entry points can be easily provided for our "benchmmark queries". (1) All car-models: members of clabject *Car*. (2) Physical entities of *car*: members of *CarPE* . (3) All physical entities of products: members of *ProductPE* .

3. *Hetero-Homogeneous Level-Hierarchies:* It is possible to introduce an additional level in one sub-hierarchy (e.g., for product category *car*) without affecting other sub-hierarchies (e.g., product category *book*). In Example we described how to introduce an additional level *brand* between *car category* and *car model*. Further, a particular car brand

such as *Porsche911* is modeled as an instance of *CarBrand* and a subclass of *Car* (since both *Porsche911* and *Car* represent level *model*). Furthermore, to describe common properties of objects belonging to car brand *Porsche911* at level physical-entity, class *Porsche911PE* is introduced as subclass of *CarPE*. For example, this class has attribute *porsche911Club* to indicate whether some Porsche911 car (i.e., *CarPE*) is registered with the Porsche911Club.

4. *Multiple relationship-abstractions:* The literature we are aware of does not introduce special techniques for modeling multiple relationship-abstraction with powertypes.

3.2 Ontological Metamodeling with Potency-based Deep Instantiation

Deep instantiation refers to meta modeling in which an object at some (meta-)level can describe the common properties for objects at each instantiation-level beneath that level. Metamodeling through meta classes that provide for deep instantiation has been first put forward in the area of databases by the VODAK system [63] (see Related Work Section of Chapter 2.6).

Ontological metamodeling with potency-based deep instantiation was introduced by Atkinson and Kühne [7]. Rather than using different types for each level as in VODAK, attributes definitions are annotated with a potency, where this potency denotes the instance level. Later, Kühne introduced DeepJava [66] for programming with multi-level models. DeepJava supports unbound classification levels and allows to describe not only common attributes of instances but also common properties of instances of instances, and so forth. Recently, Atkinson et al. [42, 6] also discussed connectors (relationships) in the context of ontological metamodeling with potency-based deep instantiation.

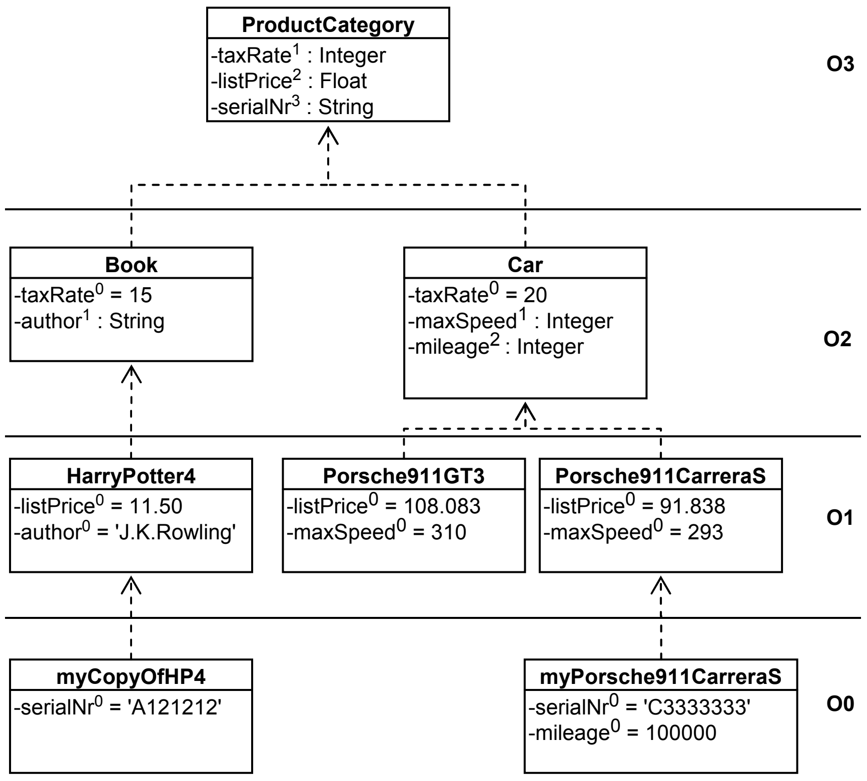


Figure 3.3: Product catalog modeled with potency-based deep instantiation

Our sample *product catalog* (see Example 1.1) consisting of abstraction levels *physical-entity*, *product model*, and *product category*, is modeled (see Figure 3.3) by representing objects at level *physical entity* as objects at classification level $O0$. Each *physical entity* (e.g. *myCopyOfHP4*, *myPorsche911CarreraS*) is modeled as some instance of a product model (e.g. *HarryPotter4*, *Porsche911GT3*, *Porsche911CarreraS*). Each product model is a clabject at classification level $O1$, i.e., is a class for objects at classification level $O0$ as well as an instance of some product category (e.g. *Book*, *Car*) at level $O2$. Each product category is again a clabject, i.e., a class for objects at classification level $O1$ and instance of *ProductCategory* at classification level $O3$.

To define that physical entities of product category *car* additionally have a *mileage*, and car models additionally have a *maximum speed*, attributes *maxSpeed* and *mileage* are introduced with potency 1 and 2, resp., at clabject *Car*. To access attributes defined at a higher level (upward navigation) it is possible to follow the instance-of relationships with method *type()*. The path from *myPorsche911CarreraS* to *taxRate* defined at *Car* is denoted as: myPorsche911CarreraS.type().type().taxRate.

Kühne and Schreiber [66] implement an extension of Java for potency-based deep instantiation. Conceptually interesting is their support for type parameters, which fundamentally extends the power of potency-based deep instantiation, supporting a kind of multi-level abstraction of relationships.

Evaluation

1. *Compactness:* Potency-based deep instantiation supports modular, redundancy-free modeling. All information concerning all levels of one domain-category can be described locally to one clabject. For example, all information about domain concept *car* is encapsulated in a single model element *Car* which is related to *ProductCategory* only once (see Figure 3.3).

2. *Query flexibility:* To our knowledge, extensions of classes are not maintained by DeepJava. But to maintain and retrieve the different member sets of our sample queries should be easy in principle, given the set theoretic interpretation of deep instantiation [67]. Every clabject with potency $p > 0$ can be viewed as having p member sets, one for each level beneath, e.g., *Car* = {{*Porsche911GT3*, *Porsche911CarreraS*},{{},{*myPorsche911CarreraS*}}}.

3. *Hetero-Homogeneous Level-Hierarchies:* Adding an intermediate classification level to a sub-hierarchy (e.g. *car-brand* under product-category *car* with instance *Porsche911*) requires *global* model changes: When introducing a new classification level O_{new} (e.g., $O2$ for *car-brand* with instance *Porsche911*) below level O_n (e.g., $O2$, see Figure3.3,

becoming $O3$ with instances *Car*, etc.) and above level O_{n-1} (e.g., $O1$, see Figure3.3, with instances *Porsche911CarreraS*, etc.), global model changes are necessary: (1) For each clabject at a classification level above O_{new} (in our case *ProductCategory* and *Book*) potencies of attributes that affect objects at classification levels below O_{new} have to be changed (in our case *listPrice* and *serialNr* in clabject *ProductCategory* and *author* in clabject *Book*). (2) For each clabject at level O_n, a (dummy) clabject at level O_{new} has to be introduced, e.g., *BookBrandDummy*, and each clabject at level O_{n-1} has to be re-classified to be instance-of the respective clabject at level O_{new} (*HarryPotter4* is instance-of *BookBrandDummy*). Furthermore, (3) upward navigation paths have to be changed: the navigation from *myPorsche911CarreraS* to its *taxRate* changes from *myPorsche911CarreraS.type().type().taxRate* to *myPorsche911CarreraS.type().type().type().taxRate.* We conclude that due to these necessary changes in existing model elements, a given design or implementation cannot be easily extended with additional levels.

4. *Multi-level relationships:* Modeling of multiple relationship-abstractions has recently been discussed by Atkinson et al. [42, 6]. They introduced connectors in multi-level modeling environments, focusing on the graphical representation of connectors. It remains, however, unclear how these connectors can be exploited for querying. These approach does also not provide for the compact representation of (implicitly) specialized relationship classes and metaclasses. The constraints modeled in Figure 1.3 can be implemented in DeepJava using type parameters [66], which, however, only captures aspect (c) *Specialization/Instantiation of Range.*

3.3 Materialization

Materialization [30, 98, 23] is a generic relationship type which can be regarded as a special kind of aggregation that also incorporates aspects of generalization and instantiation. Each materialization relationship connects a more abstract class c_a (playing the *model* role), e.g. *ProductModel*, with a more concrete class c_c (playing the *materialization* role), e.g. *ProductPhysicalEntity*. An instance o_c of the more concrete class is a materialization of exactly one instance o_a of the more abstract class. Object o_c both inherits attribute values from object o_a (shared values) and instantiates attributes defined at class c_c as well as attributes introduced at object o_a. Thus, similar to clabjects (see Sections 3.2 and 3.1), o_a plays both the role of an object and the role of a class.

Our sample three-level *product catalog*, see Example 1.1, can be modeled by applying materialization relationships repeatedly. Class *ProductPhysicalEntity* materializes class *ProductModel* which, in turn, materializes class *ProductCategory*. Each class represents one level of abstraction (or, in this context, one level of materialization).

Powerful attribute propagation mechanisms [98, 23] facilitate definition of shared attribute values at the more abstract object o_a that are propagated to the more concrete object o_c as well as definition of attributes at o_a, that are instantiated at o_c. Type 1 propagation (T1) simply propagates values of attributes (mono- or multivalued) from o_a to o_c, e.g. *taxRate=20* from *Car* to *Porsche911CarreraS*. For better readability, we have marked propagated T1-attributes with a "/"-character, such as */taxRate=20* at object *Porsche911CarreraS*. Type 2 propagation (T2) allows to define an attribute at c_a, which is instantiated at o_a with a set of possible values. This attribute is instantiated at o_c with one value (T2mono) or a set of values (T2multi) from the set of possible values defined at o_a (not considered in our example). Type 3 propagation (T3) allows to define a multi-valued attribute at c_a, instantiated with a set of values at o_a. Each of these attribute values defined

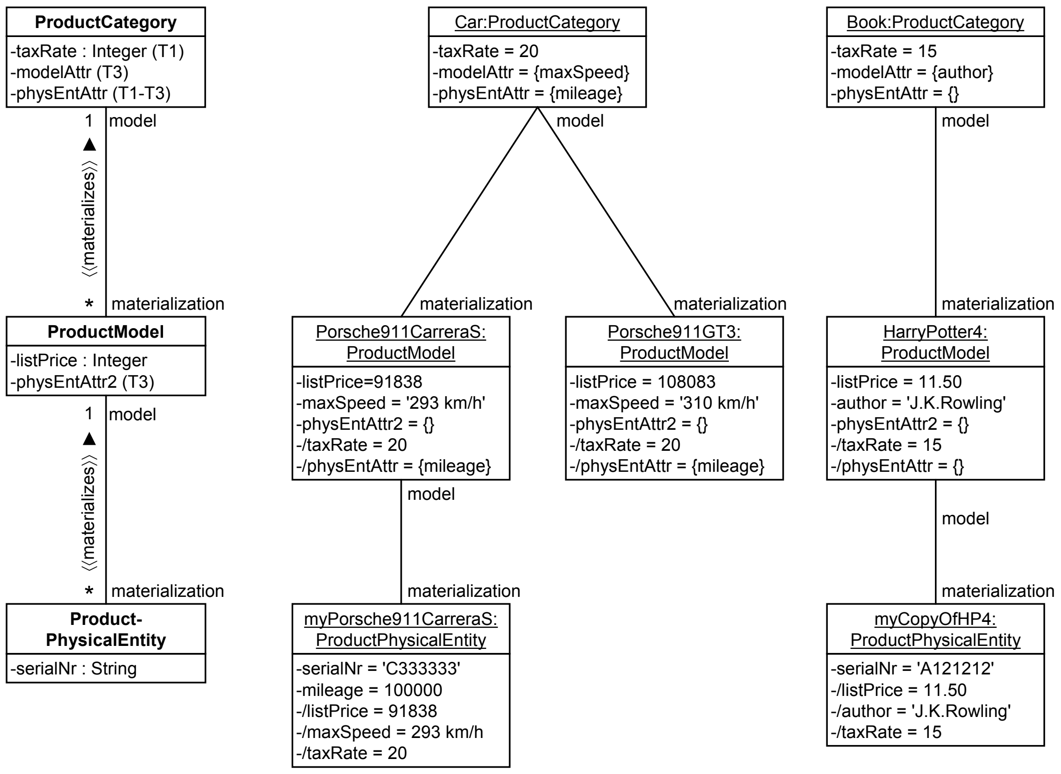


Figure 3.4: Product catalog modeled with materialization (using a respective UML-notation by Olivé[94]) and applying composite attribute propagation mechanism (T1-T3) to mimic deep instantiation

at o_a is transformed to an attribute in o_c and instantiated there For example, class *ProductCategory* defines an attribute *modelAttr* with T3 that is instantiated at object *Car:ProductCategory* with $\{maxSpeed\}$, and *maxSpeed* is instantiated at object *Porsche911CarreraS:ProductModel* (see Figure 3.4).

Attribute values defined at objects at higher materialization levels can be accessed at objects at lower materialization levels by traversal of materialization links. For example, attribute value *taxRate = 20* at *Car:ProductCategory*, can be accessed at *myPorsche911CarreraS* by *myPorsche911CarreraS.model.model.taxRate*. Alternatively, if *taxRate* is propagated using T1, it can be directly accessed at *myPorsche911CarreraS* by *myPorsche911CarreraS.taxRate*.

Composite attribute propagation types, as described in [23], allow to define attributes that have impact on two materialization levels below. In particular, a combination of propagation modes T1 and T3 can be used to mimic potency-based instantiation. The combination of propagation modes T1 and T3 has, to the best of our knowledge, not been considered in literature, but it can be used as follows to mimic potency-based instantiation. An abstract class c_a defines an attribute with T1-T3 propagation. The attribute is instantiated with a set of values at instance o_a of class c_a and, due to propagation mode T1 (which is first applied), propagated to every materialization o_c of abstract object o_a. The attribute is propagated further in T3 mode to every materialization o_{cc} of o_c, i.e., o_{cc} receives an attribute for every value of that attribute at o_c and instantiates it. In this way, attributes with T1-T3 resemble attributes with potency 2 (see Section 3.2), and attributes with T1-T1-T3 resemble attributes with potency 3, and so forth. We illustrate this approach by the following example.

As described in our sample problem statement, product models and physical entities belonging to product category *car* differ from other product models and physical entities in that they provide for attributes *maxSpeed* and *mileage*, respectively. To be able to define these peculiarities, attribute *modelAttr(T3)* and *physEntAttr(T1-T3)* are introduced at class *ProductCategory*. As described in the previous paragraph, these attributes serve as placeholders for attributes introduced in instances of *ProductCategory*. At *Car:ProductCategory*, *modelAttr* is instantiated with {*maxSpeed*} to express that there is an attribute *maxSpeed* in every materialization of *Car*, i.e. in *Porsche911CarreraS* and in *Porsche911GT3*. Furthermore, attribute *physEntAttr* is instantiated with {*mileage*} to define that there is an attribute mileage in every materialization of a materialization of *Car*, i.e. in *myPorsche911CarreraS*. Alternatively, one could explicitly specialize classes *ProductModel* and *ProductPhysicalEntity* by subclasses *CarModel* and *CarPhysicalEntity*, respectively. We do not further consider this alternative, because subclassing of classes which are part of materialization hierarchies is not considered in literature on materialization and is somewhat counter-intuitive to the idea of materialization.

Another powerful composite attribute propagation type, T3-T2, is discussed in [23], but not needed to model our sample product catalog.

Evaluation

1. *Compactness:* Materialization allows modular and redundancy-free models. All peculiarities of domain concept *car*, including those concerning objects at level product model and physical entity, can be described locally to one model element, namely *Car:ProductCategory*. Thus, materialization effectively reduces accidental complexity.

2. *Query flexibility:* Materialization hierarchies (as in [23]) provide a predefined entry point for all objects at a specific materialization level by the class representing this materialization level, e.g. all objects at level *physical entity* can be addressed via class *ProductPhysicalEntity*. Relevant literature does not consider pre-defined query entry points that refer to all objects of one materialization level that directly or indirectly materialize some abstract object. However, one should in principle be able to retrieve such sets of objects rather easily with materialization by queries like (in SQL-Syntax): *SELECT * FROM ProductModel p WHERE p.model = Car* or to access all *physical cars*: *SELECT * FROM ProductPhysicalEntity p WHERE p.model.model = Car* .

3. *Hetero-Homogeneous Level-Hierarchies:* Literature on materialization does not discuss how to introduce additional materialization levels locally to a specific object in the materialization hierarchy, like level *brand* local to *Car:ProductCategory*. Since the modeling solution with materialization presented herein mimics potency-based deep instantiation using T1-T3-attribute propagation, additional levels would lead

to similar problems as when using potency-based deep instantiation, i.e., lead to global model changes. Using materialization, propagation modes of attributes need to be adapted analogously as described above for potency values of potency-based deep instantiation.

4. *Multi-level relationships:* Pirotte et al.[98] introduce attribute propagation. Since attributes can be regarded as (directed) relationships, T2 and T3 attribute propagation are a restricted form of multi-level relationships. Thus, T2 and T3 partly capture (a) *(meta-)classification of relationships* (c) *specialization of range*, in a (d) *compact* manner, but lack (b) *adressability/queryability of relationships at different levels*. Pirotte et al. also shortly mention materialization of relationships, but do not discuss materialization of relationships in detail. They basically consider a relationship as a class that can be materialized like any other class. Pirotte et al. also sketch the idea of materialization of aggregation. For example, a product model is composed of various parts (bill of materials) and this bill of materials can be materialized to the level of *physical entities*. This idea could be elaborated and extended for materialization of relationships in order to support multiple relationship-abstractions.

3.4 Summary

In this chapter we reviewed three multi-level modeling techniques. We exemplified them based on our sample problem and evaluated them with regard to our requirements for multi-level modeling.

The results of our evaluations of the various modeling techniques are summarized in Table 3.1. This comparison of different modeling techniques is complemented by a corresponding evaluation in Chapter 9.

	Mater.	Deep I.	Powertypes
Compactness	+	+	–
Query Flexibility	∼	∼	+
Heterogenous Level-Hier.	–	–	+
Relationship-Abstraction	∼	∼	–

Table 3.1: Evaluation of different multi-level abstraction techniques. Used symbols: '+' (full support), '∼' (limited support), '–' (no support)

We wish to stress that this evaluation only addresses the suitability of modeling techniques for multi-level abstraction, especially for domains such as exemplified in our sample problem (see Section 1.2). It does, however, not evaluate the overall quality of these approaches or their suitability for other problem domains.

Chapter 4

M-Objects and M-Relationships

Contents

In this chapter we discuss *multi-level objects* (*m-objects*) and *multi-level relationships* (*m-relationships*), which are the basic constructs of our multi-level modeling approach. We give formal definitions of their basic structure and of global consistency criteria. In Section 4.1, we present the concept of m-objects and their consistent concretization in hetero-homogeneous concretization hierarchies. In Section 4.2, we discuss the concept of m-relationships and their consistent concretization in hetero-homogeneous concretization hierarchies of m-relationships. The m-object and m-relationship approach builds on our basic assumptions and on our analysis of the classical abstraction principles, as discussed in Chapter 2.

This chapter is based on a previous paper [86], since then, we made major extensions to the m-object and m-relationship approach. Especially by providing stricter consistency criteria. Additionally we introduce inheritance mechanisms and some shorthand notations.

4.1 Multi-Level Objects

In this section we introduce, exemplify and formally define m-objects and their concretization. We first describe m-objects and how one m-object can concretize another m-object. We then look at the roles which an m-object plays in such a concretization. Building on a formal definition of m-objects, we specify rules for consistent concretization of m-objects and for consistent concretization hierarchies. Note that the basic concepts of m-object hierarchies could alternatively be described by providing a mapping to UML and OCL, or to the Higher-order Entity-Relationship Model (HERM) [122]. Our concise definitions are independent of a specific conceptual modeling language and well suited for defining consistency rules as well as operators for working with m-objects (see Chapters 5 and 6).

An m-object encapsulates and arranges abstraction levels in a linear order from the most abstract to the most concrete one. Thereby, it describes itself and the common properties of the objects at each level of the concretization hierarchy beneath itself. An m-object that concretizes another m-object, the parent, inherits all levels except for the top-level of the parent. It may also specialize the inherited levels or even introduce new levels. An m-object specifies concrete values for the properties of the top-level. This top-level has a special role in that it describes the m-object itself. All other levels describe common properties of m-objects beneath itself.

Definition 4.1 (M-Object). *An m-object $o \in O$, where O is the universe of m-objects, is described by a 6-tuple $(L_o, A_o, p_o, l_o, d_o, v_o)$. This description consists of a set of levels L_o, taken from a universe of levels L, and a set*

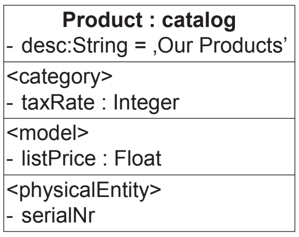


Figure 4.1: M-object *Product* representing a three-level *product catalog*

of attributes A_o, taken from a universe of attributes A. The levels L_o are organized in a linear order, as defined by partial function parent $p_o : L_o \to L_o$, which associates with each level its parent level. Each attribute is associated with one level, defined by function $l_o : A_o \to L_o$, and has a domain, defined by function $d_o : A_o \to D$ (where D is a universe of data types). Optionally, an attribute has a value from its domain, defined by partial function $v_o : A_o \to V$, where V is a universe of data values, and $v_o(a) \in d_o(a)$ if $v_o(a)$ is defined.

For notational convenience, we define in the following a couple of shorthand notations.

Definition 4.2 (M-Object Shorthand Notations). *Level-hierarchy $P_o \subseteq (L_o \times L_o)$ is defined as: $(l', l) \in P_o \overset{\text{def}}{=} p_o(l') = l$. We denote its transitive closure by P_o^+ and its transitive-reflexive closure by P_o^*.*

The top-level *of m-object o, denoted as $\hat{l}_o$, is the single level in L_o that has no parent level, i.e.: $\hat{l}_o = l \overset{\text{def}}{=} l \in L_o \wedge p_o(l)$ is undefined.*

The second-top-level*, if such exists, of o, denoted as $\hat{\hat{l}}_o$ is the single level in L_o that has $\hat{l}_o$ as its parent, i.e.: $\hat{\hat{l}}_o = l \overset{\text{def}}{=} l \in L_o \wedge p_o(l) = \hat{l}_o$.*

The bottom-level *of o, denoted as $\check{l}_o$, is the single level in L_o that is not parent of any other level in L_o, i.e.: $\check{l}_o = l \overset{\text{def}}{=} l \in L_o \wedge (\nexists l' \in L_o : p_o(l') = l)$.*

The top-level attributes *of o, denoted as $\hat{A}_o$, also referred to as* own-attributes of o*, are given by the set of attributes associated with the top-level of o, i.e.: $\hat{A}_o \overset{\text{def}}{=} \{a \in A_o \mid l_o(a) = \hat{l}_o\}$.*

Further we say that an m-object o is at level l if l is its top-level, $\hat{l}_o = l$.

Example 4.1. Consider m-object *Product* of Figure 4.1. This m-object represents the sample product catalog (see Example 1.1) which consists of product descriptions at three levels of abstraction, namely at levels *product category*, *product model*, and *product physical entity*. The top-level of m-object *Product* represents the *product catalog* itself, which is described by own-attribute *desc* with data type *String* as domain and value *"Our Products"*, this is the instance-facet of m-object *Product*, which is an instance of the singleton-class also represented by this top-level. The schema of the product catalog is described by the order of levels and by the member-attributes defined with these levels. The second-top-level, labeled *category*, represents the class of all *product categories* within the product catalog, and defines their common structure by member-attribute *taxRate* with domain *Integer*. Level *model* represents the class of all *product models* and defines their common structure by member-attribute *listPrice* with domain *Float*. The bottom-level, labelled *physicalEntity*, represents the class of all *product physical entities* with member-attribute *serialNr*, having *String* as domain. The hierarchy of levels represents an aggregation hierarchy as described in Chapter 2: each *physical entity* belongs to one *product model* and each *product model* belongs to one *product category*. We say, m-object *Product* is at level *catalog* since this level is its top-level (note that the name of the top-level is always depicted next to the name of the m-object, separated by a colon).

An m-object may *concretize* another m-object, which is referred to as its *parent*. Such a concretization relationship between two m-objects comprises aspects of instantiation-, specialization- and decomposition-relationships between the levels of an m-object and the levels of its parent, as follows:

- *Classification – Instantiation:* Each m-object can be regarded as an instance of its parent m-object. In particular, the top-level of an m-object is an instance of the second-top-level of its parent m-object. An m-object must adopt all levels from its parent except for the parent's

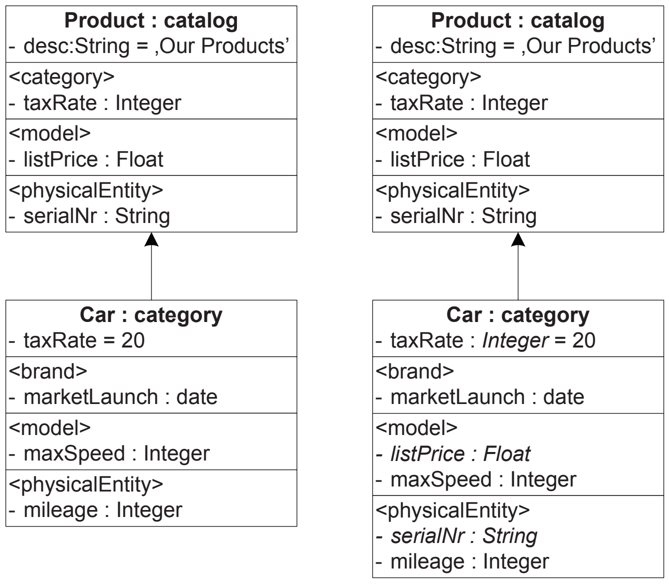


Figure 4.2: M-object *Car* concretizes m-object *Product*. Alternative graphical representations without (left) as well as with inherited attributes (right)

top-level and it can specify values for its attributes. For a discussion of the metaclass-facets of an m-object we refer to Chapter 6.

- *Generalization – Specialization:* The level descriptions of an m-object correspond to subclasses of the corresponding levels of its parent. The m-object can add attributes to levels and define new levels. By doing so, an m-object must not change the relative order of levels it inherits from its parent.

- *Aggregation – Decomposition:* The concretization path between m-objects of different levels expresses an aggregation hierarchy (as described in Chapter 2) at the instance level. The aggregation schema of the sub-hierarchy of an m-object is given by the order of its abstraction levels.

Example 4.2 (Concretization)**.** M-object *Car* concretizes m-object *Product* (see left part of Figure 4.2). This concretization relationship is depicted by an arrow from *Car* to *Product*. This concretization relationship represents, first, that product category *Car* is part of the sample *product catalog*. Second, m-object *Car* instantiates the second-top-level of *Product*, which is *category*. Since m-object *Product* defines member-attribute *taxRate*, m-object *Car* has *taxRate* as an own-attribute. *Car* instantiates *taxRate* with value *20*.

M-object *Car* further represents the fragment of the sample product catalog that describes *cars* at different levels of abstraction. This is referred to as the sub-hierarchy of *Car*. M-object *Car* specializes the schema of its sub-hierarchy by specializing classes at level *model*, introducing additional member-attribute *maxSpeed*, and *physicalEntity*, introducing additional member-attribute *mileage*. It further specializes the schema of the aggregation hierarchy by introducing an additional abstraction level, namely *brand*, above level *model*, to represent that each *Car model* belongs to a *Car brand*. *Car brands* are described by attribute *marketLaunch*.

Note that the right part of Figure 4.2 depicts the full description of m-object *Car*, including attributes inherited from m-object *Product*. The inheritance mechanism in place will be defined below.

When concretizing m-objects repeatedly, the resulting m-objects form concretization hierarchies (see Example 4.3).

Definition 4.3 (Concretization Hierarchies of M-Objects)**.** *Concretization hierarchies of m-objects are defined by relation $H \subseteq O \times O$ which forms a forest (set of trees). Let $o', o \in O$, then o' is said to be a* direct concretization of *or* child of *o iff $(o', o) \in H$, and to be a* descendant of *o iff $(o', o) \in H^+$, alternatively written as $o' \prec o$, and to be a* descendant of or equal to *o iff $(o', o) \in H^*$, alternatively written as $o' \preceq o$. In the inverse direction, o is said to be* parent of, *be* ancestor of, *or be* ancestor of or equal to *o', respectively. H^+ is the transitive closure of H and H^* is the transitive-reflexive closure of H. We alternatively refer to the descendants of an m-object o as* concretizations of *o and to its ancestors as* abstractions of *o.*

Every concretization hierarchy has a single root m-object *$\hat{o} \in \hat{O}$ which represents the hierarchy. The set of root m-objects, $\hat{O} \subseteq O$, is defined as consisting of all m-objects that have no parent m-object: $\hat{O} \stackrel{\text{def}}{=} \{\hat{o} \in O \mid \nexists \dot{o} \in O : \hat{o} \prec \dot{o}\}$. Every m-object o belongs to exactly one hierarchy (represented*

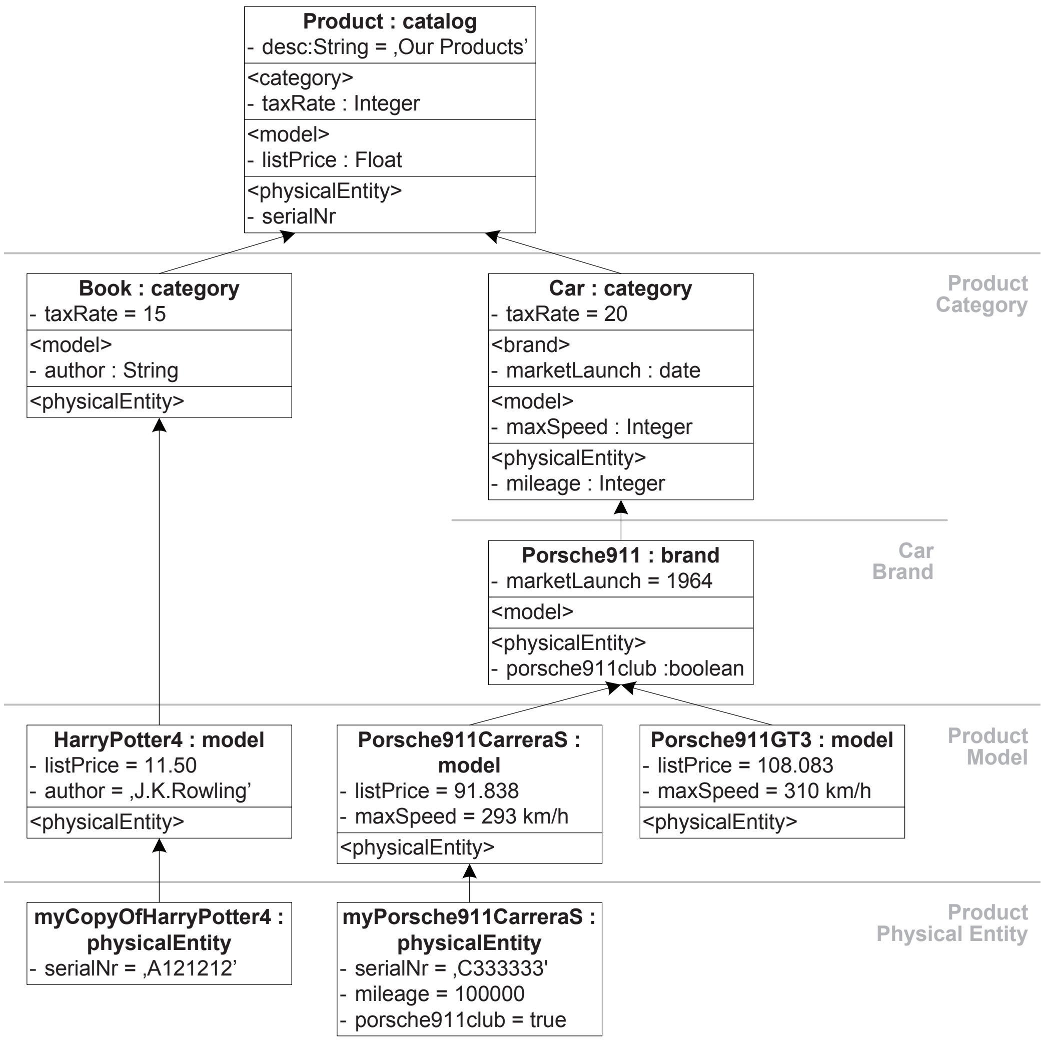


Figure 4.3: Concretization hierarchy of a product catalog along multiple levels (inherited attributes not shown)

by a root m-object). The root-m-object of o, denoted as $\hat{o}_o$, is the single root-m-object that is an ancestor of or equal to o: $\hat{o}_o \overset{\text{def}}{=} \hat{o} \in \hat{O} : o \preceq \hat{o}$.

M-objects, levels, and attributes have *names*, defined by function name : $O \cup L \cup A \rightarrow N$, where N is the universe of names. Additionally, hierarchies of m-objects provide a namespace, this namespace is assigned to the root m-object and shared by all descendant m-objects, it is defined by function $ns : O \rightarrow N$. Names of m-objects, levels, and attributes are unique within one hierarchy, that is, levels, attributes, and m-objects may be externally identified by the namespace provided by their hierarchy together with their names.

M-objects organized in a concretization hierarchy inherit properties from their parent m-objects and, thus, may only be partially defined. The inheritance mechanism is formally defined in the following. Examples of applying these inheritance rules are given in Chapter 5 in the context of creating m-object hierarchies in a stepwise manner.

Definition 4.4 (M-Object Inheritance)**.** *Given a delta description of a (child) m-object o', $(L^{\Delta}_{o'}, A^{\Delta}_{o'}, p^{\Delta}_{o'}, l^{\Delta}_{o'}, d^{\Delta}_{o'}, v^{\Delta}_{o'})$, its full description $(L_{o'}, A_{o'}, p_{o'}, l_{o'}, d_{o'}, v_{o'})$ is derived as follows: If o' does not concretize another m-object o, i.e., $\nexists o \in O : (o', o) \in H$, then $(L_{o'}, A_{o'}, p_{o'}, l_{o'}, d_{o'}, v_{o'}) := (L^{\Delta}_{o'}, A^{\Delta}_{o'}, p^{\Delta}_{o'}, l^{\Delta}_{o'}, d^{\Delta}_{o'}, v^{\Delta}_{o'})$. Otherwise, that is, o' concretizes another m-object o, its parent m-object, denoted as $(o', o) \in H$, then its namespace is that of its parent, i.e., $ns(o') := ns(o)$, and its full description $(L_{o'}, A_{o'}, p_{o'}, l_{o'}, d_{o'}, v_{o'})$ is given as follows:*

- *the set of levels $L_{o'}$ of the child m-object is the union of levels introduced by the child m-object, $L^{\Delta}_{o'}$, and levels of the parent m-object, L_o, without the top-level of the parent m-object, i.e., $L_{o'} := (L^{\Delta}_{o'} \cup L_o) \setminus \{\hat{l}_o\}$*
- *the set of attributes of the child m-object, $A_{o'}$, is the union of attributes introduced at the child m-object, $A^{\Delta}_{o'}$, and attributes of the parent m-object, A_o, without the attributes of the top-level of the parent m-object, i.e., $A_{o'} := (A^{\Delta}_{o'} \cup A_o) \setminus \{\hat{A}_o\}$*

- *the order of levels given by the parent m-object may be overwritten by the child m-object. The second-top-level of o is the top-level of o' and has no parent level within o'. For every level $l \in L_{o'}$:*
$$p_{o'}(l) := \begin{cases} p^{\Delta}_{o'}(l) & \textit{if defined} \\ \textit{undefined} & p^{\Delta}_{o'}(l) \textit{ undefined } \wedge l = \hat{\hat{l}}_o \\ p_o(l) & \textit{otherwise} \end{cases}$$

- *the assignment of every attribute $a \in A_{o'}$ to a level is given by*
$$l_{o'}(a) := \begin{cases} l^{\Delta}_{o'}(a) & \textit{if defined} \\ l_o(a) & \textit{otherwise} \end{cases}$$

- *the assignment of a data type to every attribute $a \in A_{o'}$ is given by*
$$d_{o'}(a) := \begin{cases} d^{\Delta}_{o'}(a) & \textit{if defined} \\ d_o(a) & \textit{otherwise} \end{cases}$$

- *the assignment of a value to every attribute $a \in A_{o'}$ is given by*
$$v_{o'}(a) := \begin{cases} v^{\Delta}_{o'}(a) & \textit{if defined} \\ v_o(a) & \textit{otherwise} \end{cases}$$

In the following definition of a consistent concretization, we assume that partially defined m-objects have already been extended as just described.

Definition 4.5 (Consistent Concretization)**.** *Let $o' \in O$ be a direct concretization of $o \in O$, $(o', o) \in H$, then o' is a* consistent concretization *of o iff*

1. *The top-level of o' is second-top-level of o: $\hat{l}_{o'} = \hat{\hat{l}}_o$ (*second-top-level instantiation*)*

2. *Each level of o, except for its top-level, is also a level of o': $(L_o \setminus \{\hat{l}_o\}) \subseteq L_{o'}$ (*level containment*)*

3. *All attributes of o, except for the attributes of its top-level, also exist in o': $(A_o \setminus \hat{A}_o) \subseteq A_{o'}$ (*attribute containment*)*

4. *The relative order of common levels of o' and o is the same, i.e.,* $\forall l', l \in (L_o \cap L_{o'}) : (l', l) \in P_{o'}^+ \Rightarrow (l', l) \in P_o^+$ *(*level order compatibility*)*

5. *Common attributes are associated with the same level, have the same domain, and the same value, if defined: For all* $a \in (A_o \cap A_{o'})$*:*

   (a) $l_{o'}(a) = l_o(a)$ *(*stability of attribute levels*)*

   (b) $d_{o'}(a) = d_o(a)$ *(*stability of attribute domains*)*

   (c) $v_o(a)$ *is defined* $\Rightarrow v_{o'}(a) = v_o(a)$ *(*compatibility of attribute values*)*

Example 4.3 (Consistent Concretization)**.** M-object *Car* is a consistent concretization of m-object *Product* (see right part of Figure 4.2). Except for the top-level *catalog*, each level of *Product* also exists in *Car* (level containment). The same is true for all attributes, i.e. *Car* only misses attribute *desc*, which is a top-level attribute of *Product* (attribute containment). The levels have the same order in both m-objects, e.g. in each m-object level *category* comes above level *model* (level order compatibility). Both m-objects associate attribute *taxRate* with level *category* (stability of attribute levels) and define domain *Integer* for this attribute (stability of attribute domains). If m-object *Product* defined a value for this attribute, the value would need to be *20* to ensure compatibility of attribute values.

M-objects do not only inherit levels and attributes from their parents, but can also introduce new levels and add attributes to levels. In contrast to potency-based deep instantiation [7, 66] (see Chapter 3), m-objects use names instead of numeric potencies to identify levels, which enables introducing additional levels without affecting existing navigation paths.

Example 4.4 (Extensibility)**.** In Figure 4.2, m-object *Car* inherits levels *category*, *model* and *physical entity* from m-object *Product*. It adds level *brand* to define that cars are further categorized by their brand. Note that the additional level *brand* applies only to descendant m-objects of *Car* and not to other sub-hierarchies such as descendants of *Book*. Additionally, m-object *Car* extends level definitions of m-object *Product* by adding attribute *maxSpeed* to level *model*, and attribute *mileage* to level *physical entity*.

In the following Definition 4.5 we extend consistency criteria for individual concretization to consistency criteria that must hold for concretization hierarchies as a whole.

Definition 4.6 (Consistent Concretization Hierarchy of M-Objects)**.** *A concretization hierarchy $H \subseteq O \times O$ is* consistent*, iff*

1. *For each pair of child m-object o' and parent m-object o, $(o', o) \in H$, o' is a consistent concretization of o according to Definition 4.5.*
2. *Each attribute and level is introduced at only one m-object, hence, if an m-object $o \in O$ shares an attribute (or a level) with another m-object $\dot{o} \in O$, then either one of them introduced the attribute (or level, respectively) and is an ancestor of the other, or they have a common ancestor which introduced this attribute (or level, respectively):*
   (a) $\forall a \in A, o, \dot{o} \in O : a \in (A_o \cap A_{\dot{o}}) \Rightarrow \exists \ddot{o} \in O : o \preceq \ddot{o} \wedge \dot{o} \preceq \ddot{o} \wedge a \in A_{\ddot{o}}$ *(*unique induction rule for attributes*)*
   (b) $\forall l \in L, o, \dot{o} \in O : l \in (L_o \cap L_{\dot{o}}) \Rightarrow \exists \ddot{o} \in O : o \preceq \ddot{o} \wedge \dot{o} \preceq \ddot{o} \wedge l \in L_{\ddot{o}}$ *(*unique induction rule for levels*)*

Example 4.5 (Concretization Hierarchy)**.** In Figure 4.1, m-object *Porsche911* is a direct concretization of m-object *Car* and a descendant of both, m-object *Car* and m-object *Product*. M-objects *Book* and *Car* have a common attribute *taxRate*. To be consistent, these m-objects must have a common ancestor in the concretization hierarchy, which also defines this attribute, namely m-object *Product* in the example.

As introduced in Definition 4.1, levels within an m-object are totally ordered, while concretizations may introduce additional levels, but they may not change the relative order of levels. Further, each level is introduced at exactly one m-object. Thus we can derive a global partial order of levels as follows.

Definition 4.7 (Partial Order of Levels)**.** *The global partial order of levels is given by* transitive *relation* $\prec$*. A level* $l' \in L$ *is below another level* $l \in L$*, written as* $l' \prec l$*, if there is an m-object* o *in which* (l', l) *is an element of the transitive closure of parent relation* P_o*, i.e.,* $\forall l', l \in L : (\exists o \in O : (l', l) \in P_o^+) \Rightarrow l' \prec l$*. We write* $l' \preceq l$ *to denote that a level* l' *is below or equal to another level* l*, i.e.,* $l' \preceq l \overset{\text{def}}{=} l' \prec l \vee l' = l$*.*

Note that a level $l' \in L$ may be below another level $l \in L$, $l' \prec l$, even if these levels are not part of the same m-object. This follows from the transitivity of relation $\prec$. Further note that for notational convenience we overloaded symbol '$\prec$'. It refers, both, to the transitive concretization relation (as an alternative notation for H^*), as well as to the global partial order of levels. We will further overload this symbol in the context of m-relationships.

4.2 Multi-Level Relationships

In this section we introduce, exemplify and formally define multi-level relationships (m-relationships) as a high-level modeling primitive for modeling relationships at multiple levels of abstraction. We show how m-relationships connect m-objects at multiple levels, how they can be concretized, and we look at the roles which they play in concretizations. Then we define rules for consistent concretization of m-relationships. Finally, we discuss one possible naming scheme for m-relationships and introduce inheritance between m-relationships. For simplicity, we consider only binary relationships in this chapter and do not consider cardinality constraints.

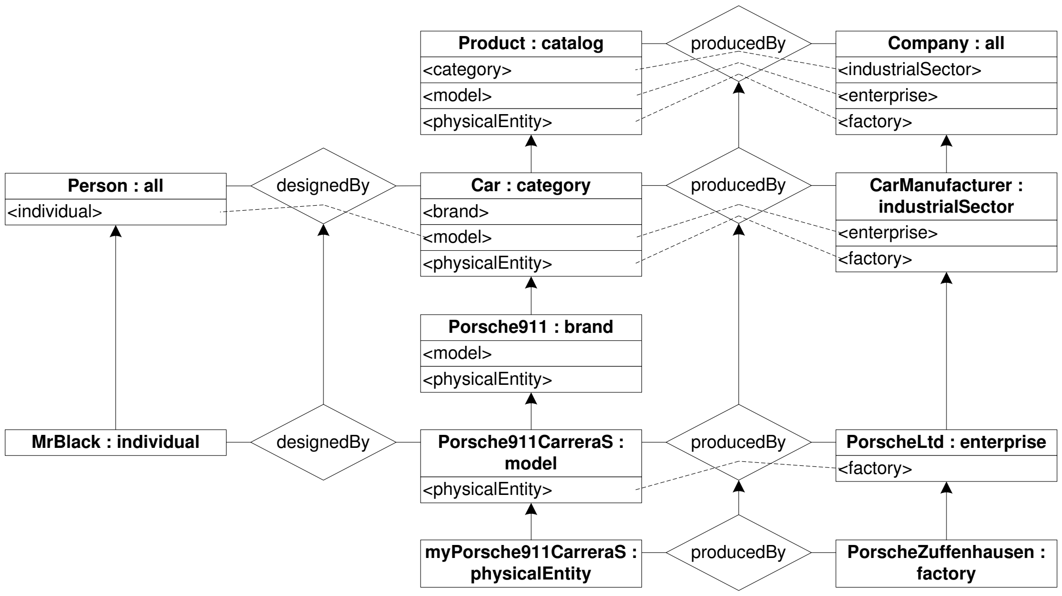


Figure 4.4: Product catalog modeled with m-objects and m-relationships

So far we have shown how to reduce accidental complexity by describing entities and entity types at multiple levels of abstraction through m-objects. M-relationships are analogous to m-objects in that they describe relationships between m-objects at multiple levels of abstraction and in that they are organized in concretization hierarchies.

The basic idea of m-relationships is to provide a high-level modeling primitive that (1) encapsulates different abstraction levels of a relationship, (2) implies extensional constraints between different abstraction levels of a relationship, (3) supports heterogenous hierarchies (e.g. additional level *brand* at category *Car*), and (4) can be exploited for navigating and querying. This leads to application models that are easier to define, to understand, and to maintain, as compared to models where these different facets of relationships are modeled as distinct model elements (as exemplified in Chapter 2).

Definition 4.8 (M-Relationship). *An m-relationship $r \in R$, where R is the universe of m-relationships, is described by a quadruple $(\hat{r}_r, s_r, t_r, C_r)$. Each m-relationship r belongs to a root-m-relationship $\hat{r}_r \in R$, connects two m-objects, $s_r \in O$ and $t_r \in O$, and has a set of connection-levels $C_r \in L \times L$.*

Each of its connection-levels $(l, \bar{l}) \in C_r$ *connects a level* l *of source m-object* s_r, $l \in L_{s_r}$, *to a level* $\bar{l}$ *of target m-object* t_r, $\bar{l} \in L_{t_r}$.

The set of connection-levels of an m-relationships expresses an aggregation schema (see Chapter 2). Each connection-level is regarded as a relationship class having relationship occurrences at that connection-level as instances. The hierarchy of component and composite relationship classes needs not to be explicitly modeled, since the partial order of pairs of abstraction levels can be derived from the partial order of abstraction levels of m-objects (see Definition 4.7). The partial order of pairs of abstraction levels is as follows.

Definition 4.9 (Partial Order of Pairs of Levels). *We say a pair of levels* $(l', \bar{l}') \in L \times L$ *is below or equal to a pair of levels* $(l, \bar{l}) \in L \times L$, *written as* $(l', \bar{l}') \preceq (l, \bar{l})$, *iff:* $l' \preceq l \wedge \bar{l}' \preceq \bar{l}$. *A pair of levels* $(l', \bar{l}')$ *is below a pair of levels* $(l, \bar{l})$, *written as* $(l', \bar{l}') \prec (l, \bar{l})$, *iff:* $(l' \prec l \wedge \bar{l}' \preceq \bar{l}) \vee (l' \preceq l \wedge \bar{l}' \prec \bar{l})$.

Example 4.6 (M-Relationship). To model that car models have a designer, Figure 4.4 depicts an m-relationship *designedBy* between source m-object *Car* and target m-object *Person*. We write *Car-designedBy-Person* to refer to this m-relationship. Its top-connection-level *(category,all)* is given by the top-levels of its source m-object *Car* and its target m-object *Person*. We say m-relationship *Car-designedBy-Person* is at connection-level *(category,all)*. It further connects level *model* with level *individual* to express that *car models* are designed by individual *persons*. M-relationships that only have two connection-levels, like *Car-designedBy-Person*, are similar to associations in the UML and relationship sets (relationship types) in the ER model.

Now consider m-relationship *producedBy* between *Product* and *Company*, referred to by *Product-producedBy-Company*, which specifies, apart from its top-connection-level *(catalog,all)*, multiple connection-levels, namely *(category,industrialSector)*, *(model,enterprise)*, *(physicalEntity,factory)*. These connection-levels express an aggregation hierarchy of relationship classes (as discussed in Chapter 2): each *producedBy*-m-relationship between a *physicalEntity* and a *factory* has to be part of exactly one *producedBy*-m-relationship between a *model* and an *enterprise*, which, in turn, has to be part of exactly one *producedBy*-m-relationship between a *category* and an *industrialSector*.

Like m-objects, m-relationships are organized in concretization hierarchies. Each such hierarchy has a root m-relationship. Each m-relationship belongs to exactly one concretization hierarchy of m-relationships, by associating the m-relationship with the root of a concretization hierarchy of m-relationships. The position of an m-relationship in a concretization hierarchy is implicitly given by the pair of connected m-objects (its coordinate).

Definition 4.10 (Concretization Hierarchies of M-Relationships)**.** *A hierarchy of m-relationships, rooted in m-relationship $\hat{r}$, consists of the set of m-relationships, denoted as $\hat{r}\langle\rangle$, being associated with this root m-relationship: $\hat{r}\langle\rangle \overset{\text{def}}{=} \{r \in R \mid \hat{r}_r = \hat{r}\}$. The root m-relationship $\hat{r}$ has itself as root m-relationship, i.e., $\hat{r}_{\hat{r}} = \hat{r}$.*

Each m-relationship r in this hierarchy, $r \in \hat{r}\langle\rangle$, connects two m-objects that are descendants of or are equal to the m-objects connected by $\hat{r}$, i.e.: $r \in \hat{r}\langle\rangle \Rightarrow s_r \preceq s_{\hat{r}} \wedge t_r \preceq t_{\hat{r}}$.

For each pair of m-objects $(o, \bar{o}) \in O \times O$ there is at most one m-relationship per hierarchy, i.e.: $\forall r, \dot{r} \in R : s_r = s_{\dot{r}} \wedge t_r = t_{\dot{r}} \wedge \hat{r}_r = \hat{r}_{\dot{r}} \Rightarrow r = \dot{r}$.

The partial order ($\preceq$) of m-relationships in concretization hierarchies is defined as: $\forall r', r \in R : r' \preceq r \Leftrightarrow \hat{r}_{r'} = \hat{r}_r \wedge s_{r'} \preceq s_r \wedge t_{r'} \preceq t_r$. We say an m-relationship r' is a descendant of (concretization of) or equal to another m-relationship r iff $r' \preceq r$. We say an m-relationship r' is a descendant of (concretization of) r, denoted as $r' \prec r$, iff $r' \preceq r \wedge r' \neq r$. We say r is a child of (direct concretization of) r', denoted as $(r', r) \in H\!R$ iff there is no other m-relationship $\dot{r}$ in between r' and r, i.e.: $(r', r) \in H\!R \Leftrightarrow r' \prec r \wedge (\nexists \dot{r} \in R : r' \prec \dot{r} \prec r)$. In the inverse direction, we say r is an ancestor of (abstraction of), or a parent of (direct abstraction of) r', respectively.

Analogously to m-relationships, arbitrary pairs of m-objects (coordinates) are partially ordered.

Definition 4.11 (Partial Order of Coordinates)**.** *A coordinate* $(o', \bar{o}') \in O \times O$ *is* below or equal to *coordinate* $(o, \bar{o}) \in O \times O$*, written as* $(o', \bar{o}') \preceq (o, \bar{o})$*, iff* $o' \preceq o \wedge \bar{o}' \preceq \bar{o}$*. We say* $(o', \bar{o}')$ *is* below *coordinate* $(o, \bar{o})$*, written as* $(o', \bar{o}') \prec (o, \bar{o})$*, iff* $(o' \prec o \wedge \bar{o}' \preceq \bar{o}) \vee (o' \preceq o \wedge \bar{o}' \prec \bar{o})$*.*

As exemplified above, connection-levels of an m-relationship are regarded as aggregation schema, where a connection-level that is below another connection-level is regarded as a component relationship class of the latter. As discussed in Chapter 2, we regard *aggregation* as a *transitive*, *antisymmetric*, and *irreflexive* relation that connects components with composites, where the component is *existentially dependent* on the composite. On the schema level, aggregation also represents a *functional dependency* between component class and composite class.

In order to ensure this *functional dependency* between component class and composite class (represented by two connection-levels, one below the other), we define that a consistent child m-relationship has to contain all connection-levels of the parent m-relationship except for its top-connection-level. If a parent m-relationship has a non-top connection-level above the top-connection-level of the child-m-relationship, then the child-m-relationship is considered inconsistent, since it violates the imposed functional and existential dependency. In this way the extension of higher connection-levels restrains the possible extension of lower connection-levels. This is what leads to the notion of *active domain*, which we will discuss in detail in Chapter 5.

Also note that an m-relationship may have more than one parent m-relationships (this is what we call *multiple concretization*). But, these parent m-relationships reside on different connection-levels, which represent different composite class. In this way, concretization hierarchies with multiple concretization can still be regarded as *exclusive part-whole relations* (as discussed in [39]).

Definition 4.12 (Consistent Concretization of M-Relationships)**.** *Let* $r' \in R$ *be a direct concretization of* $r \in R$*,* $(r', r) \in H_R$*. Child m-relationship* r' *is* *a* consistent concretization *of parent m-relationship* r *iff each connection-level of* r*, except for its top-connection-level, is also a connection-level of* r'*:* $C_r \setminus \{\hat{c}_r\} \subseteq C_{r'}$*. A property we call* connection-level containment*.*

In the case of multiple concretization, a child m-relationship has to contain the union of connection-levels of its parents (without their top-connection-levels).

Definition 4.13 (Consistent Concretization Hierarchy of M-Relationships)**.** *A* concretization hierarchy of m-relationships *is* consistent*, iff for each two m-relationships* $r', r \in R$ *within that hierarchy, where* r' *directly concretizes* r*,* $(r', r) \in H_R$*,* r' *is a consistent concretization of* r *according to Definition 4.12.*

Example 4.7 (Consistent Concretization of M-Relationships)**.** M-relationship *Car-producedBy-CarManufacturer* (see Figure 4.4) is a consistent concretization of m-relationship *Product-producedBy-Company* since each connection-level of m-relationship *Product-producedBy-Company*, except for its top-connection-level *(catalog,all)*, is also a connection-level of *Car-producedBy-CarManufacturer*.

Example 4.8 (Inconsistent Concretization of M-Relationships)**.** M-relationship *HarryPotter4-producedBy-PorscheLtd* (see Figure 4.5) is an inconsistent concretization of m-relationship *Product-producedBy-Company*, which defines a connection-level between abstraction levels *Category* and *IndustrialSector*. Since abstractions of *HarryPotter4* and of *PorscheLtd* (*Book* and *CarManufacturer*, respectively) at these abstraction levels are not connected by a *producedBy*-m-relationship, the m-relationship between *HarryPotter4* and *PorscheLtd* is inconsistent.

Depending on the number of connection-levels as well as depending on which connection-level one looks at, an m-relationship plays *different roles*, which manifest its instance-, class-, and metaclass-facets:

- *Relationship occurrence:* an m-relationship always connects the top-level of its source m-object with the top-level of its target m-object.

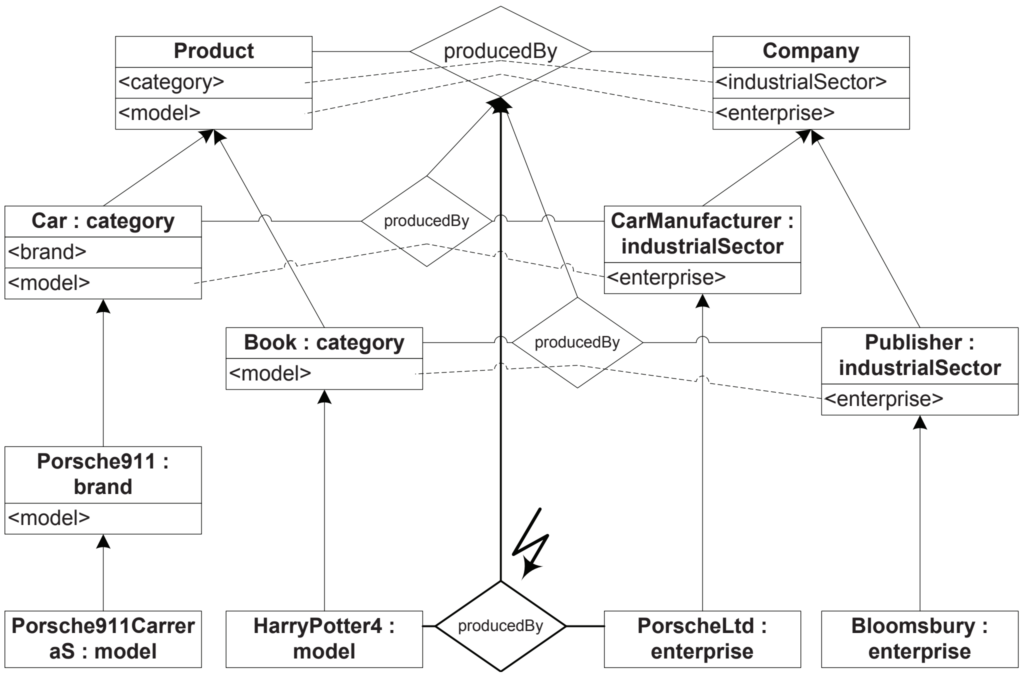


Figure 4.5: Inconsistent Concretization

This connection-level is called its *top-connection-level* and constitutes a relationship occurrence, similar to a link in the UML.

- *Relationship class:* The top-connection-level of an m-relationship is regarded as a singleton-class with the m-relationship's instance-facet as instance. An m-relationship may connect further levels of its source m-object with levels of its target m-object. These are its non-top-connection-levels. A non-top-connection-level is regarded as a relationship class, since it has a domain and an extension (see Chapter 5). An m-relationship with only one non-top-connection-level (a two-level relationship) is similar to an association in the UML or a relationship type in the ER model. Like instantiating an association in the UML results in a link, concretizing a two-level m-relationship results in an m-relationship with at least one connection-level (a relationship occurrence).

- *Relationship meta*-class:* we are especially interested in m-relationships that go beyond these two connection-levels and have multiple, interdependent non-top-connection-levels. Concretizing a three-level m-relationship results in an m-relationship with at least two connection-levels (which play the roles of relationship occurrence and relationship class). Concretizing a four-level m-relationship results in an m-relationship with at least three connection-levels (which play the roles of relationship occurrence, relationship class and relationship metaclass). An m-relationship that has n non-top-connection-levels, with $n \geq 1$, can thus be seen as a relationship meta^{n-1}-class.

With regard to the different facets of an m-relationship, a concretization relationship between a child m-relationship and a parent m-relationship not only expresses *aggregation* but also *meta*-classification* and *specialization*, depending on which facets of the m-relationships one is looking at.

With regard to *meta*-classification*, if the child m-relationship resides on a connection-level that is present in the parent m-relationship, than it is regarded as instance of the relationship class, represented by this connection-level of its parent. Since connection-levels of parent m-relationships, in this way, are considered as relationship classes, connection-levels of parents of these parent m-relationships can be considered as metaclasses, and, in turn, connection-levels of the next higher abstraction, can be regarded as meta^2-classes, and so forth.

With regard to *specialization*, connection-levels of child m-relationships are regarded as subclasses of corresponding connection-levels of the parent m-relationship, by restricting the domain at lower connection-levels to pairs of m-objects that are descendants of m-objects connected by the child m-relationship. Another aspect of specialization is the possibility to introduce additional connection levels in child m-relationships, thus specializing the aggregation schema of the parent m-relationship. Note that the role of specialization and classification in m-relationship hierarchies becomes more apparent when considering that each connection-level of an m-relationship may introduce member-attributes that are to be instantiated by descendant m-relationships at that connection-level, and, in this way, define and specialize the *structure* of m-relationships at that level (a feature discussed in Chapter 8).

Example 4.9 (Aspects of Concretization of M-Relationship)**.** Consider m-relationship *Product-producedBy-Company* in Figure 4.4 which is concretized by *Car-producedBy-CarManufacturer*. This concretization-relationship comprises (i) *Aggregation/Decomposition*: *Car-producedBy-CarManufacturer* (its instance facet) is part of *Product-producedBy-Company* (of its instance facet) (ii) *Meta***-Classification/Instantiation*: The instace-facet of *Car-producedBy-CarManufacturer* is instance of the class-facet at connection-level *(category,industrialSector)* of *Product-producedBy-Company*. The class-facet at connection-level *(model,enterprise)* of *Car-producedBy-CarManufacturer* is instance of a metaclass-facet at connection-level *(model,enterprise)* of *Product-producedBy-Company* (iii) *Generalization/Specialization*: Class-facets of *Product-producedBy-Company* at connection levels *(model,enterprise)* and *(physicalEntity, factory)* are specialized: its static domains are restrained to the cross-product of descendants of m-object *Car* at level *model* (resp. physicalEntity) and descendants of m-object *CarManufacturer* at level *enterprise* (resp. *factory*).

Considering external identifiers of m-relationships: each root-m-relationship $\hat{r} \in \hat{R}$ has assigned a label that serves as namespace for its concretization hierarchy of m-relationships and is thus shared by all its descendants. Note that $\hat{R} \subseteq R$ is the set of root-m-relationships within a multi-level model and, thus, represents the set of concretization hierarchies of m-relationships. The namespace of an m-relationship is defined by total function $ns : R \to N$. The namespace is unique within the set of root-m-relationships and can thus be employed to refer to a hierarchy of m-relationships. An m-relationships can be externally identified by its namespace together with the names of its target-m-object and its source-m-object. Each m-relationship may optionally have a label defined by function name : $R \to N \cup \{\varepsilon\}$, this label is only informative.

Like m-objects, m-relationships may be only partially defined. A partially defined m-relationship inherits namespace and connection-levels from its parent m-relationship as defined in the following definition.

Definition 4.14 (M-Relationship Inheritance). *Given is an m-relationship $r \in R$ and its direct concretization $r' \in R$, $(r', r) \in H_R$. The namespace of r' is that of its root-m-relationship, i.e., $ns(r') := ns(\hat{r}_{r'})$. Its connection-levels $C_{r'}$ are given by the union of connection-levels introduced at r', $C^{\Delta}_{r'}$, and connection-levels of its parent m-relationships, without the parents' top-connection-levels, i.e., $C_{r'} := (C^{\Delta}_{r'} \cup \bigcup_{r \in R:(r',r) \in H_R}(C_r \setminus \{\hat{c}_r\}))$.*

Inheritance in concretization hierarchies of m-relationships gives rise to a short-hand graphical notation for m-relationships. By now, we only provided a uniform graphical representation of m-relationships and their concretizations, by labeling each m-relationship with the name of the m-relationship hierarchy together with, if available, specific names of single m-relationships, and by graphically representing each connection-level at each m-relationship (see Figure 4.4). Typically, however, single m-relationships do not come with specific names, and in most cases one does not add additional connection-levels at concretized m-relationships. Thus, graphical representation of labels and connection-levels at each concretized m-relationships is redundant and leads to graphical representations of multi-level models that are overly verbose and thus hard to comprehend. For the purpose of simplifying graphical representation we introduce a short-hand for concretized m-relationships, omitting their label and omitting inherited concretization levels. This concise representation of m-relationship hierarchies is exemplified in Figure 4.6. A similar approach is presented in [6] for the concise graphical representation of links and associations in the UML when modeling with multiple ontological classification levels.

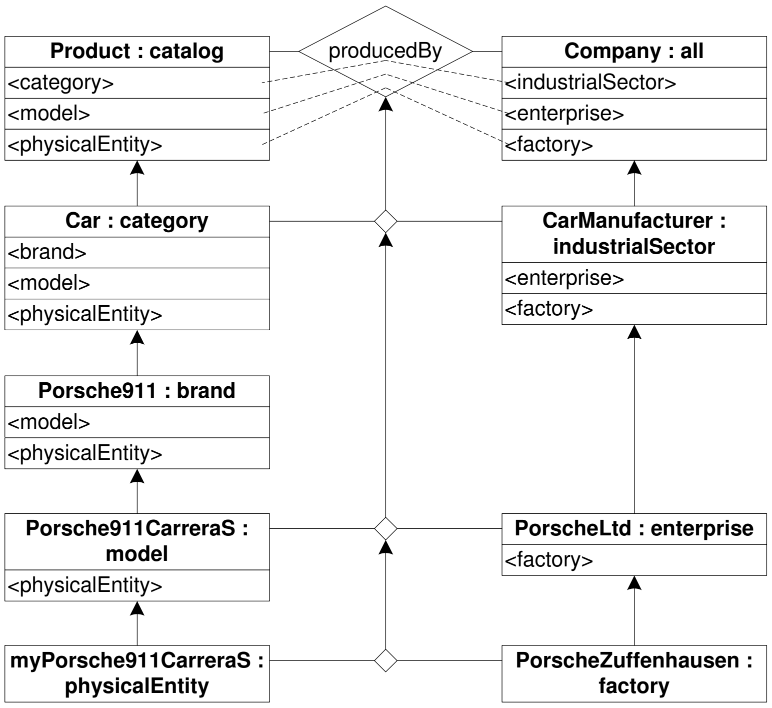


Figure 4.6: Abbreviated graphical representation of m-relationship hierarchies

4.3 Summary

In this chapter we discussed how to model entities and entity types at multiple levels of abstraction using m-objects. We also discussed how to model relationships at multiple levels of abstraction using m-relationships. We provided set theoretic definitions of their basic structure and of global consistency criteria. We complemented the approach by a discussion of hetero-homogeneous abstraction hierarchies and of how they are represented by concretization hierarchies of m-objects and of m-relationships.

Multiple concretization hierarchies of m-objects connected by concretization hierarchies of m-relationships, as discussed in this chapter, comprise a *multi-level model* (m-model). Such m-models constitute the core of the structural part of what we envisage as *conceptual modeling in the large*. In Chapter 9, we will sketch future extensions, which will complement the core m-object and m-relationship approach. Some extensions will already be provided in this thesis in the context of data warehousing (see Chapter 8).

In the following chapters we will introduce operations to create and manipulate (see Chapter 5) and to query (see Chapter 6) multi-level models based on m-objects and m-relationships. In the context of querying m-objects and m-relationships we will also discuss further aspects of m-objects and m-relationships, such as their *extensions*, *metaclass-extensions*, and —regarding m-relationships— their *active domain* at various levels of abstraction. In Chapter 9 we will provide an evaluation of the m-object and m-relationship approach, complementing the comparison of different multi-level modeling techniques (see Chapter 3).

Chapter 5

Creating and Manipulating Multi-Level Models

Contents

In this chapter we define generic operations for defining and manipulating multi-level models (*m-models*) based on m-objects and m-relationships as discussed in Chapter 4. We give formal definitions of these operations together with examples of their application. In Section 5.1, we discuss prerequisites for defining operations on multi-level models. In Section 5.2, we define operations for creating, manipulating, and deleting m-objects, levels, and attributes. In Section 5.3, we define operations for creating, manipulating, and deleting m-relationships and their connection-levels. Section 5.4 summarizes the chapter.

5.1 Introduction

In this chapter we provide formal definitions of operations for creating and manipulating m-models. Based on structural definitions provided in the previous chapter we define operations in terms of their

1. *Input and output*
2. *Preconditions*: A precondition is a condition that the input and the state of the m-model must satisfy before the operation takes place (compare [94, p.257]). An operation only takes place if all its preconditions hold. A precondition is a logical expression that ranges over input of the operation and over the pre-state of the multi-level model.
3. *Effects*: Effects of an operation are defined in terms of postconditions. A postcondition is a condition that the m-model and the output satisfy after the operation took place (compare [94, p.260]). A postcondition is a logical expression that ranges over input, output, pre-state and post-state of the multi-level model. In order to refer to the state of the multi-level model before the operation took place, the universes, sets, functors, and relations are marked with ^{pre}, for example O^{pre} is the universe of m-objects immediately before the operation took place.

These definitions ensure that if an operation is applied on a consistent m-model and if all preconditions are met, that then the resulting m-model fulfils all static structural consistency criteria (no proofs are given).

A well-known problem of specifying changes in a knowledge base using this postcondition approach is the *frame problem* [18]. It "*is the problem of stating succinctly those parts of the information base that are not affected by an event, and that therefore they must remain unchanged. [...] In practice, the frame problem is greatly mitigated by the assumption of minimal set. It is assumed that the set of structural events is minimal in the sense that if* *any of them is removed from the set, then the resulting set does not achieve the desired state in the information base.*" [94, p.266]

We make the assumption of minimal set.

Following structural definitions given in the previous chapter, a multi-level model consists of universes of m-objects O, m-relationships R, attributes A, levels L, data types D, values V, and names N. O, R, A and L are *dynamic universes*, they can expand and shrink. Thus it is possible to add elements to and delete elements from these universes. Furthermore, concretization hierarchies of m-objects and m-relationships are defined in global relations H and $H\!\!R$, respectively. Naming of elements is provided by global functions name and ns. Schema and instance data is defined in the descriptions of m-objects and m-relationships.

Each m-object in the universe of m-objects $o \in O$ has as an *essential and inseparable part* a delta description $(L_o^\Delta, A_o^\Delta, p_o^\Delta, l_o^\Delta, d_o^\Delta, v_o^\Delta)$. That is, an m-object and its delta description have the same lifetime. In our formalization we assume that when adding an m-object to the universe of m-objects, that an *empty* delta-description is created as well. When deleting an m-object from the universe of m-objects, its delta description is deleted as well. For each m-object one can dynamically derive, using *inheritance* rules (see Definition 4.4), its full description $(L_o, A_o, p_o, l_o, d_o, v_o)$ (see Definition 4.1). Analogously, the delta description of an m-relationship $r \in R$, $(\hat{r}_r, s_r, t_r, C_r^\Delta)$, is considered as essential and inseparable part of the m-relationship. In the case of m-relationships only the set of connection-levels is complemented using inheritance rules (see Definition 4.14).

We assume that when deleting an element x from a universe, all references to x are deleted as well: (1) membership of x in sets such as A_o^Δ and L_o^Δ, (2) tupels that refer to x in relations, such as H, $H\!\!R$, and C_r^Δ, or in functions, such as name, ns, p_o^Δ, l_o^Δ, d_o^Δ, v_o^Δ, are deleted (3) other references to x, such as from s_r, t_r, are unset.

In this chapter we will give extensive examples of the application of the discussed operations based on our sample problem (see Example 1.1). Our example starts with an empty multi-level model. That is, the dynamic universes of m-objects, m-relationships, attributes, and levels are empty, i.e., $O = \{\}$, $R = \{\}$, $A = \{\}$, $L = \{\}$. Also, global relations are empty, i.e., $H = \{\}$, and $H\!R = \{\}$, as well as global functions name and ns. We assume that the static universe of data types contains some common primitive data types, i.e., $\{Date,\ Integer,\ String,\ Float,\ Boolean\} \subseteq D$ and that the static universe of names N consists of all strings of alphanumeric characters, for example no longer than 30 characters, beginning with a letter.

5.2 Creating and Manipulating M-Objects

We now define the following operations for creating and manipulating m-objects. Note that square brackets denote optional parameters. In our formalization, omitting an optional parameter is equivalent to setting this parameter to ε.

- *new(n,[n′],[n″])* creates a new root m-object with namespace n, optionally a specific m-object-name n', and optionally a name of its top-level.
- *o.addLevel(n,l′)* adds a new level l to o below level l'.
- *o.addAttribute(n,l,d)* adds a new attribute a to o at level l' with domain d.
- *o.set(a,v)* sets the value of attribute a at o to value v.
- *o.concretize([n])* creates a new m-object with name n that concretizes o.
- *o.delete()* deletes m-object o.
- *o.deleteLevel(l)* deletes level l of m-object o.

- *o.deleteAttribute(a)* deletes attribute a from m-object o and from its descendants.
- *o.unset(a)* unsets attribute value a at m-object o.

Creating a New Concretization Hierarchy The first step in creating a multi-level model is to start creating concretization hierarchies by creating one or more root m-objects. Root m-objects are created using operation *new* which takes as parameter the name of the m-object hierarchy $n \in N$ which is mandatory and serves as namespace for m-objects, levels, and attributes in this hierarchy. The operation further takes as parameter m-object-name $n' \in N \cup \{\varepsilon\}$ which is optional because the root m-object of a hierarchy typically has no specific m-object name but is referred to by the name of the hierarchy (n). A newly created m-object comes with one level. Optional parameter level-name $n'' \in N \cup \{\varepsilon\}$ may provide a custom name for this level, if omitted it is labeled 'root'.

Definition 5.1 (Operation New)**.** *Given is a namespace $n \in N$, a name $n' \in N \cup \{\varepsilon\}$ and a level name $n'' \in N \cup \{\varepsilon\}$ of a to-be-created m-object. Operation new(n,n',n") is defined by the following pre- and post-conditions:*

Pre 1: $\forall o' \in O^{pre} : ns(o') \neq n$. *(there is no existing m-object with the same namespace)*

Post 1: $\exists o \in O : o \notin O^{pre}$ *(adds a new m-object o to the universe of m-objects)*

Post 2: $\exists l \in L : l \notin L^{pre}$ *(adds a new level l to the universe of levels)*

Post 3: $ns(o) = n$

Post 4: $name(o) = n'$

Post 5: $l \in L_o$

Post 6: $name(l) = \begin{cases} n'' & n'' \neq \varepsilon \\ \text{'root'} & otherwise \end{cases}$

Example 5.1 (Operation New)**.** Operation *new('Product', ε, 'catalog')* creates a new root m-object with namespace 'Product', without a specific m-object name. In the following we simply write *Product* to refer to this m-object. This informal shorthand notation leads to rather strange-looking definitions like $ns(Product)$ = 'Product' and name($Product$) = ε. We also use *Product* to refer to the whole concretization hierarchy beneath m-object *Product*. If not clear from the context we write "m-object *Product*" or "*Product* hierarchy" (without quotation marks), respectively, to distinguish between them. The operation also creates empty object-local sets and functions, referred to as $L^{\triangle}_{Product}$, $A^{\triangle}_{Product}$, $p^{\triangle}_{Product}$, $l^{\triangle}_{Product}$, $d^{\triangle}_{Product}$, $v^{\triangle}_{Product}$. It also creates a new level named 'catalog', in the following simply referred to as *catalog*, and adds it to the set of levels of m-object *Product*, i.e., $L^{\triangle}_{Product} = \{catalog\}$. Figure 5.1 depicts m-object *Product*.

Figure 5.1: Newly created m-object *Product*

Adding Levels A new level l can be added to an m-object o using operation *addLevel* which takes as parameter the name $n \in N$ of the to-be-introduced level and its parent level $l' \in L$. The name of the new level must be unique within the namespace of o (Pre. 5.2.1) and there must not be any concretizations of o below level l' (Pre. 5.2.2). Given that these preconditions are met, the operation adds a new level l to the universe of levels. This new level is labeled n (Post. 5.2.2) and is added to the set of levels of o (Post. 5.2.3). The provided parent-level l' is set as the parent-level of l (Post. 5.2.4). If parent-level l' used to be parent of another level l'' then the newly introduced level l becomes the new parent-level of l'' (Post. 5.2.5).

Definition 5.2 (Operation Add Level)**.** *Given are an m-object $o \in O$ together with a name $n \in N$ and a parent-level $l' \in L$ of a to-be-introduced* *level. Operation $o.addLevel(n, l')$ is defined by the following pre- and post-conditions:*

Pre 1: $\forall l'' \in L, \forall o' \in O : l'' \in L_{o'} \wedge ns(o') = ns(o) \Rightarrow name(l'') \neq n$ *(name of level is unique within an m-object hierarchy)*

Pre 2: $\forall o', o'' \in O : (o', o) \in H^* \wedge l' = \hat{l}_{o'} \Rightarrow (o'', o') \notin H^+$ *(no m-objects below parent level)*

Post 1: $\exists l \in L : l \notin L^{pre}$ *(adds a new level l to the universe of levels L*

Post 2: $name(l) = n$

Post 3: $l \in L_o^{\Delta}$

Post 4: $p_o^{\Delta}(l) = l'$

Post 5: $\forall l'' \in L_o : p_o^{pre}(l'') = l' \Rightarrow p_o^{\Delta}(l'') = l$ *(re-organized level-hierarchy)*

Note that there are different design options concerning preconditions for operation *addLevel*: We may alternatively want to guarantee, that the bottom-level is the same for all m-objects within a concretization hierarchy. This could be achieved by adding rule *stable bottom level:* $(\forall o' \in O : l' = \check{l}_o \Rightarrow (o, o') \notin H)$ to the set of preconditions, which says that only root m-objects may introduce a level below the former bottom level.

Example 5.2 (Operation Add Level)**.** We now add levels to m-object *Product* applying the following operations (Figure 5.2 depicts m-object *Product* after application of these operations):

1. *Product.addLevel('category',catalog)* adds level *category* to m-object *Product* below level *catalog*. The set of levels and the order of levels of *Product* after this operation are as follows: $L^{\Delta}_{Product} = \{catalog, category\}$, $p^{\Delta}_{Product} = \{category \mapsto catalog\}$

2. Now we might skip, for the moment, level *model* and introduce first level *physicalEntity* below level *category* by operation

Product.addLevel('physicalEntity',category) resulting in the following state of the m-model: $L^{\triangle}_{Product} = \{catalog, category, physicalEntity\}$, $p^{\triangle}_{Product} = \{category \mapsto catalog, physicalEntity \mapsto category\}$

3. Finally we add level *model* below level *category* by operation *Product.addLevel('model',category)*. Postcondition *re-organized level-hierarchy* (Post. 5.2.5) guarantees that level *model* is added between levels *category* and *physicalEntity* by redefining parent-level of *physicalEntity* to level *model*, i.e., $L^{\triangle}_{Product} = \{catalog, category, physicalEntity, model\}$, $p^{\triangle}_{Product} = \{category \mapsto catalog, physicalEntity \mapsto model, model \mapsto category\}$.

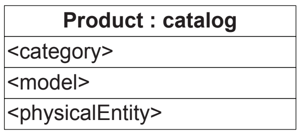


Figure 5.2: Resulting m-object *Product* after *add level* operations

Adding Attributes A new attribute can be added to an m-object o using operation *addAttribute* which takes as parameter the name n of the to-be-introduced attribute, the level l to which it should be assigned, and a data type d that serves as its domain. To be applicable, the name n of the new attribute must be unique within the namespace of o (Pre. 5.3.1) and level l has to be a level of o (Pre. 5.3.2). Given that these preconditions are met, the operation adds a new attribute a to the universe of attributes. The name of this new attribute is set to the provided name n (Post. 5.3.2). The new attribute a is added to the set of attributes of o (Post. 5.3.3), is assigned to level l (Post. 5.3.4), and is given domain d (Post. 5.3.5) in the delta definition of m-object o.

Definition 5.3 (Operation Add Attribute)**.** *Given are an m-object $o \in O$, a name $n \in N$ of a to-be-introduced attribute, the level $l \in L$ it should be assigned to, and its domain $d \in D$. Operation $o.addAttribute(n, l, d)$ is defined by the following pre- and post-conditions:*

Pre 1: $\forall a' \in A, \forall o' \in O : a' \in A_{o'} \wedge ns(o') = ns(o) \Rightarrow name(a') \neq n$ *(unique attribute-name)*

Pre 2: $l \in L_o$

Post 1: $\exists a \in A : a \notin A^{pre}$ *(adds a new attribute a to the universe of attributes)*

Post 2: $name(a) = n$

Post 3: $a \in A_o^{\Delta}$

Post 4: $l_o^{\Delta}(a) = l$

Post 5: $d_o^{\Delta}(a) = d$

Example 5.3 (Add Attribute)**.** Operation *Product.addAttribute('desc', catalog, String)* adds attribute *desc* to top-level *catalog* of m-object *Product* and defines its domain to be data type *String*. Further attributes are added to lower levels by operations *Product.addAttribute('taxRate', category, Integer)*, *Product.addAttribute('listPrice', model, Float)*, and *Product.addAttribute('serialNr', physicalEntity, String)*, leaving relevant object-local sets and functions as follows: $A^{\Delta}_{Product} = \{desc, taxRate, listPrice, serialNr\}$, $l^{\Delta}_{Product} = \{desc \mapsto catalog, taxRate \mapsto category, listPrice \mapsto model, serialNr \mapsto physicalEntity\}$, $d^{\Delta}_{Product} = \{desc \mapsto String, taxRate \mapsto Integer, listPrice \mapsto Float, serialNr \mapsto String\}$. Figure 5.3 depicts m-object *Product* after application of these operations.

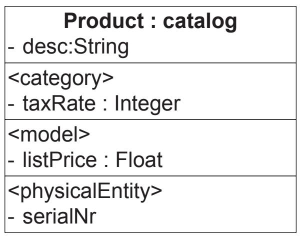


Figure 5.3: M-object *Product* after *add attribute* operations

Setting Values An attribute a can be set to value v at m-object o using operation $o.set(a, v)$ given that attribute a is an attribute of o (Pre. 5.4.1) and given that v is a member of the domain of attribute a (Pre. 5.4.2). If these preconditions are met, the operation sets the value of a to v (Post. 5.4.1), and has to ensure that this value is not overwritten in delta-descriptions of descendant m-objects of o, that is, value assignments to attribute a in descendant m-objects are deleted (Post. 5.4.2). The latter postcondition is applicable when setting values of attributes that are assigned to non-top-levels of o.

Definition 5.4 (Operation Set Value)**.** *Given are an m-object $o \in O$, an attribute $a \in A$ and a value $v \in V$. Operation $o.set(a, v)$ is defined by the following pre- and post-conditions:*

Pre 1: $a \in A_o$

Pre 2: $v \in d_o(a)$

Post 1: $v_o^{\Delta}(a) = v$

Post 2: $\forall o' \in O : (o', o) \in H^+ \Rightarrow v_{o'}^{\Delta}(a)$ *is undefined*

For notational convenience one can also use the usual dot-notation (see Ex. 5.4) as shorthand for setting (and also for querying) attribute values.

Example 5.4 (Set Values)**.** Operation *Product.setValue(desc, 'Our Products')* sets the value of attribute *desc* at m-object *Product* to 'Our Products'. Instead of *Product.setValue(desc, 'Our Products')* one can synonymously write: *Product.desc = 'Our Products'*. The object-local attribute-value function is then given as: $v_{Product}^{\Delta} = \{desc \mapsto \text{'Our Products'}\}$. Figure 5.4 depicts m-object *Product* after application of this operation.

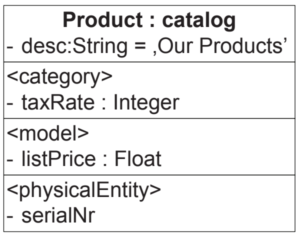


Figure 5.4: M-object *Product* after *set value* operation

Concretizing M-Objects An existing m-object o' may be concretized by a new m-object o using operation *concretize*, which takes as optional parameter the name n of the to-be-created m-objects. For the operation to be applicable the to-be-concretized m-object o' must have at least two levels (Pre. 5.5.1), since its top-level is not shared by its concretizations. If a name n is provided, this name has to be unique within this concretization hierarchy (Pre. 5.5.2). Name n may be omitted, in this case a name is generated by concatenating the name of the level that is going to be the top-level of the new m-object with an auto-incremented number. Given that the preconditions are met, the operation adds a new m-object o to the universe of m-objects, asserts that is is a direct concretization of o' (Post. 5.5.2), and sets its name (Post. 5.5.3).

Definition 5.5 (Operation Concretize)**.** *Given are a to-be-concretized m-object $o' \in O$, and optionally a name $n \in N \cup \{\varepsilon\}$ of a to-be-created m-object. Operation o'.concretize(n) is defined by the following pre- and post-conditions:*

Pre 1: $|L_{o'}| > 1$ *(o' has at least two levels)*

Pre 2: $\forall o'' \in O : name(o'') \neq n \vee ns(o'') \neq ns(o')$ *(unique m-object name)*

Post 1: $\exists o \in O : o \notin O^{pre}$ *(adds a new m-object o to the universe of m-objects)*

Post 2: $(o, o') \in H$

Post 3: $name(o) = n$

Example 5.5 (Concretize M-Object)**.** Operation *Product.concretize('Car')* concretizes m-object *Product* by adding a new m-object with name 'Car' to the universe of m-objects (in the following referred to as *Car*) and asserts that m-object *Car* is a direct concretization of m-object *Product*, i.e., $(\mathit{Car}, \mathit{Product}) \in H$. The delta descriptions of *Car* are empty, i.e., $L^{\Delta}_{\mathit{Car}} = \{\}$, $A^{\Delta}_{\mathit{Car}} = \{\}$, $p^{\Delta}_{\mathit{Car}} = \{\}$, $l^{\Delta}_{\mathit{Car}} = \{\}$, $d^{\Delta}_{\mathit{Car}} = \{\}$, $v^{\Delta}_{\mathit{Car}} = \{\}$. This delta description is the basis of the most concise graphical representation of concretization hierarchies (see left of Figure 5.5).

The full description of m-object *Car* is derived by complementing its (empty) delta description with the description of m-object *Product* using previously defined inheritance rules (see Def. 4.4): $L_{\mathit{Car}} = \{\mathit{category}, \mathit{model}, \mathit{physicalEntity}\}$, $A_{\mathit{Car}} = \{\mathit{taxRate}, \mathit{listPrice}, \mathit{serialNr}\}$, $p_{\mathit{Car}} = \{\mathit{physicalEntity} \mapsto \mathit{model}, \mathit{model} \mapsto \mathit{category}\}$, $l_{\mathit{Car}} = \{\mathit{taxRate} \mapsto \mathit{category}, \mathit{listPrice} \mapsto \mathit{model}, \mathit{serialNr} \mapsto \mathit{physicalEntity}\}$, $d_{\mathit{Car}} = \{\mathit{taxRate} \mapsto \mathit{Integer}, \mathit{listPrice} \mapsto \mathit{Float}, \mathit{serialNr} \mapsto \mathit{String}\}$, $v_{\mathit{Car}} = \{\}$. This full description is the basis of the verbose graphical representation of concretization hierarchies (see middle of Figure 5.5). The verbose graphical representation may be used when only representing the child m-object but not the parent m-object.

The standard graphical representation (as introduced in Chapter 4) of concretization hierarchies is in between depicting the delta description of an m-object and depicting the full description. Instead only those aspects of the full descriptions that seem relevant for understanding are depicted: its full level-hierarchy together with newly added attributes and inherited attributes that have been assigned a value (see right of Figure 5.5).

Concretizing an m-object also typically means to instantiate attributes and to extend level descriptions by introducing additional attributes and even by introducing additional levels using the above introduced operations. In Example 5.6 we will look at how to describe and extend a child m-object by setting its delta descriptions and also exemplify full descriptions of m-objects which are derived using inheritance rules as introduced in Definition 4.4.

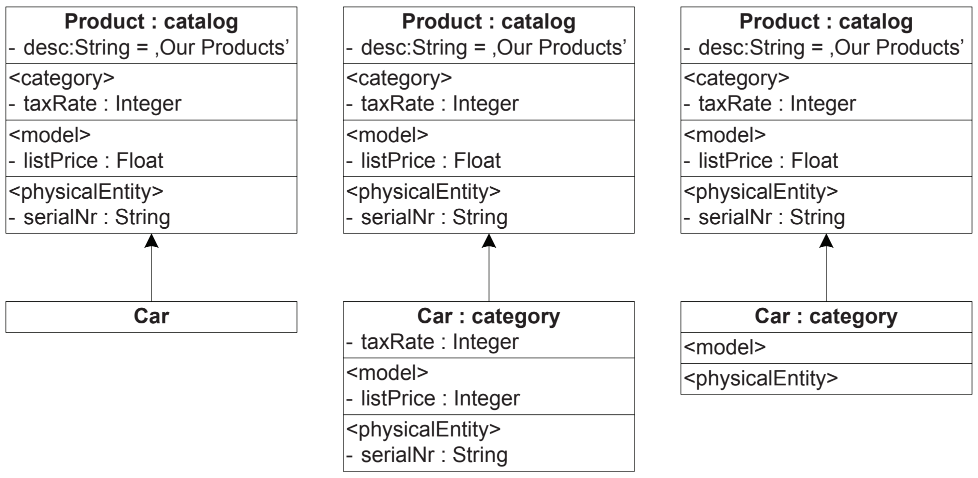


Figure 5.5: M-object *Product* with concise (left), full (center), and standard (right) graphical representations of its concretization *Car*

Example 5.6 (Concretize M-Object - Instantiate and Specialize)**.** Our just created m-object *Car* (see Ex. 5.5 and Figure 5.5) has an empty delta description. Now we add additional levels and attributes and set attribute values:

Operation *Car.addLevel('brand', category)* adds a level named 'brand' below level category and adds it to the set of levels of *Car*, i.e., $L^{\Delta}_{Car} = \{brand\}$, and defines that level *brand* has level *category* as its parent, and that *brand* is parent of *model*, which formerly had *category* as its parent level, i.e., $p^{\Delta}_{Car} = \{brand \mapsto category, model \mapsto brand\}$. The full description of *Car* is: $L_{Car} = \{category, model, physicalEntity, brand\}$, $p_{Car} = \{physicalEntity \mapsto model, model \mapsto brand, brand \mapsto category\}$. Further attributes are introduced by operations *Car.addAttribute('market-Launch', brand, date)*, *Car.addAttribute('maxSpeed', model, Integer)* and *Car.addAttribute('mileage', physicalEntity, Integer)*. The value of attribute *taxRate* at top-level *category* is set to *20* by operation *Car.taxRate = 20*.

These operations change delta descriptions of *Car* to: $A^{\Delta}_{Car} = \{marketLaunch, maxSpeed, mileage\}$, $l^{\Delta}_{Car} = \{marketLaunch \mapsto brand, maxSpeed \mapsto model, mileage \mapsto physicalEntity\}$, $d^{\Delta}_{Car} =$

$\{marketLaunch \mapsto date, maxSpeed \mapsto Integer, mileage \mapsto Integer\}$, $v^{\Delta}_{Car} = \{taxRate \mapsto 20\}$ (see left of Figure 5.6).

The full description of *Car* is then: $A_{Car} = \{taxRate, listPrice, serialNr, marketLaunch, maxSpeed, mileage\}$, $l_{Car} = \{taxRate \mapsto category, listPrice \mapsto model, serialNr \mapsto physicalEntity, marketLaunch \mapsto brand, maxSpeed \mapsto model, mileage \mapsto physicalEntity\}$, $d_{Car} = \{taxRate \mapsto Integer, listPrice \mapsto Float, serialNr \mapsto String, marketLaunch \mapsto date, maxSpeed \mapsto Integer, mileage \mapsto Integer\}$, $v_{Car} = \{taxRate \mapsto 20\}$ (see right of Figure 5.6).

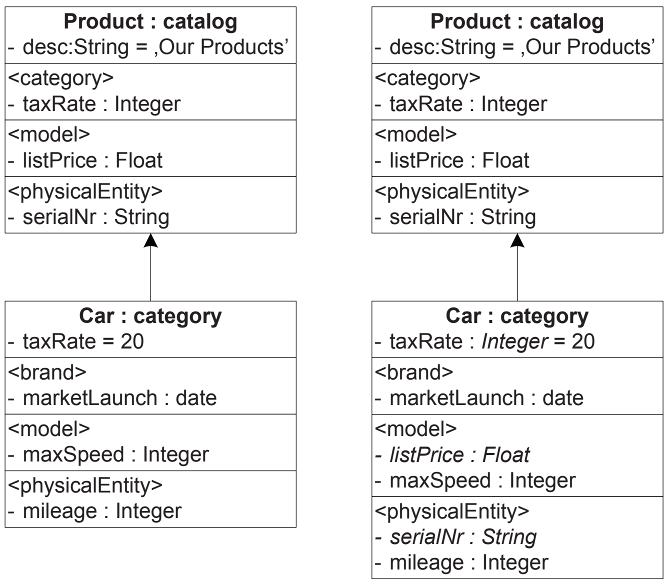


Figure 5.6: Standard representation of child m-object *Car* (left) and its full representation including inherited attributes (right)

Deleting M-Objects An m-object o may be deleted using operation *o.delete()*, given that it has no descendant m-objects (Pre. 5.6.1) and given that it is not part of an m-relationship (Pre. 5.6.2). If these preconditions are fulfilled, the operation deletes m-object o from the universe of m-objects and deletes all object-local sets and functions. It also deletes attributes and levels that where introduced at o from the respective universes (Post. 5.6.3 and Def. 5.6.4).

Definition 5.6 (Operation Delete)**.** *Given is an m-object $o \in O$. Operation* o.delete() *is defined by the following pre- and post-conditions:*

Pre 1: $\forall o' \in O : (o', o) \notin H^+$

Pre 2: $\forall r \in R : s_r \neq o \wedge t_r \neq o$

Post 1: $o \notin O$ *(deletes o from the universe of m-objects)*

Post 2: deletes object-local sets and functions $L_o^\triangle, A_o^\triangle, p_o^\triangle, l_o^\triangle, d_o^\triangle, v_o^\triangle$

Post 3: $\forall a \in A^{pre} : a \in A_o^{\triangle,pre} \Rightarrow a \notin A$

Post 4: $\forall l \in L^{pre} : l \in L_o^{\triangle,pre} \Rightarrow l \notin L$

Example 5.7 (Delete M-Object)**.** Operation *Car.delete* is applicable since m-object *Car* has no concretizations and is not part of any m-relationships. The operation deletes m-object *Car* from the universe of m-objects and deletes object-local sets and functions, $L_{Car}^\triangle$, $A_{Car}^\triangle$, $p_{Car}^\triangle$, $l_{Car}^\triangle$, $d_{Car}^\triangle$, $v_{Car}^\triangle$. It further deletes attributes that were introduced at *Car*, i.e., *marketLaunch*, *maxSpeed*, and *mileage*, from the universe of attributes A. Moreover, it deletes level *brand*, which was introduced at *Car*, from the universe of levels L.

Deleting Levels A level l of m-object o can be deleted using operation $o.deleteLevel(l)$. The to-be-deleted level l must have been introduced at o (Pre. 5.7.1), must not be the top-level of o, and there must not be any descendant m-object at level l (Pre. 5.7.2). There must also be no m-relationships having l as part of a connection-level (Pre. 5.7.3). Given that these preconditions are met, the operation deletes l from the universe of levels. We assume that, when removing levels from the universe of levels, all references to these levels are deleted as well. After its execution, in any m-object o', any level l' that formerly had l as its parent has then the parent of l as its parent (Post. 5.7.2). Note that o' is o or a descendant of o following from the unique induction rule for levels. Each attribute a that has been assigned to level

l in any m-object o' is deleted from the universe of attributes (Post. 5.7.3), note again that o' is o or a descendant of o. This operation is exemplified in Example 5.8.

Definition 5.7 (Operation Delete Level)**.** *Given are an m-object $o \in O$ and a to-be-deleted level $l \in L$. Operation $o.deleteLevel(l)$ is defined by the following pre- and post-conditions:*

Pre 1: $l \in L_o^\Delta$

Pre 2: $\forall o' \in O : \hat{l}_{o'} \neq l$

Pre 3: $\forall r \in R, \forall l', l'' \in L : (l', l'') \in C_r \Rightarrow l' \neq l \wedge l'' \neq l$

Post 1: $l \notin L$ *(deletes l from the universe of levels)*

Post 2: $\forall l' \in L, \forall o' \in O : p_{o'}^{\Delta,pre}(l') = l \Rightarrow p_{o'}^{\Delta}(l') = p_{o'}^{pre}(l)$

Post 3: $\forall o' \in O, \forall a \in A^{pre} : l_{o'}^{pre}(a) = l \Rightarrow a \notin A$

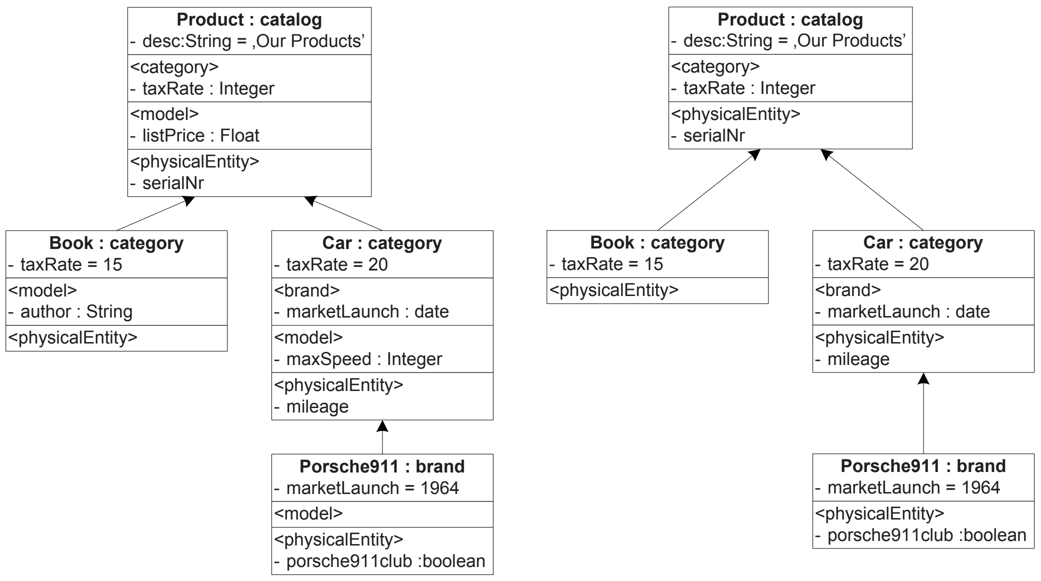


Figure 5.7: A sample m-model before (left) and after (right) operation *Product.deleteLevel(model)*

Example 5.8 (Delete Level). Consider m-objects as depicted in the left part of Figure 5.7. Operation *Product.deleteLevel(model)* is applicable, because level *model* is introduced at m-object *Product*, because it is not the top-level of *Product*, because there are no m-objects at level *model*, and because there are no m-relationships having connecting level *model*. The right part of Figure 5.7 depicts the multi-level model after application of *Product.deleteLevel(model)*. Level *model* and all attributes assigned to level *model* (*listPrice*, *maxSpeed*, and *author*) have been deleted. In m-object *Product*, the parent level of *physicalEntity* has been changed to *category*; in m-object *Car*, where level *brand* was introduced between *model* and *category*), the parent-level of *physicalEntity* has been changed to *brand*.

Deleting Attributes Attributes can be deleted at m-objects where they were introduced. It is not possible to delete inherited attributes. We assume that, when removing attributes from the universe of attributes, all references to these attributes are deleted as well.

Definition 5.8 (Operation Delete Attribute). *Given are an m-object $o \in O$ and a to-be-deleted attribute $a \in A$. Operation $o.deleteAttribute(a)$ is defined by the following pre- and post-conditions:*

Pre 1: $a \in A_o^{\Delta,pre}$ (a was introduced at o)

Post 1: $a \notin A$ (deletes a from the universe of attributes)

Example 5.9 (Delete Attribute). Consider attribute *taxRate* at level *model* introduced in m-object *Product* and its values in m-objects *Car* and *Book* (see left part of Figure 5.8). Operation *Product.deleteAttribute(taxRate)* can be applied since attribute *taxRate* was introduced in m-object *Product*. In result, it deletes *taxRate* from the universe of attributes and, consequently, also from the set of attributes of *Product*, $A_{Product}$, and also deletes respective values in m-objects *Book* and *Car* (see right part of Figure 5.8).

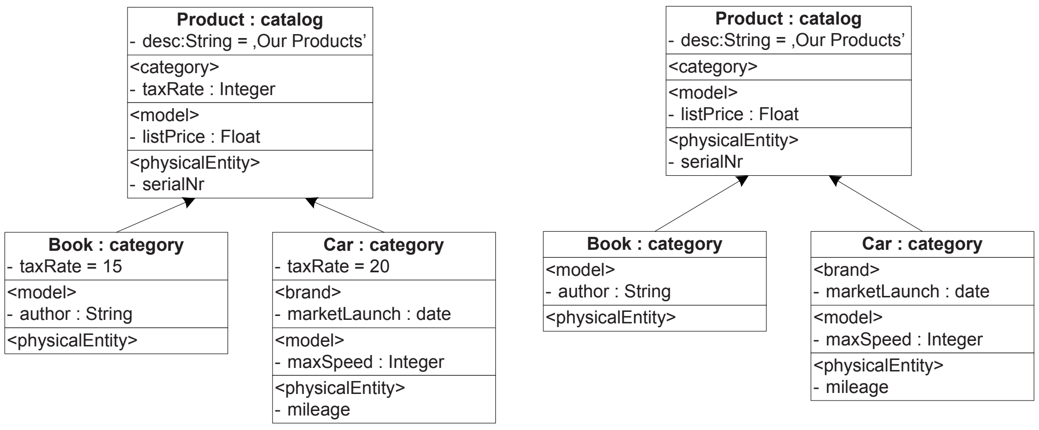


Figure 5.8: A sample m-model before (left) and after (right) operation *Product.deleteAttribute(taxRate)*

Unsetting Values The value of an attribute a at an m-object o can be unset (set to nil). Operation $o.unset(a)$ can be applied if a value is assigned to a in the delta-description of o, i.e., the value is not inherited from ancestor m-objects, (Pre. 5.9.1). Its only effect is to unset the value of a (Post. 5.9.1).

Definition 5.9 (Operation Unset Value)**.** *Given are an m-object $o \in O$ and an attribute $a \in A$. Operation $o.unset(a)$ is defined by the following pre- and post-conditions:*

Pre 1: $v_o^\Delta(a)$ is defined.

Post 1: $v_o^\Delta(a)$ is undefined

5.3 Creating and Manipulating M-Relationships

We now define the following operations for creating and manipulating m-relationships.

- *newMRel(n,o,o',[n'])* creates a new root m-relationship with namespace n, source m-object o, target m-object o', and optionally a name n'.
- *r.addConnLevel(l,l')* adds connection-level (l,l') to m-relationship r.
- *r.concretize(o,o',[n])* creates a new m-relationship with name n, if provided, that concretizes r and connects source m-object o and target m-object o'.
- *r.delete* deletes m-relationship r.
- *r.delConnLevel(l,l')* deletes connection-level (l,l') from r.

Creating Root M-Relationships The first step in creating a hierarchy of m-relationships is to create a root m-relationship r. Root m-relationships are created using operation newMRel which takes as parameter the name n of the m-relationship hierarchy. Parameter n is mandatory and serves as namespace for descending m-relationships. Furthermore, parameters specify references to a source m-object o and to a target m-object o'. Optionally one may give a specific name n' for m-relationship r. Name n' is typically omitted, i.e., $n' = \varepsilon$, since the root m-relationship may be identified by the name n of the m-relationship hierarchy. The only precondition for this operation to be applicable is that namespace n has not been used already (Pre. 5.10.1). The operation adds a new m-relationship to the universe of m-relationships and sets its source m-object to o and its target m-object to o' (Post. 5.10.3). Moreover, it introduces one connection-level that connects the top-levels of o and o' (Post. 5.10.4), and sets namespace and name of the newly created m-relationship (Post. 5.10.5 and Def. 5.10.6).

Definition 5.10 (Operation New M-Relationship)**.** *Given is a namespace $n \in N$, a name $n' \in N \cup \varepsilon$, a source m-object $o \in O$, and a target m-object $o' \in O$. Operation newMRel$(n, o, o', [n'])$ is defined by the following pre- and post-conditions:*

Pre 1: $\forall r' \in R : ns(r') \neq n$

Post 1: $\exists r \in R : r \notin R^{pre}$ *(adds a new m-relationship r to the universe of m-relationships)*

Post 2: $\hat{r}_r = r$ *(refers to itself as root m-relationship)*

Post 3: $s_r = o \wedge t_r = o'$

Post 4: $C_r^{\Delta} = \{(\hat{l}_o, \hat{l}_{o'})\}$

Post 5: $ns(r) = n$

Post 6: $name(r) = n'$

Example 5.10 (Operation New M-Relationship)**.** Consider m-objects *Product* and *Company* (see Figure 5.9). Operation *newMrel('producedBy', Product, Company)* creates a new root m-relationship labelled 'producedBy'; this label serves as namespace for the whole hierarchy of m-relationships beneath this relationship. As is typical for root m-relationships, there is no specific name for this m-relationship. In the following we simply write *producedBy* to refer to this m-relationship. We also use *producedBy* to refer to the whole concretization hierarchy beneath m-relationship *producedBy*. To make explicit that we refer to this *producedBy*-relationship we also write *Product-producedBy-Company*. The operation also creates local variables $s_{producedBy}$ and $t_{producedBy}$ and sets it to m-objects *Product* and *Company*, respectively, i.e., $s_{producedBy} = Product$, $t_{producedBy} = Company$. It also creates a new connection-level that connects the top-level of source m-object *Product*, which is *catalog*, with the top-level of target m-object *Company*, which is *root*, i.e., $C_{producedBy}^{\Delta} = \{(catalog, root)\}$.

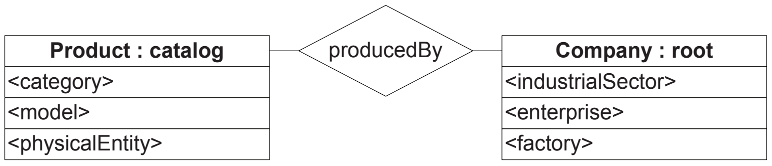


Figure 5.9: Resulting root m-relationship *producedBy* after application of operation *newMRel('producedBy', Product, Company)*

Adding connection-levels Immediately after creation, a new root m-relationship r has one connection-level. Additional connection-levels are created using operation *addConnLevel* which takes as parameter two levels l and l'. For the operation to be applicable l has to be a level of the source m-object of r and l' has to be a level of the target m-object of r (Pre. 5.11.1); there must not be an existing connection-level in r that connects these two levels (Pre. 5.11.2). Also, there must not be a descendant m-relationship below this connection-level (Pre. 5.11.3), since this could lead to inconsistencies in concretizations of r below connection-level (l,l') with regard to rule *connection-level containment* (see Definition 4.12). Given that these preconditions are met, the operation adds a new connection-level to the delta description of r (Post. 5.11.1).

Definition 5.11 (Operation Add Connection-Level)**.** *Given is an m-relationship r, a source-level l, and a target-level l'. Operation r.addConnLevel(l,l') is defined by the following pre- and post-conditions:*

Pre 1: $l \in L_{s_r} \wedge l' \in L_{t_r}$

Pre 2: $(l, l') \notin C_r$

Pre 3: $\forall r' \in R : r' \preceq r \Rightarrow (l, l') \preceq \hat{c}_{r'}$

Post 1: $(l, l') \in C_r^{\Delta}$

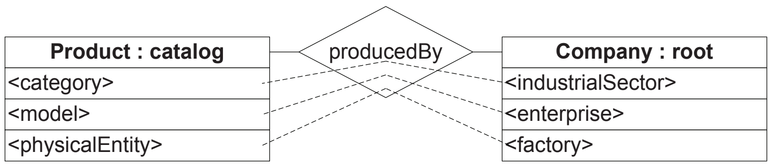


Figure 5.10: Resulting root m-relationship *producedBy* after application of *addConnLevel*-operations

Example 5.11 (Operation Add Connection-Level)**.** We now add further connection-levels to m-relationship *producedBy*.

1. *producedBy.addConnLevel(model,enterprise)* adds connection-level *(model,enterprise)* to m-relationship *producedBy*
2. *producedBy.addConnLevel(physicalEntity,factory)* adds connection-level *(physicalEntity,factory)* to m-relationship *producedBy*
3. *producedBy.addConnLevel(category,industrialSector)* adds connection-level *(category,industrialSector)* to m-relationship *producedBy*

The resulting set of connection-levels is then: $C^{\Delta}_{producedBy} = \{$*(catalog, root), (model, enterprise), (physicalEntity, factory), (category, industrialSector)* $\}$ as depicted in Figure 5.10, note that non-top-connection-levels are depicted as dashed lines.

Also note that pairs of levels and thus connection-levels are ordered (as introduced in Section 4.2), in this example the order is: $(\text{physicalEntity}, \text{factory}) \prec (\text{model}, \text{enterprise}) \prec (\text{category}, \text{industrialSector}) \prec (\text{catalog}, \text{root})$.

As a design variant, introducing a connection-level below the former bottom-level may only be allowed in root m-relationships that have not been concretized (*stable bottom connection-level*).

Concretizing M-Relationships Concretizing an m-relationship r' refers to introducing a new m-relationship r below r', that is in the (sub-)hierarchy of r', this is done using operation *concretize* which takes as parameter a source m-object o and a target m-object o', and optionally a name n. To be applicable, the coordinate (o, o') has to be below the pair of source- and target m-objects of r' (Pre. 5.12.1). In order to ensure consistent concretization (see Def. 4.12), the to-be created m-relationship must not have a parent which has its second-top connection-level above o and o'. Note that this also has to consider m-relationships that are not in the sub-hierarchy of r' (Pre. 5.12.2). There must not be an existing m-relationship r'', connecting o and o', in this hierarchy (Pre. 5.12.3). If these preconditions are fulfilled, operation *r'.concretize(o, o', n')* adds a new m-relationship r to the universe of m-relationships (Post. 5.12.1), sets its root m-relationship to the root m-relationship of r' (Post. 5.12.2), sets its source to o and its target to o' (Post. 5.12.3) and asserts its name (Post. 5.12.4).

Definition 5.12 (Operation Concretize M-Relationship)**.** *Given are an m-relationship $r' \in R$, a source m-object $o \in O$, a target m-object $o' \in O$, and optionally a name $n \in N \cup \{\varepsilon\}$. Operation r'.*concretize(o, o', n') *is defined by the following pre- and post-conditions:*

Pre 1: $(o, o') \prec (s_{r'}, t_{r'})$

Pre 2: $\forall r'' \in R : \hat{r}_{r''} = \hat{r}_{r'} \wedge (o, o') \prec (s_{r''}, t_{r''}) \wedge (\nexists r''' \in R : \hat{r}_{r'''} = \hat{r}_{r'} \wedge (o, o') \preceq (s_{r'''}, t_{r'''}) \prec (s_{r''}, t_{r''})) \Rightarrow \nexists (l, l') \in (C_{r''} \setminus \{\hat{c}_{r''}\}) : (\hat{l}_o, \hat{l}_{o'}) \prec (l, l'))$ *(*ensuring consistent concretization (Def. 4.12)*)*

Pre 3: $\nexists r'' \in R : r'' \preceq r' \wedge (o, o') = (s_{r''}, t_{r''})$

Post 1: $\exists r \in R : r \notin R^{pre}$ *(adds a new m-relationship r to the universe of m-relationships)*

Post 2: $\hat{r}_r = \hat{r}_{r'}$

Post 3: $s_r = o \wedge t_r = o'$

Post 4: $name(r) = n$

Example 5.12 (Operation Concretize M-Relationship)**.** M-relationship *producedBy* between m-objects *Product* and *Company* is concretized between m-objects *Car* and *CarManufacturer* using operation *Product-producedBy-Company.concretize(Car, CarManufacturer)* (see top of Figure 5.11). This operation is applicable since, first, *Car* is at level *category* and *CarManufacturer* is at level *industrialSector*, hence their connection is at connection-level *(category, industrialSector)*, which is the second-top-level of *Product-producedBy-Company*; second, *Car* is a concretization of *Product* and *CarManufacturer* is a concretization of *Company*; and,

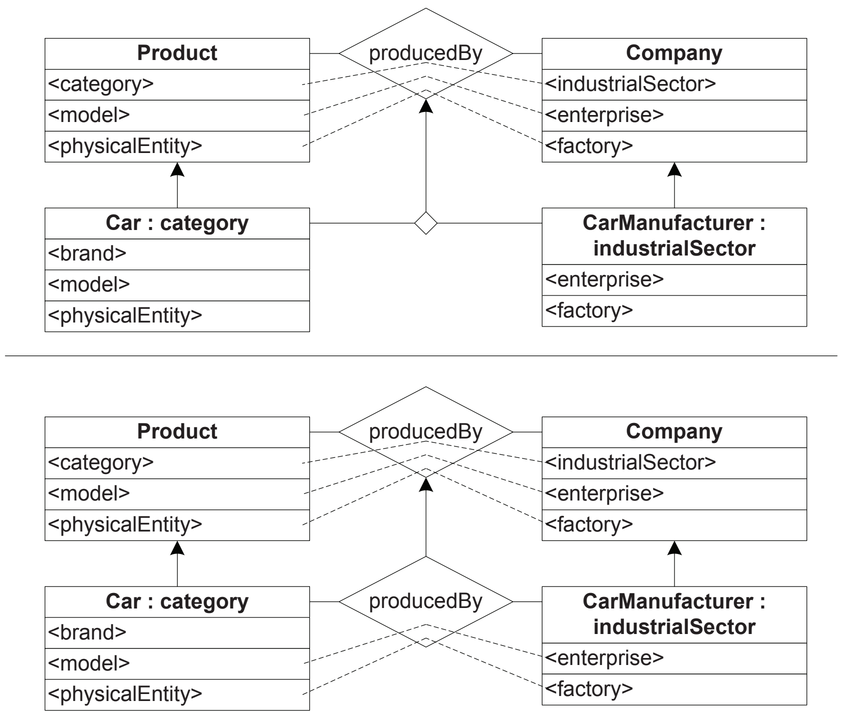


Figure 5.11: Resulting m-relationship *Car-producedBy-CarManufacturer* after application of operation *concretize*. Top: inherited connection-levels not shown. Bottom: Inherited connection-levels shown

third, there is no existing *producedBy*-m-relationship between *Car* and *CarManufacturer*. The operation adds a new m-relationship, referred to as *Car-producedBy-CarManufacturer*, to the universe of m-relationships and asserts that it is a direct concretization of *Product-producedBy-Company*, i.e., (*Car-producedBy-CarManufacturer*, *Product-producedBy-Company*)$\in H\!R$. It sets source and target of the new m-relationship to *Car* and *CarManufacturer*, respectively, i.e., $s_{\text{Car-producedBy-CarManufacturer}} =$ *Car* and $t_{\text{Car-producedBy-CarManufacturer}} =$ *CarManufacturer* and does not assign a specific name, i.e., *name*(*Car-producedBy-CarManufacturer*) $= \varepsilon$. The newly created m-relationship does not introduce additional connection-levels, i.e., $C^{\Delta}_{\text{Car-producedBy-CarManufacturer}} = \{\}$, but it inherits connection-levels from its parent m-relatioship *Product-producedBy-Company*, using inheritance rules as introduced in Def. 4.14 resulting in a set of connection-levels $C_{\text{Car-producedBy-CarManufacturer}} =$

$\{$(model, enterprise), (physicalEntity, factory), (category, industrialSector)$\}$ (see bottom of Figure 5.11).

Deleting M-Relationships An m-relationship r may be deleted, using operation *delete*, given that r is a leaf m-relationship, that is an m-relationship without concretizations.

Definition 5.13 (Operation Delete M-Relationship)**.** *Given is an m-relationship $r \in R$. Operation r.delete() is defined by the following pre- and post-conditions:*

Pre 1: $\nexists r' \in R : r' \prec r$ (r has no concretizations)

Post 1: $r \notin R$ (deletes r from the universe of m-relationships)

Deleting connection-levels A connection-level (l, l') of an m-relationship r can be deleted using operation *delConnLevel*, which takes as parameter a source level l and a target level l', which identify the to-be-deleted connection-level. For this operation to be applicable, there must be a connection-level that connects l and l' and that was introduced in r (Pre. 5.14.1). The to-be-deleted connection-level must not be the top-connection-level of r (Pre. 5.14.2). Furthermore, there must not be any concretizations of r at connection-level (l, l') (Pre. 5.14.3). Given that these preconditions are met, the operation deletes the given connection-level from the set of connection-levels defined in the delta description of r (Post. 5.14.1).

Definition 5.14 (Operation Delete connection-level)**.** *Given is an m-relationship $r \in R$, a source-level $l \in L$ and a target-level $l' \in L$. Operation r.delConnLevel(l,l') is defined by the following pre- and post-conditions:*

Pre 1: $(l, l') \in C_r^{\Delta}$

Pre 2: $(l, l') \neq \hat{c}_r$

Pre 3: $\forall r' \in R : r' \prec r \Rightarrow \hat{c}_{r'} \neq (l, l')$

Post 1: $(l, l') \notin C_r^{\Delta}$

5.4 Summary

In this chapter we gave formal definitions, in terms of pre- and postconditions, of generic operations for defining and manipulating multi-level models based on m-objects and m-relationships. The operational model of the m-objects/m-relationships approach is complemented by generic query operations as introduced in the next chapter. Custom operations can be encoded in terms of these generic operations (see Chapter 9 for a sketch of methods in multi-level models).

These generic operations can be implemented on top of object-oriented programming languages, or on top of object-oriented or object-relational databases and then constitute the basis of a multi-level programming or, respectively, a multi-level database framework.

Chapter 6

Querying Multi-Level Models

Contents

In this chapter we show how to query multi-level models. We introduce, in Section 6.1, operators for querying m-objects and sets of m-objects. These operators comprise a query algebra that is closed on sets of m-objects. In addition to their formal definition we give extensive examples of their use. In Section 6.2 we overload these operators for their application on m-relationships and sets of m-relationships. Complementing our object-centered query approach, we also introduce an operator for navigation along m-relationships. In Section 6.3 we discuss two basic assumptions of the m-object/m-relationship approach, namely top-down modeling and the closed world assumption. We give examples of how top-down multi-level modeling may be embedded in an organizational context and provide additional query operators that support multi-level modeling under these basic assumptions. Section 6.4 summarizes the chapter.

6.1 Querying M-Objects

In this section we introduce syntax and semantics of operations for querying m-objects and hierarchies of m-objects. First we show how to refer to m-objects using their namespace and name, and how to refer to attributes at different levels of an m-object. We then introduce operators that allow to refer to a specific abstraction (ancestor) of an m-object at a given level using *upward navigation* and to the set of descendant m-objects at a given level, i.e., its *class extension* at this level. Next, we introduce a *selection* operator that allows to reduce a set of m-objects to those m-objects whose attribute values fulfil a given selection criteria. These operators are specific to the m-object/m-relationship approach and are complemented by the usual set operators *union*, *intersection* and *minus*.

Some of the operators discussed in this section have already been defined in our previous work [86] for their application on single m-objects and are now overloaded for their application on sets of m-objects. Our operators form an *algebra* that is closed on sets of m-objects. The section is complemented by showing how to explicitly refer to the different meta*-class facets of an m-object, this, however, is not part of the query approach.

The presented query approach is *object-preserving* rather than object-generating (see [102] for an in-depth discussion of this distinction). Result sets of operations always consist of existing m-objects organized in existing concretization hierarchies, operations do not generate new m-objects or new hierarchies and they do not manipulate hierarchies or the inner structure of m-objects.

Referring to M-Objects Internally, each m-object is identified by its OID. Externally, each m-object is identified by its name together with its namespace. To separate name and namespace we use a hash sign (#). The namespace of an m-object may be omitted whenever it follows unambiguously from the context of an expression. To refer to the root m-object of a concretization hierarchy we simply write the namespace of the hierarchy.

Example 6.1 (Referring to M-Objects)**.** To refer to m-object *Car* in our sample problem (see Example 1.1) we write *Product#Car* or only *Car* if it is clear from the context that we are talking about concretization hierarchy *Product*. To refer to the root m-object of hierarchy *Product* we simply write *Product*.

Referring to Attributes Attribute names are unique within m-object hierarchies (the name of the hierarchy serves as namespace) and thus unique within m-objects. Thus an *attribute* can be referred to by its namespace and its name, separated by a hash symbol. The *value of an attribute* at a specific m-object is referred to by a reference to the m-object (external identifier or variable name) and the attribute's name, separated by a dot. Note that there is no difference between referring to attributes at top-levels of m-objects and referring to attributes at non-top-levels.

Example 6.2 (Referring to Attributes)**.** In order to refer to attribute *taxRate*, which is introduced at m-object *Car* in hierarchy *Product*, one writes *Product#taxRate*. If the namespace can be inferred from the context one may omit the namespace and simply write *taxRate*. The value of attribute *taxRate* at m-object *Car*, which is *20*, is referred to by *Car.taxRate*. One can likewise refer to non-top attributes, e.g., *Car.mileage* refers to the shared value of attribute *mileage* at level *physicalEntity* of m-object *Car* (which is not set in our example).

When applied to sets of m-objects, attribute value operator '.' is applied on each element of the set and returns the union of the results as result.

Example 6.3 (Attribute Values of Sets of M-Objects)**.** To refer to the set of values of attribute *taxRate* of m-objects *Car* and *Book* one writes

{*Car, Book*}.*listPrice*

and gets {*15, 20*} as result.

Upward Navigation Upward navigation operations provide direct and *stable access* from an m-object to its abstraction (i.e., ancestor m-object) at a given higher abstraction level. The operator is overloaded for its application on sets of m-objects. By *stable access* we refer to a navigation path that needs not to be changed with regard to additional intermediate levels in certain sub-hierarchies (such as level *brand* in the sub-hierarchy of *Car*).

Definition 6.1 (Upward Navigation)**.** *The ancestor m-object of m-object $o \in O$ at level $l \in L$, denoted as $o[l]$, is defined by*

$$o[l] \overset{\text{def}}{=} o' \in O : (o, o') \in H^* \wedge \hat{l}_{o'} = l.$$

The set of ancestor m-objects of m-objects $O' \subseteq O$ at level l, denoted as $O'[l]$, is defined by

$$O'[l] \overset{\text{def}}{=} \bigcup_{o \in O'}(o[l]).$$

Example 6.4 (Upward Navigation)**.** To navigate from m-object *myPorsche911CarreraS* to its ancestor at level *category* one writes

myPorsche911CarreraS[category]

and gets *Car* as result. To navigate from m-objects *myPorsche911CarreraS*, and *HarryPotter4* to their abstractions at level *category* one writes

{*myPorsche911CarreraS, HarryPotter4*}*[category]*

and gets {*Car, Book*} as result.

Upward navigation makes attribute values of higher-level m-objects also available at their descendant m-objects. From the viewpoint of some o, attributes at higher level m-objects are "global" attributes (e.g., *taxRate* is defined once per *product category*). As a shorthand notation, one may refer to attribute values of ancestor m-objects without explicit upward navigation, which is possible because every m-object has at most one ancestor at a given level and since attribute names are unique within hierarchies.

Example 6.5 (Attribute Values at Ancestor M-Objects)**.** To refer to the value of attribute *taxRate* applicable for *myPorsche911CarreraS*, which is provided by its ancestor at level *category*, one writes *my-*

Porsche911CarreraS[category].taxRate. As a shorthand alternative one can also write *myPorsche911CarreraS.taxRate*.

Class Extension An m-object represents, for each level of direct or indirect descendants, the class of descendant m-objects at that level. The notion of class refers first to the common structure (often referred to as *type* or *intension* of a class) and second to the set of descendant m-objects at a specific level (often referred to as *extension* or *members* of a class). To refer to the set of m-objects at level l beneath m-object o, we write $o\langle l\rangle$. Note, that this class extension operator is overloaded for its application to sets of m-objects. Also note that there is a special kind of singleton classes, namely those given by an m-object and its top-level, $o\langle\hat{l}_o\rangle$.

Definition 6.2 (Class Extension). *The class extension of m-object $o \in O$ at level $l \in L_o$, denoted as $o\langle l\rangle$, is defined by*

$o\langle l\rangle \overset{\text{def}}{=} \{o' \mid (o', o) \in H^* \wedge \hat{l}_{o'} = l\}$.

The set of m-objects that are in the class extension at level l of an m-object in a set of m-objects $O' \subseteq O$, denoted as $O'\langle l\rangle$, is defined by

$O'\langle l\rangle \overset{\text{def}}{=} \bigcup_{o \in O'} o\langle l\rangle$.

Example 6.6 (Class Extension). Figure 6.1 depicts various class extensions of different m-objects at different levels. Consider m-object *Product* and its descendants at level *model*, which are m-objects *HarryPotter4*, *Porsche911CarreraS*, and *Porsche911GT3*. We call this set of m-objects the extension of *Product* at level *model*, denoted as *Product⟨model⟩*. Note that there is a singleton class of m-object *Product* at level *catalog* that only consists of m-object *Product*. These and other class extensions of m-object *Product* are:

Product⟨catalog⟩ = {Product}
Product⟨category⟩ = {Book, Car}
Product⟨model⟩ = {HarryPotter4, Porsche911CarreraS, Porsche911GT3}
Product⟨physicalEntity⟩ = {myCopyOfHP4, myPorsche911CarreraS}

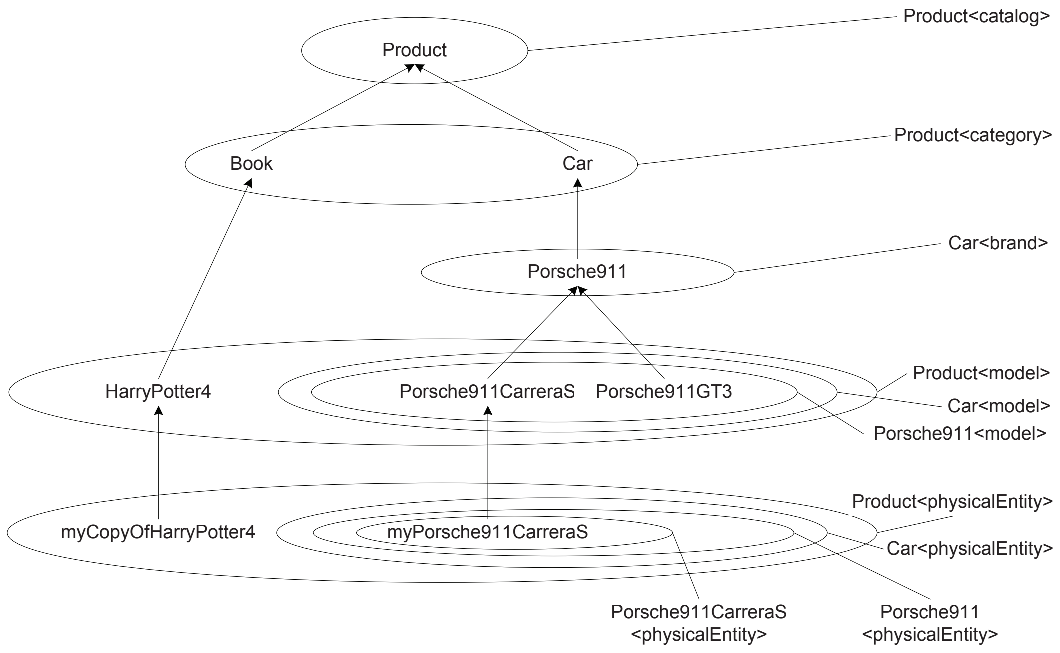


Figure 6.1: Class extensions of different m-objects at different levels

Not only root m-objects have class extensions but also their concretizations. Class extensions of lower level m-objects are subsets of respective class extensions of their ancestor m-objects.

Example 6.7 (Class Extension - Subset Hierarchy)**.** Consider again sets and subsets as depicted in Figure 6.1. M-Object *Car*, which is a concretization of m-object *Product*, has the following class extensions:

$Car\langle category\rangle = \{Car\}$ (not depicted)
$Car\langle brand\rangle = \{Porsche911\}$
$Car\langle model\rangle = \{Porsche911CarreraS, Porsche911GT3\}$
$Car\langle physicalEntity\rangle = \{myPorsche911CarreraS\}$

Note that these class extensions of m-object *Car* are subsets of respective class extensions of m-object *Product* at the different levels of *Car*:

$Car\langle category\rangle \subseteq Product\langle category\rangle$
$Car\langle model\rangle \subseteq Product\langle model\rangle$
$Car\langle physicalEntity\rangle \subseteq Product\langle physicalEntity\rangle$

Note that *Car*$\langle$*brand*$\rangle$ has no superset at m-object *Product*, since level *brand* was introduced at m-object *Car* and is not available at m-object *Product*.

Furthermore, class extensions of m-object *Porsche911* are subsets of respective class extensions of m-object *Car*, and class extensions of m-object *Porsche911CarreraS* are subsets of respective class extensions of m-object *Porsche911*.

The class extension operator is overloaded for its application on sets of m-objects. This is necessary since operators presented in this section constitute an algebra which is closed on sets of m-objects. Note that the class extension operator can be applied on sets of m-objects at different levels of abstraction.

Example 6.8 (Class Extension Operations on Sets of M-Objects)**.** To refer to the set of m-objects at level *physicalEntity* beneath m-object *Book* or beneath m-object *Porsche911CarreraS*, one writes

{*Book*, *Porsche911CarreraS*}$\langle$*physicalEntity*$\rangle$

which evaluates to {*myCopyOfHarryPotter4*, *myPorsche911CarreraS*}. This is synonym to the union of class extensions of m-objects *Book* and *Porsche911CarreraS* at level *physicalEntity*.

Selection In addition to referring to descendant m-objects at a given level of a given m-object one often also wants to retrieve only those m-objects from a set of m-objects that fulfil a given selection criterion.

Selection criteria are formulated as predicates. A *predicate* p is a boolean expression over values of *top-level* attributes of an m-object o, and over values of top-level attributes of ancestor m-objects of this m-object o. A predicate p applied to an m-object o, denoted as $o.eval(p)$, either evaluates to *true*, evaluates to *false*, or to *unknown*. A predicate p evaluates to *unknown* if neither m-object o nor an ancestor of o specify attributes or attribute values which allow to evaluate the predicate.

Example 6.9 (Simple Predicate)**.** One could be interested in products that have a *taxRate* of *15*. The respective predicate is (*taxRate* = *15*). Applied to m-objects, this predicate either evaluates to *true*, *false*, or *unknown*, as illustrated in the following examples:

Product.eval(taxRate=15)	*unknown*
Book.eval(taxRate=15)	*true*
myCopyOfHarryPotter4.eval(taxRate=15)	*true*
Porsche911CarreraS.eval(taxRate=15)	*false*

Note that predicate *(taxRate=15)*, when applied on *Product*, evaluates to *unknown* since *Product* has not specified a shared value for *taxRate*.

Basic expressions can be combined to composite expressions using logical connectors such as *conjunction* ∧, *disjunction* ∨, and *negation* ¬, with the usual semantics. When evaluating composite expressions, non-applicable sub-expressions make the composite expression not applicable.

Example 6.10 (Composite Predicate)**.** One could also be interested in products that have a *taxRate* of *15* and a *listPrice* above *10*. The respective composite predicate is *(taxRate=15 ∧ listPrice>10)*, see the following examples:

Product.eval(taxRate=15 ∧ listPrice>10)	*unknown*
Book.eval(taxRate=15 ∧ listPrice>10)	*unknown*
myCopyOfHarryPotter4.eval(taxRate=15 ∧ listPrice>10)	*true*
Porsche911CarreraS.eval(taxRate=15 ∧ listPrice>10)	*false*

Predicates can be predefined at m-objects and associated with a level. For this purpose we treat predicates as a specific kind of attributes that take boolean expression as values. Then these predicates can be introduced at the higher level m-object and be instantiated at lower level m-objects.

Example 6.11 (Predefined Predicates)**.** Consider our sample product hierarchy. We want to define that *Cars* are expensive if they have a *listPrice* of above *100000*, and that *Books* are expensive if they have a *listPrice* of above *10*. For this purpose we introduce a predicate-valued attribute *expensive* at level *category* of m-object *Product*, and instantiate it at product

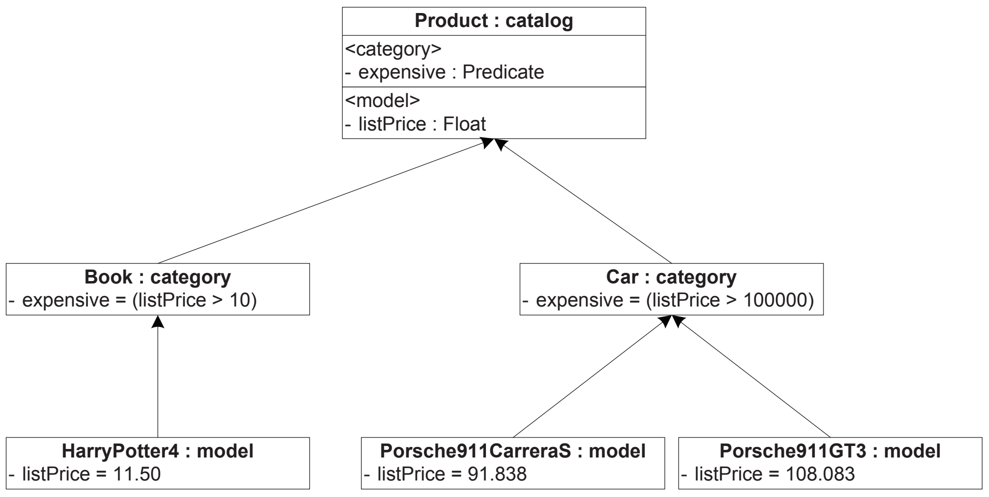


Figure 6.2: Simplified product hierarchy with predefined predicate *expensive*

categories *Book* and *Car* as follows: *Book.expensive = (listPrice>10)* and *Car.expensive = (listPrice>100000)* (see Figure 6.2). Then, evaluation of predicate *expensive* for a given m-object means to first determine the respective definition of predicate *expensive* and then to evaluate it, as exemplified in the following:

Book.expensive()	*(listPrice>10)*	*unknown*
HarryPotter4.expensive()	*(listPrice>10)*	*true*
Porsche911CarreraS.expensive()	*(listPrice>100000)*	*false*
Porsche911GT3.expensive()	*(listPrice>100000)*	*true*

A selection operation is then used to retrieve all m-objects from a set of m-objects for which a given predicate evaluates to true.

Definition 6.3 (Select σ)**.** *Given are a boolean predicate p and a set of m-objects $O' \subseteq O$, then the result set of* selection-*operation $\sigma_p O'$ is defined as:*

$$\sigma_p O' \overset{\text{def}}{=} \{o \in O' \mid p(o)\}$$

Example 6.12 (Selection Operation)**.** To retrieve all *product models* having a *listPrice* of above *100000* we write $\sigma_{(listPrice>100000)} Product\langle model\rangle$. This expression is evaluated in two steps. First, class extension operation *Product*$\langle$*model*$\rangle$ results in {*Porsche911GT3, HarryPotter4, Porsche911CarreraS*}. Second, selection operation $\sigma_{(listPrice>100000)}$\{*Porsche911GT3, HarryPotter4, Porsche911CarreraS*\} applies predicate *(listPrice>100000)* on each m-object in the intermediate result and evaluates to {*Porsche911GT3*}, since *Porsche911GT3* is the only m-object having a *listPrice* of above *100000*.

To retrieve all product models, which are expensive in context of their product category, we reuse predicate *expensive* as introduced in Ex. 6.11 and write $\sigma_{expensive()}$*Product*$\langle$*model*$\rangle$. This operation results in {*HarryPotter4, Porsche911GT3*}.

Complex Query Expressions Simple query expressions, such as exemplified above, can be combined to complex query expressions by combining a sequence of simple query expressions with the usual set operations *union* $\cup$, *intersection* $\cap$, and *minus* $\setminus$. In the following we will present some common patterns. Table 6.1 gives an overview of the available operators at different precedence levels. Operations with higher precedence are executed before operations with lower precedence. Operations that have the same precedence are executed from left to right. Parentheses '()' are used to group operations in order to change precedence.

Selection operations are used to retrieve all m-objects that fulfil a given predicate. Predicates may only refer to attribute values of the retrieved m-objects or of their ancestors. It is not possible to query m-objects with predicates on their descending m-objects. This is because there are two different ways to pose such queries: First, one could ask for m-objects that have at least one descendant m-object satisfying a given predicate (existential quantification). Second, one could ask for m-objects that *only* have m-objects at a given level satisfying a given predicate (universal quantification).

Multi-level queries with *existential quantification over descendant m-objects* are expressed by a selection operation on lower level m-objects followed by upward navigation. More precisely, to retrieve all m-objects from an extension of an m-object o at level l that have at least one descendant m-object at level l' satisfying a predicate p, we first retrieve all descendant m-objects at level l' using class extension operation, followed by a selection operation, and an upward navigation operation, i.e., $(\sigma_p o\langle l'\rangle)[l]$.

Example 6.13 (Existential Quantification)**.** To retrieve all product categories with at least one product model having a *listPrice* of above *100000* one writes

$(\sigma_{(\mathit{listPrice}>100000)} \mathit{Product}\langle \mathit{model}\rangle)[\mathit{category}]$

and gets $\{\mathit{Car}\}$ as result.

Universal quantification is expressed, as usual, by negated existential quantification. Consider that one wants to retrieve from a class extension of m-object o at level l those m-objects for which all descendants at level l', with $l' \prec l$, satisfy a given predicate p. This is expressed as a selection operation on class extension of o at level l' with negated predicate p followed by upward navigation to level l', the result of this operation is then subtracted from the class extension of o at level l, i.e., $o\langle l\rangle \setminus (\sigma_{\neg p} o\langle l'\rangle)[l]$.

Example 6.14 (Universal Quantification)**.** To retrieve product categories that only contain product models with a listPrice above 100000, i.e., all product categories not having any product model with a listPrice of 100000 or below, one writes

$\mathit{Product}\langle \mathit{category}\rangle \setminus (\sigma_{\neg(\mathit{listPrice}>100000)} \mathit{Product}\langle \mathit{model}\rangle)[\mathit{category}]$

and gets $\{\}$ as result.

Metaclass Extension An m-object does not only define a class for each of its levels, but also, implicitly, multiple metaclasses. In this thesis we limit the discussion of metaclasses on their extensional role, i.e. a meta-class considered as a set of classes, and do not deal with the structural role of meta-classes (meta-types).

Definition 6.4 (Metaclasses and their Extensions)**.**

1. *For each m-object o and each pair of levels $(l_0, l_1) \in P_o^+$ meta-class $o\langle l_1\langle l_0\rangle\rangle$ is defined as the set of classes containing for each m-object o' that is a descendant of o at level l_1 class $o'\langle l_0\rangle$, i.e.* $o\langle l_1\langle l_0\rangle\rangle \overset{\text{def}}{=} \{o'\langle l_0\rangle \mid o' \in o\langle l_1\rangle\}$.

2. *For $o \in O$ and $l_0 \in L_o$ meta-class $o\langle\langle l_0\rangle\rangle$ is the set of all classes containing m-objects at level l_0 and that are descendants of o, i.e.,* $o\langle\ \langle l_0\rangle\rangle \overset{\text{def}}{=} \{o'\langle l_0\rangle \mid (o', o) \in H^*\}$

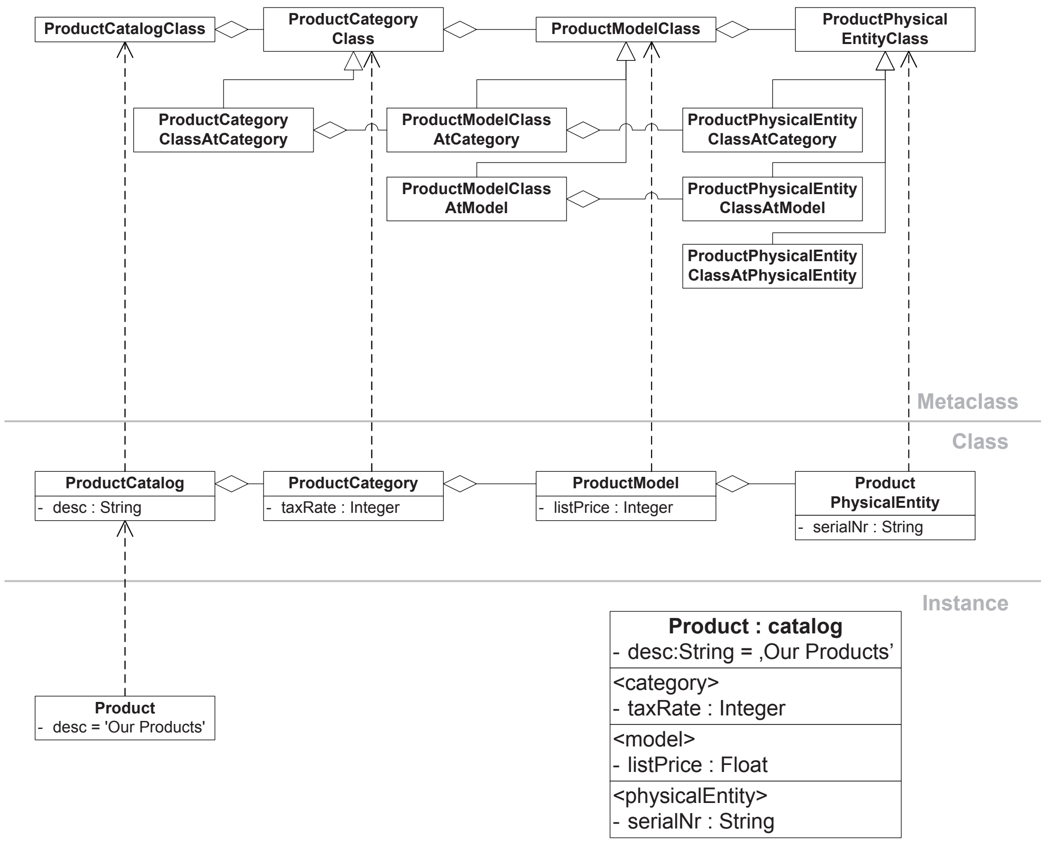


Figure 6.3: M-object *Product* and its object-, class- and metaclass-facets

Example 6.15 (Metaclasses and their Extension)**.** Figure 6.1 shows m-object *Product* and explicitly represents object-, class- and metaclass-facets of *Product* using our simplified graphical notation as introduced in Chapter 2. The depicted classes and metaclasses can be queried using above defined class extension and metaclass extension operations. For example, *Product⟨category⟨model⟩⟩* refers to the extension of metaclass *ProductModelClassAtCategory* in Figure 6.1.

In the following, the notion of metaclass extension is illustrated by the different metaclasses of Product and their extensions. (Singleton-meta-classes like Product⟨catalog⟨category⟩⟩ are omitted):

Product⟨category⟨model⟩⟩ =
{Car⟨model⟩, Book⟨model⟩}

Product⟨category⟨physical entity⟩⟩ =
{Car⟨physical entity⟩, Book⟨physical entity⟩}

Product⟨model⟨physical entity⟩⟩ =
{HarryPotter4⟨physical entity⟩, Porsche911CarreraS⟨physical entity⟩, Porsche911GT3⟨physical entity⟩ }

Product⟨⟨model⟩⟩ =
{Product⟨model⟩, Car⟨model⟩, Book⟨model⟩, Porsche911⟨model⟩}

Product⟨⟨physical entity⟩⟩ =
{Product⟨physical entity⟩, Car⟨physical entity⟩, Book⟨physical entity⟩, HarryPotter4⟨physical entity⟩, Porsche911⟨physical entity⟩, Porsche911CarreraS⟨physical entity⟩, Porsche911GT3⟨physical entity⟩}

Metaclasses can be further classified to meta^2-classes, which, in turn, can be further classified to meta^3-classes, and so forth. M-objects implicitly comprise such meta^{2+}-classes. In the following we provide a query operator that allows to refer to their extensions.

Definition 6.5 (Meta^{2+}-Classes and their Extensions)**.**

1. *For each m-object o and (n+1)-tuple of levels* $(l_0, l_1, \ldots, l_n)$*, where n>1 and for i=1..n :* $(l_{i-1}, l_i) \in P_o^+$*, meta*^{n}*-class* $o\langle l_n \langle l_{n-1} \ldots \langle l_0 \rangle \ldots \rangle\rangle$ *is defined as the following set of meta*$^{n-1}$*-classes:*
   $\{o' \langle^{n-1} l_{n-1} \ldots \langle^0 l_0 \rangle^0 \ldots \rangle^{n-1} \mid o' \in o\langle l_n \rangle\}$*.*

2. *For each m-object o and each level* $l_0 \in L_o$ *meta*^{n}*-class* $o\langle^n \langle^{n-1} \ldots \langle^0 l_0 \rangle \ldots \rangle\rangle$ *(with* $n > 1$*) is the following set of meta*$^{n-1}$*-classes*

$\{o' \langle^{n-1} l_{n-1} \ldots \langle^0 l_0 \rangle^0 \ldots \rangle^{n-1} \mid (o', o) \in H^* \wedge (\forall i \in \mathit{1..n\text{-}1} : (l_{i-1}, l_i) \in P^+_{o'})\}$

Example 6.16 (Meta^{2+}-Classes and their Extensions)**.** The following meta^2-classes are based on m-object *Product*, as given by Figure 4.1 (Singleton-meta^n-classes like Product⟨catalog⟨category⟨model⟩⟩⟩ are omitted):

Product⟨category⟨model⟨physical entity⟩⟩⟩ =
{*Car⟨model⟨physical entity⟩⟩*, *Book⟨model⟨physical entity⟩⟩*}

Product⟨⟨⟨physical entity⟩⟩⟩ =
{*Product⟨category⟨physical entity⟩⟩*, *Product⟨model⟨physical entity⟩⟩*, *Car⟨brand⟨physical entity⟩⟩*, *Car⟨model⟨physical entity⟩⟩*, *Book⟨model⟨physical entity⟩⟩*}

6.2 Querying M-Relationships

In this section we overload previously introduced query operators for querying m-relationships and sets of m-relationships. Additionally we introduce an operator for *navigation along m-relationships* in both directions. In order to better exemplify these operators, we slightly adapt our running example.

Example 6.17 (Adapted Example)**.** We introduce an additional industrial sector, namely *publisher*, and assert that *book*s are produced by *publisher*s. Companies at level *enterprise* now have two attributes, *country* and *revenue*. We assert that enterprise *PorscheLtd* is a German company and has a revenue of *10,000.0000* (note that this is an arbitrary value), and introduce one enterprise of industrial sector *publisher*, namely *Bloomsbury*. We state that it is *UK*-based and has a revenue of *11,000.000*. We assert that *HarryPotter4* is produced by *Bloomsbury* and that *Porsche911CarreraS* and *Porsche911GT3* are produced by *PorscheLtd*.

Because they are not relevant for discussing the basics of querying m-relationships, we ignore level *brand* of sub-hierarchy *Car*, attributes *maxSpeed* and *mileage*, as well as m-objects *myCopyOfHarryPotter4* and *myPorsche911CarreraS* at level *physicalEntity* and m-object *PorscheZuffenhausen* at level *factory* (see Figure 6.4).

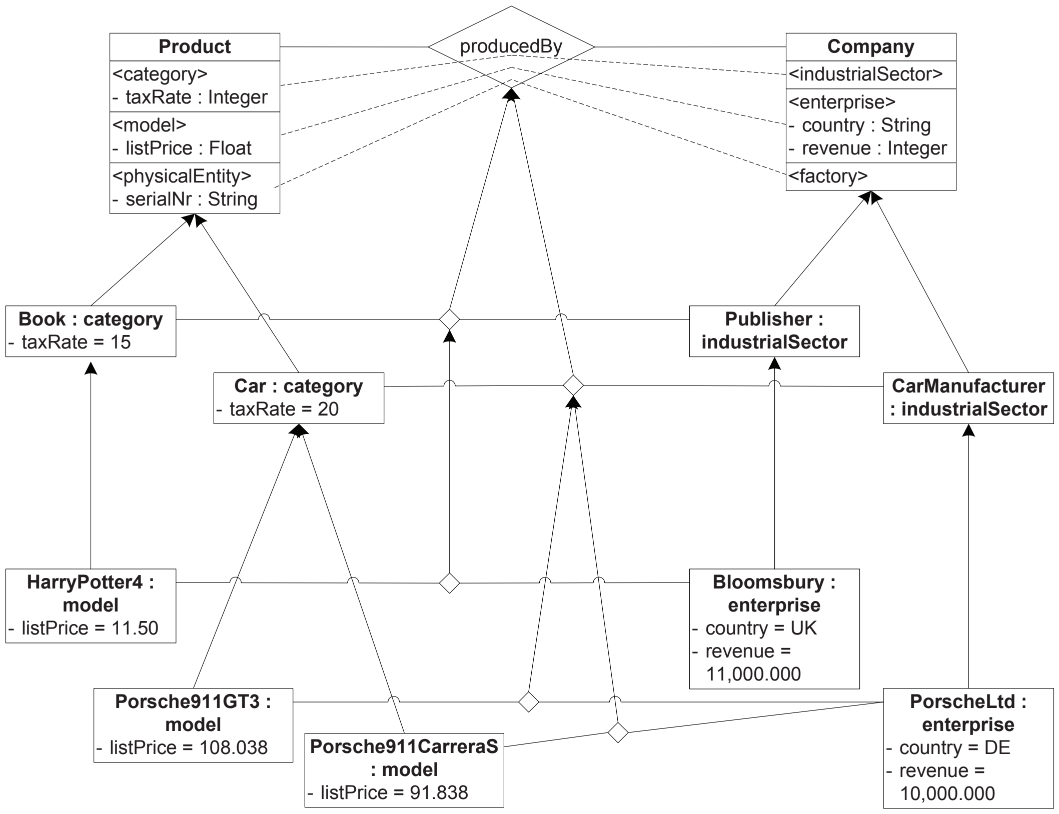


Figure 6.4: Sample multi-level model adapted for exemplifying m-relationship queries

Referring to M-Relationships Each m-relationship is identified by its namespace (that is the name of the m-relationship hierarchy), its optional name, and identifiers of its source and target m-objects. Identifiers of source m-object, namespace and, optionally, name of the m-relationship, as well as identifier of target m-object are separated by dashes (see Example 6.18). To refer to the root m-relationship of a hierarchy of m-relationships one simply writes the namespace of the hierarchy.

Example 6.18 (Referring to M-Relationships)**.** In order to refer to the root *producedBy* m-relationship (between *Product* and *Company*) one simply writes *producedBy*. In order to refer to the *producedBy*-relationship between m-objects *Car* and *CarManufacturer*, which is a non-root m-relationship, one writes *Car-producedBy-CarManufacturer*.

Referring to Source and Target M-Objects To refer to the source or target m-object of an m-relationship we write a dot followed by a role name (in the simple case either *.source* or *.target*). Applied to a set of m-relationships one can retrieve either the set of source m-objects or the set of target m-objects.

Definition 6.6 (Source and Target M-Objects)**.** *Given is an m-relationship* $r \in R$ *and a role name, either* source *or* target*, then*

$r.source \overset{\text{def}}{=} s_r$

$r.target \overset{\text{def}}{=} t_r.$

Given is a set of m-relationships $R' \in R$ *and a role name, either* source *or* target*, then*

$R'.source \overset{\text{def}}{=} \bigcup_{r \in R'}(s_r)$

$R'.target \overset{\text{def}}{=} \bigcup_{r \in R'}(t_r).$

Instead of default role names (*source*, *target*) one could also assign custom role names to source and targets of m-relationships. In this case one could refer to source and target m-objects using these custom role names. Alternatively, if unambiguous, one can also use names of connected m-object hierarchies instead of role names.

Example 6.19 (Referring to Source and Target M-Objects)**.** Consider m-relationship *Car-producedBy-CarManufacturer*. In order to refer to the *taxRate* of its source m-object one writes

Car-producedBy-CarManufacturer.source.taxRate

and gets *20* as result. Alternatively, since the given m-relationship connects two distinct m-object hierarchies, one could also write *Car-producedBy-CarManufacturer.Product.taxRate*. Consider that we gave custom role name *producer* to the target-role of m-relationship *producedBy*. Then we could write *Car-producedBy-CarManufacturer.producer* to refer to m-object *Car-Manufacturer* which plays the *producer*-role in this m-relationship.

Upward Navigation Upward navigation operations provide direct and stable access from an m-relationship to its abstraction (i.e., ancestor m-relationship) at a given higher connection-level. The operator is again overloaded for its application on sets of m-relationships.

Definition 6.7 (Upward Navigation in M-Relationship Hierarchies)**.** *The ancestor m-relationship of m-relationship* $r \in R$ *at connection-level* $(l, \bar{l})$, *denoted as* $r[l, \bar{l}]$, *is defined by*

$r[l, \bar{l}] \overset{\text{def}}{=} r' \in R : r \preceq r' \wedge \hat{c}_{r'} = (l, \bar{l}).$

The set of ancestor m-relationships of m-relationships $R' \subseteq R$ *at connection-level* $(l, \bar{l})$, *denoted as* $R'[l, \bar{l}]$, *is defined by*

$R'[l, \bar{l}] \overset{\text{def}}{=} \bigcup_{r \in R'}(r[l, \bar{l}]).$

If the given connection-level $(l, \bar{l})$ is no connection-level above a given m-relationship, then upward navigation results in ε.

Example 6.20 (Upward Navigation for M-Relationships)**.** To navigate from m-relationship *myPorsche911CarreraS-producedBy-PorscheZuffenhausen* to its ancestor m-relationship at connection-level *(category, industrialSector)* one writes

myPorsche911CarreraS-producedBy-PorscheZuffenhausen[category, industrialSector]

and gets m-relationship *Car-producedBy-CarManufacturer* as result.

M-Relationship Extensions Analogous to class extensions of m-objects, also m-relationships have extensions at their various connection-levels. The extension of an m-relationship at a given connection-level is the set of descendant m-relationships at this connection-level. The m-relationship extension operator is overloaded for its application to sets of m-relationships.

Definition 6.8 (M-Relationship Extension)**.** *The extension of m-relationship $r \in R$ at connection-level $(l, \bar{l})$, with $(l, \bar{l}) \in C_r$, denoted as $r\langle l, \bar{l}\rangle$, is defined by*

$$r\langle l, \bar{l}\rangle \overset{\text{def}}{=} \{r' \in R \mid r' \preceq r \wedge \hat{c}_{r'} = (l, \bar{l})\}.$$

The union of extensions of a set of m-relationships $R' \subseteq R$ at connection-level $(l, \bar{l})$, with $(l, \bar{l}) \in \bigcap_{r \in R'}(C_r)$, denoted as $R'\langle l, \bar{l}\rangle$, is defined by

$$R'\langle l, \bar{l}\rangle \overset{\text{def}}{=} \bigcup_{r \in R'}(r\langle l, \bar{l}\rangle).$$

Example 6.21 (Extensions of M-Relationships)**.** Considering m-relationships depicted in Figure 6.4, the extension at connection-level *(category, industrialSector)* of m-relationship *producedBy* between *Product* and *Company* denoted as

producedBy⟨*category,industrialSector*⟩

is {*Book-producedBy-Publisher, Car-producedBy-CarManufacturer*}. Its extension at connection-level *model-enterprise*, denoted as

producedBy⟨*model,enterprise*⟩

is {*HarryPotter4-producedBy-Bloomsbury, Porsche911CarreraS-producedBy-PorscheLtd*}.

The extension of m-relationship *Car-producedBy-CarManufacturer*, which is a concretization of *producedBy*, at connection-level *(model,enterprise)*, denoted as *Car-producedBy-CarManufacturer*⟨*model,enterprise*⟩ is {*Porsche911CarreraS-producedBy-PorscheLtd*} and is a subset of *producedBy*⟨*model,enterprise*⟩.

Navigation along M-Relationships M-relationship extensions, as introduced above, are the basis of relationship-centered querying. Now, from an object-centered perspective, one wants to be able to navigate m-relationships from source m-object to target m-object (or vice versa from target to source). Navigation along multi-level relationships is similar to dereferencing references in SQL-3. But, navigation in the context of concretization hierarchies of m-relationships is multi-faceted and expresses that all concretizations of a given m-relationship are to be traversed, leading to an m-object at some specified level.

Based on the definition of m-relationship extensions it is straightforward to define an operator for navigation along m-relationships. The set of m-objects at target level l that can be reached from m-object o by navigating along m-relationship r —or along one of its concretizations— is the set of m-objects which, together with o, are member of the extension of r at the connection-level given by the top-level of o and target-level l. M-relationships can likewise be navigated from target m-objects to source m-objects.

Definition 6.9 (Navigation Along M-Relationships)**.** *The set of target m-objects at level l that is reached from source m-object $o \in O$ by traversing m-relationship $r \in R$ or a concretization of r, denoted as $o{\rightarrow}r(l)$, is defined by*

$$o{\rightarrow}r(l) \overset{\text{def}}{=} \{\bar{o} \in O \mid \exists r' \in R : r' \in r\langle \hat{l}_o, l\rangle \wedge s_{r'} = o \wedge t_{r'} = \bar{o}\}$$

and in the opposite direction:

$$o{\rightarrow}r^{-}(l) \overset{\text{def}}{=} \{\bar{o} \in O \mid \exists r' \in R : r' \in r\langle l, \hat{l}_o\rangle \wedge s_{r'} = \bar{o} \wedge t_{r'} = o\}$$

In contrast to our original definitions in [86] we now write $o{\rightarrow}r(l)$ instead of $o{\rightarrow}r\langle l\rangle$ to avoid confusion with class extension operations $o\langle l\rangle$. Also note that we use a superscripted dash $(o \rightarrow r^{-}(l))$ to denote inverse navigation along m-relationships. This seems more intuitive than the left-arrow-notation $(o \quad r(l))$ used in [86].

Example 6.22 (Navigation along M-Relationships)**.** In order to navigate from m-object *Porsche911CarreraS* to its producer at level *enterprise* one writes

Porsche911CarreraS $\rightarrow$*producedBy(enterprise)*

and gets $\{$*PorscheLtd*$\}$ as result. Vice versa, to navigate from m-object *PorscheLtd* to the product models that it produces, one writes

$PorscheLtd \rightarrow producedBy^{-}(model)$

and gets {*Porsche911CarreraS, Porsche911GT3*} as result.

Like other operators, operators for navigation along m-relationships are overloaded for their application on sets of m-objects.

Definition 6.10 (Navigation Along M-Relationships (Set))**.** *The set of target m-objects at level $l \in L$ that is reached from a set of source m-objects $O' \subseteq O$ by traversing m-relationship $r \in R$ or a concretization of r, denoted as $O' \rightarrow r(l)$, is defined by*

$$O' \rightarrow r(l) \overset{\text{def}}{=} \bigcup_{o \in O'} (o \rightarrow r(l))$$

and, analogously, in the opposite direction from target m-objects to source m-objects:

$$O' \rightarrow r^{-}(l) \overset{\text{def}}{=} \bigcup_{o \in O'} (o \rightarrow r^{-}(l))$$

Example 6.23 (Navigation along M-Relationships (Set))**.** In order to navigate from the extension of m-object *Product* at level *model* to its producers at level *enterprise* one writes

$Product\langle model \rangle \rightarrow producedBy(enterprise)$

and gets {*Bloomsbury, PorscheLtd*} as result.

Note that one often wants to navigate not only to directly related m-objects but also to their ancestors at a higher abstraction level. In such a case one has to first follow an explicit m-relationship, and then navigate to an ancestor m-object using upward navigation (see Example 6.24). An alternative to this kind of m-relationship navigation to higher levels is provided in Section 6.3.

Example 6.24 (Navigation to Abstractions of Target M-Objects)**.** Car physical entity *myPorsche911CarreraS* is not linked directly to the enterprise by which it was produced but only to its producing factory. Thus one cannot directly navigate to its producing enterprise. In order to navigate from *myPorsche911CarreraS* to its producer at level *enterprise* one writes *myPorsche911CarreraS* $\rightarrow$*producedBy(factory)[enterprise]*

and gets {PorscheLtd} as result.

M-relationship navigation can be combined with m-object query expressions. In the following we give some illustrative examples. Table 6.1 gives an overview of available operators at different precedence levels.

1	$\langle\rangle$, []	Upward navigation and class extension
2	σ	Selection
3	$\rightarrow$	Navigation along m-relationships
4	$\cup$, $\cap$, $\setminus$	Usual set operations

Table 6.1: Query operators and their precedence (from high to low)

Example 6.25 (Algebra Expressions)**.** Consider products and companies in our running example (Ex. 1.1) and the following query examples:

1. All companies at enterprise level that produce some car models with a listPrice of more than 100000:
   $\sigma_{listPrice>100000}Car\langle model\rangle \rightarrow producedBy(enterprise)$

2. Product models that are produced by Austrian enterprises:
   $\sigma_{country=A}Company\langle enterprise\rangle \rightarrow producedBy^{-}(model)$

3. Product models that are produced by Austrian enterprises and have a listPrice above 100000:
   $\sigma_{listPrice>100000}(\sigma_{country=A}Company\langle enterprise\rangle$
   $\rightarrow producedBy^{-}(model))$

Selection of M-Relationships In addition to referring to descendant m-relationships at a given connection-level one often also wants to retrieve only those m-relationships from a set of m-relationships that fulfil a given selection criterion. This is analogous to selection operations on sets of m-objects.

Selection criteria for m-relationships are formulated as predicates. An *m-relationship predicate* is a boolean expression over attributes and predicates of the source m-object (and its ancestors), and of the target m-object (and its ancestors) of an m-relationship, and, if provided, also over attributes of the m-relationship itself (note that m-relationship attributes are not discussed in this chapter). A predicate is either *true*, *false* or *not applicable* for an m-relationship and formulated similar to m-object predicates. To refer to attributes or predicates of the source (or target) m-object we write 'source.' (or 'target.', resp.) followed by the name of the attribute or predicate.

Example 6.26 (Simple Predicate on M-Relationships)**.** One could be interested in *producedBy*-m-relationships that have a source with a *listPrice* of above *100000*. The respective predicate is (*source.listPrice* > *100000*). Applied to m-relationships, this predicate either returns *true*, *false*, or *not applicable*, as illustrated in the following examples:

producedBy.eval(source.taxRate=15)	*unknown*
Car-producedBy-CarManufacturer.eval(source.taxRate=15)	*false*
Book-producedBy-Publisher.eval(source.taxRate=15)	*true*
HarryPotter4-producedBy-Bloomsbury.eval(source.taxRate=15)	*true*

More interestingly, m-relationship predicates allow to compare attribute values of the source m-object with attribute values of the target m-object.

Example 6.27 (Relating Attributes in M-Relationship Predicates)**.** One could be interested in *producedBy*-m-relationships connecting an m-object with a *listPrice* that is in certain ratio to the revenue of its producing company. For example, to find out those producedBy-m-relationships where the listPrice of the product is more than one percent of the total revenue of the producer, the respective predicate is (*source.listPrice* > (*target.revenue*/100)).

A selection operation is then used to retrieve all m-relationships from a set of m-relationships for which the given predicate evaluates to true.

Definition 6.11 (Select σ)**.** *Given are boolean predicate p and a set of m-relationships $R' \subseteq R$, then the result set of* selection-*operation $\sigma_p R'$ is defined as follows:*
$\sigma_p R' \overset{\text{def}}{=} \{r \in R' \mid p(r)\}$

Example 6.28 (Select Operation)**.** In order to retrieve those *producedBy*-m-relationships at connection-level *(model,enterprise)* where the source m-object (or one of its ancestors) has defined a *listPrice* that is higher than one percent of the total revenue of the producing company one writes

$\sigma_{(source.listPrice > target.revenue/100)} producedBy\langle model, enterprise\rangle$

and gets {*Porsche911GT3-producedBy-PorscheLtd*} as result.

Complex Query Operations Simple query expressions on m-relationships, as exemplified above, to complex query expressions by combining a sequence of simple query expressions with the usual set operations *union* ∪, *intersection* ∩, and *minus* \. This is again analogous to query expressions on sets of m-objects.

Example 6.29 (Complex Query Operations on M-Relationships)**.** In order to retrieve all *producedBy*-m-relationships at connection-level *(model,enterprise)* but without concretizations of *Car-producedBy-CarManufacturer* one writes

producedBy⟨*model, enterprise*⟩ \ *Car-producedBy-CarManufacturer*⟨*model, enterprise*⟩

and gets {*HarryPotter4-producedBy-Bloomsbury*} as result.

6.3 Active Domain

In Chapter 5 we discussed how to create and manipulate multi-level models and implicitly made two basic assumptions: first, the *closed world assumption* and, second, the assumption that multi-level models are created *top-down*, from abstract to concrete. Furthermore, we assume that in an organizational context, m-objects and m-relationships at different levels are created and manipulated by different users having different user rights.

In this section we will first discuss these basic assumptions in detail and recount our sample problem in an organizational context with focus on these two assumptions. We will then analyze how top-down multi-level modeling can be effectively supported by additional query functionality. Especially concerning modelers at lower levels that need to know which m-objects they may connect with m-relationships. For this purpose we introduce the concepts of *active domain of m-relationships*.

The *active domain* at a connection-level of an m-relationship r is the set of pairs of source and target-m-objects, which may be connected by an m-relationship, concretizing r, without the need to introduce additional m-relationships above this connection level.

We also provide an object-centered perspective on active domains of m-relationships. From the viewpoint of a given m-object, the *active target domain* for a specific target-level of an m-relationship r is the set of m-objects to which an m-relationship that concretizes r can be established from the given m-object without the need to introduce additional higher-level m-relationships.

Top-Down Closed-World Assumption The m-object and m-relationship approach makes the *top-down closed-world assumption* which combines top-down modeling and closed world assumption. While this is often implicitly assumed in information systems engineering it is important to explicitly make these assumptions, since – with the advent of decentralized information systems like the Semantic Web – these assumptions are often rightfully contested. In this subsection we will discuss the motivations and implications of making these assumptions. We will first look at top-down modeling and then at the closed world assumption and at their combined use. Especially when modeling with m-relationships it is important to be aware of these assumptions since the expressiveness of m-relationships relies on them.

Note, that in the examples given in this section we embed our sample problem in a broader, organizational context that goes beyond the scope of this thesis, but seems appropriate to illustrate the big picture that guides our research. Along these lines, we assume that there are different users, that play different roles at different levels in an organization. We also assume some kind of user rights management that gives different permissions to different users and makes use of the multi-level structure of multi-level models.

The m-object/m-relationship approach is a *top-down* (in contrast to *bottom-up*) multi-level modeling approach. M-objects and m-relationships at the most abstract level are created first and are then concretized at lower levels of abstraction. M-objects and m-relationships at higher levels provide schemas for lower level m-objects and m-relationships. That is, they play a normative role with regard to lower levels. We also assume that m-objects and m-relationships at different abstraction levels are often provided by different modelers, where modelers at higher levels have more normative power than modelers at lower levels. This is analogous to the hierarchical structure of normative bodies in government and business. For example, lawmakers within a state may only work within the boundaries defined by lawmakers of their country which have to work within the boundaries defined by super-national organizations such as the European Union. Similarly, in a company, a department manager may only make decisions within the boundaries defined by the board of directors

This is in contrast to de-centralized environments like the Web, where a bottom-up approach would be more appropriate. There, one would first model concrete objects and relationships and then model their abstractions at higher levels. Adapting m-objects and m-relationships for bottom-up abstraction remains open for future work.

While bottom-up abstraction is important, we believe that in a subject-domain that is at least partially centralized, like when modeling organizations at multiple levels, a top-down approach is more appropriate. Furthermore, top-down modeling is very common and modelers are familiar with it, two examples are: (1) Large, best practice software solutions (as discussed in Section 1.1) come in a very generic form and can be customized for specific needs. (2) In object-oriented modeling/programming, one typically starts with the most generic classes and then specializes them and then instantiates them.

In addition to being a top-down approach, m-objects and m-relationships make the *closed world assumption (CWA)*. Making the closed world assumption, all *relevant* information about a subject domain – that is, information for which a schema is provided and which is within the scope of the information system – is assumed to be available in the information system. Everything that is not represented in the information system is assumed to be either not relevant or not existing. Example 6.30 exemplifies the CWA in the context of our sample problem.

Making the closed world assumption is again in contrast to de-centralized information system which – in their context – rightfully make the *open world assumption*, where one cannot infer anything from non-available information.

Example 6.30 (Closed World Assumption)**.** The scope of our sample product hierarchy are products sold by our sample company. This sample product hierarchy has two m-objects, *Book* and *Car*, at level *category*. Making the closed world assumption we can thus infer that our company does not sell any products other than books and cars. The scope of m-relationship *producedBy* is given by the companies that are possible suppliers for products sold by our company. If there is no producedBy-relationship between a specific product and a specific company available in the m-model, we can assume that this company is – at the moment – not considered to be a possible supplier for this product.

Top-down multi-level modeling and closed-world assumption have implications that become apparent only when looking at their combined use. A lower level m-object or m-relationship can only be created if a respective higher level m-object is already available in the m-model. That is, lower levels are dependent on the set of m-objects or set of m-relationships at the next higher level. This restrictive role of extensions of higher levels in a concretization hierarchy is especially strong if m-objects and m-relationships are created at different times and by different modelers and especially if modelers at lower levels do not have permission to introduce higher-level m-objects or higher-level m-relationships. In the following we exemplify this constraining role of level extensions in m-object hierarchies (see Ex. 6.31) and in m-relationship hierarchies (see Ex. 6.32).

Example 6.31 (Top-Down Modeling)**.** Consider our sample product hierarchy with levels *category*, *model*, and *physicalEntity*. In our sample company, like in most companies, information about objects at these different levels is provided by different departments (note, that this is still a simplification since data management concerning products is typically further fragmented in companies). In our sample company, data management concerning product categories is the privilege of a central administrative department that directly reports to the CEO. Data about product models is managed by members of the product management department, and data about *physical entities* is managed by the *production* or the *logistics* departments. Hence, a member of the product management department may only create new product models, that fall within one of the two product categories, which are *Book* and *Car*. Along these lines, a member of the production department may only create representations of physical entities of products that belong to one of the product models that have previously been created by a member of the product management department. Furthermore, each product category is assigned to a product manager by our central administrative department. It assigned *Mr.Black* to be responsible for category *Book*, which might mean that he is the one who has to care about all the book data.

Example 6.32 (Top-down Modeling of M-Relationships)**.** Consider our sample product hierarchy and our hierarchy of companies and the hierarchy of *producedBy* relationships between them. Information concerning *producedBy* relationships at connection-levels *category-industrialSector*, *model-enterprise* and *physicalEntity-factory* is again managed by different departments. Data at connection-level *category-industrialSector* is managed by our central administrative department. Data at connection-level *model-enterprise* is managed by the strategic purchasing department. A *producedBy*-m-relationship at level *model-enterprise* typically comes with information about framework contracts and pricing information (note, that attributes of m-relationships are out of scope of this thesis and are only mentioned to clarify the big picture). Purchasing of product physical entities is in the responsibility of the operative purchasing department and leads to *producedBy*-m-relationships at connection-level *physicalEntity-factory*. The constraints imposed by m-relationships in this setting mean, that operative purchasing staff can only buy product physical entities from factories, for which the strategic purchasing department has introduced a *producedBy*-relationship at level *model-enterprise*. In our example (see Figure 4.4), our central administration department introduced a single *producedBy*-m-relationship, namely between *Car* and *CarManufacturer*. Hence, the strategic purchasing department may only introduce *producedBy*-m-relationships between *Car* models and enterprises of sector *CarManufacturer*. They introduced only one such m-relationship, namely between *Porsche911CarreraS* and *PorscheLtd*. That means, that our operative purchasing staff may only introduce *producedBy*-relationships between physical entities of *Porsche911CarreraS* and factories of *PorscheLtd*.

While not relevant in the context of this chapter, we now also review, in order to complement the big picture, top-down specialization in an organizational context. Modelers at higher levels (i.e., modelers that have the permission to create and manipulate m-objects and m-relationships at higher levels), also define which data has to be provided by modelers at lower levels and also define the levels at which this data has to be provided. That is, modelers at higher levels define, refine, and extend schemas for lower level data.

Example 6.33 (Specialization)**.** Consider our sample company with its hierarchically organized data administration (see Ex. 6.31). Our central administrative department not only defines the set of product categories relevant in our company, it also defines – reflecting strategic information needs – which data has to be provided by product managers for each car model and for each book title (books at level model) and which data has to be provided by production and logistics staff concerning each product physical entity that leaves the company. At the next level, product managers not only create and manipulate data about product models, they also – reflecting their information needs – specialize which data has to be provided by members of the production and logistics departments about physical entities of products.

Query Support for Modeling M-Relationships As already mentioned in Chapter 4, an m-relationship has extensions, static domains, and active domains at its connection-levels. At a given connection-level, its *extension* is given by the set of descendant m-relationships at this connection-level (see Definition 6.21). Its *static domain* at a specific connection-level is given by the cross-product of the extensions of its source m-object and its target m-object at respective abstraction levels, as defined in the following.

Definition 6.12 (Static Domain)**.** *The* domain *of an m-relationship $r \in R$ at a given connection-level $(l, \bar{l}) \in C_r$, denoted as $r\langle{:}l, \bar{l}{:}\rangle$, is the cross-product of the extensions of the connected m-objects at the given level:*

$r\langle{:}l, \bar{l}{:}\rangle \overset{\text{def}}{=} s_r\langle l\rangle \times t_r\langle \bar{l}\rangle$.

We further define an operator to refer to the direct-subsuming m-relationships —with regard to a given m-relationship hierarchy— of a given coordinate $(o, \bar{o})$. These direct-subsuming m-relationships are the one to look at when introducing a new m-relationship into a concretization hierarchy of m-objects.

Definition 6.13 (Direct-Subsuming M-Relationships)**.** *Given are a coordinate $(o, \bar{o}) \in O \times O$ and an m-relationship $r \in R$ with $(o, \bar{o}) \prec (s_r, t_r)$. The direct-subsuming m-relationships of $(o, \bar{o})$ with regard to r, denoted as $\varsigma_{(o,\bar{o})}(r)$, are m-relationships that are within the same hierarchy as r, connecting a pair of m-objects above $(o, \bar{o})$, and having no concretizations, which* *are above* $(o, \bar{o})$*:* $\varsigma_{(o,\bar{o})}(r) \overset{\text{def}}{=} \{r' \in R \mid r' \preceq \hat{r}_r \wedge (o, \bar{o}) \prec (s_{r'}, t_{r'}) \wedge \nexists r'' \in R : (o, \bar{o}) \preceq (s_{r''}, t_{r''}) \wedge r'' \prec r'\}$.

When making the top-down/closed-world assumption and modeling at a specific abstraction level (i.e., connection-level) one can only create m-relationships at coordinates for which parent m-relationships have already been created at the next higher connection-level. This set of coordinates is called the *active domain* of an m-relationship at this connection-level. The active domain is the set of coordinates for which m-relationships at the given connection-level may be introduced without the need of introducing m-relationships at higher connection-levels. In an organizational context the active domain defines which m-relationship may be introduced by an organizational unit without relying on a superordinate organizational unit.

Definition 6.14 (Active Domain)**.** *The* active domain *of an m-relationship* $r \in R$ *at a given connection-level* $(l, \bar{l}) \in C_r$*, denoted as* $r\langle\cdot l, \bar{l}\cdot\rangle$*, is a subset of the domain of* r *at that connection level and only consists of coordinates having no direct-subsuming m-relationship with regard to* r*, that has a second-top-level above* $(l, \bar{l})$*:*

$r\langle\cdot l, \bar{l}\cdot\rangle \overset{\text{def}}{=} \{(o, \bar{o}) \in r\langle: l, \bar{l}:\rangle \mid \forall r' \in \varsigma_{(o,\bar{o})}(r) \Rightarrow \nexists(l', \bar{l}') \in (C_{r'} \setminus \{\hat{c}_{r'}\}) : (l, \bar{l}) \prec (l', \bar{l}')$

Example 6.34 (Active Domain)**.** Consider m-objects and m-relationships depicted in Figure 6.5. To better exemplify active domains, we added, in difference to Figure 6.4, an additional book title (*LoR*, short for *Lord of the Rings*) and an additional publisher enterprise, namely *GeoAllenUnwin*. Note that we ignore attribute definitions, since they are not necessary for explaining active domains.

Our central administrative department has asserted that *Book*s are produced by *Publisher*s and that *Car*s are produced by *CarManufacturer*s. Now, our strategic purchasing department wants to link *Product model*s with their producing companies at level *enterprise* (it already connected *HarryPotter4* with *Bloomsbury*, and *Porsche911CarreraS* with *PorscheLtd*) . To find out

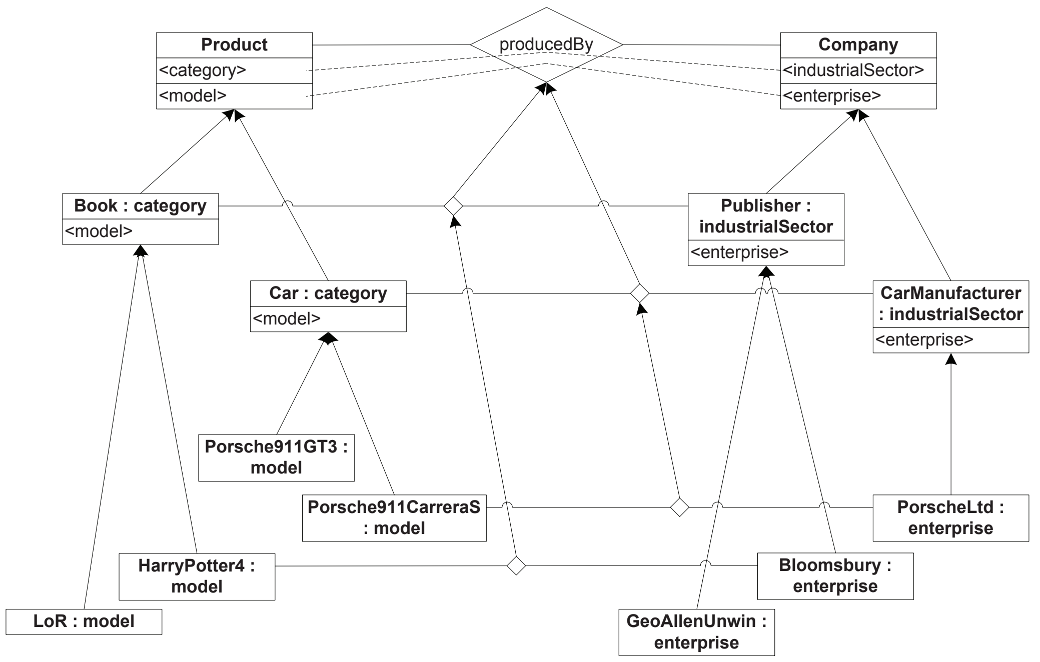


Figure 6.5: Sample multi-level model adapted for exemplifying active domain queries

about the m-relationships that are possible with regard to *producedBy* relationships asserted at the next higher level, it asks for the *active domain* of m-relationship *producedBy* at connection-level *(model, enterprise)*:

producedBy$\langle$·*model, enterprise*·$\rangle$

and gets {*(LoR, Bloomsbury), (LoR, GeoAllenUnwin), (HarryPotter4, Bloomsbury), (HarryPotter4, GeoAllenUnwin), (Porsche911GT3, PorscheLtd), (Porsche911CarreraS, PorscheLtd)*} as result. These are the coordinates where the purchasing department may create a *producedBy*-m-relationship.

The distinction between *static domain*, *active domain*, and *extension* of an m-relationship at a given connection-level is exemplified in Table 6.2. Static domains and active domains of m-relationships are sets of coordinates. Differently, an extension of an m-relationship at a given connection-level (see Def. 6.8) is a set of m-relationships. With regard to consistency criteria for

Static Domain *producedBy*⟨:*model*,*enterprise*:⟩	Active Domain *producedBy*⟨·*model*, *enterprise*·⟩	Extension *producedBy*⟨*model*, *enterprise*⟩
(LoR, GeoAllenUnwin)	*(LoR, GeoAllenUnwin)*	
(LoR, Bloomsbury)	*(LoR, Bloomsbury)*	
(LoR, PorscheLtd)		
(HarryPotter4, GeoAllenUnwin)	*(HarryPotter4, GeoAllenUnwin)*	
(HarryPotter4, Bloomsbury)	*(HarryPotter4, Bloomsbury)*	*HarryPotter4-producedBy-Bloomsbury*
(HarryPotter4, PorscheLtd)		
(Porsche911CarreraS, GeoAllenUnwin)		
(Porsche911CarreraS, Bloomsbury)		
(Porsche911CarreraS, PorscheLtd)	*(Porsche911CarreraS, PorscheLtd)*	*Porsche911CarreraS-producedBy-PorscheLtd*
(Porsche911GT3, GeoAllenUnwin)		
(Porsche911GT3, Bloomsbury)		
(Porsche911GT3, PorscheLtd)	*(Porsche911GT3, PorscheLtd)*	

Table 6.2: Static domain, active domain, and extension of m-relationship producedBy at connection-level (model, enterprise) as depicted in Figure 6.5

concretization of m-relationships (see Definition 4.12), static domain only considers rule *'source and/or target concretization'*. Active domain, as introduced above, also considers rule *'connection-level containment'* as well as rule *'direct-subsuming parent'*.

Active Domain Navigation Active domain navigation provides an object-centered perspective on active domains of m-relationships. While navigation along m-relationships is based on extensions of m-relationships, active domain navigation is based on active domains of m-relationships.

The *active target domain* of m-relationship r at level l for m-object o is the set of m-objects to which m-object o may establish or already has an outgoing m-relationship (an m-relationship of which o is the source) which is r or a descendant of r. In the opposite direction, the *active source domain* of m-relationship r at level l for m-object o is the set of m-objects to which m-object o may establish or has an incoming m-relationship (an m-relationship of which o is the target) that is r or a descendant of r.

Definition 6.15 (Active Domain Navigation)**.** *The active target domain of m-relationship $r \in R$ at level $l \in L$ for m-object $o \in O$, denoted as $o \twoheadrightarrow r(l)$, is defined by:*

$$o \twoheadrightarrow r(l) \overset{\text{def}}{=} \{\bar{o} \in O \mid (o, \bar{o}) \in r\langle\!\cdot \hat{l}_o, l \cdot\!\rangle\}$$

In the opposite direction, the active source domain of m-relationship $r \in R$ for level $l \in L$, for m-object $o \in O$, denoted as $o \twoheadrightarrow r^-(l)$, is defined by:

$$o \twoheadrightarrow r^-(l) \overset{\text{def}}{=} \{\bar{o} \in O \mid (\bar{o}, o) \in r\langle\!\cdot l, \hat{l}_o \cdot\!\rangle\}$$

Example 6.35 (Active Domain Navigation)**.** Consider m-objects and m-relationships in Figure 6.5 and the following active domain navigation operations and their results

LoR↠producedBy(enterprise)
  = {*GeoAllenUrwin, Bloomsbury*}
HarryPotter4↠producedBy(enterprise)
  = {*GeoAllenUrwin, Bloomsbury*}
Porsche911GT3↠producedBy(enterprise)
  = {*PorscheLtd*}
PorscheLtd↠producedBy$^-$(model)
  = {*Porsche911GT3, Porsche911CarreraS*}

6.4 Summary

In this chapter we complemented the core m-object/m-relationship approach with a discussion on how to query multi-level models. We introduced an object-centered query approach based on an algebra that is closed on sets of m-objects, together with a relationship-centered query approach based on an algebra that is closed on sets of m-relationships. The object-centered query approach is complemented by an operator to navigate along m-relationships. Querying m-objects and m-relationships is object-preserving rather than object-generating (see [102] for this distinction).

To encode custom behavior of m-objects and m-relationships, methods without side-effects can be formulated in terms of generic query operations as introduced in this Chapter. Methods with side-effects additionally contain generic manipulation operations (see previous chapter). Similar to predefined predicates, as introduced in this chapter, methods can be concretized at lower level m-objects and apply dynamic binding.

In this chapter we also discussed how multi-level modeling with m-objects and m-relationships is applied in an organizational context. For this purpose we provided additional query support (active domain) for top-down multi-level modeling in a closed-world setting.

In subsequent chapters we will discuss how m-objects/m-relationships can be applied to ontology engineering and to data warehousing. The OWL mapping presented in the next chapter is primarily based on addressing class extensions of m-objects as well as m-relationship extensions, as discussed in this chapter, by OWL DL class expressions. The OLAP query approach presented in Chapter 8 is loosely based on query operations introduced in this chapter.

Chapter 7

Multi-Level Modeling in OWL

Contents

In this chapter we show how to transfer ideas of multi-level modeling from conceptual modeling to web ontology engineering by providing semantic-preserving mappings from m-objects and m-relationships to the decidable fragment of OWL (web ontology language), extended by integrity constraints. In Section 7.1 we give a very short introduction to OWL and outline our motivation for mapping m-objects and m-relationships to OWL. Section 7.2 shows how to map m-objects to OWL and how to check consistency of and to query multi-level models using OWL reasoners. Section 7.3 shows how to map m-relationships to OWL and shows how m-relationships can be queried using OWL reasoners. Section 7.4, which concludes the chapter, briefly describes appropriate tool support.

This chapter is based on a paper [87] we presented at a workshop at the 28th International Conference on Conceptual Modeling (ER 2009).

7.1 Introduction

The *web ontology language* (OWL) [78] is the de facto standard language for ontologies on the semantic web [112, 50]. An ontology is a conceptual schema of a domain, or in other words, a formal representation of a conceptualization of a domain [35]. OWL is used for various tasks in information retrieval, in information integration, and in conceptual modeling. OWL reasoners support different reasoning tasks (see [24]), most notably checking of class membership (querying of instance data), as well as subsumption and consistency checking (support for terminology development and conceptual modeling). Since the web is an open, de-centralized information system [15], OWL makes different basic assumptions than database models. OWL makes the *open world assumption* which is in contrast to the *closed world assumption*, that is, OWL reasoners assume that there might always be additional information that is not available. Furthermore OWL does not make the *unique name assumption*, that is, one object might come with different names.

OWL has three sub-languages: OWL Lite, OWL DL, and OWL Full. OWL Full is the most expressive sub-language and does not pose any restrictions on the use of its language features, but reasoning with OWL Full is, in the general case, undecidable. OWL Lite was designed to be easily implementable and easy to reason with, however it is not very expressive. We are, thus, primarily interested in OWL DL which is based on description logic $\mathcal{SHOIN}$ (see [13] for an introduction to Description Logics), has model-theoretic semantics, and provides for decidable reasoning. OWL 2 [1] is the most recent version of OWL and comes with additional sub-languages and additional language constructs. Its decideable (DL) fragment is based on description logic $\mathcal{SROIQ}$ [51] and comes with improved expressivity.

OWL supports metamodeling in two flavors. Metamodeling in OWL Full allows to treat classes as individuals but is not decideable. OWL 2 supports a very basic, but decideable approach to metamodeling called punning or contextual semantics [80]: one symbol can be used to refer both to a class as well as to an individual; the decision whether a symbol is interpreted as class, property, or individual is context-dependent. For example, the symbol *Car* interpreted as class refers to the set of physical entities that belong to product category *car*, while *Car* interpreted as individual can be classified as *ProductCategory* and has assigned a value for property *taxRate*.

In conceptual modeling more powerful approaches to ontological metamodeling (or multi-level modeling) exist: materialization[98], potency-based deep instantiation[7], and m-objects/m-relationships[86]. These approaches not only support to treat classes as individuals but also to describe domain concepts with members at multiple levels of abstraction. For example, domain concept *Product* has members at different levels of abstraction: *Car* at level *category*, *Porsche911CarreraS* at level *model* and *MyPorsche911CarreraS* at level *physical entity*.

Previous work has shown how to bridge the gap between Model-Driven Architecture and OWL [28], discussed the relation between ontologies and (meta)modeling [40], and sketched how to represent materialization with Description Logics [17]. Herein, we continue the latter line of work for the more powerful m-object/m-relationships approach by introducing a detailed mapping from m-objects/m-relationships to OWL.

Our mapping from m-objects/m-relationships to OWL is motivated as follows: (1) For conceptual modeling: to provide querying facilities and decideable consistency checking for multi-level models using OWL reasoners. (2) For ontology engineering: to extend the metamodeling features of OWL 2 (punning) to objects that represent classes at multiple levels of abstraction, and still remain within the decideable and first-order fragment of OWL. In the latter case, the outcome of the mapping serves as a basis for multi-level ontology engineering and can be augmented and combined with further OWL axioms and ontologies.

Some of the consistency criteria of m-objects/m-relationships have to be evaluated as integrity constraints which are not available in OWL. For this we rely on an existing approach [81] that allows to combine open world and closed world reasoning.

7.2 Mapping M-Objects to OWL

The basic goals of mapping m-objects to OWL are (1) to preserve their semantics, fulfilling their basic goals and requirements, and (2) to provide an OWL-representation that allows OWL reasoners (a) to detect inconsistencies and (b) to execute queries at the different levels of abstraction. In this section we describe each step of the mapping from m-objects to OWL and exemplify the more interesting steps by showing mapping-output for m-object *Car* (see Figure 1.4 in Chapter 1). Listing 7.2 shows the mapping output for m-object *Product*, of which *Car* is a concretization of. The mapping procedure is summarized in Algorithm 1, which is independent from the naming scheme of the m-object/m-relationship approach, thus we do not refer to naming functions (*name* and *ns*). To improve readability, we use Description Logic syntax [12] and for brevity we do not explicitly introduce entities (i.e. individuals, classes, and properties). Also note, that Description Logic syntax does not differentiate between data properties and object properties.

Since the m-object approach makes the *closed world assumption* (CWA), which OWL does not, the mapping approach has to assure that axioms that are meant to be interpreted as integrity constraints do not lead to unwanted inferences, e.g., wrong classifications. For this we rely on an existing approach [81] that allows to designate certain TBox axioms as integrity constraints. In our mappings such axioms are marked by '`IC:`'. For TBox (schema) reasoning these axioms are treated as usual, but for ABox (data) reasoning, they are treated only as checks and do not derive additional information.

Concretization hierarchies of m-objects are mapped to OWL by representing *each m-object as individual*, e.g., Car, and *each abstraction level as primitive class*, e.g., Category. Each m-object is assigned to an abstraction level by a class assertion. Each m-object with a parent m-object is connected to this parent using functional property concretize (see Algorithm 1, lines 5 – 6). E.g., individual *Car* is member of class *Category* and conretizes *Product*:

```
1: Category(Car)
2: concretize(Car, Product)
```

Values of top-level attributes, i.e. attribute values that describe the m-object itself (a.k.a. *own-slots*), are represented as property assertions (see Algorithm 1, line 7). E.g. product category *Car* has assigned a *taxRate* of *20*:

```
3: taxRate(Car, 20)
```

As explained in Section 6.1, a level of an m-object is seen as a class that collects all direct and indirect concretizations of the m-object at the respective level. Such a class can be used (1) as entry point for queries, (2) to define extensional constraints, and (3) to define and refine common characteristics of its members. The class of individuals that belong to abstraction level l and that are direct or indirect concretizations of individual o, i.e. $o\langle l\rangle$, corresponds to class expression ($\exists$concretize_t.$\{[o]\} \sqcap [l]$). concretize_t is defined as transitive super-property of concretize. E.g., the class of all car models, $Car\langle Model\rangle$, corresponds to class expression ($\exists$concretize_t.{Car} $\sqcap$ Model).

Thus, a concretization hierarchy implicitly introduces *multiple subsumption hierarchies*, one for each level, as explained in Chapter 6 (see Figure 7.1 for the resulting classes and subclasses in our example). E.g., from (concretize(Car, Product)) an OWL reasoner infers that ($\exists$concretize_t.{Product} $\sqcap$ Model) subsumes ($\exists$concretize_t.{Car} $\sqcap$ Model) and that ($\exists$concretize_t.{Product} $\sqcap$ PhysicalEntity) subsumes

($\exists \mathsf{concretize\_t}.\{\mathsf{Car}\} \sqcap \mathsf{PhysicalEntity}$). This powerful feature of OWL facilitates inheritance between these classes as well as consistency checks.

Common characteristics of the members of a certain level of an m-object are defined by subclass axioms with the respective class expression at the left hand side. Attributes, in particular, are represented by data properties and respective value and number restrictions (see Algorithm 1, lines 8 – 9). Values of attributes might be shared with m-objects at lower levels (see Algorithm 1, line 10), no example shown. To preserve the semantics of the m-object approach, these axioms are to be interpreted as integrity constraints. E.g., *Car⟨brand⟩s* have an attribute *marketLaunch*, *Car⟨model⟩s* have an attribute *maxSpeed* (in addition to attribute *listPrice* from *Product⟨model⟩*), and *Car⟨physicalEntity⟩s* have an attribute *mileage* (in addition to *serialNr* from *Product⟨physicalEntity⟩*):

4: $\texttt{IC:}\exists \mathsf{concretize\_t}.\{\mathsf{Car}\} \sqcap \mathsf{Brand} \sqsubseteq \forall \mathsf{marketLaunch.Date} \sqcap {=}1\ \mathsf{marketLaunch}.\top$

5: $\texttt{IC:}\exists \mathsf{concretize\_t}.\{\mathsf{Car}\} \sqcap \mathsf{Model} \sqsubseteq \forall \mathsf{maxSpeed.Integer} \sqcap {=}1\ \mathsf{maxSpeed}.\top$

6: $\texttt{IC:}\exists \mathsf{concretize\_t}.\{\mathsf{Car}\} \sqcap \mathsf{PhysicalEntity} \sqsubseteq \forall \mathsf{mileage.Integer} \sqcap {=}1\ \mathsf{mileage}.\top$

A level l of an m-object o ensures that concretizations of o at lower levels also concretize a concretization of o at level l (see Algorithm 1, lines 11 – 12). This allows *stable upward navigation* and supports *heterogenous level hierarchies* by allowing that every m-object may introduce new abstraction levels for its descendants, that do not apply for descendants of other m-objects. E.g., all *Car Models* belong to a *Car Brand*:

7: $\texttt{IC:}\exists \mathsf{concretize\_t}.\{\mathsf{Car}\} \sqcap \mathsf{Model} \sqsubseteq \exists \mathsf{concretize\_t}.(\exists \mathsf{concretize\_t}.\{\mathsf{Car}\} \sqcap \mathsf{Brand})$

The mapping also ensures that each attribute is inducted at only one level of one m-object (see Algorithm 1, lines 13 – 14), and that each level is inducted at only one m-object (see Algorithm 1, lines 15 – 16); no examples shown.

To ensure that an m-object belongs to one abstraction level at most, all levels are pairwise disjoint (see Algorithm 1, line 17). E.g., an individual at level *Brand* cannot, at the same time, be at level *Model*:

Algorithm 1 Mapping M-Objects to OWL

```
    Input: a set $O$ of m-objects $o = (L_o, A_o, p_o, l_o, d_o, v_o)$, a concretization hierarchy $H$, and a universe of levels $L$, as described in Def. 4.1.
    Output: a set of OWL axioms.
 1: assert: $\top \sqsubseteq \leqslant 1\ \mathsf{concretize}$
 2: assert: $\mathsf{concretize} \sqsubseteq \mathsf{concretize\_t}$
 3: assert: $\mathsf{concretize\_t}^+ \sqsubseteq \mathsf{concretize\_t}$
 4: for all $o \in O$ do
 5:     assert: $[\hat{l}_o]([o])$
 6:     if $\exists o' : (o, o') \in H$ then assert: $\mathsf{concretize}([o],[o'])$
 7:     for all $a \in \hat{A}_o : v_o(a)$ is defined do assert: $[a]([o],[v_o(a)])$
 8:     for all $a \in (A_o \setminus \hat{A}_o)$ do
 9:         assert IC: $\exists \mathsf{concretize\_t}.\{[o]\} \sqcap [l_o(a)] \sqsubseteq \forall [a].[d_o(a)] \sqcap {=}1\ [a].\top$
10:         if $v_o(a)$ is defined then assert: $[l_o(a)] \sqcap \exists \mathsf{concretize\_t}.\{[o]\} \sqsubseteq \exists [a].\{[v_o(a)]\}$
11:     for all $(l, l') \in P_o : l' \neq \hat{l}_o \wedge (\nexists o' \in O : (o, o') \in H \wedge (l, l') \in P_{o'})$ do
12:         assert IC: $\exists \mathsf{concretize\_t}.\{[o]\} \sqcap [l] \sqsubseteq \exists \mathsf{concretize\_t}.(\exists \mathsf{concretize\_t}\{[o]\} \sqcap [l'])$
13:     for all $a \in A_o : \nexists o' \in O : (o, o') \in H \wedge a \in A_{o'}$ do
14:         assert IC: $\exists [a].\top \sqsubseteq (\exists \mathsf{concretize\_t}.\{[o]\} \sqcup \{[o]\}) \sqcap [l_o(a)]$
15:     for all $l \in L_o : l \neq \hat{l}_o \wedge (\nexists o' \in O : (o, o') \in H \wedge l \in L_{o'})$ do
16:         assert IC: $[l] \sqsubseteq \exists \mathsf{concretize\_t}.\{[o]\}$
17: for all $l \in L, l' \in (L \setminus \{l\})$ do assert: $[l] \sqcap [l'] \sqsubseteq \bot$
18: for all $o \in O, o' \in (O \setminus \{o\})$ do assert: $[o] \not\approx [o']$
    Notation: '$[o]$' denotes a variable that is to be substituted by its actual value. 'assert: ' adds the subsequent OWL axiom to the mapping output. 'IC:' denotes an OWL axiom that is to be interpreted as integrity constraint.
```

8: $\mathsf{Brand} \sqcap \mathsf{Model} \sqsubseteq \bot$

The m-object approach makes the *unique name assumption.* Thus we state that each pair of m-objects (cartesian product) is a member of the inequality predicate (see Algorithm 1, line 18). E.g., the symbols *Car* and *Porsche911* refer to different individuals:

9: $\mathsf{Car} \not\approx \mathsf{Porsche911}$

Listing 1 Mapping output for m-object *Product.* (Disjointness and Inequality Axioms are not shown)

$\mathsf{Catalog(Product)}$

$\mathsf{desc(Product, "Our\ Products")}$

$\mathtt{IC}{:}\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcap \mathsf{Category} \sqsubseteq \forall \mathsf{taxRate.Integer} \sqcap {=1}\ \mathsf{taxRate}.\top$

$\mathtt{IC}{:}\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcap \mathsf{Model} \sqsubseteq \forall \mathsf{listPrice.Float} \sqcap {=1}\ \mathsf{listPrice}.\top$

$\mathtt{IC}{:}\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcap \mathsf{PhysicalEntity} \sqsubseteq \forall \mathsf{serialNr.String} \sqcap {=1}\ \mathsf{serialNr}.\top$

$\mathtt{IC}{:}\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcap \mathsf{Model} \sqsubseteq \exists \mathsf{concretize\_t}.(\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcap \mathsf{Category})$

$\mathtt{IC}{:}\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcap \mathsf{PhysicalEntity} \sqsubseteq \exists \mathsf{concretize\_t}.(\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcap \mathsf{Model})$

$\mathtt{IC}{:}\exists \mathsf{desc}.\top \sqsubseteq (\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcup \{\mathsf{Product}\}) \sqcap \mathsf{Catalog}$

$\mathtt{IC}{:}\exists \mathsf{taxRate}.\top \sqsubseteq (\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcup \{\mathsf{Product}\}) \sqcap \mathsf{Category}$

$\mathtt{IC}{:}\exists \mathsf{listPrice}.\top \sqsubseteq (\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcup \{\mathsf{Product}\}) \sqcap \mathsf{Model}$

$\mathtt{IC}{:}\exists \mathsf{serialNr}.\top \sqsubseteq (\exists \mathsf{concretize\_t}.\{\mathsf{Product}\} \sqcup \{\mathsf{Product}\}) \sqcap \mathsf{PhysicalEntity}$

$\mathtt{IC}{:}\mathsf{Category} \sqsubseteq \exists \mathsf{concretize\_t}.\{\mathsf{Product}\}$

$\mathtt{IC}{:}\mathsf{Model} \sqsubseteq \exists \mathsf{concretize\_t}.\{\mathsf{Product}\}$

$\mathtt{IC}{:}\mathsf{PhysicalEntity} \sqsubseteq \exists \mathsf{concretize\_t}.\{\mathsf{Product}\}$

7.3 Mapping M-Relationships to OWL

There are basically two alternative representations of m-relationships in OWL, as properties or as individuals. At first sight a mapping to property assertions (*property approach*) seems intuitive. However, since property assertions do not have identifiers it is not possible to directly represent concretiza-

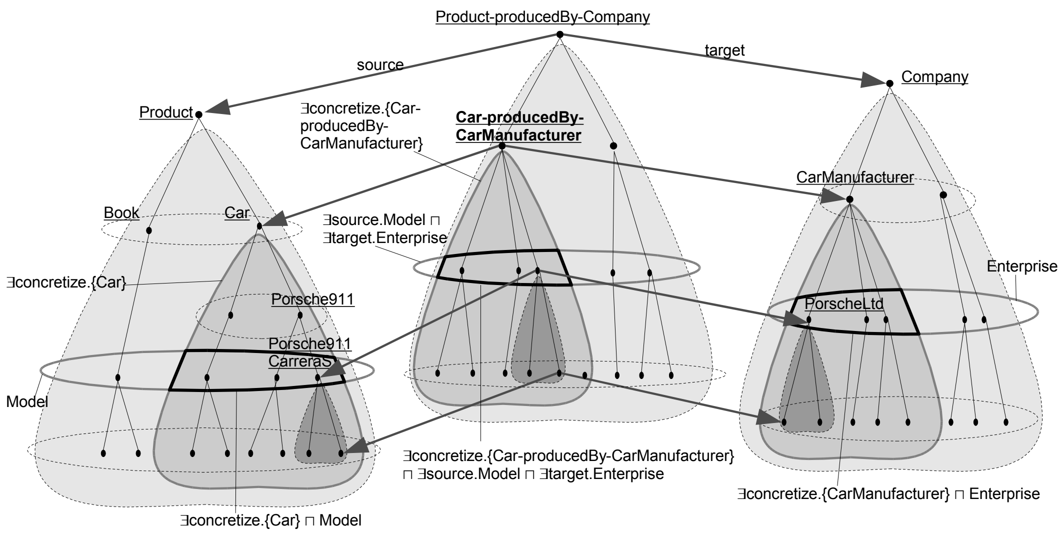

Product-producedBy-Company
source
target
Product
Company
∃concretize.{Car-producedBy-CarManufacturer}
Car-producedBy-CarManufacturer
CarManufacturer
Book
Car
∃source.Model ⊓ ∃target.Enterprise
Enterprise
∃concretize.{Car}
Porsche911
PorscheLtd
Porsche911 CarreraS
Model
∃concretize.{Car-producedBy-CarManufacturer} ⊓ ∃source.Model ⊓ ∃target.Enterprise
∃concretize.{CarManufacturer} ⊓ Enterprise
∃concretize.{Car} ⊓ Model

Figure 7.1: Sets and subsets of m-objects and m-relationships. M-relationship *Car-producedBy-CarManufacturer* and its members, domain, and range at level *(model, enterprise)* are emphasized

.

tion links. Thus it would be necessary to represent each connection-level of an m-relationship as property and redundantly represent a concretization-link between two m-relationships by several sub-property-axioms. To avoid this redundancy we suggest to map m-relationships to individuals (*objectification approach*) which allows to directly represent concretization-links between m-relationships. We again describe each step of the mapping and exemplify them based on m-relationship *producedBy* between *Car* and *CarManufacturer* from Example 1.1 (see Figure 1.4). Listing 7.3 shows the mapping output for m-relationship *Product-producedBy-Company*, of which *Car-producedBy-CarManufacturer* is a concretization of. The mapping procedure is summarized in Algorithm 2.

The *objectification approach* represents each m-relationship as individual that is linked to its parent relationship and to its source- and target-m-objects by property assertions, using functional properties concretize, source, and target, respectively (see Algorithm 2, lines 3 – 5). E.g., m-relationship *producedBy* between *Car* and *CarManufac-*

turer, named *Car-producedBy-CarManufacturer*, concretizes m-relationship *Product-producedBy-Company*, its source is *Car* and its target is *CarManufacturer*:

10: concretize(Car-producedBy-CarManufacturer, Product-producedBy-Company)
11: source(Car-producedBy-CarManufacturer, Car)
12: target(Car-producedBy-CarManufacturer, CarManufacturer)

An m-relationship constrains domain and range of its concretizations as defined in Chapter 4 (*source- or target level concretization*) (see Algorithm 2, line 6). E.g., all direct or indirect concretizations of *Car-producedBy-CarManufacturer* have a direct or indirect concretization of *Car*, or *Car* itself, as source, and a direct or indirect concretization of *CarManufacturer*, or *CarManufacturer* itself, as target. Either *Car* or *CarManufacturer* must be concretized:

13: IC:$\exists$concretize_t.$\{$Car-producedBy-CarManufacturer$\} \sqsubseteq$ ($\forall$source.($\exists$concretize_t. $\{$Car$\} \sqcup \{$Car$\}$) $\sqcap$ $\forall$target.$\exists$concretize_t.$\{$CarManufacturer$\}$)$\sqcup$ ($\forall$source.$\exists$concretize_t. $\{$Car$\}$ $\sqcap\forall$target.($\exists$concretize_t.$\{$CarManufacturer$\}$ $\sqcup$ $\{$CarManufacturer$\}$))

Algorithm 2 Mapping M-Relationships to OWL

```
    Input: A set $R$ of m-relationships $r = (s_r, t_r, C_r)$ and a concretization hierarchy $H_R$, as described in Def. 4.8.
    Output: a set of OWL axioms.
1:  assert: $\top \sqsubseteq \leqslant 1 source \sqcap \leqslant 1 target$
2:  for all $r \in R$ do
3:      if $\exists r' : (r, r') \in H_R$ then assert: concretize($[r]$,$[r']$)
4:      assert: source($[r]$,$[s_r]$)
5:      assert: target($[r]$,$[t_r]$)
6:      assert IC: $\exists$concretize_t.$\{[r]\} \sqsubseteq$ ($\forall$source.($\exists$concretize_t.$\{[s_r]\} \sqcup \{[s_r]\}$ ) $\sqcap$ $\forall$target.$\exists$concretize_t.$\{[t_r]\}$) $\sqcup$ ($\forall$source.$\exists$concretize_t.$\{[s_r]\}$ $\sqcap\forall$target.($\exists$concretize_t.$\{[t_r]\} \sqcup \{[t_r]\}$ ))
7:      for all $(l, l') \in C_r : l \neq \hat{l}_{s_r} \vee l' \neq \hat{l}_{t_r}$ do
8:          assert IC: $\exists$concretize_t.$\{[r]\}\sqcap$($\exists$source.$\exists$concretize_t.$[l]$ $\sqcup$ $\exists$target.$\exists$concretize_t.$[l']$) $\sqsubseteq$ $\exists$concretize_t.($\exists$concretize_t.$\{[r]\}$ $\sqcap\exists$source.$[l]$ $\sqcap\exists$target.$[l']$)
9:      for all $r' \in R : r \neq r'$ do assert: $[r] \not\approx [r']$
```

Analogous to levels of m-objects, which ensure safe upward navigation, connnection levels of m-relationships ensure safe navigation along m-relationships at higher levels. A connection-level (l, l') of an m-relationship r ensures that concretizations of r at levels below (l, l') concretize an m-relationship at level (l, l') that concretizes r. This has to be interpreted as integrity constraint (see Algorithm 2, lines 7 – 8). E.g., all m-relationships below connection-level *(Model,Enterprise)* that concretize *Car-producedBy-CarManufacturer* have to concretize an m-relationship that concretizes *Car-producedBy-CarManufacturer* at level *(Model,Enterprise)* (An analogous axiom has to be asserted for connection-level *(PhysicalEntity,Factory)*):

14: $\texttt{IC:}\exists\mathsf{concretize\_t}.\{\mathsf{Car\text{-}producedBy\text{-}CarManufacturer}\}\sqcap(\exists\mathsf{source}.\exists\mathsf{concretize\_t}.\mathsf{Model} \sqcup \exists\mathsf{target}.\exists\mathsf{concretize\_t}.\mathsf{Enterprise}) \sqsubseteq \exists\mathsf{concretize\_t}.(\exists\mathsf{concretize\_t}.\{\mathsf{Car\text{-}producedBy\text{-}CarManufacturer}\} \sqcap\exists\mathsf{source}.\mathsf{Model} \sqcap\exists\mathsf{target}.\mathsf{Enterprise})$

To enforce the *unique name assumption* we assert that each pair of m-relationships (cartesian product) belongs to the inequality predicate (see Algorithm 2, line 9). E.g., *Car-producedBy-CarManufacturer* and *Product-producedBy-Company* refer to different individuals:

15: $\mathsf{Car\text{-}producedBy\text{-}CarManufacturer} \not\approx \mathsf{Product\text{-}producedBy\text{-}Company}$

OWL reasoners can be employed to navigate m-relationships (as defined in Chapter 6) by class expressions. For example, to query which *IndustrialSector* might produce *MyPorsche911CarreraS* (see Figure 1.4) we ask for the members of class $(\mathsf{IndustrialSector} \sqcap \exists\mathsf{target}^{-}.((\exists\mathsf{concretize\_t}.\{\mathsf{Product\text{-}producedBy\text{-}Company}\} \sqcup \{\mathsf{Product\text{-}producedBy\text{-}Company}\}) \sqcap \exists\mathsf{source}.(\exists\mathsf{concretize\_t}^{-}.\{\mathsf{MyPorsche911CarreraS}\} \sqcup \{\mathsf{MyPorsche911CarreraS}\})))$.

Listing 2 Mapping output for M-Relationship *Product-producedBy-Company*

source(Product-producedBy-Company, Product)

target(Product-producedBy-Company, Company)

`IC:`$\exists$concretize_t.{Product-producedBy-Company} $\sqsubseteq$ ($\forall$source.($\exists$concretize_t.{Product} $\sqcup$ {Product}) $\sqcap$ $\forall$target.$\exists$concretize_t.{Company})$\sqcup$ ($\forall$source.$\exists$concretize_t. {Product} $\sqcap\forall$target.($\exists$concretize_t.{Company} $\sqcup$ {Company}))

`IC:`$\exists$concretize_t.{Product-producedBy-Company}$\sqcap$($\exists$source.$\exists$concretize_t.Category $\sqcup$ $\exists$target.$\exists$concretize_t.IndustrialSector) $\sqsubseteq$ $\exists$concretize_t.($\exists$concretize_t.{Product-producedBy-Company} $\sqcap\exists$source.Category $\sqcap\exists$target.IndustrialSector)

`IC:`$\exists$concretize_t.{Product-producedBy-Company}$\sqcap$($\exists$source.$\exists$concretize_t.Model $\sqcup$ $\exists$target.$\exists$concretize_t.Enterprise) $\sqsubseteq$ $\exists$concretize_t.($\exists$concretize_t.{Product-producedBy-Company} $\sqcap\exists$source.Model $\sqcap\exists$target.Enterprise)

`IC:`$\exists$concretize_t.{Product-producedBy-Company}$\sqcap$($\exists$source.$\exists$concretize_t.PhysicalEntity $\sqcup$ $\exists$target.$\exists$concretize_t.Factory) $\sqsubseteq$ $\exists$concretize_t.($\exists$concretize_t.{Product-producedBy-Company} $\sqcap\exists$source.PhysicalEntity $\sqcap\exists$target.Factory)

Product-producedBy-Company $\not\approx$ Car-producedBy-CarManufacturer

Product-producedBy-Company $\not\approx$ Porsche911CarreraS-producedBy-PorscheLtd

Product-producedBy-Company $\not\approx$ MyPorsche911CarreraS-producedBy-PorscheZuffenhausen

7.4 Summary

In this chapter we showed how to apply the m-object/m-relationship approach to ontology engineering with OWL. We provided detailed mappings from our abstract m-object/m-relationship model to the decidable fragment of OWL and provided example mapping outputs.

To represent integrity constraints (closed world constraints) we applied an existing approach [81] that allows to combine open world and closed world reasoning. As an alternative to OWL with integrity constraints we want to investigate in future work how m-objects and m-relationships can be mapped on semantic web formalisms with native support for closed world reasoning, such as F-Logic [62] and its derivations OWL-Flight [24] and WSML [70].

In regard to tool support, we are extending the ontology editor Protégé [90] with a plugin for modeling with m-objects and m-relationships. Multi-level models designed with this Protégé extension can then - using an additional export plug-in - be mapped to OWL axioms (and integrity constraints) as described in the previous chapters. The result of this mapping may be freely augmented with open world OWL axioms and thereby integrated with ordinary OWL ontologies.

Apart from tool-support, in future work on multi-level modeling in the semantic web we are going to address (1) mapping of extended m-objects and m-relationships to OWL, (2) scalability, especially concerning reasoning performance, (3) collaborative, bottom-up multi-level modeling, and, most importantly, (4) multi-level modeling and reasoning under the open-world assumption.

Chapter 8

Hetero-Homogeneous Hierarchies in Data Warehouses

Contents

Data warehouses facilitate multi-dimensional analysis of data from various data sources. While the original data sources are often heterogeneous, current modeling and implementation techniques discard and, thus, cannot exploit these heterogeneities. In [88] we introduced *Hetero-Homogeneous Hierarchies* based on extended m-objects and m-relationships to model dimension hierarchies and cubes with inherent heterogeneities. Sub-dimension-hierarchies can be specialized to contain additional levels and additional non-dimensional attributes. Sub-cubes can be specialized towards additional measures, more fine-grained facts, and differing units of measure.

This self-contained chapter is structured as follows: In Section 8.1 we give an overview and examples of the different types of supported heterogeneities. In Section 8.2 and Section 8.3 we show how to model hetero-homogeneous dimension hierarchies and hierarchies of m-cubes, respectively, and provide structural definitions and consistency criteria. In Section 8.4 we show how to query m-cubes and, for this purpose, introduce an m-cube algebra. In Section 8.5 we briefly survey related work. In Section 8.6, which concludes the chapter, we give an outlook on future work.

8.1 Introduction

Data Warehouses facilitate multi-dimensional analysis of data integrated from various data sources. Available and interesting data is often heterogeneous concerning available measures, granularity of measures, units of measures, applicable aggregation levels, and interesting secondary information (non-dimensional attributes). However, to ease querying and storing multi-dimensional data, current modeling and implementation techniques force to fully homogenize available data according to a global multi-dimensional schema.

Our approach is summarized by the oxymoron term *hetero-homogeneous hierarchies*. A hetero-homogeneous hierarchy is a hierarchy with a single root node that is (1) homogeneous in regard to a minimal common schema shared by all sub-hierarchies, where a sub-hierarchy is a hierarchy rooted in a child of the root node, (2) heterogeneous in regard to the specialized schemas of sub-hierarchies.

We discuss our approach by a running example, starting with a homogeneous schema that can be modeled using the Dimensional Fact Model [31] (see Figure 8.1). Consider a homogeneous *sales* cube with dimensions *product*, *time*, and *location*. Dimension *product* defines dimension level *category* with non-dimensional attribute *category manager* and dimension level *model*

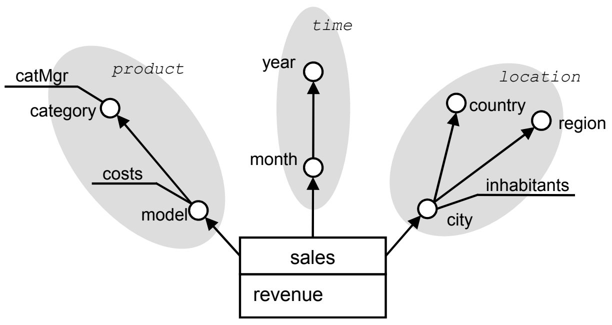


Figure 8.1: Homogeneous cube modeled with the Dimensional Fact Model

with non-dimensional attribute *costs*. The *product* dimension defines *category* above *model*. Dimension *time* defines level *year* above level *month*. Dimension *location* defines levels *country*, *region*, and *city*. Level *city* has non-dimensional attribute *inhabitants* defined. Level *city* is below levels *country* and *region*, which are in no order to each other, i.e., data at level *city* can alternatively be aggregated to level *country* or to level *region*. The *sales* cube defines a measure *revenue*.

Dimension hierarchies can be hetero-homogeneous with regard to

- *non-dimensional attributes*, e.g., in sub-hierarchy *Car* of dimension *product*, dimension instances at level model have an additional non-dimensional attribute *maxSpeed*.
- *additional levels*, e.g., in sub-hierarchy *Switzerland* of dimension *location* there is an additional level *kanton* between *city* and *country* and an additional level *store* below *city*.

Cubes can be hetero-homogeneous in that

- sub-cubes, such as *car sales in Switzerland in the year 2009* may have *additional measures*, e.g., *quantity sold*,
- different sub-cubes may give *different units* for the same measure, e.g., values of measure *revenue* are provided in *Swiss francs*

- base facts for various measures are provided at *mixed granularities*, e.g., base facts for measure *cheapestOffer* are provided at level *category*, *year*, *country* while base facts for measure *revenue* are provided at level *model*, *month*, *year*.

- different sub-cubes may provide base facts for the same measure at *different granularities*, e.g., measure *revenue* originally defined for level *model*, *month*, and *city* is now available more detailed at level *model*, *month*, and *store*.

To represent such kind of situations one needs a design approach to represent hetero-homogeneous hierarchies in dimensions and cubes. Such an approach should allow for an instance-based specialization of dimensions and cubes.

In this chapter, we show how hetero-homogeneous hierarchies in data warehouses can be modeled by a revised and extended form of m-objects and m-relationships. Note that this chapter is, in difference to previous chapters, self-contained and redefines previously defined concepts such as m-objects and m-relationships.

Hetero-homogeneous dimension hierarchies are modeled as concretization hierarchies of m-objects. An m-object encapsulates and arranges dimension levels in a partial order from abstract to concrete. It describes itself and the common properties of the objects at each level of the dimension hierarchy beneath itself. An m-object that concretizes another m-object inherits dimension levels and non-dimensional attributes of the parent. It may also introduce additional levels and additional non-dimensional attributes. For modeling dimension hierarchies we extend the original definitions of m-objects from a total order of levels to partially ordered level-hierarchies and introduce consistency criteria to avoid conflicts due to multiple concretization.

Cube schemas as well as facts are modeled as m-relationships. For modeling cube schemata and facts we extend m-relationships from binary m-relationships to n-ary m-relationships that may define measures and assert measure values. We define consistency criteria that avoid conflicts due to multiple inheritance and avoid overlapping primary fact instances.

Most current approaches to data warehousing are centered around the notion of a *cube*. Our conceptual approach to modeling and querying data warehouses is centered around multi-level cubes (m-cubes). An *m-cube* represents a cube of cubes, given by the cartesian product of dimension levels and, on a more fine-grained level, a set of coordinates, given by the cartesian product of dimension instances.

We also introduce an m-cube algebra with closed m-cube operations *dice*, *slice*, *import-union*, and *projection*; together with *fact*- and *cube-extraction* operations. Other common data warehouse operations like roll-up, drill-down, drill-across are subsumed by these operations. To cope with heterogeneous measure units we also support unit conversion. In order to exploit heterogeneities in m-cubes, queries are typically double-staged: after selecting a sub-m-cube, using dice, the query can make use of additional schema information like additional measures, refined granularity, additional non-dimensional attributes, and additional cube levels.

8.2 Hetero-Homogeneous Dimension Hierarchies

In this section we first revisit and extend m-objects and then we show how to model hetero-homogeneous dimensions with them.

8.2.1 M-Objects revisited

An m-object, as originally introduced, encapsulates and arranges abstraction levels in a linear order from the most abstract to the most concrete one. Thereby, it describes itself and the common properties of the objects at each level of the concretization hierarchy beneath itself. An m-object specifies concrete values for the properties of its top-level. This top-level describes the m-object itself. All other levels describe common properties of m-objects beneath itself.

We now give revised definitions that support m-objects with a partial (non-linear) order of levels.

Definition 8.1 (M-Object). *An m-object o is described by a 6-tuple $(L_o, A_o, P_o, l_o, d_o, v_o)$ where $L_o \subseteq L_D$ is a set of levels from a universe of levels and $A_o \subseteq A_D$ is a set of attributes from a universe of attributes. The levels L_o are organized in a partial order, as defined by parent relation $P_o \subseteq L_o \times L_D$, which associates with each level its parent levels. Each attribute is associated with one level, defined by function $l_o : A_o \to L_o$, and has a domain, defined by function $d_o : A_o \to$ data types. Optionally, an attribute has a value from its domain, defined by partial function $v_o : A_o \to V$, where V is a universe of data values, and $v_o(a) \in d_o(a)$ iff $v_o(a)$ is defined.*

An m-object has a single top-level, $\hat{l}_o := l \in L_o : \nexists l' \in L_o : (l, l') \in P_o$.

We say o is at level l, if l is its top-level. We further say level l' is a child of level l iff $(l', l) \in P_o$, and l' is a descendant of, or below, l iff $(l', l) \in P_o^+$, where P_o^+ is the transitive closure of P_o, and l' is a descendant of or the same as l iff $(l', l) \in P_o^*$, where P_o^* is the transitive-reflexive closure of P_o.

M-objects, levels, and attributes have names, defined by function $name : O \cup L \cup A \to names$, where $names$ is the universe of names. Names of m-objects, attributes, and levels are unique within one dimension.

Example 8.1 (M-object Car)**.** Product category *car* (see Figure 8.2) has three levels *category*, *brand*, and *model* and defines a value for attribute *catMgr*.

An m-object can *concretize* another m-object, which is referred to as its parent, by introducing new levels, introducing new attributes, and providing values for attributes. The concretization relationship comprises aspects of classification, generalization and aggregation. A concretization relationship between two m-objects does not reflect that one m-object is at the same time an *instance of*, *component of*, and *subclass of* another m-object as a whole. Rather, a concretization relationship has to be interpreted in a multi-faceted way. This is exemplified by the following example.

Example 8.2 (Concretization)**.** M-object *Car* concretizes *Product*. The concretization relationship is to be interpreted in a multi-faceted way: m-object *Car* is instance of level *category* of m-object *Product* because level *category*, which is the first non-top-level of m-object *Product*, is its top-level. It also specifies a value for its attribute *catMgr*. M-object *Car* specializes m-object *Product* by introducing a new level *brand* and adding attribute *maxSpeed* to level *model*. The level *model* of m-object *Car* is regarded as a subclass of level *model* of m-object *Product*.

A child m-object o' chooses its single top-level from the common second-top-levels of its parent m-objects. It 'inherits' from each parent m-object o all levels below its own top-level, together with the relative order of these common levels. It also 'inherits' attributes associated with common levels, together with the properties of these attributes, as defined by functions l_o, d_o, and v_o. In the case of multiple concretization the top-level of the child m-object must be a common second-top-level of the parent m-objects.

For simplicity, we do not define this inheritance mechanism (note that this inheritance mechanism could be defined in line with Def. 4.4) and instead assume that each m-object is fully described. We summarize the consistency criteria in the following definition.

Definition 8.2 (Consistent Concretization). *An m-object o' is a* consistent concretization *of another m-object o iff*

1. *The top-level of o' is a second-top-level in o: $(\hat{l}_{o'}, \hat{l}_o) \in P_o$*
2. *Each level of o, from $\hat{l}_{o'}$ downwards, is also a level of o': $l \in L_o : (l, \hat{l}_{o'}) \in P_o^* \Rightarrow l \in L_{o'}$ (*level containment*)*
3. *All attributes of o, associated with a level that is shared by o and o', also exist in o', $\{a \in A_o \mid l_o(a) \in L_{o'}\} \subseteq A_{o'}$ (*attribute containment*)*
4. *The relative order of common levels of o and o' is the same: $l, l' \in (L_{o'} \cap L_o) : (l, l') \in P_{o'}^+ \Leftrightarrow (l, l') \in P_o^+$ (*level order compatibility*)*
5. *Levels newly introduced in o' have parents only within o': $\forall (l, l') \in P_{o'} : l \in (L_{o'} \setminus L_o) \Rightarrow l' \in L_{o'}$ (*locality of level order*).*
6. *Common attributes are associated with the same level, have the same domain, and the same value, if defined: For $a \in (A_{o'} \cap A_o)$:*
   (a) *$l_o(a) = l_{o'}(a)$ (*stability of attribute levels*)*
   (b) *$d_o(a) = d_{o'}(a)$ (*stability of attribute domains*)*
   (c) *$v_o(a)$ is defined $\Rightarrow v_o(a) = v_{o'}(a)$ (*compatibility of attribute values*)*

8.2.2 Modeling Hetero-Homogeneous Dimension Hierarchies with M-Objects

We now describe how a homogeneous dimension hierarchy can be modeled by m-objects: (1) The dimension is represented by a hierarchy of m-objects. (2) Each dimension level corresponds to a level of the root m-object. (3) Each level schema is represented by the attributes associated with that level of the root m-object. (4) A dimension instance of some dimension level is represented by an m-object, whose top-level is the dimension-level. (5) Attribute values associated with the top-level of an m-object describe the dimension instance that the m-object represents.

Example 8.3 (Homogeneous Dimension Hierarchies)**.** Consider Figure 8.4 ignoring all relationship symbols. M-objects *Product*, *Time*, and *Location* represent the dimensions of the Dimensional Fact Model depicted in Figure 8.1. M-objects beneath the gray line depict dimension instances.

Additional non-dimensional attributes can be introduced at various levels for the successors of some dimension instance as follows: The m-object representing this dimension instance is extended by attribute definitions at that level; the m-object now serves also as dimension schema for the sub-hierarchy rooted at this dimension instance.

Additional levels can be introduced for the successors of some dimension instance as follows: The m-object representing this dimension instance is extended with additional levels and now serves also as dimension schema for the sub-hierarchy rooted in this dimension instance.

Example 8.4 (Hetero-Homogeneous Dimension Hierarchy)**.** In the dimension hierarchy *product* (see Figure 8.2), m-object *Car* introduces additional attribute *maxSpeed* at level *model* and additional level *brand*.

A data warehouse comprises multiple dimensions. Each dimension D organizes a set of m-objects $O_D \subseteq O$ in a hierarchy H_D, with levels L_D, taken from a universe of levels L, and describes m-objects using attributes A_D, taken from a universe of attributes A. Each m-object, but the root m-object, has one or more parent-m-objects as defined by acyclic relation $H_D : O_D \times O_D$. Let $o, o' \in O_D$, then o' is said to be a *direct concretization of* o or o' *concretizes* o, iff $(o', o) \in H_D$, to be an *indirect concretization of* o iff $(o', o) \in H_D^+$, to be equal to or an indirect concretization of o iff $(o', o) \in H_D^*$. H_D^+ and H_D^* denote the transitive and transitive-reflexive closure, resp., of H_D.

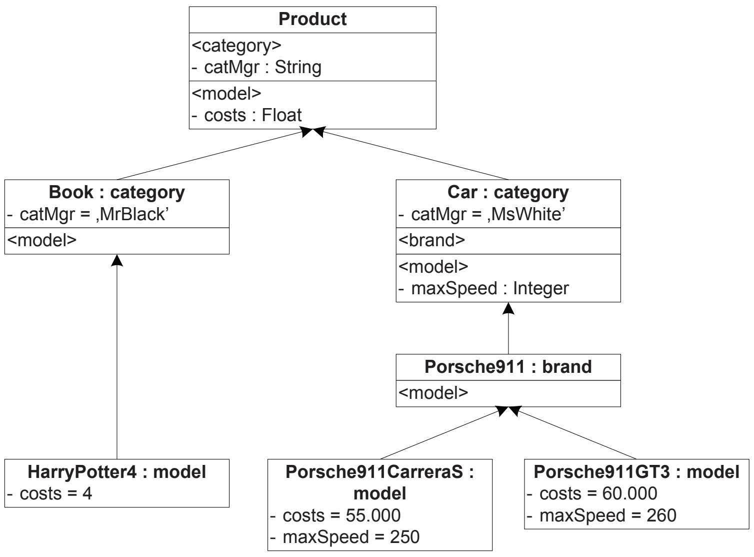


Figure 8.2: Hierarchy of m-objects representing hetero-homogeneous dimension hierarchy *product*. Attributes are only shown at m-objects where they are introduced or instantiated

In case of multiple concretization, stemming from level hierarchies that are not in a total but only in a partial order (see Figure 8.3), we avoid conflicts due to 'multiple inheritance' by ensuring that each attribute and each level is inducted at exactly one m-object. We only consider dimensions with such concretizations of m-objects to be consistent.

Definition 8.3 (Consistent Dimension). *A dimension $D = (O_D, A_D, L_D, H_D)$ is* consistent*, iff*

1. *Each $o \in O_D$ is an m-object according to Definition 8.1.*

2. *For each pair of m-objects $(o', o) \in H_D$, o' is a consistent concretization of o according to Definition 8.2.*

3. *Each attribute and level is introduced at only one m-object:*

   (a) $a \in (A_o \cap A_{o'}) : \exists \bar{o} \in O : (o, \bar{o}) \in H_D^* \wedge (o', \bar{o}) \in H_D^* \wedge a \in A_{\bar{o}}$ *(*unique induction rule for attributes*)* *(b)* $l \in (L_o \cap L_{o'}) : \exists \bar{o} \in O : (o, \bar{o}) \in H_D^* \wedge (o', \bar{o}) \in H_D^* \wedge l \in L_{\bar{o}}$ *(*unique induction rule for levels*)*

4. *If an m-object o' with top-level l is a direct or indirect concretization of m-object o where $(l, l') \in P_o$ then o' must concretize an m-object $\hat{o}$ with top-level l'.*

5. *An m-object o may not directly or indirectly concretize two m-objects o', o'' that are at the same level, i.e., $(o, o') \in H^* \wedge (o, o'') \in H^* \Rightarrow \hat{l}_{o'} \neq \hat{l}_{o''}$ (*unique level predecessor*)*

Levels in a dimension, L_D, are implicitly partially ordered. This follows from the unique induction rule for levels and level order compatibility. We say, $l' \in L_D$ is a descendant of $l \in L_D$, written as $l' \prec l$, if there is an m-object $o \in O_D$ in which l' is a descendant of l. We write $l' \preceq l$ to denote that l' is either descendant of or equal to l. Also note that $\prec$ and $\preceq$ are transitive, i.e.: $\forall l', l \in L_D : (\exists o \in O_D : (l', l) \in P_o^*) \vee (\exists l'' \in L_D : l' \preceq l'' \wedge l'' \preceq l) \Rightarrow l' \preceq l$.

Example 8.5 (Consistent Hetero-Homogeneous Dimension Hierarchy)**.** Consider dimension hierarchy *location* in Figure 8.3. M-object *Lausanne* is an indirect concretization of m-object *location* via kanton *Vaud* and country *Switzerland*. As level *region* is also a parent level of level *city* in m-object *Location*, *Lausanne* must also concretize an m-object at level *region*. This is with m-objects *Alps* the case.

8.3 Hetero-Homogeneous Cubes

In this section we revisit and extend definitions of m-relationships and, then, we show how to model hetero-homogeneous cubes with them.

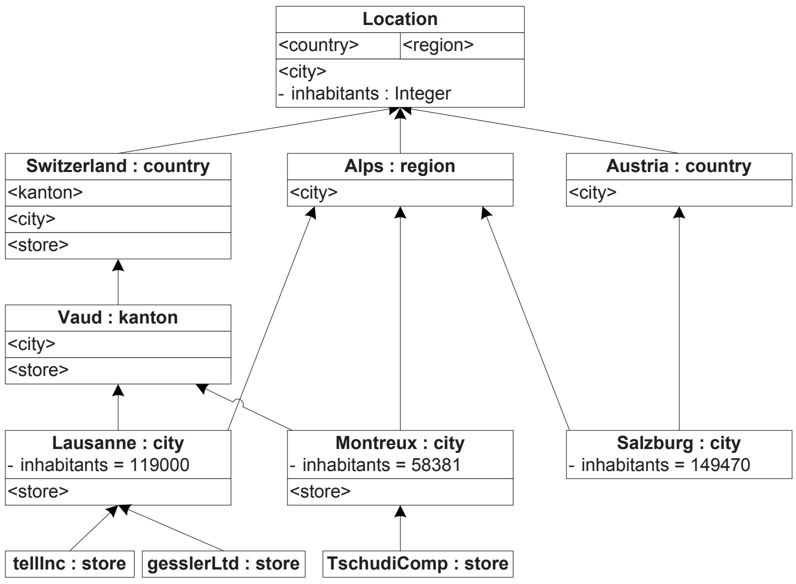


Figure 8.3: Hetero-homogeneous dimension hierarchy *location* with multiple concretization

8.3.1 M-Relationships revisited

M-relationships as described in previous chapters are analogous to m-objects in that they describe relationships between m-objects at multiple levels of abstraction. They have the following features: (1) M-relationships at different abstraction levels can be arranged in concretization hierarchies, similar to m-objects. (2) An m-relationship represents different abstraction levels of a relationship, namely one relationship occurrence and multiple relationship classes. Such a relationship class collects all descending m-relationships that connect m-objects at the respective levels. (3) An m-relationship implies extensional constraints for its concretizations at multiple levels. (4) M-relationships can cope with heterogenous hierarchies and (5) m-relationships can be exploited for querying and navigating.

While our original approach (see Chapter 4) considered only binary m-relationships without relationship attributes, the revised definition below covers for n-ary m-relationships that are described by attributes. Taking into account the data warehouse context the attributes are measures, have an associated aggregation function, and a connection-level indicating at which detail measure values are provided.

Definition 8.4 (M-Relationship)**.** *An m-relationship* $r = (o_1, ..., o_n; M, b, u, f, v)$ *between m-objects* $o_1, ..., o_n$*, its coordinate (denoted also by* coord(r)*), is described by a set of measures* M*. Its top-connection-level* $\hat{c}_r$ *is implicitly given by the top-levels of the referenced m-objects, i.e.,* $\hat{c}_r := (\hat{l}_{o_1}, ..., \hat{l}_{o_n})$*. Each measure* $m \in M$ *is described by*

1. *a connection-level, as defined by total function* $b : M \to (L_{o_1} \times ... \times L_{o_n})$
2. *a unit of measure, as defined by total function* $u : M \to U$*, where* U *is a universe of measure units.*
3. *a distributive aggregation function, as defined by total function* $f : M \to \{\textsc{Sum}, \textsc{Max}, \textsc{Min}\}$*.*
4. *an asserted value (primary fact), as defined by partial function* $v : M \to V$*. A measure* $m \in M$ *has an asserted value iff the connection-level of* m *is equivalent to the top-connection-level of* r*, i.e.:* $v(m)$ *is defined* $\Leftrightarrow b(m) = \hat{c}_r$*.*

When talking about different m-relationships, e.g. r and r', we alternatively use subscripts (e.g., M_r and $M_{r'}$) or quotes (e.g. M, b, being features of r and M', b' being features of r') to denote the context of sets and functions.

Definition 8.5 (Measure Units and Measure Types)**.** *Each measure unit* $u \in U$ *is member of one measure type* $t \in T$*, where* T *is a universe of measure types, as defined by total function* $type : U \to T$*.*

Example 8.6 (M-Relationship)**.** Consider m-relationship *sales* in Figure 8.4 between m-objects *Product*, *Time*, and *Location*. It defines measure *revenue* at connection-level ⟨*model,month,city*⟩ with unit of measure € and aggregation function Sum.

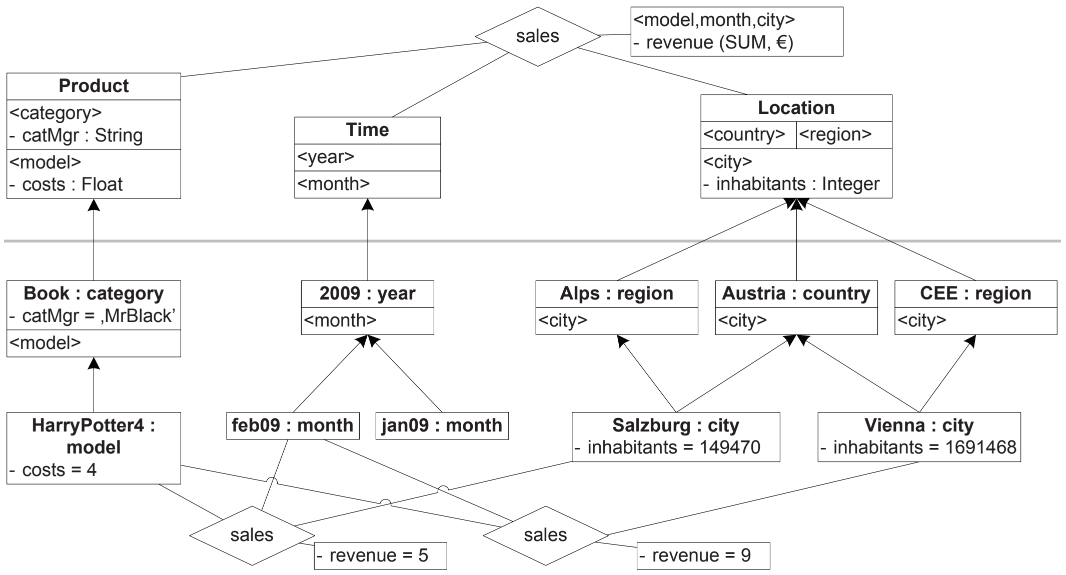


Figure 8.4: Homogeneous data warehouse modeled with m-objects and m-relationships

To discuss concretization of m-relationships we need the notions of *partial order of connection-levels* and *partial order of coordinates*. In the context of data warehouses, connection-levels are synonymously referred to as *cube coordinates*; *coordinates* are also referred to as *cell coordinates*. The connection-level (or cube coordinate) of a cell $(o_1, \ldots, o_n)$, denoted as $\hat{c}_{(o_1,\ldots,o_n)}$ is defined as $(\hat{l}_{o_1}, \ldots, \hat{l}_{o_n})$.

Definition 8.6 (Partial Order of connection-levels)**.** *Given a coordinate $(o_1, ..., o_n)$ and the levels of the m-objects of that coordinate, $L_{o_1}, ..., L_{o_n}$, and two connection-levels $(l'_1, ..., l'_n), (l_1, ..., l_n) \in (L_{o_1} \times ... \times L_{o_n})$. We say $(l'_1, ..., l'_n)$ is a* descendant *of $(l_1, ..., l_n)$, written as $(l'_1, ..., l'_n) \preceq (l_1, ..., l_n)$, iff for i=1..n each level l'_i is a descendant of l_i, i.e.,*
$(l'_1, ..., l'_n) \preceq (l_1, ..., l_n) \Leftrightarrow l'_1 \preceq l_1 \wedge ... \wedge l'_n \preceq l_n$.

Definition 8.7 (Partial Order of Cell Coordinates)**.** *Given n dimensions, $D1, ..., Dn$, of n disjoint sets of m-objects, $O_{D1}, ..., O_{Dn}$). We say coordinate $(o'_1, ..., o'_n) \in (O_{D1} \times ... \times O_{Dn})$ is a* descendant of or equal to *coordinate $(o_1, ..., o_n) \in (O_{D1} \times ... \times O_{Dn})$, written as $(o'_1, ..., o'_n) \preceq (o_1, ..., o_n)$, iff $\forall_{i=1}^{n}$: $(o'_i, o_i) \in H^*_{Di}$. In this case we also speak of a* sub-coordinate*.*

Coordinate $(o'_1, ..., o'_n)$ *is a* descendant of - *or* proper sub-coordinate *of* - *coordinate* $(o_1, ..., o_n)$*, written as* $(o'_1, ..., o'_n) \prec (o_1, ..., o_n)$*, iff for all dimensions i=1..n,* o'_i *is a descendant of or is equal to* o_i*, and for at least one dimension* j*,* o'_j *is a concretization of* o_j*, i.e.,* $\forall_{i=1}^{n} : (o'_i, o_i) \in H^*_{Di} \wedge \exists_{j=1}^{n} : (o'_j, o_j) \in H^+_{Dj}$.

Coordinate $(o'_1, ..., o'_n)$ overlaps with *coordinate* $(o_1, ..., o_n)$*, written as* $(o'_1, ..., o'_n) \between (o_1, ..., o_n)$*, iff they have some (sub-)coordinates in common, that is, for all dimensions,* $i = 1..n$*, the respective dimension m-objects* o_i, o'_i *are either equal or in a concretization relationship:* $(o'_1, ..., o'_n) \between (o_1, ..., o_n) \Leftrightarrow (\forall_{i=1}^{n} : (o'_i, o_i) \in H^*_{Di} \vee (o_i, o'_i) \in H^*_{Di})$

An m-relationship is concretized by substituting one or more of the m-objects in its coordinate by descendant m-objects. The descendant m-relationship must provide values for the measures at its top-connection-level and may add measures, and move the connection-level of a measure to a more specific connection-level.

Definition 8.8 (Consistent Concretization of M-Relationships)**.** *An m-relationship* $r' = (o'_1, ..., o'_n; M', b', u', f', v') \in R$ *is a consistent concretization of another m-relationship* $r = (o_1, ..., o_n; M, b, u, f, v) \in R$*, iff*

1. $(o'_1, ..., o'_n) \prec (o_1, ..., o_n)$
2. *every measure* m *of* r*,* $m \in M$*, with a base-level that is below or equal to the top-level of* r'*, is also a measure of* r'*, every other measure of* r *is not a measure of* r' (measure containment)*:*
   $\{m \in M \mid b(m) \preceq \hat{c}_{r'}\} \subseteq M'$
   $\{m \in M \mid b(m) \not\preceq \hat{c}_{r'}\} \cap M' = \emptyset$
3. *for each measure* m *shared by* r *and* r'*, the base-level of* m *at* r' *is the same or below the base-level of* m *at* r*:* $\forall m \in (M \cap M') : b'(m) \preceq b(m)$ (assured granularity)
4. *Common measures are associated with measure units of the same measure type and the same aggregation function: For* $m \in (M \cap M')$*:*

(a) $type(u'(m)) = type(u(m))$ (stability of measure types)

(b) $f'(m) = f(m)$ (stability of aggregation functions)

Example 8.7 (Concretization of M-Relationships)**.** M-relationship *sales* between m-objects *HarryPotter4*, *feb09*, and *Salzburg* concretizes *sales* between m-objects *Product*, *Time*, *Location*. Its top-connection-level is ⟨*model,month,city*⟩, thus it defines a value for measure *revenue*. An example for introducing additional levels and moving measures to more specific connection-levels will be given later.

To denote that m-relationship $r' \in R$ is a concretization of $r \in R$ we write $r' \prec r$. To denote that r' is a concretization of or equal to r, we write $r' \preceq r$.

8.3.2 Modeling Hetero-Homogeneous M-Cube-Hierarchies with M-Relationships

We first describe how a homogeneous cube of n dimensions can be modeled by m-relationships: (1) A cube is represented by a concretization hierarchy of n-ary m-relationships. (2) The root m-relationship connects the root m-objects of these n dimensions. (3) The root m-relationship has measures associated with a single connection-level which consists of the bottom levels of these n dimensions and gives the measures of the cube. (4) The cells or facts of the cube are represented by m-relationships that concretize the root m-relationship and connect n m-objects that are at the connection-level for which the measures of the root m-relationship are defined and give values for these measures.

Example 8.8 (Homogeneous Cube)**.** Figure 8.4 depicts a homogeneous cube schema *sales* (above the gray horizontal line) and its facts (below the gray line). The cube schema corresponds to the Dimensional Fact Model depicted in 8.1. The cube extension has two facts. Note, while the m-cube approach

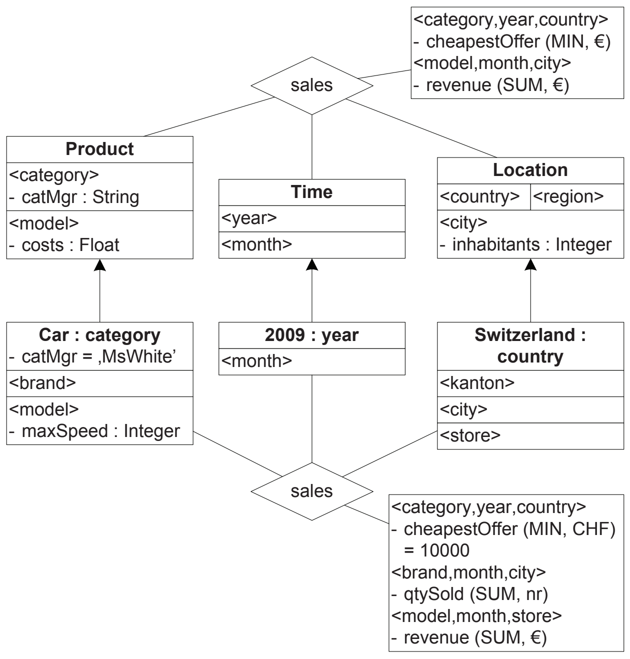


Figure 8.5: Concretization of m-relationship sales, also representing a hetero-homogeneous m-cube

provides a coherent model both for cube- and dimension-schemas as well as their instances, its graphical representation is obviously not meant to be used to fully model a cube with all its facts and dimension instances; it is rather used, analogously to object diagrams in UML, to model exemplary dimension instances and facts together with dimension and cube schemas.

We now describe how a hetero-homogeneous cube of n dimensions can be modeled by m-relationships: Cubes can be hetero-homogeneous in that (1) sub-cubes have *additional measures*, (2) different sub-cubes may give *different units* for the same measure (3) various measures are provided at *mixed granularities*(4) different sub-cubes may provide the same measure at *different granularities* (see examples given in Section 8.1).

Additional Measures can be introduced at a sub-cube identified by a coordinate $(o_1, ..., o_n)$ as follows: An m-relationship for this coordinate is introduced. This m-relationship defines a measure for the connection-level at which values for the measure are provided.

Different sub-cubes with *different units* for the same measure are supported as follows: An m-relationship for the coordinates of each sub-cube is introduced and gives a different unit of measure.

Cubes with measures that are provided at *mixed granularities* can be represented as follows: An m-relationship is introduced that associates these measures with different connection-levels.

Cubes in which different sub-cubes provide the same measure at *different granularities* are represented as follows: An m-relationship is introduced for the cube and provides a measure at some connection-level. For each sub-cube that provides this measure at a more detailed granularity, an m-relationship is introduced and associates this measure with a more specific connection-level.

Example 8.9 (Hetero-Homogeneous Cubes)**.** Figure 8.5 depicts a fragment of a hetero-homogeneous cube. - Note, this example is different from previous ones for sake of presentation and simplicity. - M-relationship *sales* between m-objects *Product*, *Time*, and *Location* introduces two measures at mixed granularities. Measure *cheapestOffer* is defined for connection-level ⟨*category,year,country*⟩ and measure *revenue* for connection-level ⟨*model,month,city*⟩. M-relationship *sales* between category *car*, year *2009*, and country *Switzerland* concretizes the above m-relationship as follows: (1) It introduces an additional measure *qtySold* for connection-level ⟨*model,month,city*⟩. (2) It moves measure *revenue* from connection-level ⟨*model,month,city*⟩ to ⟨*model,month,store*⟩. Thus it provides for different granularity of measure *revenue*: the cube will have stored revenue values for models of cars, months in 2009, and stores in Switzerland, but not for other product categories, months in other years, stores in other countries. (3) It provides a different unit of measure for *cheapestOffer*, that is Swiss francs instead of €.

The notion of a multi-level cube (m-cube), as defined below, generalizes the cube in the Dimensional Fact Model [31].

Definition 8.9 (Multi-Level Cube)**.** *A multi-level cube* $C = (D_1, ..., D_n; S, R)$ *connects* n *dimensions,* $D_1, ..., D_n$*. Its root coordinate* S *is identified by a tuple* $(o_1, ..., o_n) \in O_{D_1} \times ... \times O_{D_n}$*.* R *is a set of m-relationships which represent the measure schema and the base facts of* C*. The set of cell coordinates* X *of m-cube* C *is defined as* $X := \{(o_1, \ldots, o_n) \in O_{D_1} \times ... \times O_{D_n} \mid (o_1, \ldots, o_n) \preceq S\}$*.*

The m-relationships of C that provide a measure-value are called the base facts or base cells of C. The set of measures which are introduced somewhere in m-cube C, denoted as M_C, are defined as $M_C := \bigcup_{r \in R_C} M_r$.

When talking about different m-cubes, e.g. C and C', we alternatively use subscripts (e.g., X_C, S_C) or quotes (e.g. D_1, S, being features of C and D'_1, S' being features of C') to denote the context of sets and functions. Whenever the context is clear we use unquoted variables (e.g. D_1, S).

We now define consistency criteria that avoid conflicts due to multiple inheritance and avoid overlapping facts. For this definition we use the set $\hat{R}_r$ of *directly subsuming m-relationships* of an m-relationship r, $\hat{R}_r := \{r' \in R \mid r \preceq r' \wedge \nexists r'' \in R : r \preceq r'' \prec r'\}$.

Definition 8.10 (Consistent M-Cube)**.** *A multi-level cube* $C = (D_1, ..., D_n; S, R)$ *with root coordinate* $S = (o_1, ..., o_n)$ *is consistent iff*

1. *there is one m-relationship in* R *that corresponds to root coordinate* S*.*
2. *for each cell coordinate* $x \in X$ *there is at most one corresponding m-relationship in* R*.*
3. *For each pair of m-relationships* $r, r' \in R$*, if* r' *is a concretization of* r*,* $r' \preceq r$*, then* r' *is a consistent concretization of* r *according to Def. 8.8.*

4. *Each measure is introduced at only one m-relationship:* $\forall r, r' \in R : \exists m \in \{M_r \cap M_{r'}\} \Rightarrow \exists r'' \in R : m \in M_{r''} \wedge \mathit{coord}(r) \preceq \mathit{coord}(r'') \wedge \mathit{coord}(r') \preceq \mathit{coord}(r'')$ (unique induction rule for measures)

5. *For each measure m shared by two overlapping m-relationships r and r', $\mathit{coord}(r) \between \mathit{coord}(r')$, if r defines a value for m than r' must not define a value for m: $v_r(m)$is defined $\Rightarrow$ $v_{r'}$is not defined.* (unique assertion of values)

6. *For each non-empty cell coordinate $x \in X$ (a cell coordinate is non-empty if there is at least one m-relationship beneath it), for each pair r, r' of direct subsuming m-relationships of x that contain a measure m with base-level below or equal to the connection-level of x, denoted as $\hat{c}_x$, the measure unit and the base level for m are the same at r and r':*
$\forall x \in X, \forall r, r' \in \hat{R}_x, \forall m \in M_r \cap M_{r'}$:
$(\exists r'' \in R : \mathit{coord}(r'') \preceq x) \wedge (b_r(m) \preceq \hat{c}_x \vee b_{r'}(m) \preceq \hat{c}_x)) \Rightarrow$

    (a) $u_r(m) = u_{r'}(m)$ (unit conflict avoidance)

    (b) $b_r(m) = b_{r'}(m)$ (base level conflict avoidance)

Unit conflict avoidance and base level conflict avoidance (Def. 8.10, item 6) ensure that that no coordinate has conflicting direct subsuming m-relationships. Possible conflicts due to multi-dimensional concretization are solved explicitly by an m-relationship directly beneath the conflicting m-relationships.

An m-cube represents hetero-homogeneous base facts as possibly extracted and loaded from various source OLTP databases. An m-cube defined with root coordinate $(o_1, ..., o_n)$ implicitly also represents a cube of cubes. This cube of cubes consists of a set of homogeneous cubes, one for each n-tuple of levels in the cartesian product of the levels of the m-objects of the root coordinate. The cells of such a cube are given by the cartesian product of m-objects at those levels.

Example 8.10. Figure 8.6 depicts the homogeneous cubes of m-cube *sales*; Figure 8.7 shows sample measure values at coordinates of these cubes whereby we ignore dimension *time* for simplicity.

A hetero-homogeneous cube exists for each sub-coordinate and consists of those m-relationships of the given cube that are descendants of that sub-coordinate.

Example 8.11. Sub-m-cube *sales(Car, Time, Switzerland)* takes a closer look at car sales in Switzerland. Figure 8.8 depicts the homogeneous cubes of this sub-m-cube ignoring dimension *time*. Note, that the dimension levels identifying these cubes are not shown. Additional cubes become available for the additional dimension levels *kanton* and *store* defined for country *Switzerland* (see Figure 8.3) and additional level *brand* defined for category *car*. Further, additional measure *qtySold* is available for the cubes at connection-level ⟨*brand,city*⟩ and above, since this measure has been defined for cars in Switzerland for this level (see Figure 8.5). Note that for descendant connection-levels the cubes show a null value for this measure.

The aggregate cell (or fact) has the root coordinate of the (sub-)m-cube and a value for each measure that is provided for this coordinate or can be calculated from the base cells of the m-cube.

Example 8.12. The top-left entry in Figure 8.7 and the top-left entry in Figure 8.8 represent the aggregate cells of coordinates *(product,location)* and *(car,switzerland)* respectively.

Thus, a multi-level cube implicitly describes all (roll-up) cubes that can be derived from a base cube and its dimensions.

In the subsequent section we introduce an m-cube algebra whose operations can be used to extract the aggregate cell, a homogeneous cube, or a hetero-homogeneous cube from an m-cube. There we also describe how the measure values of these cubes and the aggregate cell (fact) are determined.

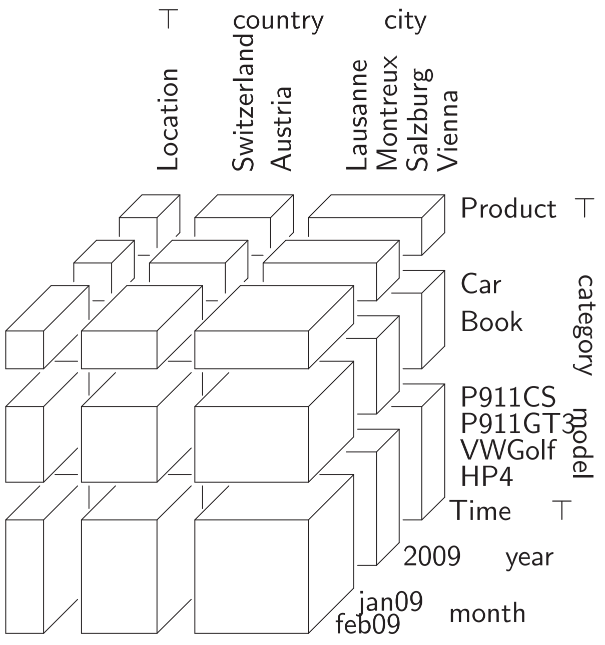


Figure 8.6: Visualization of homogeneous cubes derived from dimensions *product*, *time*, and *location* of m-cube *sales*. For simplicity, dimension level *region* is not shown

8.4 Querying M-Cubes

In this section we introduce an algebra for multi-level cubes. There are three types of operators

1. closed m-cube operators (*dice* δ, *slice* σ, *import-union* $\cup_i$, *projection* π) apply to m-cubes and produce m-cubes as result

2. the *fact extraction* operator φ applies to an m-cube and extracts all measure values of a given cell into a relation with a single tuple

3. the (roll-up) *cube* operator κ applies to an m-cube and produces a homogeneous roll-up cube (a relation with primary and/or aggregated facts) as result, which allows to apply traditional cube operations and facilitates integration with current data warehouse technology

Some common cube-operations like *roll-up* and *drill-down* are not part of the algebra but are defined as mappings from one κ-application to another κ-application.

Based on this m-cube algebra we propose a two-stage approach to analyze data in hetero-homogeneous m-cubes, (1) *selecting a (sub-)m-cube* and (2) specifying the *query* based on the schema of the selected (sub-)m-cube. Queries consistent with the schema of the (sub-)m-cube return homogeneous and correct answers. Note that a sub-m-cube typically has a richer schema than a more general m-cube (as exemplified in Figure 8.8).

The *query* consists of (i) optionally a set of boolean *predicates* to narrow the analysis on cells whose m-objects fulfil the predicate (corresponds to operation *slice* in Def. 8.15) (ii) optionally a set of *measures of interest* (corresponds to operation *projection* in Def. 8.12), (iii) optionally a *measure unit* for each measure and (iv) a *cell coordinate* to retrieve facts of a single cell, or a *cube coordinate* to retrieve facts of all cells within the specified cube (corresponds to operations *fact extraction* in Def. 8.20, and *cube extraction* in Def. 8.22, respectively). If not specified explicitly all available measures are considered and values are converted to the measure unit specified at the specified (sub-)m-cube (see Figure 8.9 for an example query and its results).

8.4.1 Closed M-Cube Operations

The *dice*-operator selects a sub-m-cube from an m-cube.

Definition 8.11 (Dice δ). *Given an input m-cube* $C = (D_1, ..., D_n, S, R)$, *coordinate* $(o_1, ..., o_n)$, *and that there is an m-relationship* $r = (o_1, ..., o_n, M, b, u, f, v) \in R$, *then* $\delta_{o_1,...,o_n} C$ *results in output-cube* $C' = (D_1, ..., D_n, S', R')$ *with*
$S' = (o_1, ..., o_n)$
$R' = \{r' \in R \mid coord(r') \preceq (o_1, ..., o_n)\}$

		$\top$	country		city			
		Location	Switzerland	Austria	Lausanne	Montreux	Vienna	Salzburg
$\top$	Product	20	13	7	6	7	7	4
category	Car	13	10	3	5	5	3	4
	Book	7	3	4	1	2	4	4
	P911CS	7	5	2	3	2	2	4
model	P911GT3	7	5	2	3	2	2	4
	VWGolfXY	6	5	1	2	3	1	4
	HP4	7	3	4	1	2	4	4

Figure 8.7: Homogeneous cubes of a sample m-cube *sales* showing values for measure *revenue* as defined by m-relationship *sales* between *Product* and *Location*, ignoring dimension *Time* and level *region*.

Example 8.13. Dice operation $\delta_{(Car,2009,Switzerland)}sales$ retrieves a sub-m-cube *car09SalesCHaux* containing m-relationships with coordinates that are descendants of *(Car,2009,Switzerland)*.

The projection operator applied on an m-cube, returns an m-cube with a reduced set of measures.

Definition 8.12 (Projection π). *Given an input m-cube $C = (D_1, ..., D_n, S, R)$, and a set of measures $\mathcal{M} \subseteq M_C$, then $\pi_{\mathcal{M}}C$ results in output-cube $C' = (D_1, ..., D_n, S, R')$, with R' defined as follows: for each $r = (o_1, ..., o_n; M, b, u, f, v) \in R$ there is a $r' = (o_1, ..., o_n; M', b', u', f', v') \in R'$, with $M' := M \cap \mathcal{M}$, and for each $m \in M'$: $b'(m) := b(m)$, $u'(m) := u(m)$, $f'(m) := f(m)$, and $v'(m) := v(m)$.*

Example 8.14. Projection operation $\pi_{revenue,qtySold}car09SalesCHaux$ retrieves an m-cube *car09SalesCH* containing only measures *revenue* and *qtySold*. Figure 8.8 depicts the cube of cubes of this m-cube (without dimension *time*).

	country	kanton	city		store		
	Switzerland	Vaud	Lausanne	Montreux	tellInc	gesslerLtd	TschudiComp
Car	28/17	28/17	18/11	10/6	9/	9/	10/
P911	19/10	19/10	11/**6**	8/**4**	5/	6/	8/
VWGolf	9/7	9/7	7/**5**	2/**2**	4/	3/	2/
P911CS	10/	10/	5/	5/	**3**/	**2**/	**5**/
P911GT3	9/	9/	6/	3/	**2**/	**4**/	**3**/
VWGolfXY	9/	9/	6/	3/	**4**/	**3**/	**2**/

Figure 8.8: Homogeneous cubes of sub-m-cube with root coordinate (*Car*,*Switzerland*) of m-cube *sales* depicting measures *revenue* and *qtySold* where available

	tellInc	gesslerLtd
P911CS	3	2
P911GT3	2	4
VWGolfXY	4	3

Figure 8.9: Homogeneous cube of sales revenue for car sales in 2009 in Switzerland in big cities with cells at level ⟨*model*,*Time*,*store*⟩

As prerequisites for predicates used as selection criteria in slice-operation we redefine the notions of stable upward navigation and class extension.

Definition 8.13 (Upward Navigation)**.** *The ancestor m-object of m-object* $o \in O_D$ *at level* $l \in L_o$*, denoted as* $o[l]$*, is defined by*
$o[l] \overset{\text{def}}{=} o' : (o, o') \in H_D^* \wedge \hat{l}_{o'} = l.$

An m-object represents for each level of direct or indirect descendants the class of descendant m-objects of that level. To refer to the set of m-objects at level l beneath m-object o, we write $o\langle l\rangle$. For example, $car\langle model\rangle$ refers to the set of m-objects at level *model* beneath m-object *Car*.

Definition 8.14 (Class Extension)**.** *The class of m-objects of m-object* $o \in O_D$ *at level* $l \in L_o$*, denoted as* $o\langle l \rangle$*, is defined by*
$o\langle l \rangle \overset{\text{def}}{=} \{o' \mid (o', o) \in H_D^* \wedge \hat{l}_{o'} = l\}$.

A *predicate* is a boolean expression over attributes of a class of m-objects, $o\langle l \rangle$ and of its ancestors (using upward navigation). Note, that predicates could be predefined at m-objects and associated with a level like attributes. Then these predicates could be overwritten in concretizations.

A *slice*-operation on a given m-cube selects all coordinates at a given level that fulfill the given criteria and returns an m-cube with all m-relationships from the given m-cube that are between descendants of the given coordinates, between ancestors of the given coordinates, or are at these coordinates. Dimensions $D_1, ..., D_n$ and root coordinate $S = (o_1, ..., o_n)$ are the same in both the input m-cube and the output m-cube. An outer-slice has the same output but additionally consists of all m-relationships from the input m-cube that are above the cube-level of the selection.

Definition 8.15 (Slice σ)**.** *Given are an input m-cube* $C = (D_1, ..., D_n, S, R)$ *with* $S = (o_1, ..., o_n)$ *and selection predicates* $(p_1, l_1), ..., (p_n, l_n)$*. For* $\sigma_{(p_1,l_1),...,(p_n,l_n)}$ *to be applicable on* C*, there must not be an m-relationship in* R *with an asserted measure value above cube-level* $(l_1, ..., l_n)$*. The slice operation* $\sigma_{(p_1,l_1),...,(p_n,l_n)}C$ *results in output cube* $C' = (D_1, ..., D_n, S, R')$ *where* R' *is given as follows. Let the selected cells be given by* $\bar{X} := \{o \in o_1\langle l_1 \rangle \mid p_1(o)\} \times ... \times \{o \in o_n\langle l_n \rangle \mid p_n(o)\}$*; and let the included m-relationships be given by* $\bar{R} := \{r \in R \mid \exists x \in \bar{X} : r \preceq x\}$*. Then* $R' := \bar{R} \cup \{r \in R \mid \exists \bar{r} \in \bar{R} : \bar{r} \preceq r\}$*.*

Example 8.15 (Slice)**.** Slice-operation $\sigma_{(inhabitants>100000, city)}$*car09SalesCH* selects m-cube *car09SalesCHinBigCities*, which comprises m-relationships representing car sales of 2009 in Switzerland in cities with more than 100000 inhabitants.

Definition 8.16 (Outer Slice $\bar{\sigma}$)**.** *Outer Slice is defined as Slice in Def. 8.15 with the difference that* R' *is defined as follows:*
$R' := \bar{R} \cup \{r \in R \mid (l_1, ..., l_n) \preceq \hat{c}_r\}$

Import Union inserts a cube into an existing cube. It can be seen as a bulk operation for inserting m-relationships. The resulting cube needs to be consistent according to Def. 8.9.

Definition 8.17 (Import Union $\cup_i$)**.** *Given two input cubes, main cube $C = (D_1, ..., D_n, S, R)$ and to-be-imported cube $C' = (D_1, ..., D_n, S', R')$, with $\nexists r \in R : r \preceq S'$ and $S' \preceq S$, then $C \cup_i C'$ results in output cube $C'' = (D_1, ..., D_n, S, R'')$ with $R'' := R \cup R'$.*

8.4.2 Fact and Cube Extraction

Before defining fact and cube extraction operators we need to investigate which measures are available for a given coordinate. A measure at a given coordinate may be provided by an m-relationship of the m-cube, i.e., be an asserted fact, or be derived through application of the aggregation function provided with the measure definitions.

Definition 8.18 (Common Measures at Coordinates)**.** *Given a coordinate $x = (o_1, ..., o_n)$ from a consistent m-cube $C = (D_1, ..., D_n; S, R)$, its set of measures, M_x, is given by the union of measures of its direct subsuming m-relationships $\hat{R}_x$, given that the measure's connection-level is below or equal to the level of x: $M_x :=$*
$\{m \in \bigcup_{r \in \hat{R}_x} M_r \mid \forall r \in \hat{R}_x : b_r(m) \preceq (\hat{l}_{o_1}, ..., \hat{l}_{o_n})\}$

For each measure $m \in M_x$, given one of its direct subsuming m-relationships $r \in \hat{R}_x$ that contains m, $m \in M_r$, the base-level, unit-of measure, and aggregation function are those defined at r:

1. $b_x(m) := b_{r'}(m)$
2. $u_x(m) := u_{r'}(m)$
3. $f_x(m) := f_{r'}(m)$

Conversion between measure units is facilitated by multi-polymorphic function *conv*. It applies, dependent on the pair of source and target measure units, a simple arithmetic expression on the numeric input value to produce an output value. We assume, that there is a conversion expression for each pair of measure units that are members of the same measure type. Context-sensitive unit conversion, e.g. time-dependent currency conversion, is facilitated by extending function *conv* to take dimension objects, i.e. a cell-coordinate, as additional parameters. The extended *conv*-method is multi-polymorphic in the two measure-units and in these dimension-objects. We do not further discuss this extension and refer the interested reader to [103].

Given a source measure unit $u_s \in U$, a target measure unit $u_t \in U$, with $type(u_s) = type(u_t)$, and an input value $v \in V$, operation $conv(u_s, u_t, v)$ returns a value that is the conversion of value v from measure unit u_s to measure unit u_t.

We now define how measure values are derived from asserted facts.

Definition 8.19 (Aggregation of Measures *val*). *Given an m-cube* $C = (D_1, ..., D_n, S, R)$*, a cell* $x = (o_1, ..., o_n)$ *with* $\exists r \in R : r \preceq x$*, measure* $m \in M_x$*, and measure unit* $u \in U$*, with* $type(u) = type(u(m))$*, then the value of measure* m *at coordinate* x *converted to unit* u*,* $val(m, x, u)$*, is calculated by applying aggregation function* $f_x(m)$ *on the set of converted* m*-values of* m*-relationships below or at cell* x*, given by* $R_x := \{r \in R \mid r \preceq x\}$*; or null if this set is empty, i.e.:*

$$val(m, x, u) :=$$
$$\begin{cases} f_x(m)(\bigcup_{r \in R_x} conv(u_r(m), u, v_r(m))) & \exists r \in R_x : (v_r(m) \text{ is defined}) \\ null & otherwise \end{cases}$$

We are now ready to define the fact extraction operator.

Definition 8.20 (Fact Extraction φ). *Given an m-cube* $C = (D_1, ..., D_n, S, R)$*, a cell* $x = (o_1, ..., o_n)$ *with* $\exists r \in R : r \preceq x$*, and a mapping from measures to measure units* $(m_1 \mapsto u_1, ..., m_k \mapsto u_k)$*, then fact extraction operation* $\varphi_{(o_1,...,o_n),(m_1 \mapsto u_1,...,m_k \mapsto u_k)} C$ *returns a relation with* *schema $(D_1, ..., D_n, m_1 : u_1, ..., m_k : u_k)$ and an instance consisting of one tuple $(o_1, ..., o_n, val(m_1, x, u_1), ..., val(m_k, x, u_k))$.*

When leaving out the mapping from measures to measure units, fact extraction results in a relation with all measures that are available at the respective cell and converts each measure to the respective unit of measure defined at this cell (see Def. 8.21).

Definition 8.21 (Fact Extraction Shorthand φ)**.** *Given cell $x = (o_1, ..., o_n)$ with $\exists r \in R : r \preceq x$ with measures $M_x = \{m_1, ..., m_k\}$ and units of measures $u_x = \{m_1 \mapsto u_1, ..., m_k \mapsto u_k\}$, then $\varphi_{o_1,...,o_n} C$ is a shorthand for $\varphi_{(o_1,...,o_n),(m_1 \mapsto u_1,...,m_k \mapsto u_k)} C$.*

A cube extraction operation returns a homogeneous cube, consisting of a tuple for each non-empty cell at a given cube-level.

Definition 8.22 (Cube Extraction κ)**.** *Given an m-cube $C = (D_1, ..., D_n, S, R)$, a cube-level $(l_1, ..., l_n) \in (L_{D_1}, ..., L_{D_n})$, and a mapping from measures to measure units $(m_1 \mapsto u_1, ..., m_k \mapsto u_k)$.*

The set of non-empty cells of C at level $(l_1, ..., l_n)$, denoted as $C\langle l_1, ..., l_n \rangle$, is given by $\{(o_1, ..., o_n) \in X \mid (\exists r \in R : r \preceq (o_1, ..., o_n)) \wedge \hat{l}_{o_1} = l_1 \wedge ... \wedge \hat{l}_{o_1} = l_1\}$

The result of cube extraction operation $\kappa_{(l_1,...,l_n),(m_1 \mapsto u_1,...,m_k \mapsto u_k)} C$ is the relation given by union of facts of all non-empty cells at level $(l_1, ..., l_n)$:
$\kappa_{(l_1,...,l_n),(m_1 \mapsto u_1,...,m_k \mapsto u_k)} C :=$
$\bigcup_{x \in C\langle l_1,...,l_n \rangle} (\varphi_{x,(m_1 \mapsto u_1,...,m_k \mapsto u_k)} C)$

Example 8.16. Given our m-cube *car09SalesCHinBigCities* of car sales, the homogeneous cube with measure *revenue* of sales rolled up to level *model,store* can be extracted by applying projection and subsequent cube extraction operators, e.g., $\kappa_{(model,store),(revenue \mapsto €)} \pi_{revenue}$ *car09SalesCHinBigCities*. Figure 8.9 depicts the result of this query as cross table.

In order to retain measures that are available at some but not all cells of a cube, we use outer union [21] on facts extracted according to Def. 8.21. Note that we accept null values and heterogenous measure units in the resulting cube (see Def. 8.23).

Definition 8.23 (Outer Cube Extraction $\bar{\kappa}$). *The result of $\bar{\kappa}_{(l_1,\dots,l_n)}C$ is the relation given by outer union, denoted as $\bar{\bigcup}$, on facts of all non-empty cells, at level $(l_1, ..., l_n)$:*
$\bar{\kappa}_{(l_1,\dots,l_n)}C := \bar{\bigcup}_{x \in C\langle l_1,\dots,l_n\rangle}(\varphi_c C)$

8.5 Related Work

Heterogeneities in data warehouses are widely acknowledged as an important research direction and have received considerable attention in the literature, especially on data warehouse integration [127, 14], summarizability [55], OLAP visualization [75, 22], and conceptual modeling [74]. These works especially discuss heterogeneities in dimension hierarchies, such as non-covering, non-strict, and asymmetric hierarchies. However, to the best of our knowledge, none of these approaches provides for a top-down modeling approach of hetero-homogeneous dimension and cube hierarchies.

Conceptual data warehouse design has attracted a lot of work, various approaches are based on entity-relationship modeling, such as [116], on the UML, such as [129], or on abstract state machines [132]. The well-established Dimensional Fact Model [31] has been used in this chapter as starting point to illustrate homogeneous data warehouse schemas and how hetero-homogeneous hierarchies extend them.

An important area of work concerns summarizability [71, 55] and formal aspects of aggregation in data warehouses [72]. In this context [34] introduce the notions of distributive, algebraic, and holistic aggregation functions. In this chapter we only considered measures based on distributive aggregation functions, a restriction we will relax in future work.

8.6 Summary and Outlook

In this chapter we discussed hetero-homogeneous hierarchies based on m-objects and m-relationships and discussed their application to data warehousing. We revisited structural definitions and consistency criteria of m-objects in order to allow for multiple concretizations. Dimension hierarchies based on extended m-objects can contain heterogeneities with regard to non-dimensional attributes and level hierarchies. We also revisited structural definitions and consistency criteria of m-relationships to allow for n-ary m-relationships described by attributes. Multi-level cubes based on extended m-relationships may contain base facts at mixed granularities, and sub-m-cubes may introduce additional measures, provide measures at finer granularity, and provide measures with differing measure units. We provided a query algebra for online-analytical processing (OLAP) that can cope with these heterogeneities.

Data warehouses with hetero-homogeneous hierarchies can be implemented on top of object-relational database management systems. A *proof-of-concept prototype* has been implemented, under our supervision, as an extension package[1] for Oracle DB. For a detailed description, including usage instructions, implementation details, and extensive examples, we refer the interested reader to Schütz's Master's Thesis [111]. This implementation can also serve as a starting point for implementing a general-purpose multi-level database framework.

We believe that hetero-homogeneous hierarchies are a very promising approach to modeling, implementing, and querying data warehouses with inherent heterogeneities. Interesting issues which we will investigate in the future are:

[1]The extension package for Oracle DB can be found on the department's web page at `http://www.dke.jku.at/research/publications/MT1002_sources.zip`

- *Aggregation operations.* In this chapter we limited the discussion on measures based on distributive aggregation functions Sum, Max, Min. We excluded operation Count due to the lack of a meaningful definition of its semantics in the presence of different and mixed granularities. Future work needs to address peculiarities of aggregation operations in multi-level cubes, in the flavor of [72], especially concerning empty cells, as well as algebraic and holistic aggregation operations.

- *Efficiency.* In this chapter we discussed a conceptual modeling and querying approach and described its implementation as a proof-of-concept prototype. In future work we will investigate how hetero-homogeneous hierarchies can be stored and queried more efficiently.

Chapter 9

Summary and Outlook

Contents

This concluding chapter is structured as follows. In Section 9.1, we evaluate m-objects and m-relationships with regard to comparison criteria introduced in Chapter 1, thus, complementing our comparison of multi-level modeling techniques in Chapter 3. In Section 9.2 we summarize the contributions of this thesis. In Section 9.3 we give an outlook on future work.

9.1 Evaluation

In Chapter 1 we presented a sample problem statement and introduced appropriate comparison criteria for multi-level abstraction techniques. Following these criteria, we evaluated, in Chapter 3, three modeling techniques concerning their suitability for multi-level abstraction. In this section we evaluate m-objects and m-relationships accordingly. The results of our evaluations of the various modeling techniques are summarized in Table 9.1. We wish to stress that this evaluation only addresses the suitability for modeling hetero-homogeneous abstraction hierarchies. It does, however, not evaluate the overall quality of these approaches or their suitability for other problem domains.

1. *Compactness:* M-objects allow modular and redundancy-free modeling by encapsulating all information concerning one domain concept into one m-object. For example, all information concerning domain concept *car* is encapsulated into one m-object, *Car*.

2. *Query flexibility:* Pre-defined qualified identifiers can be used to refer to the set of m-objects at a specific level that belong to a specific m-object. The set of all m-objects that are descendants of a given m-object at some specified level is identified by qualifying the name of the m-object by that level. For example, (1) to refer to all models of product-category *Car* one writes *Car*⟨*model*⟩, (2) to refer to the physical cars of product category *car*: *Car*⟨*physicalEntity*⟩ , and (3) to refer to all physical entities of products: *Product*⟨*physicalEntity*⟩ (see Chapter 6 for details).

3. *Hetero-Homogeneous Level-Hierarchies:* Hierarchies of m-objects are easy to extend by an additional level (e.g., *brand*) in one sub-hierarchy (e.g., the sub-hierarchy rooted by *car*) without affecting other sub-hierarchies (e.g., the sub-hierarchy rooted by *book*). Figure 1.4 shows level *brand* with m-object *car*.Adding this level to cars, but not to books, requires no changes above m-object *car*, nor in sibling hierarchies of m-objects (such as *book*).

4. *Multiple relationship abstractions:* M-relationships between m-objects are like m-objects described at multiple levels, associating one or more pairs of levels of the linked m-objects. M-relationships are interpreted in a multi-faceted way, once as relationship occurrence (or sometimes also called relationship instance or link), and once as relationship class

(or sometimes also called relationship type or association), or also as meta-relationship class. They take on these multiple roles depending on what pairs of linked levels one considers and on whether one considers, the linked m-objects in an instance or a class-role. As m-relationships may also be concretized along the concretization hierarchy of linked m-objects, they support multiple relationship abstractions.

M-Relationships cover (a) *(Meta-)Classification of Relationships*, (b) *Adressability/Queryability of Relationships at different levels*, and (c) *Specialization/Instantiation of Domain and Range*

	Mater.	Deep I.	Powertypes	M-Objects
Compactness	+	+	–	+
Query Flexibility	∼	∼	+	+
Heterogenous Level-Hier.	–	–	+	+
Relationship-Abstraction	∼	∼	–	+

Table 9.1: Evaluation of different multi-level abstraction techniques. Used symbols: '+' (full support), '∼' (limited support), '–' (no support)

9.2 Contributions

The main part of this thesis is dedicated to conceptual structural modeling. Contributions to conceptual modeling can be applied, in general, to conceptual and logical database design, to knowledge representation, as well as to software engineering (model-driven development, programming language design). The main contributions of this thesis are summarized in the following.

We introduced the notion of *hetero-homogeneous abstraction hierarchies* as the core construct of conceptual modeling in the large. In such a hierarchy, each element belongs to a certain abstraction level and can be regarded as abstraction of all its descendants (its sub-hierachy). Each element can specialize the schema of its sub-hierarchy by introducing additional abstraction levels and by specializing the structure of elements at a given abstraction level.

As a prerequisite for modeling hetero-homogeneous abstraction hierarchies, we revisited basic notions of conceptual modeling as well as basic abstraction principles, namely *classification*, *generalization*, and *aggregation*. We provided a *concise conceptual framework* together with a simple graphical notation which solely build on the classical abstraction principles. In particular, we also discussed aggregation, specialization, and meta-classification of relationships (issues that are widely neglected in the conceptual modeling literature).

We showed how *hetero-homogeneous abstraction hierarchies* of objects as well as of relationships can be represented using this *conceptual framework*: Abstract domain concepts (such as *Car*) as well as concrete domain entities (such as *a physical Car*) are represented as objects, residing on the instance level, having different *principles of identity* (at different levels of abstraction or granularity), and being arranged in a specific kind of aggregation hierarchy. Objects at higher levels of abstraction (such as *Car*) additionally have multiple class-facets (such as *CarModel* or *CarPhysicalEntity*) as well as metaclass-facets (such as *PhysicalEntityClassOfCarModel*). We backed up aspects of this approach by relating it to insights from *ontological analysis*, particularly on Guizzardi's analysis of the ontological foundations of conceptual structural models [39]. However, representing hetero-homogeneous abstraction hierarchies solely based on the classical abstraction principles leads to overly-verbose models. Such models consist of a myriad of classes and metaclasses as well as aggregation-, instantiation-, and generalization-relationships between them. We concluded that this is not practicable.

To overcome the issue of overly-verbose models, we introduced a modeling technique that allows to represent a domain concept together with its class- and metaclass-facets in a single model element called *multi-level object* (*m-object*). Analogously, we provided for representing a relationship together with its relationship-class- and relationship-metaclass-facets in a single model element called *multi-level relationship* (*m-relationship*). Arrangement of m-objects and m-relationships in so called *concretization hierarchies* comprises aspects of specialization, meta*-classification, as well as aggregation. We provided a concise formalization of m-objects and m-relationships in terms of set-theoretic definitions of their structure as well as of their consistent arrangement in concretization hierarchies. The modeling technique was complemented by a simple graphical notation.

We introduced formal definitions of *operations for creating and manipulating multi-level models* based on m-objects and m-relationships. These formal definitions can be employed as a blueprint for implementing multi-level databases as well as for implementing multi-level programming languages.

We extensively discussed *querying multi-level models*. We introduced an object-centered query approach which comprises of query operations on m-objects. These operations form an algebra that is closed on sets of m-objects (*m-object algebra*). Additionally, we introduced a relationship-centered query approach, in terms of an algebra that is closed on sets of m-relationships (*m-relationship algebra*). We also showed how these two algebras are combined. This formal query approach, similar to relational algebra, can serve as blueprint for implementing query support on top of multi-level databases.

We introduced the notion of *Active Domain* to multi-level modeling and discussed query support for this construct. An active domain is given by the set of possible relationships at a lower level of abstraction that may possibly be introduced with regard to existing relationships at higher abstraction levels. This is especially important in organization, where different levels of a multi-level model are maintained by different users, having differing user rights.

As a basis for re-using the contributions of this thesis for the *Semantic Web*, we provided a *mapping from m-objects and m-relationships to OWL*, extended by integrity constraints. We admit, however, that the feasibility of this mapping approach may be hampered by the computational complexity of reasoning with such multi-level models. It is open to future work to simplify the approach in order to provide for better computational properties as well as to also provide for the bottom-up modeling style which is prevalent in the Web.

We applied the notion of hetero-homogeneous hierarchies to *Data Warehousing* in order to provide a solution to the long-pending problem of *heterogeneities in dimension hierarchies.* Based on an extended form of m-objects, we introduced *hetero-homogeneous dimension hierarchies.* In such dimension hierarchies, sub-hierarchies may refine their schema by introducing *additional dimension levels* and *additional non-dimensional attributes.* Our consistency criteria ensure that the schema of each sub-hierarchy complies with the schema of the dimension hierarchy of which is is a fragment of.

Based on an extended form of m-relationships, we also allowed for heterogeneities in data warehouse cubes. In a *hetero-homogeneous multi-level cube*, different measures may be represented at different levels of granularity. Sub-Cubes may —for the facts within that sub-cube— introduce additional measures, refine the level of granularity of measures, and use differing units of measure. We provided formal consistency criteria that ensure that the refined schema of a sub-cube is consistent with regard to the cube it is a fragment of. This global consistency is necessary in order to ensure that queries against the schema of a cube can be evaluated in face of heterogeneities in sub-cubes.

We discussed and formally defined operations for *querying hetero-homogeneous hierarchies in data warehouses.* These operations form an algebra that is closed on multi-level cubes (*m-cube algebra*). This query approach serves as basis for on-line analytical processing (OLAP) with hetero-homogeneous hierarchies in data warehouses.

Based on our formal definitions of hetero-homogeneous dimension hierarchies, of hetero-homogeneous multi-level cubes, of the m-cube algebra, as well as on our implementation concept, an *extension package for Oracle DB* was implemented (as part of a master's thesis [111]) . This proof-of-concept prototype allows to create, maintain, and query Data Warehouses which are comprised of hetero-homogeneous dimension hierarchies and hetero-homogeneous multi-level cubes.

9.3 Outlook

The scope of this thesis is limited to the core constructs of conceptual modeling, namely objects (entities, individuals) and relationships between them, together with their representation at higher levels of abstraction. It remains open to future work to extend this approach with further constructs of conceptual modeling, examples include *relationships between relationships* (as in the higher-order entity relationship model [122]), *key constraints* and *cardinality constraints* [124], further abstraction principles such as *Role/Player* [33, 130], as well as *meta-classification of attributes.* Furthermore, extensions introduced in the context of data warehousing (multiple concretization, n-ary m-relationships, m-relationships with attributes) can be generalized and be integrated in the core m-object/m-relationship approach.

In this thesis, we only considered extensional aspects of metaclass-facets of m-objects and m-relationships. We did not consider typing aspects of metaclasses, such as *member-metatype* (see Chapter 2 for a discussion of the different aspects of metaclasses). We believe, that considering typing aspects of metaclass-facets of m-objects facilitates integration of the m-object approach with current approaches to linguistic metamodeling, like the Meta-Object Facility (MOF) [95], which focus on the *member-metatype* aspect of metaclasses. This remains open for future work.

Future work also needs to analyze the implications of *orthogonal abstraction hierarchies.* We are particularly interested in the *specialization of m-objects/m-relationships* orthogonal to their concretization as well as in the composition of multiple m-relationships and m-objects to *structured* *m-objects*. Such orthogonal abstractions will, presumably, result in multiple interdependent concretization hierarchies. Previous work on combining specialization and structured objects [114, 60, 128], as well as the work of Thalheim and colleagues on component development [120, 123, 126, 100], can serve as a starting point.

Concerning tool support for m-objects and m-relationships, we are currently investigating different directions. First, as part of an ongoing master's thesis [25], an extension to the ontology editor and knowledge-base framework Protégé [90] for multi-level modeling with m-objects and m-relationships is being implemented. As part of another master's thesis [52], an export-plugin for this extension of Protégé is being developed, implementing our mappings from m-objects and m-relationships to OWL as discussed in Chapter 7. Second, also as part of an ongoing master's thesis [49], a multi-level programming framework based on *Ruby* [27] is being developed. Database extensions for managing persistent multi-level models can be implemented in line with our implementation of hetero-homogeneous hierarchies in data warehouses. The implementation of hetero-homogeneous hierarchies in data warehouses is going to be complemented by an SQL-like language as well as a graphical modeling and visualization tool.

The core ideas of this thesis can be applied to modeling of static behavior (methods, object life cycles) in order to provide for multi-level modeling of business processes. Previous work on object-oriented modeling of processes and of their specialization [104, 105, 119, 106] will serve as a starting point.

It remains to future work to apply our core ideas to modeling of active behavior (events/situations, actions) in order to provide for multi-level modeling of business rules. Previous work on object-oriented modeling of active behavior [68, 69, 92] will serve as a starting point.

Our contributions to data warehousing may be employed as a basis for extending *Active Data Warehousing* [79] with multi-level modeling of analysis rules. Previous work on Active Data Warehouses [107, 121, 120] can serve as a starting point.

List of Tables

List of Figures

Bibliography

[1] OWL 2 Web Ontology Language : Document Overview : W3C Recommendation 27 October 2009, October 2009.

[2] Thomas Aschauer, Gerd Dauenhauer, and Wolfgang Pree. Multi-level Modeling for Industrial Automation Systems. In *EUROMICRO-SEAA*, pages 490–496, 2009.

[3] Thomas Aschauer, Gerd Dauenhauer, and Wolfgang Pree. Representation and Traversal of Large Clabject Models. In *MoDELS*, pages 17–31, 2009.

[4] Thomas Aschauer, Gerd Dauenhauer, and Wolfgang Pree. Towards a generic architecture for multi-level modeling. In *WICSA/ECSA*, pages 121–130, 2009.

[5] Colin Atkinson. Meta-Modeling for Distributed Object Environments. In *EDOC*. IEEE Computer Society, 1997.

[6] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Trans. Software Eng.*, 35(6):742–755, 2009.

[7] Colin Atkinson and Thomas Kühne. The Essence of Multilevel Metamodeling. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of the 4th International Conference on the UML 2001, Toronto, Canada*, LNCS 2185, pages 19–33. Springer Verlag, October 2001.

[8] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Sci. Comput. Program.*, 44(1):5–22, 2002.

[9] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[10] Colin Atkinson and Thomas Kühne. Concepts for Comparing Modeling Tool Architectures. In *MoDELS*, pages 398–413, 2005.

[11] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software and System Modeling*, 7(3):345–359, 2008.

[12] Franz Baader. Description Logic Terminology. In Baader et al. [13], pages 485–495.

[13] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[14] Stefan Berger and Michael Schrefl. From Federated Databases to a Federated Data Warehouse System. In *HICSS*, page 394. IEEE Computer Society, 2008.

[15] Tim Berners-Lee, Wendy Hall, James A. Hendler, Kieron O'Hara, Nigel Shadbolt, and Daniel J. Weitzner. A Framework for Web Science. *Foundations and Trends in Web Science*, 1(1):1–130, 2006.

[16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[17] Alexander Borgida and Ronald J. Brachman. Conceptual Modeling with Description Logics. In Baader et al. [13], pages 349–372.

[18] Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the Frame Problem in Procedure Specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[19] Michael L. Brodie. Association: A Database Abstraction for Semantic Modelling. In Peter P. Chen, editor, *ER*, pages 577–602. North-Holland, 1981.

[20] Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[21] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.

[22] Alfredo Cuzzocrea and Svetlana Mansmann. OLAP Visualization: Models, Issues, and Techniques. In J. Wang, editor, *Encyclopedia of Data Warehousing and Mining, Second Edition.* Information Science Reference, 2009.

[23] Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. Materialization and Its Metaclass Implementation. *IEEE Trans. Knowl. Data Eng.*, 14(5):1078–1094, 2002. 0605.

[24] Jos de Bruijn, Rubén Lara, Axel Polleres, and Dieter Fensel. OWL DL vs. OWL flight: conceptual modeling and reasoning for the semantic Web. In Allan Ellis and Tatsuya Hagino, editors, *WWW*, pages 623–632. ACM, 2005.

[25] Alois Diwold. Multi-Level-Modellierung: Implementierung eines Protégé-Plugins (Working Title). Master's thesis, Institut für Wirtschaftsinformatik - Data & Knowledge Engineering, Johannes Kepler Universität Linz, 2010. (Work in Progress).

[26] Klaus Ettmayer. Ruby on Rails with Roles - Ein verteiltes Rollenmodell für RESTful Webservices. Master's thesis, Institut für Wirtschaftsinformatik - Data & Knowledge Engineering, Johannes Kepler Universität Linz, 2009. (In German. Available online: http://www.dke.uni-linz.ac.at/research/publications/MT1001.pdf).

[27] David Flanagan and Yukihiro Matsumoto. *The ruby programming language.* O'Reilly, 2008.

[28] Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. Bridging MDA and OWL Ontologies. *J. Web Eng.*, 4(2):118–143, 2005.

[29] Ralf Gitzel, Ingo Ott, and Martin Schader. Ontological Extension to the MOF Metamodel as a Basis for Code Generation. *Comput. J.*, 50(1):93–115, 2007.

[30] Robert C. Goldstein and Veda C. Storey. Materialization. *IEEE Trans. Knowl. Data Eng.*, 6(5):835–842, 1994.

[31] Matteo Golfarelli, Dario Maio, and Stefano Rizzi. The Dimensional Fact Model: A Conceptual Model for Data Warehouses. *Int. J. Cooperative Inf. Syst.*, 7(2-3):215–247, 1998.

[32] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software and System Modeling*, 5(1):72–90, 2006.

[33] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Trans. Inf. Syst.*, 14(3):268–296, 1996.

[34] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[35] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *Int. J. Hum.-Comput. Stud.*, 43(5-6):907–928, 1995.

[36] Nicola Guarino. Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human-Computer Studies*, 43(5/6):625–640, 1995.

[37] Nicola Guarino. The Ontological Level: Revisiting 30 Years of Knowledge Representation. In *Conceptual Modeling: Foundations and Applications*, pages 52–67, 2009.

[38] Nicola Guarino and Christopher A. Welty. Evaluating ontological decisions with OntoClean. *Commun. ACM*, 45(2):61–65, 2002.

[39] Giancarlo Guizzardi. *Ontological Foundations for Structural Conceptual Models*. PhD thesis, University of Twente, The Netherlands, 2005.

[40] Giancarlo Guizzardi. On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta)Models. In Olegas Vasilecas, Johann Eder, and Albertas Caplinskas, editors, *DB&IS*, volume 155 of *Frontiers in Artificial Intelligence and Applications*, pages 18–39. IOS Press, 2006.

[41] Giancarlo Guizzardi. The Problem of Transitivity of Part-Whole Relations in Conceptual Modeling Revisited. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *CAiSE*, volume 5565 of *Lecture Notes in Computer Science*, pages 94–109. Springer, 2009.

[42] Matthias Gutheil, Bastian Kennel, and Colin Atkinson. A Systematic Approach to Connectors in a Multi-level Modeling Environment. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 843–857. Springer, 2008.

[43] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[44] Henderson-Sellers and Gonzalez-Perez. Connecting Powertypes and Stereotypes. *Journal of Object Technology*, 4:83–96, 2005.

[45] Carlos A. Heuser and Günther Pernul, editors. *Advances in Conceptual Modeling - Challenging Perspectives, ER 2009 Workshops CoMoL,*

ETheCoM, FP-UML, MOST-ONISW, QoIS, RIGiM, SeCoGIS, Gramado, Brazil, November 9-12, 2009. Proceedings, volume 5833 of *Lecture Notes in Computer Science*. Springer, 2009.

[46] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1), 2004.

[47] Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, and Werner Retschitzegger. *UML@Work*. dpunkt.verlag, 3rd edition edition, 2005. PR.MS.221.

[48] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979.

[49] Christian Horner. Multi-Level Programmierung mit M-Objects in Ruby (Preliminary Version). Master's thesis, Institut für Wirtschaftsinformatik - Data & Knowledge Engineering, Johannes Kepler Universität Linz, 2010.

[50] Ian Horrocks. Ontologies and the semantic web. *Commun. ACM*, 51(12):58–67, 2008.

[51] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The Even More Irresistible SROIQ. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 57–67. AAAI Press, 2006.

[52] Joachim Huber. Multi-Level Modellierung und OWL: Implementierung von Mapping und Werkzeug-Unterstützung (Preliminary Version). Master's thesis, Institut für Wirtschaftsinformatik - Data & Knowledge Engineering, Johannes Kepler Universität Linz, 2010.

[53] Michael Huemer. Rollen im Web of Data: Analyse, RDF-basiertes Rollenmodell und zugehöriger Data Browser. Master's thesis, Institut für Wirtschaftsinformatik - Data & Knowledge Engineering, Johannes Kepler Universität Linz, 2009. (In German. Available online: http://www.dke.uni-linz.ac.at/research/publications/MT0903.pdf).

[54] Richard Hull and Roger King. Semantic database modeling: survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.

[55] Carlos A. Hurtado and Alberto O. Mendelzon. Reasoning about Summarizability in Heterogeneous Multidimensional Schemas. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 375–389. Springer, 2001.

[56] Stefan Jablonski, Roland Kaschek, and Bernhard Thalheim. Preface to CoMoL 2009. In Heuser and Pernul [45], page 1.

[57] Matthias Jarke, Stefan Ehrer, Rainer Gallersdörfer, Manfred A. Jeusfeld, and Martin Staudt. ConceptBase - A Deductive Object Base Manager. Technical report, Informatik V, RWTH Aachen, 1993.

[58] Matthias Jarke, Rainer Gallersdörfer, Manfred A. Jeusfeld, and Martin Staudt. ConceptBase - A Deductive Object Base for Meta Data Management. *J. Intell. Inf. Syst.*, 4(2):167–192, 1995.

[59] Manfred A. Jeusfeld. Metamodeling and Method Engineering with ConceptBase. In Manfred A. Jeusfeld, Mathias Jarke, and John Mylopoulos, editors, *Metamodeling for Method Engineering.* MIT Press, 2009.

[60] Gerti Kappel and Michael Schrefl. Local Referential Integrity. In Günther Pernul and A. Min Tjoa, editors, *Entity-Relationship Approach - ER'92, 11th International Conference on the Entity-Relationship Approach, Karlsruhe, Germany, October 7-9, 1992, Proceedings*, volume 645 of *Lecture Notes in Computer Science*, pages 41–61. Springer, 1992.

[61] Stuart Kent. Model Driven Engineering. In *IFM*, pages 286–298, 2002.

[62] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *J. ACM*, 42(4):741–843, 1995.

[63] Wolfgang Klas and Michael Schrefl. *Metaclasses and Their Application - Data Model Tailoring and Database Integration*. Springer, 1995.

[64] Thomas Kühne. What is a Model? In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/23> [date of citation: 2005-01-01].

[65] Thomas Kühne. Contrasting Classification with Generalisation. In *APCCM*, pages 71–78, 2009.

[66] Thomas Kühne and Daniel Schreiber. Can programming be liberated from the two-level style: multi-level programming with deepjava. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 229–244. ACM, 2007.

[67] Thomas Kühne and Friedrich Steimann. Tiefe Charakterisierung. In Bernhard Rumpe and Wolfgang Hesse, editors, *Modellierung*, volume 45 of *LNI*, pages 109–119. GI, 2004.

[68] Peter Lang, Werner Obermair, and Michael Schrefl. Situation Diagrams. In Roland Wagner and Helmut Thoma, editors, *DEXA*, volume 1134 of *Lecture Notes in Computer Science*, pages 400–421. Springer, 1996.

[69] Peter Lang, Werner Obermair, and Michael Schrefl. Modeling Business Rules with Situation/Activation Diagrams. In W. A. Gray and Per-Åke Larson, editors, *ICDE*, pages 455–464. IEEE Computer Society, 1997. ASW.

[70] Holger Lausen, Jos de Bruijn, Axel Polleres, and Dieter Fensel. WSML - a Language Framework for Semantic Web Services. In *Rule Languages for Interoperability*, 2005.

[71] Hans-Joachim Lenz and Arie Shoshani. Summarizability in OLAP and Statistical Data Bases. In *SSDBM*, pages 132–143, 1997.

[72] Hans-Joachim Lenz and Bernhard Thalheim. OLAP Databases and Aggregation Functions. In *SSDBM*, pages 91–100. IEEE Computer Society, 2001.

[73] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009.

[74] Elzbieta Malinowski and Esteban Zimányi. Hierarchies in a multidimensional model: From conceptual modeling to logical representation. *Data Knowl. Eng.*, 59(2):348–377, 2006.

[75] Svetlana Mansmann and Marc H. Scholl. Extending Visual OLAP for Handling Irregular Dimensional Hierarchies. In A. Min Tjoa and Juan Trujillo, editors, *DaWaK*, volume 4081 of *Lecture Notes in Computer Science*, pages 95–105. Springer, 2006.

[76] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251 – 266, 1995.

[77] Ernst Mayr. Biological Classification: Toward a Synthesis of Opposing Methodologies. *Science*, 214:510–516, October 1981.

[78] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language : Overview : W3C Recommendation 10 February 2004, February 2004.

[79] Mukesh K. Mohania, Ullas Nambiar, Michael Schrefl, and Millist W. Vincent. Active and Real-Time Data Warehousing. In Liu and Özsu [73], pages 21–26.

[80] Boris Motik. On the Properties of Metamodeling in OWL. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2005.

[81] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the Gap Between OWL and Relational Databases. In *Proc. of the 16th International World Wide Web Conference (WWW 2007)*, 2007.

[82] Renate Motschnig-Pitrik and Veda C. Storey. Modelling of set Membership: The Notion and the Issues. *Data Knowl. Eng.*, 16(2):147–185, 1995.

[83] John Mylopoulos, Philip A. Bernstein, and Harry K. T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Trans. Database Syst.*, 5(2):185–207, 1980.

[84] John Mylopoulos, Alexander Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing Knowledge About Information Systems. *ACM Trans. Inf. Syst.*, 8(4):325–362, 1990.

[85] John Mylopoulos and Harry K. T. Wong. Some Features of the TAXIS Data Model. In *VLDB*, pages 399–410, 1980.

[86] Bernd Neumayr, Katharina Grün, and Michael Schrefl. Multi-Level Domain Modeling with M-Objects and M-Relationships. In *Sixth Asia-Pacific Conference on Conceptual Modelling (APCCM 2009), Wellington, New Zealand*, 2009.

[87] Bernd Neumayr and Michael Schrefl. Multi-level Conceptual Modeling and OWL. In Heuser and Pernul [45], pages 189–199.

[88] Bernd Neumayr, Michael Schrefl, and Bernhard Thalheim. Hetero-Homogeneous Hierarchies in Data Warehouses. In Sebastian Link and Aditya K. Ghose, editors, *Proceedings of the Seventh Asia-Pacific Conference on Conceptual Modelling (APCCM 2010), Brisbane, Australia,*

January 18-21, 2010. Conferences in Research and Practice in Information Technology, Vol. 110., 2010.

[89] Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology.

[90] Natalya Fridman Noy, Ray W. Fergerson, and Mark A. Musen. The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility. In Rose Dieng and Olivier Corby, editors, *EKAW*, volume 1937 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2000.

[91] James J. Odell. *Advanced Object-Oriented Analysis & Design Using UML (also published as James Odell: Power Types. JOOP 7(2): 8-12 (1994))*, chapter Power Types, pages 23–32. Cambridge University Press, 1998.

[92] A. Olivé and R. Raventós. Modeling events as entities in object-oriented conceptual modeling languages. *Data & Knowledge Engineering*, 58(3):243–262, 2006.

[93] Antoni Olivé. Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In *CAiSE*, pages 1–15, 2005.

[94] Antoni Olivé. *Conceptual Modeling of Information Systems.* Springer, 2007.

[95] OMG. Meta Object Facility Core Specification Version 2.0, January 2006.

[96] OMG. Unified Modeling Language (UML), version 2.1.1, Superstructure Specification, February 2007.

[97] Joan Peckham and Fred Maryanski. Semantic data models. *ACM Comput. Surv.*, 20(3):153–189, 1988.

[98] Alain Pirotte, Esteban Zimányi, David Massart, and Tatiana Yakusheva. Materialization: A Powerful and Ubiquitous Abstraction Pattern. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB*, pages 630–641. Morgan Kaufmann, 1994. 0605.

[99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education, 2004.

[100] Klaus-Dieter Schewe and Bernhard Thalheim. Component-driven engineering of database applications. In Markus Stumptner, Sven Hartmann, and Yasushi Kiyoki, editors, *APCCM*, volume 53 of *CRPIT*, pages 105–114. Australian Computer Society, 2006.

[101] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[102] Marc H. Scholl and Hans-Jörg Schek. Supporting Views in Object-Oriented Databases. *IEEE Data Eng. Bull.*, 14(2):43–47, 1991.

[103] Michael Schrefl, Gerti Kappel, and Peter Lang. Modeling Collaborative Behavior Using Cooperation Contracts. *Data Knowl. Eng.*, 26(2):191–224, 1998.

[104] Michael Schrefl and Markus Stumptner. Behavior Consistent Extension of Object Life Cycles. In *OOER*, pages 133–145, 1995.

[105] Michael Schrefl and Markus Stumptner. Behavior Consistent Refinement of Object Life Cycles. In *ER*, pages 155–168, 1997.

[106] Michael Schrefl and Markus Stumptner. Behavior-consistent specialization of object life cycles. *ACM Trans. Softw. Eng. Methodol.*, 11(1):92–148, 2002. 0605.

[107] Michael Schrefl and Thomas Thalhammer. On Making Data Warehouses Active. In *DaWaK*, pages 34–46, 2000.

[108] Michael Schrefl and Thomas Thalhammer. Using roles in Java. *Softw., Pract. Exper.*, 34(5):449–464, 2004.

[109] Michael Schrefl, A. Min Tjoa, and Roland Wagner. Comparison-Criteria for Semantic Data Models. In *ICDE*, pages 120–125. IEEE Computer Society, 1984.

[110] Randall T. Schuh. *Biological Systematics: Principles and Applications.* Cornell University Press, 2000.

[111] Christoph Schütz. Extending data warehouses with hetero-homogeneous dimension hierarchies and cubes – A proof-of-concept prototype in Oracle. Master's thesis, Johannes Kepler Universität Linz, 2010. http://www.dke.jku.at/research/publications/MT1002.pdf.

[112] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006. 0605.

[113] Herbert A. Simon. *The Sciences of the Artificial.* The Mit Press, third edition edition, 1995.

[114] John Miles Smith and Diane C. P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Trans. Database Syst.*, 2(2):105–133, 1977.

[115] Il-Yeol Song and Peter P. Chen. Entity Relationship Model. In Liu and Özsu [73], pages 1003–1009.

[116] Il-Yeol Song, Ritu Khare, Yuan An, Suan Lee, Sang-Pil Kim, Jinho Kim, and Yang-Sae Moon. SAMSTAR: An Automatic Tool for Generating Star Schemas from an Entity-Relationship Diagram. In *ER*, pages 522–523, 2008.

[117] Herbert Stachowiak. *Allgemeine Modelltheorie.* Springer-Verlag, Wien and New York, 1973.

[118] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.*, 35(1):83–106, 2000.

[119] Markus Stumptner and Michael Schrefl. Behavior Consistent Inheritance in UML. In *ER*, pages 527–542, 2000.

[120] Thomas Thalhammer and Michael Schrefl. Realizing active data warehouses with off-the-shelf database technology. *Softw., Pract. Exper.*, 32(12):1193–1222, 2002.

[121] Thomas Thalhammer, Michael Schrefl, and Mukesh K. Mohania. Active data warehouses: complementing OLAP with analysis rules. *Data Knowl. Eng.*, 39(3):241–269, 2001.

[122] B. Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology.* Springer, 2000.

[123] B. Thalheim. Database component ware. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 13–26. Australian Computer Society, Inc. Darlinghurst, Australia, Australia, 2003.

[124] Bernhard Thalheim. Fundamentals of Cardinality Constraints. In *ER*, pages 7–23, 1992.

[125] Bernhard Thalheim. Component Construction of Database Schemes. In Stefano Spaccapietra, Salvatore T. March, and Yahiko Kambayashi, editors, *ER*, volume 2503 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2002.

[126] Bernhard Thalheim. Component development and construction for database design. *Data Knowl. Eng.*, 54(1):77–95, 2005.

[127] Riccardo Torlone. Two approaches to the integration of heterogeneous data warehouses. *Distributed and Parallel Databases*, 23(1):69–97, 2008.

[128] Olga De Troyer and René Janssen. On Modularity for Conceptual Data Models and the Consequences for Subtyping, Inheritance & Overriding. In *ICDE*, pages 678–685. IEEE Computer Society, 1993.

[129] Juan Trujillo, Manuel Palomar, Jaime Gómez, and Il-Yeol Song. Designing Data Warehouses with OO Conceptual Models. *IEEE Computer*, 34(12):66–75, 2001.

[130] Roel Wieringa, Wiebren de Jonge, and Paul Spruit. Using Dynamic Classes and Role Classes to Model Object Migration. *TAPOS*, 1(1):61–83, 1995.

[131] Morton E. Winston, Roger Chaffin, and Douglas Herrmann. A Taxonomy of Part-Whole Relations. *Cognitive Science*, 11(4):417–444, 1987.

[132] Jane Zhao and Klaus-Dieter Schewe. Using Abstract State Machines for Distributed Data Warehouse Design. In *APCCM*, pages 49–58, 2004.