AN OBJECT-ORIENTED APPROACH TO THE TRANSLATION BETWEEN MOF METASCHEMAS

APPLICATION TO THE TRANSLATION BETWEEN UML AND SBVR

RUTH RAVENTÓS PAGÈS

DOCTORAL THESIS

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ADVISOR: ANTONI OLIVÉ RAMON

BARCELONA, 2009

A thesis presented by Ruth Raventós Pagès in partial fulfillment of the requirements for the degree of *Doctor per la Universitat Politècnica de Catalunya*

Al Jordi

A la Laia, la Núria, la Maria,
el Pau i els seus avis

Acknowledgements

I would like to express my sincere gratitude to all who have supported and contributed the achievement of this goal.

First of all, I would like to thank my advisor Dr. Antoni Olivé for the trust he has placed in me. He, with no doubt, has been the best advisor I could ever have had, and has become a good friend in whom I will always trust. Working with him has indeed been a great pleasure. His rigor, guidance, patience and support during our discussions have taught me how to enjoy researching.

Thanks to all my colleagues in the *Grup de Modelització Conceptual,* in the *Secció de Sistemes d'Informació* at the UPC and in the *Information Systems* Department at ESADE for giving me their utmost support.

I am particularly grateful to Dr. Pericles Locopoulos for giving me the opportunity to join the Manchester University for a three-month period and for receiving me as part of his team.

Thanks to all the reviewers that have contributed with their comments in different stages of this research.

I would also like to thank the examiners of the thesis board: Dr. Pericles Loucopoulos, Dr. Paolo Atzeni, Dr. Martin Gogolla, Dr. Ernest Teniente and Dr. Maria Ribera Sancho for accepting to be members of this panel.

Thanks to many friends that have always encouraged me to do the thesis and especially to Alfred and Assumpta, Eduard and Anna, Miquel, Cristina and Jordi.

Finally, to my large family who has also made possible this project. They have supported me in the tough and happy moments. I would like to thanks to my parents who have been always a model of how to enjoy life and to my daughters Laia, Núria and Maria and to my son Pau. They have never missed a chance to remind me what most important thing in my life is. And above all, to Jorge, is it necessary to say why?

This research has partly been supported by Spain's Ministerio de Ciencia y Tecnología under project TIN2005-06053 and Spain's Ministerio de Ciencia e Innovación project TIN2008-00444.

Abstract

Since the 1960s, many formal languages have been developed in order to allow software engineers to specify conceptual models and to design software artifacts. A few of these languages, such as the Unified Modeling Language (UML), have become widely used standards. They employ notations and concepts that are not readily understood by "domain experts," who understand the actual problem domain and are responsible for finding solutions to problems.

The Object Management Group (OMG) developed the Semantics of Business Vocabulary and Rules (SBVR) specification as a first step towards providing a language to support the specification of "business vocabularies and rules." The function of SBVR is to capture business concepts and business rules in languages that are close enough to ordinary language, so that business experts can read and write them, and formal enough to capture the intended semantics and present them in a form that is suitable for engineering the automation of the rules.

The ultimate goal of business rules approaches is to build software systems directly from vocabularies and rules. One way of reaching this goal, within the context of model-driven architecture (MDA), is to transform SBVR models into UML models. OMG also notes the need for a reverse engineering transformation between UML schemas and SBVR vocabularies and rules in order to validate UML schemas.

This thesis proposes an automatic approach to translation between UML schemas and SBVR vocabularies and rules, and vice versa. It consists of the application of a new generic schema translation approach to the particular case of UML and SBVR.

The main contribution of the generic approach is the extensive use of object-oriented concepts in the definition of translation mappings, particularly the use of operations (and their refinements) and invariants, both formalized in the Object Constraint Language (OCL). Translation mappings can be used to check that two schemas are translations of each other, and to translate one into the other, in either direction. Translation mappings are declaratively defined by means of preconditions, postconditions and invariants, and they can be implemented in any suitable language. The approach leverages the object-oriented constructs embedded in Meta Object Facility (MOF) metaschemas to achieve the goals of object-oriented software development in the schema translation problem.

The generic schema translation approach and its application to UML schemas and SBVR vocabularies and rules is fully implemented in the UML-based Specification Environment (USE) tool and validated by a case study based on the conceptual schema of the Digital Bibliography & Library Project (DBLP) system.

Table of contents

List of Figures

Acronyms

DBLP	Digital Bibliography & Library Project
CIM	Computation Independent Model
ER	Entity-Relationship
MDA	Model Driven Architecture
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
SBVR	Semantics and Business Vocabulary & Rules
UML	Unified Modeling Language
USE	UML-based Specification Environment tool
XML	Extensible Markup Language

1 Introduction

This chapter introduces the research presented in this thesis and its background, explains the motivation for pursuing this work, provides an overview of the approach taken and details the structure of the thesis.

1.1 Motivation

Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specification of software behavior, and to their evolution over time and across software families. (Zave 1997)

Requirements engineering is a complex process that usually consists of three phases: requirements elicitation, requirements specification and requirements validation.

During the requirements elicitation phase, the various parties (e.g., users, designers, and managers) analyze their particular problems and needs and decide on the configuration of the system to be built. Needs and goals, defined at the business level, are translated into business requirements. Those business requirements which are to be solved by the software system are elicited.

To ensure that the business requirements document is complete and accurate, all knowledge for operating the organization and dealing with its environment should be captured in languages (such as ordinary English) that the "domain experts"—e.g., healthcare experts, finance experts, transportation experts, business managers, etc.—can read and write. Moreover, businesses change constantly and new decisions must be made accordingly in the business environment. Business experts should have mechanisms to easily incorporate these changes in the business requirements document.

In the requirements specification phase, the system's functional requirements (i.e., what it must do) and non-functional requirements (i.e., its global properties) are defined. The functional requirements are the capabilities and behaviors that must be performed, and the business rules are what the functional requirement knows—the decisions, guidelines and controls that are behind the functionality. That is, when defining a functionality, businesspeople identify the business rules that constrain it. The result of the requirements specification phase is a set of documents, called specifications, that precisely describe the system that the users require and that the designers have to design and build (Olivé 2007). The specification of the functional requirements is formally represented in what is called the conceptual schema. Conceptual schemas are described in a particular conceptual modeling language. Nowadays, UML (Rumbaugh, Jacobson and Booch 2004) is the modeling language that is most commonly used to specify conceptual schemas in the field of software engineering.

UML and other software languages have been designed for use by software engineers, whose ultimate goal is to design software artifacts. Consequently, they employ notations and concepts that are not readily understood by business experts. For example, when defining the functionalities of a rental car company, the user may identify the business rule that "each rental authorizes at most three additional drivers" (from the EU-Rent Example (Object Management Group 2008a)). In UML, a business rule may be specified by a graphical symbol in a modeling diagram (e.g., the multiplicity symbol) or as a constraint specified in OCL. For example, the aforementioned business rule of the rental car company could be described as the multiplicity symbol "0..3" of a member end of the association between rental and additional driver.

During the requirements validation phase, the quality of the conceptual schema is mainly determined by its correctness and completeness. A conceptual schema is complete if it satisfies the following condition:

All relevant general static and dynamic aspects, i.e., all rules, laws, etc., of the universe of discourse should be described in the conceptual schema. The information system cannot be held responsible for not meeting those described elsewhere, including in particular those in application programs. (Griethuysen 1982)

A conceptual schema is correct if the knowledge that it defines is true for the domain and relevant to the functions that the system must perform (Olivé 2007).

Good communication and understanding between domain experts and software engineers may be the best way to guarantee a high-quality conceptual schema. For this reason, over the last two decades, many efforts have been made to create tools that can express business concepts and business rules in languages that are close enough to ordinary language, so that business experts can read and write them, and formal enough to capture the intended semantics and present it in a form that is suitable for engineering the automation of the rules.

Some approaches to business rules tools represent business rules as sentence templates (Halle 2001, Morgan 2002, Wan Kadir and Loucopoulos 2003, Ross 2005 and Loucopoulos and Wan Kadir 2008) that can easily be mapped to ordinary language. Other approaches represent rules by using mathematical logic, which can easily be mapped to software tools. Examples include External Rule Language (ERL) (McBrien et al. 1991) and Courteous

Logic Program (CLP) (Grosof, Labrou and Chan 1999), which is encoded using XML to produce Business Rule Markup Language (BRML). BRML is the predecessor of Rule Markup Language (RuleML) (Boley, Tabet and Wagner 2001), an XML-based markup language that permits web-based rule storage, interchange, retrieval and firing/application.

Recently, the Object Management Group (OMG) published *Semantics of Business Vocabulary and Business Rules (SBVR) v.1.0* (Object Management Group 2008a) as an Available Specification. It defines the metamodel for documenting the semantics of business vocabulary, business facts and business rules. SBVR claims to be optimally conceptualized for businesspeople and already includes predefined alternative, non-normative notations for expressing concepts and rules by means of English statements (either in SBVR Structured English or in BRS RuleSpeak (Object Management Group 2008a)). Business rules in SBVR are structured by logical semantic formulations, which facilitates their automation in software systems.

The ultimate goal of SBVR and other business rules approaches is to build software systems directly from the vocabulary and business rules specifications (Date 2000).

Before the publication of the SBVR specification, OMG adopted *model-driven architecture* (MDA) (Object Management Group 2003), an approach to defining and using models at different levels of abstraction in software development. MDA specifies three system viewpoints: a computation-independent viewpoint, a platform-independent viewpoint and a platform-specific viewpoint. MDA also specifies three default system models corresponding to the three MDA viewpoints. The *computation-independent model* (CIM) is a description of a system based on the computation-independent model. It is assumed that the primary user of the CIM is the domain practitioner. In fact, SBVR specifies a metamodel to describe CIMs. A *platform-independent model* (PIM) is a description of a system from the platform-independent viewpoint. A PIM describes the conceptual model of the system to be built. UML is the standard language proposed by OMG to build PIMs. Finally, a *platform-specific model* is a description of a system from the platform-specific viewpoint. PSM is a version of PIM that includes the technical information required to develop the model in a tool.

Therefore, within the MDA, reaching the ultimate goal of the aforementioned business rules approaches implies transforming SBVR models into UML models. The need for this transformation was introduced by the OMG in Annex K of the SBVR specification (OMG 2008a). The same Annex K also explains the need to transform UML models into SBVR models. The OMG calls this reverse engineering transformation.

The main purpose of this thesis is to provide a translation specification between UML models and SBVR models and vice versa.

1.2 Problem description

The problem of automatic translation between UML and SBVR can be formulated as a particular application of a more generic problem called schema translation.

Schema translation has been considered an important practical problem in the fields of databases and information systems engineering since the mid 1970s (Chen 1976, Griethuysen (ed.) 1982). The problem is now even more important due to the need for translation among ontology languages (Concho, Fernandez López and Gómez-Pérez 2003) and for translation among "models" of the OMG's MDA software development approach (Object Management Group 2003).

Many ad hoc solutions to the schema translation problem have been proposed. A comprehensive analysis of these solutions is beyond the scope and purpose of this thesis, but Chapter 2 provides a summary of surveys in (among others) Rahm and Bernstein (2001), Shvaiko and Euzenat (2005), Czarnecki and Helsen (2006) and Mens and Van Gorp (2006). Most work on schema translation is currently described within the context of the *model management* framework (Bernstein 2003). This framework provides several generic operators that manage schemas and schema mappings. One of the operators is *ModelGen*, whose purpose is to automatically translate a source schema expressed in one metaschema into an equivalent target schema expressed in a different metaschema, along with the mapping constraints between the two schemas (Bernstein and Melnik 2007). Within this framework, a specification of the *ModelGen* operator would be the solution to our research goal.

MDM (Atzeni and Torlone 1996) was one of the first generic implementations of *ModelGen*, which was followed by MIDST (Model-Independent Schema and Data Translation) (Atzeni, Capellari and Bernstein 2006). MIDST represents schemas and metaschemas as instances of the relational metaschema; schema translations are built by combining elementary translations specified by Datalog rules defined at the metaschema level. Moreover, MIDST has a superschema and a supermetaschema, which have all the constructs known to the system. The super metaschema acts as a pivot, so it is sufficient to have translation rules for each metaschema to and from the supermetaschema. Three similar approaches have been proposed by Boyd and McBrien (2005), Hainaut (2005) and Bowers and Delcambre (2006) using the HDM, GER and ULD languages, respectively. None of these solutions is contextualized in the object-oriented paradigm.

In the context of model-driven architecture (MDA), the OMG has proposed QVT as a family of languages for representing model-to-model transformations (including translations). QVT-Relations is used to declaratively specify relationships between MOF metaschemas, using an approach similar to that of Gogolla (2005) and Bézivin et al. (2006). QVT-Operational Mappings is used to provide an imperative implementation of those relationships. Operational Mappings is a new language, although it includes OCL, which is extended with procedural constructs. The MOMENT-QVT tool is a model-transformation engine that provides partial support for QVT-Relations (Boronat, Carsí and Ramos 2005b). As yet there is no tool that provides total support for the QVT-Relations language.

Therefore, in order to build an automatic translation between UML and SBVR, we have considered several alternatives. One alternative is to build an ad hoc solution. We discarded this alternative for the same reason given by Atzeni (Atzeni 2007):

A major feature of any significant attempt to the schema translation problem would be generality: we need approaches that are maintainable and scale.

A second alternative is to adapt an existing generic approach to the particular case of UML and SBVR. As stated above, in the context of model management, some generic applications have been developed on relational databases instead of object-oriented schemas; others focus on translating object-oriented schemas, but use a third language for the schema-mapping specification between the two schemas. The use of a third language to represent mappings between two metaschemas adds complexity to the schema translation problem. Moreover, an in-depth study of such language would be necessary in order to demonstrate the consistency and correctness of the translations.

A third alternative—the one which is explored in this thesis—is to create a new generic approach to the schema translation problem. The advantages of this generic approach are explained in detail in Chapter 3.

Automatic translation between UML and SBVR metamodels is more complex than the generic schema translation problem for the following reasons:

- UML and SBVR metamodels are very complex structures. The *Structure* package of UML includes 55 metaclasses, which are instances of MOF. SBVR includes 109 metaclasses, which are instances of MOF. The specifications of the two metamodels are described very differently. The UML document first shows the abstract syntax of the metamodel in UML diagrams and then describes all of the concepts shown in the diagrams. Each concept is described separately according to a structured format that includes the following clauses: *Heading, Description, Generalizations, Attributes, Associations, Constraints, Additional operations, Semantics, Semantics variation points, Notation, Presentation options, Style guidelines, Examples* and *Changes from previous UML*. The SBVR specification is structured in several vocabularies and business rules. Within each vocabulary, the concepts are described in accordance with the non-normative SBVR Structured English notation. In other words, each vocabulary entry may include the following clauses: *Primary Representation, Definition, Source, Dictionary Basis, General Concept, Concept Type, Necessity, Possibility, Reference Scheme, Note, Example, Synonym, Synonymous Form, See, Subject Field* and *Namespace URI*. The complexity of the metamodels and the documents that describe them makes it more difficult to understand the semantics of the defined concepts and the establishment of translation mappings between them.
- The SBVR specification includes, on the one hand, the description of concepts and, on the other, the description of the representations of concepts. However, the correspondence between meanings and representations is not always clear. SBVR proposes SBVR Structured English as a possible notation for the representation, but this language is not normative and there is no straightforward correspondence between instances of these representation concepts and the SBVR Structured English notation. Therefore, in SBVR, in order to represent concepts and business

rules in ordinary English, some constructs and additional operations or conversions may be needed.

1.3 Research contributions

As stated above, ad hoc solutions and adaptations of generic schema translation approaches fall short when building an automatic translation between UML and SBVR models.

This thesis proposes a new generic schema translation approach whose main characteristics are as follows:

- Metaschemas are represented as instances of the OMG's MOF (Meta Object Facility) (Object Management Group 2006a);
- Translations are defined in terms of schema units and characterization objects of such schema units. Schema units are units of knowledge consisting on a set of schema elements. Characterization objects of schema units roughly correspond to the "domain value object" in the object-oriented design patterns field. Operations, hosted in object types, formalized in the OCL language are provided to define the schema units, the precedence relationship among them and the characterization objects;
- Elementary translations between schema units are represented by means of operation postconditions hosted in object types, and formalized in the OCL language (Object Management Group 2006b);
- The translation relationship between two sets of schema elements that represent two schema units is split into two simpler parts: one between the schema elements of one side and the characterization objects of the other side, and one between the characterization object of the second side and its schema elements; and
- The operation postconditions are also used to check the consistency of the translations.

The application of the generic schema translation approach to the translation of UML models to SBVR models and vice versa involves the following contributions:

- Schema units (i.e., the semantic units of knowledge and the precedence relationships among them) are defined in both UML and SBVR; and
- Schema mapping translation between UML and SBVR is defined in terms of two operations, equivalents and includedIn, for each schema unit of each metamodel.

Finally, two additional contributions, derived from the problem that there is no straightforward way to express the instances of the SBVR metamodel in SBVR Structured English, have also been made:

- A very simple metamodel to support SBVR Structured English notation is defined; and

- Operations are defined to obtain the instances of this metamodel from the defined SBVR schema units.

1.4 Implementation and case study

All of the specifications presented in this thesis were validated and implemented in the UML-based Specification Environment (USE) tool (Gogolla, Büttner and Richters 2007). USE is a system for the specification of information systems developed by the Database Systems Group of the Department of Mathematics and Computer Science of the University of Bremen. It is based on a subset of UML. A USE specification contains a textual description of a model using features found in UML class diagrams. Expressions written in OCL are used to specify additional integrity constraints on the model. A model can be animated to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during an animation. For each snapshot, the OCL constraints are automatically checked.

One example has been used throughout this thesis to validate the various proposals. The example is based on the DBLP Case Study developed by Planes and Olivé (2006). The DBLP Case Study contains parts of the conceptual schema of the DBLP systems, written in UML. DBLP, a computer science bibliography website hosted at the University of Trier in Germany (http://www.informatik.uni-trier.de/~ley/db/) was originally a database and logic programming bibliography site. The DBLP server provides bibliographic information on major computer science journals and proceedings. The server initially focused on Database Systems and Logic Programming (DBLP). Now it is gradually being expanded towards other fields of computer science. It has recently been suggested that DBLP should stand for "Digital Bibliography and Library Project." The server, mirrored at five other websites, indexes more than one million articles and contains several thousand links to home pages of computer scientists (April 2008).

1.5 Structure of the thesis

Figure 1.1 shows the structure of this thesis. Chapters 2 to 8 are organized as follows:

Chapter 2 examines the state of the art of translation mappings. It illustrates usage scenarios involving translation between schemas and reviews current surveys that study existing ad hoc solutions for schema translations. It also reviews the various specifications of declarative mappings found in existing approaches. Finally, it describes the schema management approach that has recently emerged as a generic approach to the manipulation of schemas and mappings.

Chapter 3 presents a new object-oriented operation-based approach to translation between MOF metaschemas. It defines the schema unit concept. Translation mappings are defined in terms of schema units. Two small fragments of the ER and Relational metaschemas are used as running examples in order to illustrate the complete application of the method.

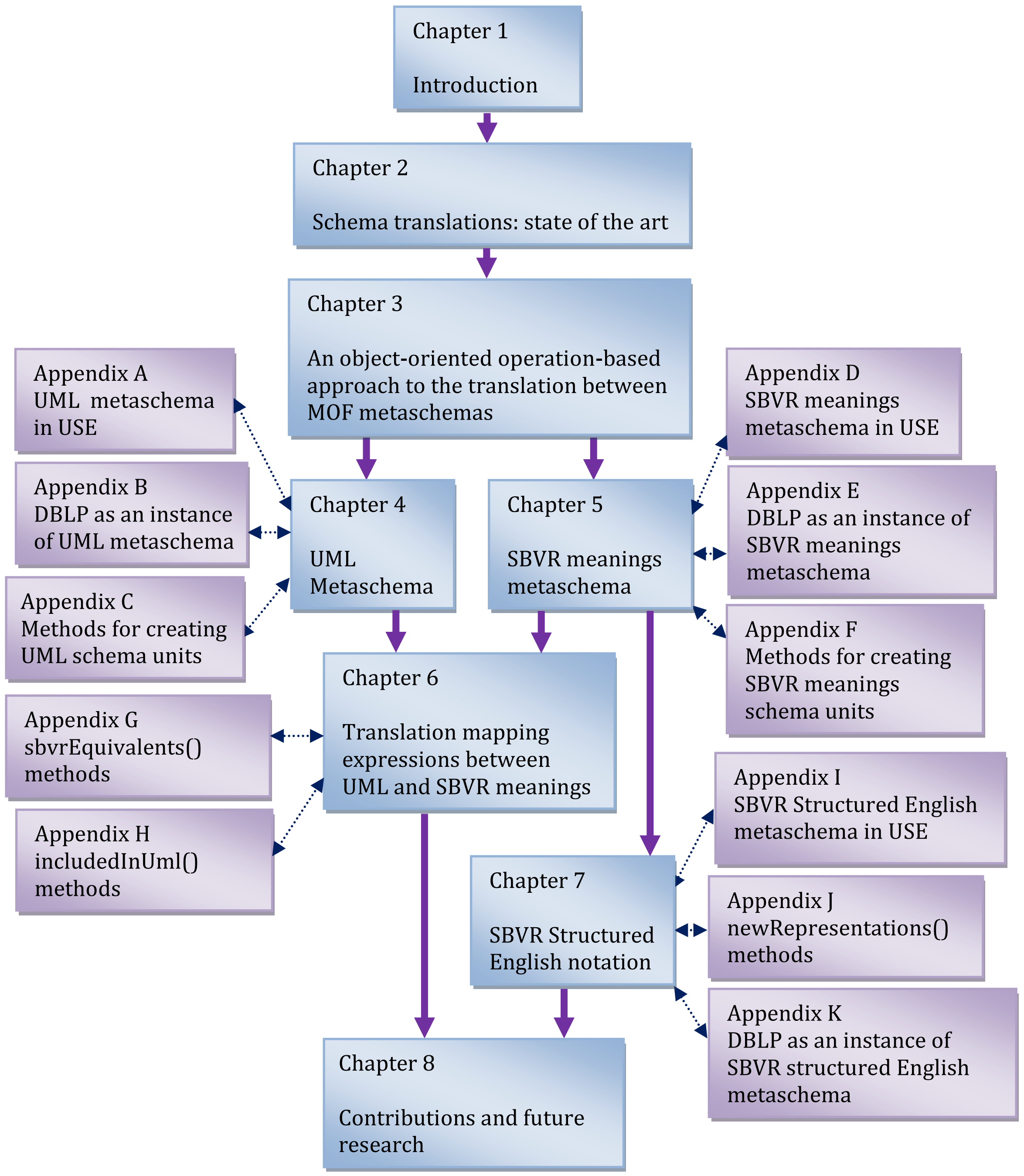

Chapter 1
Introduction
Chapter 2
Schema translations: state of the art
Chapter 3
An object-oriented operation-based approach to the translation between MOF metaschemas
Appendix A
UML metaschema in USE
Appendix B
DBLP as an instance of UML metaschema
Appendix C
Methods for creating UML schema units
Chapter 4
UML Metaschema
Chapter 5
SBVR meanings metaschema
Appendix D
SBVR meanings metaschema in USE
Appendix E
DBLP as an instance of SBVR meanings metaschema
Appendix F
Methods for creating SBVR meanings schema units
Chapter 6
Translation mapping expressions between UML and SBVR meanings
Appendix G
sbvrEquivalents() methods
Appendix H
includedInUml() methods
Appendix I
SBVR Structured English metaschema in USE
Chapter 7
SBVR Structured English notation
Appendix J
newRepresentations() methods
Appendix K
DBLP as an instance of SBVR structured English metaschema
Chapter 8
Contributions and future research

Figure 1.1 Thesis organization roadmap

Chapter 4 presents the UML metamodel. It begins by showing the DBLP example as an instance of the metamodel. It then describes its schema units, the precedence relationships among them and the characterization objects that define them.

Chapter 5 presents the SBVR meanings metamodels. First, it gives a general overview of the metamodel. Then, as in the previous chapter, it describes its schema units, the precedence relationships among them and the characterization objects that define them.

Chapter 6 describes the application of the translation approach proposed in Chapter 3 to the UML and SBVR meanings metaschemas, described in Chapters 4 and 5, respectively.

This chapter defines the necessary set of operations for translating schema units from UML to SBVR and vice versa.

Chapter 7 overviews the SBVR Structured English notation and describes the part of SBVR that refers to representations rather than meanings. This chapter also provides the set of operations for deriving the instances of SBVR representation from SBVR meanings.

Chapter 8 concludes this thesis by discussing the overall contribution of this research in the context of related work in this area. In addition, it discusses the limitations of the approach and points out areas for future research.

Finally, the **Appendices** (see Figure 1.1) describe the implementation, in USE, of the specifications of the schemas, metaschemas and operation methods referred to in Chapters 3 to 7.

2 Schema translations: state of the art

In an effort to investigate the appropriate approach for specifying translation mappings between SBVR vocabularies and UML models, this chapter reviews the literature on translation mappings. The need to translate, to transform, to integrate and to exchange information or knowledge is common to many application contexts. These needs, which require the manipulation of models and mappings between models, have been studied for more than three decades. In the database field, the problem of metadata manipulation (i.e., manipulation of metaschemas of databases) includes data integration (Batini, Lenzerini and Navathe 1986), data translation (Shu et al. 1977) and database design (Wiederhold 1977). In website and portal management, metadata is used to generate entire websites from databases (Fernandez et al. 1998, Mecca et al. 1998). In software engineering, metaschemas are used to describe the structure, interfaces and behavior of software components (Object Management Group 2006c). All types of metaschema-related applications involve the manipulation of schemas and mappings between such schemas.

In current practice, schema translation problems have often been tackled by means of ad hoc solutions, for example, by writing code for each specific application. Therefore, solutions may be very different from one another. Nevertheless, they usually divide the translation problem into two subproblems: (i) the *match*: how to obtain the translation mappings between two given schemas, that is, the relationship between the elements of the two, and (ii) the translation: how to apply the mapping functions in order to actually translate one schema to another. Several surveys have reviewed existing matching approaches aimed at solving the match problem, not only for translation purposes but also for data integration or data exchange. Other surveys have reviewed existing approaches that perform translations or, more generally, transformations between schemas. The classification dimensions proposed in all of these surveys give a good overview of the main features of current ad-hoc solutions related to translation mappings.

Ad hoc solutions are very heavy and hard to maintain, and there is still a compelling need for a general solution able to handle, in a uniform way, the great diversity of formats and types of information available (Atzeni 2007, Bernstein 2003, Bernstein et al. 2000, Bernstein and Melnik 2007).

In this direction, a quite recent approach to the generic manipulation of schemas and mappings, called *schema management*[1] (Atzeni 2007, Bernstein 2003, Bernstein et al. 2000, Bernstein and Melnik 2007), has been proposed. Its goal is to factor out the similarities of the metadata problems studied in the literature and to develop a set of high-level operators that can be utilized in various scenarios. According to Atzeni, Bernstein and Melnik, among others (Atzeni 2007, Bernstein 2003, Bernstein et al. 2000, Bernstein and Melnik 2007), five basic operators, known as *Match*, *Compose*, *Merge*, *Diff* and *SchemaGen*, can address the above problems when appropriately combined. In particular, translation between schemas may be described in terms of two of these generic operators, *Match* and *SchemaGen*. Therefore, in the schema management framework, the translation from one schema to another consists in the implementation of these two generic operators: *Match*, to obtain the mapping between the two metaschemas, and *SchemaGen*, to generate the target schema from the source schema.

Still, in ad hoc solutions and schema management, the core problem is the representation of schema mappings. There is a distinction between *engineered mappings* between schemas, which are needed in integration or translation, and *approximate mappings*, which are used in web searches and in mining heterogeneous sets of data sources (Bernstein and Melnik 2007). The former describes the exact equivalence or correspondence between elements of two different schemas and the latter usually includes additional attributes that characterize different types of correspondence among the elements.

The rest of this chapter surveys the literature that inspired the work presented in this thesis:

- Section 2.1 illustrates usage scenarios that involve translations between schemas and the common difficulties that arise when trying to solve the problems in said scenarios.
- Section 2.2 reviews current surveys that describe the features of ad hoc solutions that specify and/or implement translation mappings. Four surveys focus on matching between schemas and two surveys focus on translations and transformations between schemas.
- Section 2.3 describes various specifications of declarative translation mappings (i.e., engineered mappings) found in existing approaches.
- Section 2.4 describes the *schema management* approach: the high-level description of families of problems, the set of basic high-level operators proposed to solve these

[1] This thesis follows the terminology used in Olivé (2007). This terminology is different from that used by Bernstein & Melnik (2007) and Bézivin et al. (2006), who use the terms model and model management, respectively, to denote the concepts schema and schema management.

problems, solutions in terms of operators, and examples of existing implementations.

2.1 Application domain of schema management

This section stresses the importance of, and need for, generic schema translation and, by extension, schema management. First, it reviews the many usage scenarios that require the support of schema management by listing the families of applications that support it. Second, it summarizes the common problems found in such scenarios.

2.1.1 Families of applications that require the support of schema management

One way to characterize the application domain of schema management is to list the current categories of products that require the support of schema management. Of the numerous products in each category, only some examples are cited.

2.1.1.1 *CASE and reverse engineering tools*

Computer-aided software engineered (CASE) tools are used to assist in the development and maintenance of software. All aspects of the software-development lifecycle can be supported by CASE tools, from project management software to tools for business and requirement analysis, system design, code storage, compilers, test software and others. They usually include generators of lower-level models, and eventually code, from higher-level models. The generation of lower-level models from higher-level models involves the specification of translations between the models. This may also include using reverse engineering processors to generate higher-level models from code or lower-level models. Again, this usually involves designing translations between the models, which in turn requires an explicit representation of mappings. A list of vendors with more than 600 CASE tools is found in Lamb, Scott and Heavey (2005).

2.1.1.2 *Extract-transform-load (ETL) tools*

Extract-transform-load (ETL) tools (ETL 2007, Kimball, Caserta 2004) extract data from outside sources, transform it to fit business needs and load it into a database, usually a data warehouse. ETL tools are also used for integration with legacy systems. The first part of an ETL process extracts the data from the source schemas. Each separate schema may be an instance of a different metaschema. The extraction converts the data into a uniform format, i.e., as an instance of a given schema. The transformation stage applies a series of rules or functions to the data extracted from the source to derive the data to be loaded to the end target. The load phase loads the data into the end target, usually the data warehouse. The functions applied during the transformation stage are the applications of translation mappings defined between the source metaschema and the data warehouse metaschema.

2.1.1.3 *Message-mapping tools*

Message-mapping tools simplify the programming of message translation between different formats. These are often embedded in message-oriented transactional middleware, such as enterprise application integration (EAI) environments (Altova 2008, BEA 2007, Microsoft 2006, Stylus Studio 2008). EAI is the process of linking applications, such as supply chain management (SCM), customer relationship management (CRM) and business intelligence (BI) systems, in order to obtain financial and operational competitive advantages in business. To avoid every application having to convert data to or from every other application's formats, EAI systems usually stipulate an application-independent (or common) data format, i.e., a unique schema. The EAI system usually provides a data transformation service as well, in order to assist in the conversion between application-specific and common formats.

2.1.1.4 *Query mediators to access heterogeneous databases*

Query mediators are systems that combine the data residing at different sources and provide the user with a unified view of these data. This unified view is represented by the "global schema" and provides a reconciled view of all data, which can then be queried by the user. In database research, this is called data integration (Lenzerini 2002). In commercial IT, it is called enterprise information integration (EII) (Halevy et al. 2005) and exists in many variations, e.g., supporting web services and updates (Carey 2006). There are also custom implementations for bio-informatics and medical informatics (Davidson et al. 1999, Louie et al. 2007).

2.1.1.5 *Wrapper generation tools*

Wrapper generation tools are tools for accessing data sources from different sources and generating interfaces in a specific format for accessing and supporting the incremental updating of such sources, for example to produce an object-oriented wrapper for a relational database (Adya et al. 2007, Hibernate 2007, Oracle 2007) or to produce web wrappers for web-accessible data sources (Gruser et al. 1998). Unlike query mediators, wrappers often need to support incremental updates.

2.1.1.6 *Graphical query design tools*

Graphical query design tools can define a mapping between source schemas (e.g., relational databases) and target schemas (e.g., graphical user interfaces) (Bitpipe 2007). Usually, the source and target have different formats. These tools provide visual design environments for selecting tables and columns. They automatically build joins and Transact-SQL statements when the user selects which columns to use.

2.1.1.7 *Data translation tools*

Data translation tools can move data between different applications (Microsoft 2007). For commercial applications, their role has been partly subsumed by ETL tools. For design tools, however, they form a separate product category. For example, mechanical CAD tools need to translate between different geometric coordinate systems, assembly structures, and data formats (Bloor, Owen 1994).

2.1.2 Common problems to solve in the application domain

All of the aforementioned systems need to transform, integrate and exchange knowledge. In fact, because systems use different models to handle such knowledge, information needs to be translated from one to another. The developments in the Internet world have increased these needs, as it has become possible, at least in principle, to implement communication between systems at any level, without significant limitations in the amount of data exchanged or in the length of the interaction.

The major reasons for the complexity of these applications are as follows (Bernstein and Melnik 2007, Melnik 2004):

- Heterogeneity of representation of a particular domain, which arises because data sources are independently developed by different people and for different purposes. The data sources may use different data models, different schemas and different value encodings.
- Impedance mismatches that arise because the logical schemas required by applications are different from the physical ones exposed by data sources.
- Potpourri of tools: the solutions are language-specific, i.e., they are developed for SQL, UML, XML, or RDF and are not easily portable to other domains. For example, solutions developed for mapping database schemas are difficult to adopt for mapping websites.
- Insufficient abstraction of mapping metaschemas: mapping between metaschemas is developed using operations for the manipulation of schemas, not metaschemas. Such operations typically provide access to the individual elements of metaschemas, such as the individual attribute definitions of schemas. The programming of mapping applications with these operations requires a large amount of navigational code and incurs high development and maintenance costs.
- Unavailability of a general-purpose platform to simplify the development of mapping tools and applications. The existing general-purpose solutions typically focus on persistent storage or graphical design environments for metadata artifacts and do not go far enough to support the developers of metadata applications. In fact, many of today’s mapping-related tasks are still solved manually. An automated approach requires too much implementation effort due to the lack of a common programming platform.

The situation has become even more complicated as the number of data models has increased: ODMG (Berler et al. 2000), XSD (Peterson et al. (eds.) 2008, Sperberg-McQuen, Gao and Thompson (eds.) 2008), .NET (Microsoft 2008), .RDF (Becket 2004) and OWL (McGuiness andHarmelen 2004). Additionally, more programming languages and types of tools are appearing in the market.

2.2 Features of ad hoc solutions

In recent years, there have been so many different ad hoc approaches to solving the schema mapping problem and the schema translation problem that several surveys related thereto have been published. The dimensions proposed to classify the various approaches give a good overview of the different issues considered in the proposed solutions.

The surveys of Kalfoglou and Schorlemmer (2003) and Noy (2004) focus on the state of the art in ontology matching and approaches to integrating ontology-based information. The survey of Rahm and Bernstein (2001) classifies the schema mapping applied to database application domains. Shvaiko and Euzenat (2005) add new dimensions to the classification proposed by Rahm and Bernstein in order to apply it to information systems and ontologies, but their classification concentrates only on schema-level matching techniques. Note that all previous surveys focus on solutions to schema mapping, regardless of whether the mapping is used for integration, translation or transformation.

The surveys of Czarnecki and Helsen (2006) and Mens and Van Gorp (2006) describe and classify the existing approaches that specify and implement schema transformation and schema translation.

2.2.1 Noy classification

In the context of ontology research, Noy (2004) proposes three aspects for the classification of semantic-integration approaches:

(1) **Mapping discovery**: How the approach determines which concepts and properties represent similar notions. Mapping discovery is the major architecture used to find similarities between ontologies. The following are the two major sets of architectures:

- **Using a shared ontology**: When the goal of the approach is to facilitate knowledge sharing, a general upper ontology is used as a reference ontology in the integration process. This ontology formalizes notions such as processes and events, time and space, physical objects, and so on. Examples include the Suggested Upper Merged Ontology (SUMO) (Niles, Pease 2001) and DOLCE (Gangemi et al. 2003).
- **Using heuristics and machine-learning**: This comprises heuristic-based approaches or machine learning techniques that use various characteristics of ontologies (such as their structure, definitions of concepts or instances of classes) to find mappings.

(2) **Representation of mappings:** How mappings between ontologies are represented to enable reasoning. There is a broad spectrum of representations of mappings. The author discusses the following groups:

- **As instances of an ontology of mappings**: A mapping between two ontologies constitutes a set of instances of classes in the mapping ontology and can be used by applications to translated data from the source ontology to the target. It allows mechanisms such as the specification of recursive mappings and composed mappings.

- **As a set of bridging axioms in first-order logic**: The mappings, expressed as a set of bridging axioms relating classes and properties of the ontologies, are essentially translation rules. The rules refer to concepts from source ontologies and specify how to relate the same concepts in the other ontology. The ontologies mapped with the bridging axioms can then be treated as a single theory by a theorem prover optimized for ontology-translation tasks.
- **As views over either global or local ontologies**: A global ontology is defined to provide access to local ontologies and the mappings are defined as views over either the global or the local ontologies. In other words, a predicate from one ontology is defined as a query (and DL expression) over predicates in another ontology.

(3) **Reasoning with mappings:** What types of reasoning are involved, once the mappings are defined. For example, the mappings may be used to perform data translation, query answering or web-service composition tasks among others.

2.2.2 Kalfoglou and Schorlemmer classification

Kalfoglou and Schorlemmer (2003) classify ontology mapping approaches based on the type of work the approaches report. They distinguish the following categories:

(1) **Frameworks**: approaches that are mostly a combination of tools, providing a methodological approach to mapping; some of them are also based on theoretical work.

(2) **Methods and tools**: tools, either stand-alone or embedded in ontology development environments, and methods used in ontology mapping.

(3) **Translators**: approaches that translate vocabularies between ontologies that share the same domain.

(4) **Mediators**: tools to access, in a uniform view, vocabularies of different ontologies.

(5) **Techniques**: similar to methods and tools, but not so elaborate or as directly connected to mapping.

(6) **Experience reports**: reports on doing large-scale ontology mapping.

(7) **Theoretical frameworks**: theoretical work that has not yet been exploited by ontology mapping practitioners.

(8) **Surveys**: similar to experience reports but more comparative in style.

(9) **Examples**: a selection of original works that have been reported in the aforementioned categories.

After describing and showing examples of 35 works, the authors elaborate on important topics that emerged when examining these works. In particular, they critically review issues concerned with the relationship between ontology mapping and schema integration, the normalization of ontologies and the creation of formal instances, the role of formal theory in support of ontology mapping, the use of heuristics, the use of articulation and mapping rules, the definition of semantic bridges and the thorny issue of automated ontology mapping.

2.2.3 Rahm and Bernstein classification

The survey of Rahm and Bernstein (2001) provides a classification, in the context of the database field, of schema-matching approaches and a comparative review of matching systems.

Since the implementation of *Match* may use multiple match algorithms, or matchers, two subproblems are distinguished: (1) the implementation of individual matchers, each of which computes a mapping based on a single matching criterion, and (2) the combination of individual matchers within an integrated hybrid matcher (by using multiple matching criteria) or a composite matcher (by combining multiple match results produced by different match algorithms).

For the implementation of individual matchers, in which a mapping is computed based on a single matching criterion, the following largely-orthogonal classification criteria are considered:

(1) **Kind of information used**. Depending on the data that the mapping algorithms exploit, a matcher may be:

- Schema-level: Only schema information is considered.
- Instance-level matcher: Instances values are considered for the matching.

(2) **Granularity of match.** Depending on the schema elements or structures considered for the match, a matcher may be:

- Element-level: Individual schema elements, such as attributes, are analyzed in isolation, and their relations with other elements are ignored.
- Structure-level: Complex schema structures are considered together for the mapping.

(3) **Approach used on the mapping.** Depending on the type of comparisons made between elements, a matcher may be:

- Linguistic: Names and text are used to find similar schema elements.
- Constraint-based: Constraint information (e.g., data types, value ranges, uniqueness, optionality, relationship types, keys, cardinalities, etc.) is used to determine the similarities between elements.

(4) **Matching cardinality.** Depending on the number of elements of a source related to a certain number of elements of the target, a matcher may be:

- 1:1: One element of a schema matches to one element of the other schema.

- Set-oriented: 1:n, n:1.
- n:m: This cardinality usually requires considering the structural embedding of the schema elements and thus requires structure-level matching.

(5) **Auxiliary information used.** The matcher may rely only on the input schemas S1 and S2 or also on additional information. This additional information may be, among other things:

- Dictionaries.
- Global schemas.
- Previous matching decisions.
- User input.

A matcher that uses just one approach is unlikely to achieve as many good match candidates as one that combines several approaches. *Hybrid matchers* directly combine several matching approaches to determine match candidates based on multiple criteria or information sources. A hybrid matcher can offer better performance than the execution of multiple matchers by reducing the number of passes over the schema. *Composite matchers*, on the other hand, combine the results of several independently executed matchers, including hybrid matchers. This ability to combine matchers makes composite matchers more flexible than hybrid matchers.

2.2.4 Shvaiko and Euzenat classification

Shvaiko and Euzenat (2005) present a classification of schema/ontology matching techniques that builds on the work of Rahm and Bernstein (2001). The new criteria included are based on (i) general properties of matching techniques, (ii) interpretation of input information, and (iii) the kind of input information.

Their classification of matchers considers three major aspects: (1) granularity, (2) input interpretation, and (3) the kind of input. Further features considered are the following:

(1) **Granularity** of matching. As in Rahm and Bernstein (2001), there are two main groups of matchers:

   a. **Element-level** matching techniques, which compute mapping elements by analyzing entities in isolation, ignoring their relationships with other entities. These techniques may be the following:

      i. **String-based** techniques, which are used to match names and descriptions of schema/ontology entities. This includes name similarity, description similarity and global namespaces.

      ii. **Language-based** techniques, which can interpret a label as a word or phrase in some natural language. This includes:

         1. Tokenization: Names of entities are parsed in sequences of tokens by a tokenizer, which recognizes punctuation, cases, blank characters, digits, etc.

2. Lemmatization: The strings underlying tokens are morphologically analyzed in order to find all their possible basic forms (e.g., Kits→Kit).
3. Morphological analysis.
4. Elimination: The tokens that are articles, prepositions, conjunctions, etc. are marked to be discarded.

iii. **Constraint-based** techniques, which deal with the internal constraints applied to the definitions of entities, such as types, multiplicity of attributes and keys.

1. Datatype comparison: The various attributes of a class are compared with regard to the datatypes of their value.
2. Multiplicity comparison: Attribute values are collected by a particular construction (e.g., set, list, multiset), on which multiplicity constraints are applied.

iv. **Linguistic resources** such as common knowledge or domain-specific thesauri, which are used in order to match words (the names of schema/ontology entities are considered words of a natural language) based on the linguistic relations between them (e.g., synonyms, hyponyms).

v. **Alignment-reuse** techniques, which are an alternative way of exploiting external resources containing alignments of previously matched schemas/ontologies.

vi. **Upper-level** formal ontologies, which are external sources of common knowledge that are logic-based systems and can be exploited to analyze interpretations (e.g., SUMO or DOLCE).

b. **Structure-level** matching techniques, which compute mapping elements by analyzing entities with their relations.

i. **Graph based** techniques, which are graph algorithms that consider the input as labeled graphs. Database schemas, taxonomies and ontologies are viewed as graph-like structures containing terms and their inter-relationships.

ii. **Taxonomy-based** techniques, which are also graph algorithms, and which consider only the specialization relation.

iii. **Repositories of structures,** which store schemas/ontologies and their fragments together with the pair wise similarities between them.

iv. **Model-based** algorithms, which handle input based on its semantic interpretation. These are well-grounded deductive methods.

(2) **Input interpretation.** Techniques may generally interpret the input information in various ways. Matchers may consider:

a. **Internal** techniques, which use information that comes only with the input schemas/ontologies. This includes *syntactic* techniques, which interpret input based on its sole structure following some clearly stated algorithm, and *semantic* techniques, which use some formal semantics (e.g., model-theoretic semantics) to interpret the input and justify the results.

b. **External** techniques, which use auxiliary (external) resources or domains and common knowledge to interpret the input. These techniques do not distinguish between syntactic or semantic, since a user's input cannot be characterized as either syntactic or semantic.

(3) **Kind of input**. Algorithms may use different kinds of data. Three types are considered:

a. **Terminological**: Strings. Found in the ontology descriptions.

b. **Structural:** Structures. Found in the ontology descriptions. This requires some semantic interpretation and usually uses some semantically compliant reason to deduce the correspondences.

c. **Semantics**: Models. This includes upper-level formal ontologies, as defined above, and model-based ones (SAT and DL).

2.2.5 Czarnecki and Helsen classification

Czarnecki and Helsen (2006) propose a model to describe and classify the existing approaches to schema transformation. In their work, they consider a translation of one schema to another as a particular type of transformation in which the two schemas are equivalent and their metaschemas are different. Therefore, the features considered in transformation approaches may be applied in translation approaches.

The model considers the following features:

(1) **Specification representation**, which refers to the type of language or mechanism used to represent the specification of the transformation or matching. Some approaches express the translation expressions as preconditions and postconditions in OCL, while others express them in a relational language such as QVT-Relations, and still others express them as functions in an executable language.

(2) **Transformation rules**, which describe the smallest unit of transformation. The description of the rules includes the definition of the following:

a. Domains: how the domains (i.e., source and target models) are involved in the transformation, the metamodel, the directionality of rules, the body of the rules, and the typing of variables, logic and patterns.

b. Syntactic separation of the rules operating on the source and target models.

c. Multidirectionality: the ability to execute a rule in different directions.

d. Application conditions: how a rule is applied.

e. Intermediate structures, such as traceability links, which are necessary for transformation.

f. Parameterizations, which are used to make transformation rules more reusable.

g. Reflection and aspects, which are supported in the transformations.

(3) **Rule application control,** which refers to the strategy used to determine the specific location where a rule is applied within its source scope. Two aspects are considered:

a. Location determination: The strategy may be deterministic, non-deterministic or interactive.

b. Rule scheduling: The order is determined for the application of individual rules are applied. This may vary in four main areas: (i) form (i.e., whether scheduling can be expressed explicitly or implicitly), (ii) rule selection (i.e., explicit or nondeterministic), (iii) rule iteration (i.e., whether it includes mechanisms such as recursion, looping and fixed-point iteration), and (iv) phasing (i.e., whether the process is organized in phases).

(4) **Rule organization**: This concerns general structuring issues, such as modularization and reuse mechanisms.

(5) **Source-target relationship**: The source and target may be the same model or two different models.

(6) **Incrementality**: This refers to the ability to update existing target models based on changes in the source models. Three cases are considered: (i) target incrementality or change propagation (i.e., the ability to update the existing target models based on changes in the source models), (ii) source incrementality (i.e., the ability to minimize the amount of source that needs to be reexamined by a transformation when a source is changed), and (iii) preservation of user edits in the target (i.e., the ability to rerun a transformation on an existing user-modified target).

(7) **Directionality**: This describes whether a transformation can be executed in only one direction or in multiple directions.

(8) **Tracing**: This describes the mechanisms used to record different aspects of transformation execution.

2.2.6 Mens and Van Gorp classification

Mens and Van Gorp (2006) propose a taxonomy of model transformation that groups tools, techniques and formalisms for model transformation based on their common qualities.

The following features are considered in their classification:

(1) **Characteristics of the source and target models**, which include: (i) the number of source and target models, (ii) the technical space determined by the meta-model that is used, (iii) whether the transformation is endogenous or exogenous, (iv) whether the

transformation is horizontal (at the same level of abstraction) or vertical (at different levels of abstraction), and (v) whether the transformation is syntactic or semantic.

(2) **Characteristics of the model transformation process**: (i) the level of automation of the process, (ii) the complexity of the process, and (iii) the preservation of the source model in the target model.

(3) **Characteristics of the language or transformation tool**: (i) whether it accepts creating/reading/updating/deleting (CRUD) transformations, (ii) whether it allows suggestions when applying transformations, and (iii) whether it allows the customization or reuse of transformations.

(4) **Characteristics of the language or transformation tool to verify and guarantee correctness of transformations**: (i) whether it includes testing and validation techniques, (ii) whether it deals with incomplete or inconsistent models, (iii) whether it allows grouping, composing and decomposing transformations, (iv) whether it allows genericity of transformations, (v) whether it includes bidirectionality of transformations, and (vi) whether it supports traceability and change propagation.

(5) **Quality requirements for a transformation language or tool**: (i) usability and usefulness, (ii) verbosity versus conciseness, (iii) performance and scalability, (iv) extensibility, (v) interoperability, (vi) acceptability by user community, and (vii) standardization.

(6) **Characteristics of the mechanisms used for model transformation**: whether it relies on a declarative or operational (or imperative) approach.

2.3 Translation mappings specifications

The core problem in schema translation is the representation of translation mappings.

There is a broad spectrum of representations of translation mappings. A multitude of mapping languages have been utilized in the literature to address various schema management scenarios (Benedikt et al. 2003, Bergamaschi, Castano and Vincini 1999, Bernstein 2003, Buneman, Davidson and Kosky 1998, Claypool 2002, Halevy 2001, Hainaut 1996, Kementsietsidis, Arenas and Miller 2003, Li, Bohannon and Narayan 2003, Madhavan, Halevy 2003, Melnik, Rahm and Bernstein 2003, Mitra, Wiederhold and Kersten 2000, Popa et al. 2002, Pottinger and Bernstein 2003, Papotti and Torlone 2005). The number of languages is probably matched by the number of data models or schema languages developed for the same purpose.

In general, a schema mapping is a triple $M = (S_1, S_2, \Sigma)$ where S_1 is the source schema, S_2 is the target schema and Σ, known as mapping expression, is a set of constraints over S_1 and S_2. An instance of the mapping M is a pair $<s_1, s_2, >$ such that s_1 is an instance of S_1, s_2 is an instance of S_2, and s_1, s_2 satisfy all constraints Σ.

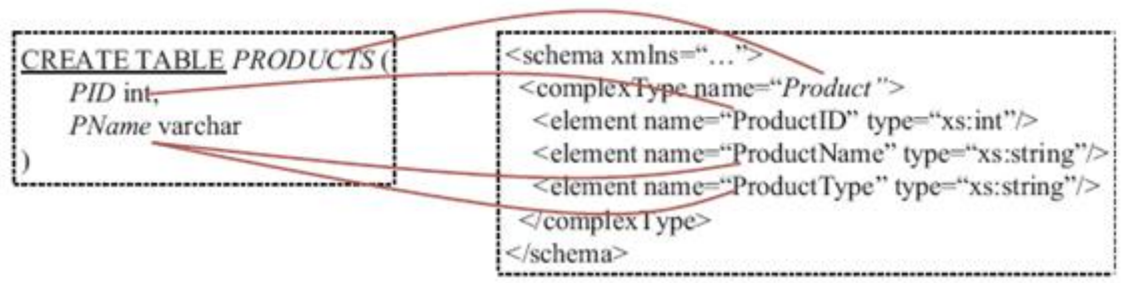
The classification of the various schema mapping approaches, summarized in Section 2.2, shows that mapping representations vary based on various aspects: the purpose of the mapping, the representation of the source and target schema, the type of language used in the representation, the kind of information used for the mapping, the granularity of the match, the cardinality of the match, etc.

The rest of this section describes, in a generic way, some of the most common approaches to the declarative specification of mappings.

2.3.1 Schema morphism expressions

A schema morphism is the simplest specification of mapping (Melnik 2004, Melnik, Rahm and Bernstein 2003, Melnik, Rahm and Bernstein 2003). Conceptually, a morphism is a set of arcs connecting the elements (e.g., relational tables or XML types) of two schemas. A morphism is clearly a weak representation of a transformation between two models, since it carries no semantics about the transformation of instances (i.e., there are no constraints). Still, morphisms are useful in metadata applications that do not require instance transformations, such as dependency tracking, schema translation (e.g., UML to IDL or ER to SQL) and impact analysis. Furthermore, morphisms can represent mappings between different kinds of schemas (e.g., relational and XML), can always be inverted and composed, and can be easily implemented and manipulated.

Figure 2.1 shows an example of a representation of morphisms between a relational table and an XML schema. Note that various kinds of schema elements, such as relations or attributes can participate in a morphism.



```
CREATE TABLE PRODUCTS (
    PID int,
    PName varchar
)
```

```
<schema xmlns="…">
  <complexType name="Product">
    <element name="ProductID" type="xs:int"/>
    <element name="ProductName" type="xs:string"/>
    <element name="ProductType" type="xs:string"/>
  </complexType>
</schema>
```

Figure 2.1 A morphism between a relational table and an XML schema (from Melnik (2004))

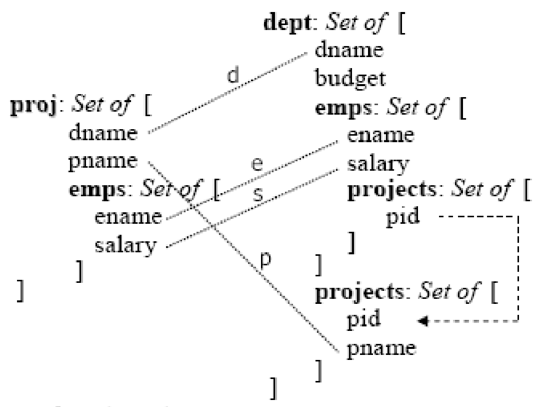
2.3.2 Schema query assertions

An alternative for defining mappings is to consider a mapping a query (e.g., an SQL query) on the source schema that produces a subset of a target relation. Thus, a mapping defines one out of possibly many ways of forming target elements. This is the most common type of mapping used for schema integration (Bergamaschi, Castano and Vincini 1999, Lenzerini 2002, Miller, Ioannidis and Ramakrishnan 1994), where a global schema is constructed from many source schemas.

In this context, a data integration system is a triple $\langle G, S, M\rangle$ where G is the global schema, S is one of the sources schemas, and M is the mapping between G and S. Several approaches are considered to specify the mapping M between G and S:

- If the sources are defined in terms of queries formulated over the global schema, the approach is called *source-centric* or *local-as-view* (*LAV)*.

- If the global schema is defined in terms of queries formulated over the sources, the approach is called *global-schema-centric* or *global-as-view* (*GAV)*.
- An approach that combines the above two approaches is called *GLAV*.
- If the mapping is between sources, without a global schema, the approach is called *peer-to-peer* (*P2P)*.



```
m1   for p in proj ⇒
     exists d' in dept, p' in d'.projects
     where d'.dname=p.dname ∧ p'.pname=p.pname

m2   for p in proj, e in p.emps ⇒
     exists d' in dept, p' in d'.projects, e' in d'.emps, p" in e'.projects
     where p".pid=p'.pid ∧
             d'.dname=p.dname ∧ p'.pname=p.pname ∧
             e'.ename=e.ename ∧ e'.salary=e.salary
```

Figure 2.2 Example of two mappings specified as GLAVs assertions (from Fuxman et al. (2006))

Figure 2.2 shows two nested relational schemas. The source schema, `proj`, is a set of records with two atomic components, `dname` (department name) and `pname` (project name), and a set-valued component, `emps`, that represents a (nested) set of employee records. The target schema is a reorganization of the source: there is, at the top level, the set of department records, with two nested sets of employee and project records. The Figure 2.2 also shows two basic mappings to describe the relationship between the source and target schemas. The first one, m_1, is a query that maps the department and project names in the source to the corresponding elements in the target. The second one, m_2, is a query that maps department and project names and their employees. Correspondences between schema elements (e.g., `dname` to `dname`) are captured by equalities between such components (e.g., `do.dname=p.dname`) grouped in the `where` clause that follows the `exists` clause of a mapping.

2.3.3 Logic-based formulas

Logic-based notation is another declarative alternative for specifying mappings (Buneman, Davidson and Kosky 1998, Calvanese, Giacomo and Lenzerini 2001, Madhavan et al. 2002). Most ontology mappings are represented by a logic-based language.

In the above example (see Figure 2.2), each mapping may be represented as an implication between a set of atomic formulas over the source schema and a set of atomic formulas over the target schema. Each atomic formula is of the form $e(x_1, \dots, x_n)$, where e denotes a set and $x_1, \dots, x_n$ are variables. The two mappings, m1 and m2, shown in the example have the following corresponding formulas (Fuxman et al. 2006):

m1: $\texttt{proj}(d, p, E_s) \rightarrow \texttt{dept}(d, !b, !E, !P) \wedge P(!x, p)$

m2: $\texttt{proj}(d, p, E_s) \wedge E_s(e, s) \rightarrow \texttt{dept}(d, !b, !E, !P) \wedge E_s(e, s, !P') \wedge P'(!x) \wedge P'(x, p)$

For each formula, the variables on the left of the implication are assumed to be universally quantified. The variables on the right that do not appear on the left are assumed to be existentially quantified. For clarity, the quantifiers are omitted and there is a question mark in front of the first occurrence of an existentially quantified variable. To illustrate, in m2, the variable E_s denotes the nested set of employee records (inside a tuple in the top-level set `proj`). The variables E, P and P' are also set variables, but existentially quantified. The variables b (for `budget`) and x (`project id`) are existentially quantified as well (but atomic). The meaning of m2 is as follows: for every source tuple (d, p, E_s) in `proj`, and for every tuple (e, s) in the set E_s, there must exist four tuples in the target as follows. First, there must be a tuple (d, b, E, P) in `dept`, where b is some "unknown" budget, E identifies a set of employee records, and P identifies a set of project records. Then, there must exist a tuple (e, s, P') in E, where P' identifies a set of project IDs. Furthermore, there must exist a tuple (x) in P', where x is an "unknown" project ID. Finally, there must exist a tuple (x, p) in the aforementioned set P, where x is the same project ID used in P'.

2.3.4 Graph transformation rules

Graph transformation rules are a visual representation alternative for schema mappings (Boyd and McBrien 2005, Grunske, Geiger and Lawley 2005, Song, Zhang and Kong 2004, Vara et al. 2007). Schemas may be represented as graphs. A graph transformation rule $p = \langle G_{LHS}, G_{RHS} \rangle$ consists of two directed typed graphs G_{LHS} and G_{RHS}, which are called the left-hand side and right-hand side of p.

Song, Zhang and Kong (2004) represent management operators based on graph transformation. Their approach is based on the reserved graph grammar (RGG) formalism (Zhang and Zhang 1997). The RGG formalism is expressed in terms of node-edge diagrams. A node is organized into a two-level hierarchy. A large rectangle is the first level, called a *super-vertex,* with small embedded rectangles as the second level, called *vertices*. Edges are used to denote relationships between nodes. An RGG consists of a set of graph grammar rules, also called *productions*, each having two graphs (the *left graph* and the *right graph*). The RGG offers a translation mechanism, i.e., graph *transformation rules* can specify an input graph of a different graph.

Figure 2.3 shows an example of a mapping represented in RGG and Figure 2.4 shows an example of *ModelGen* by graph transformation rules.

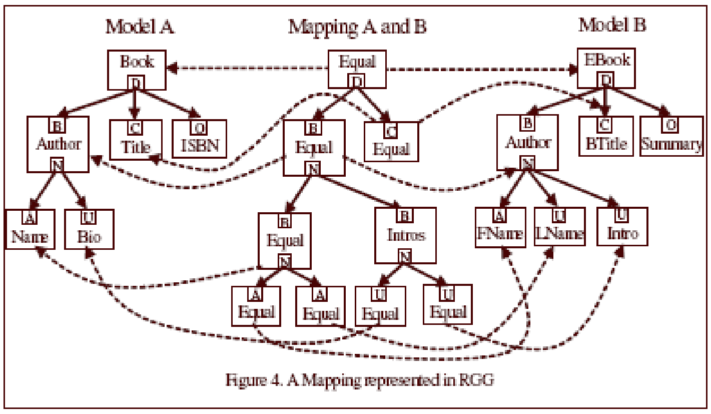


Figure 2.3 Example of a mapping represented in RGG (from Song, Zhang and Kong (2004))

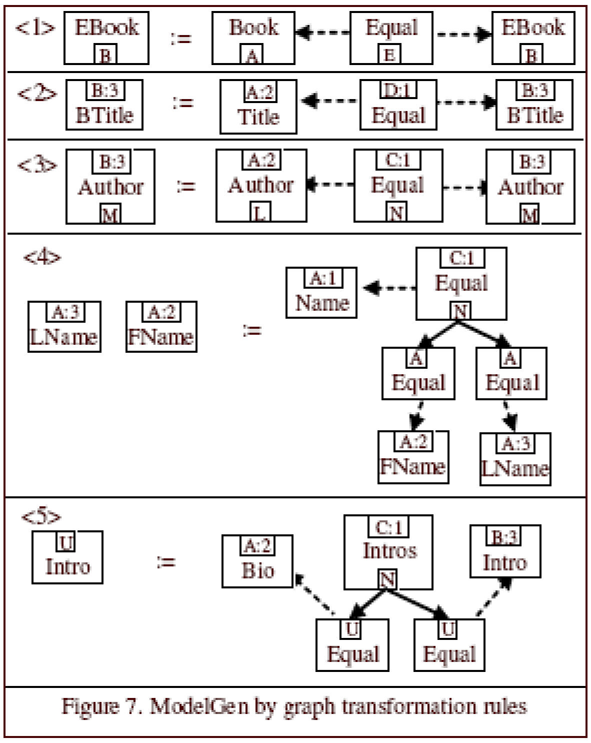

<1> EBook := Book Equal EBook
<2> BTitle := Title Equal BTitle
<3> Author := Author Equal Author
<4> LName FName := Name Equal Equal Equal FName LName
<5> Intro := Bio Intros Intro Equal Equal
Figure 7. ModelGen by graph transformation rules

Figure 2.4 Example of *ModelGen* by graph transformation rules (from Song, Zhang and Kong (2004))

2.3.5 Query/view/transformation (QVT) expressions

QVT is a family of languages for describing model transformations (Object Management Group 2007a), including schema translation mappings. QVT defines a standard way to transform source schemas into target schemas, which are instances of metaschemas that should conform to an arbitrary MOF 2.0 metametaschema.

The QVT language integrates the OCL 2.0 standard and extends it to imperative OCL. Additionally, QVT defines three domain specific languages named *Relations*, *Core* and *Operational Mappings*. These languages are organized in a layered architecture. Relations and Core are declarative languages at two different levels of abstraction, with a normative mapping between them. The Relations language has a graphical concrete syntax, it supports complex object pattern matching, and implicitly creates trace classes and their instances to record what occurred during a transformation execution. Relations can assert that other relations also hold between particular model elements matched by their patterns. The Core language is a small model/language, which only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models. It treats all of the model elements of the source, target and trace models symmetrically. The QVT/OperationalMapping language is an imperative language that extends both QVT/Relations and QVT/Core. The syntax of the QVT/OperationalMappings language provides constructs commonly found in imperative languages (e.g., loops, conditions, etc.).

A translation declaration specifies two parameters for holding the metaschemas involved in the translation. The parameters are types over the appropriate metaschemas. The execution direction is not fixed at translation definition, which means that both metaschemas involved could be source and target metaschemas and vice versa.

Each translation mapping is represented as a *relation*. A relation is defined by the declaration of two or more *domains* and a pair of *when* and *where* predicates. For example:

```
relation ClassToTable
{
  checkonly domain uml c:Class
  {namespace=p:Package{}, kind='Persistent', name=cn}
  checkonly domain rdbms: t:Table {schema=s:Schema{}, name=cn}
  when {
  PackageToSchema(p,s);
  }
  where {
  AttributeToColumn(c,t);
  }
}
```

A *domain* is a distinguished typed variable that can be matched in a model of a given model type. A domain declares a pattern, which is bound with elements from the model to which the domain is bound. Such patterns consist of a variable and a type declaration, which itself may specify some of the properties of that type. A relation can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type.

A *domain* may be invoked for *enforcement* or for *checkonly*. Enforcement of a domain is equivalent to selecting such a domain as the target. The target model may be empty or may contain existing model elements. The execution of the translation of a relation should proceed by first checking whether the relation holds and, if the check fails, by attempting to make the relation hold by creating, deleting or modifying only the target model. A domain is invoked checkonly to check the consistency of both models, i.e., to check that each model is the translation of the other.

A *when* clause specifies the conditions under which the relationship needs to hold. A *where* clause specifies the condition that must be satisfied by all elements participating in the relation, and it may constrain any of the variables in the relation and its domains. The *when* and *where* clauses can contain arbitrary OCL, but are typically expected to contain (if anything) statements about relations satisfied by variables of the domain patterns. Thus, the relation R holds if, for every match of the first domain, there exists a valid match of the second domain such that the *where* clause holds. The *when* clause "specifies the conditions under which the relationship needs to hold." At first sight, both the *when* clause and the *where* clause appear to impose extra conditions on valid matches of bindings, thus forming an intersection of relations. Operationally, the difference between the two is that the variables in the *when* clause "are already bound" "at the time the relation is invoked," and that the conditions in the *where* clause will be satisfied at the end of the invoked relation. Since this difference between the two clauses is not relational, in order to guarantee such executability the expressions occurring in a relation are required to satisfy the following conditions:

- It should be possible to organize the expressions that occur in the *when* clause, the source domains and the *where* clause into a sequential order that contains only the following kinds of expressions:

  a. An expression of the form: <object>.<property> = <variable>

  Where <variable> is a free variable and <object> is either a variable bound to an object template expression of an opposite domain pattern or a variable that gets a binding from a preceding expression in the expression order. This expression provides a binding for the variable <variable>.

  b. An expression of the form: <object>.<property> = <expression>

  Where <object> is either a variable bound to an object template expression of a domain pattern or a variable that gets a binding from a preceding expression in the expression order. There are no free variable occurrences in <expression> (variable occurrences, if any, should all have been bound in the preceding expressions).

  c. No other expression has free variable occurrences (all of their variable occurrences should have been bound in the preceding expressions).

- It should be possible to organize the expressions that occur in the target domain, into a sequential order that contains only the following kinds of expressions:

  a. An expression of the form: <object>.<property> = <expression>

  Where <object> is either a variable bound to an object template expression of the domain pattern or a variable that gets a binding from a preceding expression in the expression order. There are no free variable occurrences in <expression> (variable occurrences, if any, should all have been bound in the preceding expressions).

  b. No other expression has free variable occurrences (all of their variable occurrences should have been bound in the preceding expressions).

2.3.6 Translation schemas[2]

An alternative for specifiying schema translation problems is defined in Bézivin et al. (2006). In this approach, translation mappings can be abstracted as being *translation schemas*. A *translation schema* is nothing more than an ordinary, simple metaschema that includes the source schema, the target schema and the translation mapping expressions between the two (the set of constraints that must be satisfied when two schemas are translations of each other). The basic idea of translation is presented in Figure 2.5 (bottom), where a translation operation Mt takes a schema Ma as the source schema and produces a schema Mb as the target schema. This operation Mt is probably the most important operation in model engineering. Being models, Ma and Mb conform to metamodels MMa and MMb. Usually, the translation Mt has complete knowledge of the source metaschema MMa and the target metaschema MMb. Furthermore, the metaschemas MMa and MMb conform to a metametaschema in this figure, OMG's MOF which in turn conforms to itself.

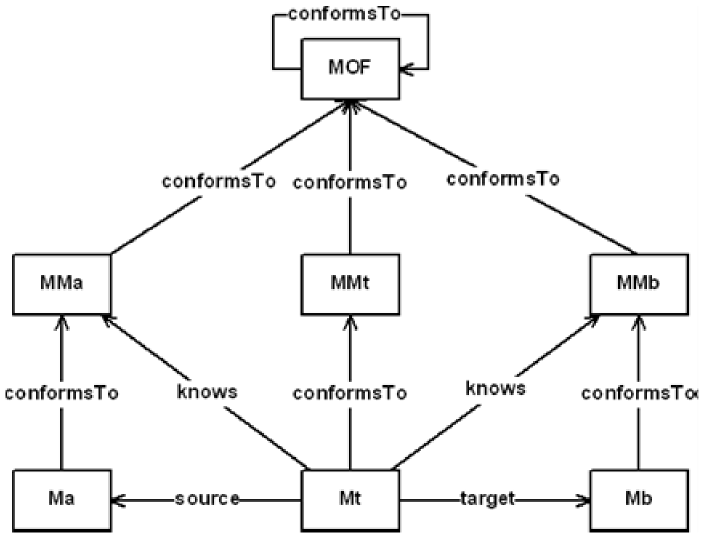


Figure 2.5 Model transformation metamodel MM MMt (from Bézivin et al. (2006))

One of the advantages of this representation is that translation schemas may be seen as translations in multiple directions. This is based on the use of direction-free minimal MOF language features: classes, associations, attributes and invariants. Another advantage is that translation schemas provide uniformity between the schema description language and the language for the translations. Additionally, the uniformity of the schema and translation language also allows for higher-order translations, i.e., translations that work on translations. It also provides the possibility of rewriting translation schemas exactly as if they were ordinary schemas, so refactorings and improvements for general schemas and UML schemas would be applicable. Moreover, standard translation schemas can be validated and checked with standard UML and OCL validation tools.

[2] Called transformation models in Bézivin et al. (2006)

2.3.6.1 *Example: ER2Rel*

Gogolla (2005) and Gogolla et al. (2002) specify the translation between the ER metaschema and the Relational metaschema. The method describes in UML and OCL the ER and Rel metaschemas, i.e., as instances of the MOF metametaschema. The two metaschemas separate the syntax from the semantics part of the two schemas. The description of the syntax of the ER metaschemas includes classes for ER schemas, entities and relationships and the description of the semantics introduces classes for ER states, instances and links. The connection between syntax and semantics is established by associations explaining that syntactical objects are interpreted by corresponding semantic objects. The RE metaschema is described similarly.

Translations between the two languages, as shown in Figure 2.6, are reflected by two associations, namely ErSchema2RelDBSchema and ErState2RelDBState. An ER schema is linked to the Relational database schema that represents the translated schema. Each ER state is associated with the Relational database state representing the same information.

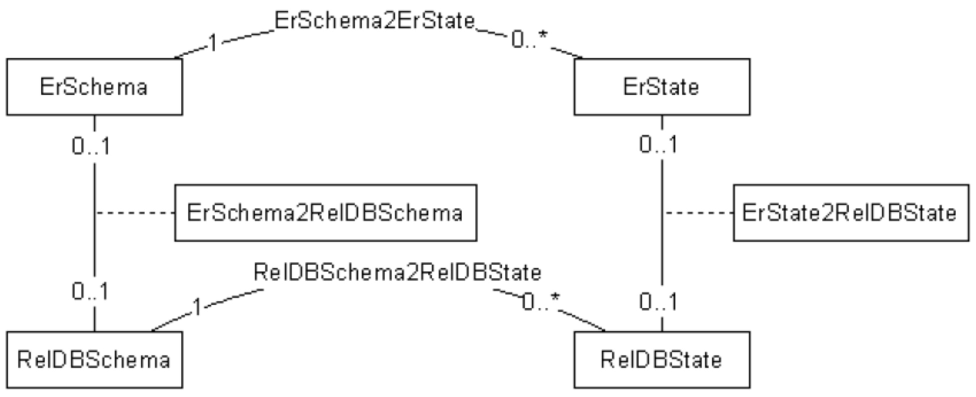


Figure 2.6 ER2Rel metamodel transformation (from Gogolla et al. (2002))

The translation class may directly access the source and target translation schemas. Therefore, semantic properties of the translation are formulated in OCL constraints. For example, the constraint that states that for every entity in the ErSchema there is a RelSchema having the same name and attributes with the same properties (i.e., name, DataType and key property) is represented in OCL as follows:

```
context self:Er2Rel_Trans inv forEntityExistsOneRelSchema:
self.relDBSchema.relSchema -> one(rl|
  e.name = rl.name and
  e.attribute -> forAll(ea| rl.attribute->one(ra|
       ea.name = ra.name and ea.dataType = ra.dataType and
       ea.isKey = ra.isKey))))
```

The ER2Rel transformation model was validated in the OCL tool USE (Gogolla, Büttner and Richters 2007).

2.4 Schema management

When used to solve all the common problems introduced in Section 2.1, ad hoc approaches are not very flexible, clearly very heavy, and hard to maintain. Therefore, a major feature of any significant approach to the problem would be generality: approaches that are maintainable and scalable. Generality requires high-level descriptions of families of problems (not just individual problems) and solutions.

Schema management (Atzeni 2007, Bernstein 2003, Bernstein et al. 2000, Bernstein, Halevy and Pottinger 2000, Bernstein and Melnik 2007, Boronat, Carsí and Ramos 2006, Melnik 2004) is an emerging approach to common problems that arise when managing schemas and mappings. It is based on the representation and management of schemas and mappings between them. The basic idea is to provide a set of operators that are specified in a generic way — that is, independently of any specific schema — to manage schemas and mappings.

Obviously, solutions specified in terms of such operators are not easy to implement. Indeed, a lot of recent research has been devoted to both the precise definition and implementation of the various operators and to the actual clarification of the features of the mapping definition languages.

The rest of this section is organized as follows. First, it describes the families of problems that arise in schema management. Secondly, it reviews a basic set of schema management operators described in the literature. Thirdly, it describes how solutions to two of the families of problems are expressed in terms of such operators. Finally, it briefly presents several examples of approaches or prototypes by implementing, fully or partially, the schema management framework.

2.4.1 Families of problems

Schema management groups most of the common problems found in the tools defined in Section 2.1 into four high-level families of problems: (1) schema transformation, (2) schema integration, (3) schema translation, and (4) propagation of changes between schemas due to evolution.

2.4.1.1 *Schema transformation*

The problem of schema transformation may be defined as follows:

Given a schema S_1, instance of a metaschema MS_1, the goal of schema transformation, also known as schema refactoring, is to obtain a schema $S_1{}'$ that represents the same knowledge as S_1 and is of "better quality" (see Figure 2.7). A variant of the schema transformation problem occurs when the schema S_1 includes instances. In that case, the goal is also to "update" the instances of the target schema.

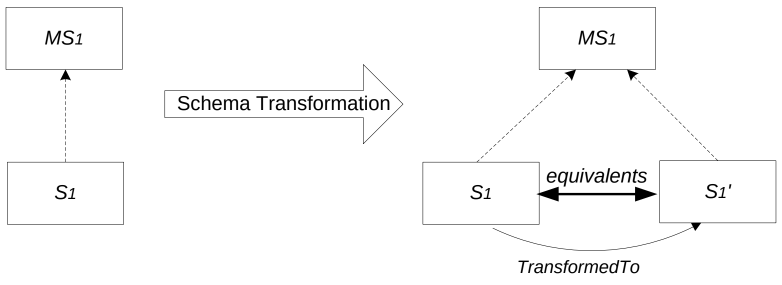


Figure 2.7 The schema transformation problem

2.4.1.2 *Schema integration*

The main objective of schema integration is the construction of a unified schema from a set of independently developed schemas. That is, given two schemas S_1 and S_2 both instances of a metaschema MS, the objective of schema integration is to obtain a third schema S_3, usually called the global schema, which expresses all the knowledge of S_1 and S_2 (see Figure 2.8).

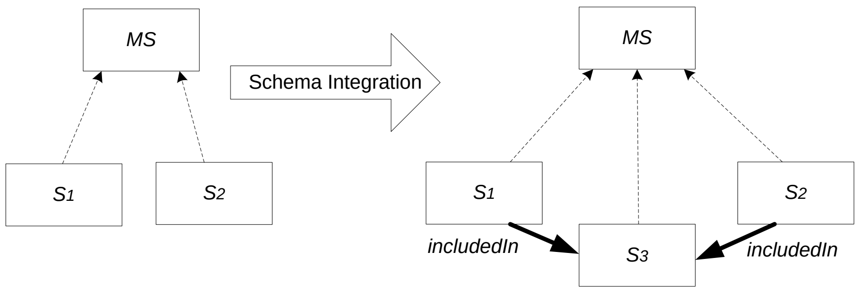


Figure 2.8 The schema integration problem

Many authors consider that schema integration is intended to merge schemas that are instances of the same metaschema. Others consider the alternative of merging schemas that are instances of different metaschemas.

2.4.1.3 *Schema translation*

Schema translation is defined as follows: given a schema S_1, instance of a metaschema MS_1, and a metaschema MS_2, different from MS_1, the goal of schema translation is to obtain a schema S_2, such that it is an instance of MS_2, with both schemas representing the same knowledge. A variant of this problem occurs when the schema S_1 includes instances (data). In that case, the goal is to generate the instance of S_2 that map to the instances of S_1 (see Figure 2.9).

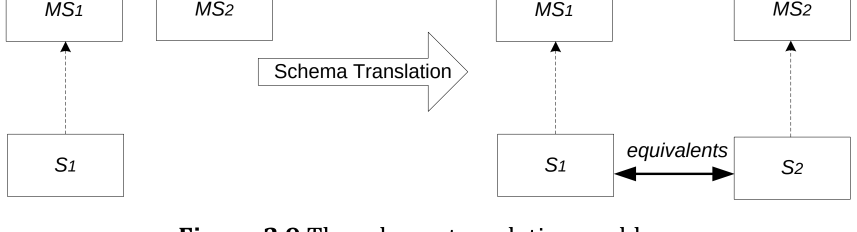


Figure 2.9 The schema translation problem

2.4.1.4 *Propagation of changes due to evolution*

In software engineering, CASE tools are used to generate (translate) lower-level models, and eventually code, from higher-level models. Analogously, reverse engineering tools are used to generate (translate) higher-level models from code or lower-level models. In this context, any evolution or change in a schema must be propagated to the translated schema.

The problem of propagation of changes due to evolution is defined as follows: given two schemas S_1 and S_2, both instances of the metaschemas MS_1 and MS_2, respectively, such that S_2 is a translation of S_1, and S_1 evolves to $S_1{}'$, the goal is to obtain, incrementally, from S_2, the target schema $S_2{}'$, which is an instance of MS_2 and a translation of $S_1^{'}$ (see Figure 2.10). A variant of this problem occurs when the schemas S_1 and S_2 include instances (data). In that case, the goal is to "update" the instances of S_2 accordingly.

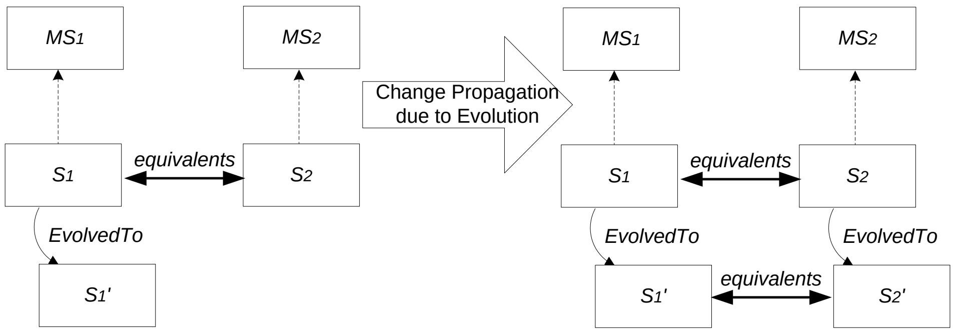


Figure 2.10 The propagation of changes due to evolution problem

2.4.2 Model management operators

In order to solve the aforementioned families of problems, schema management proposes to define schema management operators (named model management operators by Melnik (2004)) that take schemas and mappings as input and produce schemas and mappings as outputs.

The rest of this section provides formal definitions of model (schema), mapping and five basic operators, based on the work of Melnik et al. (Melnik 2004, Melnik et al. 2005). Note that the rest of this section uses the terms model and model management operator to denote the concepts of schema and schema management operator.

According to Melnik, a *model M* is a valid set of instances and a *mapping* is a relation on instances. A binary mapping is a mapping that holds between two models. In general, a mapping is an arbitrary binary relation on instances, which may be total, partial, functional, surjective, etc.

Formally, a model management operator is an n-ary predicate on schemas and mappings. The schema management operators follow the following property:

Operator closure: Let $\mathcal{L}$ be a language for specifying schemas and mappings, and let θ be a model management operator. $\mathcal{L}$ is closed under θ if, given any inputs to θ in $\mathcal{L}$, the outputs can also be expressed in $\mathcal{L}$.

Given two models, m_1 and m_2, and the mapping between them, map, the standard algebraic definitions necessary to define the operators are as follows:

- $m_1 \times m_2 =_{\text{df}} \{(x,y) | x \in m_1 \land y \in m_2\}$
- $\text{Invert}(map) =_{\text{df}} \{(y,x) | (x,y) \in map\}$
- $\text{Domain}(map) =_{\text{df}} \{x | \exists y : (x,y) \in map\}$
- $\text{Range}(map) =_{\text{df}} \text{Domain}(\text{Invert}(map))$
- $\text{Id}(m) =_{\text{df}} \{(x,x) | x \in m\}$

The following well-known properties hold:

- $\text{Domain}(\text{Id}(\text{m})) = \text{m}$
- $\text{Invert}(\text{Invert}(map)) = map$

A basic set of model management operators proposed in the literature is described below:

2.4.2.1 *Match*

Given two models m_1 and m_2, the operator returns a mapping, map, that holds between the two models, denoted map = Match(m_1, m_2). The operator Match inherently does not have formal semantics. It gives the relationship between two models in a particular application context. This relationship can sometimes be discovered semi-automatically (Bernstein et al. 2000), but Match ultimately depends on human feedback and hence may be partial or even inaccurate.

2.4.2.2 *Compose*

Given three models m_1, m_2 and m_3 and two mappings, map_{1-2}, between m_1 and m_2, and map_{2-3} between m_2 and m_3, the composed mapping between map_{1-2} and map_{2-3} is defined as follows (see Figure 2.11):

$$map_{1-2} \circ map_{2-3} =_{\text{df}} \{(x,z) | \exists y : (x,y) \in map_{1-2} \land (y,z) \in map_{2-3}\}$$

The associative property holds:

- $map_{1-2} \circ (map_{2-3} \circ map_{3-4}) = (map_{1-2} \circ map_{2-3}) \circ map_{3-4}$

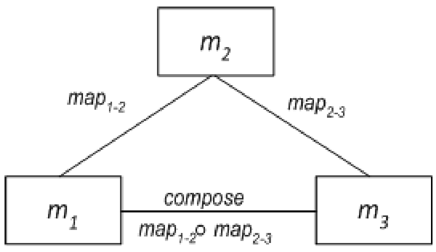


Figure 2.11 Illustration of *Compose*

2.4.2.3 *Merge*

Given two models m_1 and m_2 and the mapping between them, map_{1-2} the operator Merge gives the triple formed by a third model, m_3, and the mappings between $m_1$and m_2 and between m_2 and m_3 (see Figure 2.12):

$(m_3, map_{3-1}, map_{3-2}) = \text{Merge}(m_1, m_2, map_{1-2})$ holds only if:

- map_{3-1} and map_{3-2} are (possibly partial) surjective functions onto m_1 and m_2, respectively.
- $map_{1-2} = \text{Invert}(map_{3-1}) \circ map_{3-2}$.
- $m_3 = \text{Domain}(map_{3-1}) \cup \text{Domain}(map_{3-2})$.

The first condition states that map_{3-1} and map_{3-2} are views on m_3.

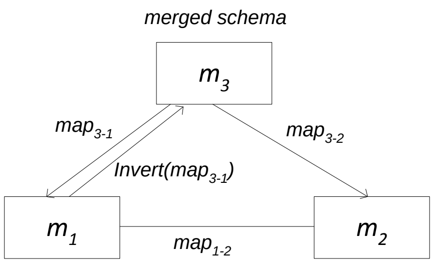


Figure 2.12 Illustration of *Merge*

2.4.2.4 *Diff*

Given two models m_1 and m_2, and the mapping between them, map_{1-2}:

$m_3 = \text{Diff}(m_1, map_{1-2})$ is the submodel of m_1 that does not participate in the mapping (see Figure 2.13).

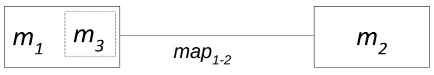


Figure 2.13 Illustration of *Diff*

2.4.2.5 *ModelGen*

Given the model m_1 of a metamodel MM_1 and a different metamodel, MM_2:

$m_2 = \text{ModelGen}(m_1, MM_2)$, where m_2 is an instance of MM_2 and corresponds to (is a translation of) m_1 (see Figure 2.14).

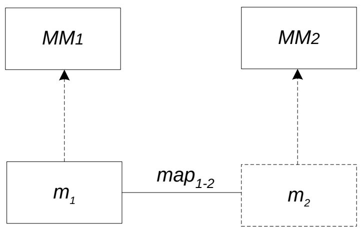


Figure 2.14 Illustration of *ModelGen*

Atzeni (2007) extends ModelGen to the data level, such that given also a database D_1 over the mode m_1, a corresponding database D_2, instance of the model m_2, is generated.

2.4.3 Solutions in terms of the application of model management operators

In this section, solutions to schema translation and change propagation due to evolution problems are defined in terms of the generic operators defined above. These solutions are mainly based on the work of Bernstein, Atzeni and Melnik, among others (Atzeni 2007, Bernstein 2003, Bernstein et al. 2000, Bernstein, Halevy and Pottinger 2000, Bernstein and Melnik 2007, Boronat, Carsí and Ramos 2006, Melnik 2004, Melnik et al. 2005). Note that this section uses the terms *model* and *model management operator* to denote the concepts *schema* and *schema management operator*.

The generic problem of model translation, defined below, can be solved by directly applying the *ModelGen* operator:

Given a model m_1 of a metamodel MM_1, and a different metamodel, MM_2, find the model m_2 of the metamodel MM_2 that is equivalent to m_1, with the following extension: if m_1 has a set of instances, D_1, find also the set of instances of m_2, D_2, that contains the same information as D_1.

1. $m_2 = \text{ModelGen}(m_1, MM_2)$ — where m_2 is an instance of MM_2, equivalent to m_1

2.4.3.1 *Propagation of change due to evolution: solution in terms of model management operators*

The generic problem of propagation of changes due to the evolution of a model is defined as follows:

Given a model m_1, instance of a metamodel MM_1, and a model m_2, instance of a metamodel MM_2, such that m_2 is a translation of m_1, and m_1 evolves to $m_1{}'$, the goal is to obtain, incrementally from m_2 the model $m_2{}'$ that is a translation of $m_1{}'$.

One solution in terms of the generic operators may be the following (see Figure 2.15):

1. $map_{1-1'} = \text{Match}(m_1, m_{1'})$, where $map_{1-1'}$ is the mapping between m_1 and $m_{1'}$ (note that only the elements of $m_{1'}$ that are not changed are mapped)
2. $map_{1'-1} = \text{Match}(m_{1'}, m_1)$, where $map_{1'-1}$ is the mapping between $m_{1'}$ and m_1 (note that only the elements of $m_{1'}$ that are not changed are mapped)
3. $map_{1'-2} = map_{1'-1} \circ map_{1-2}$, where $map_{1'-2}$ is the mapping between $m_{1'}$ and m_2 (some of the elements of $m_{1'}$ may not have a mapping in $map_{1'-2}$)
4. $map_{2-1'} = map_{2-1} \circ map_{1-1'}$, where $map_{2-1'}$ is the mapping between m_2 and $m_{1'}$ (some of the elements of m_2 may not have a mapping in $map_{2-1'}$)

Some elements of m_2 may be "orphans," i.e., they are not incident in $map_{1'-2}$.

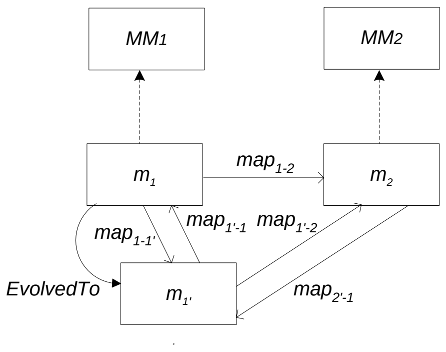


Figure 2.15 Illustration of the propagation of changes due to evolution scenario after the 4th step

Now, to proceed to eliminate the "orphans," first, the difference between m_2 and $m_{1'}$ is calculated, and then these orphans are eliminated from m_2 (see Figure 2.16):

5. $Dim_{1'} m_2, =$ Diff($m_{1'}$ $map_{1'-2}$), where $Dim_{1'} m_2$, are the set of elements of $m_{1'}$ that are not mapped in m_2
6. $Newm_2 =$ ModelGen($Dim_{1'} m_2, MM_2$)
7. $Dim_2 m_{1'}, =$ Diff(m_2 $map_{2-1'}$), where $Dim_2 m_{1'}$, are the set of elements of m_2 that are not mapped in $m_{1'}$
8. $m_{2'} = m_2 + Newm_2 - Dim_2 m_{1'}$

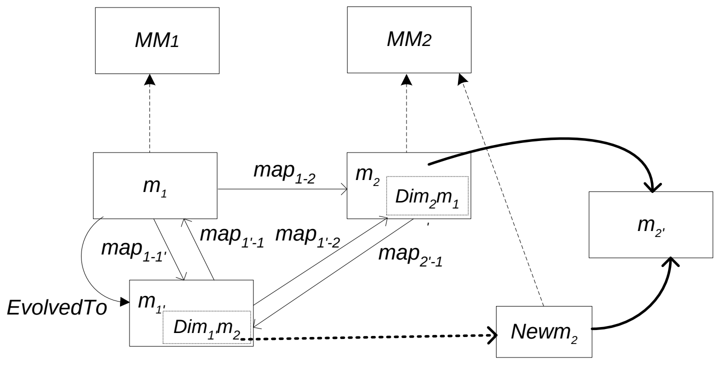


Figure 2.16 Illustration of the propagation of changes due to evolution scenario from the 5th step

2.4.4 Implementations of the model management framework

The following subsections review some prototypes that implement, partially or totally, a model management framework:

2.4.4.1 *Rondo*[3]

Melnik, Rahm and Bernstein, in (Melnik, Rahm and Bernstein 2003), describe an implementation of a model management prototype called Rondo. This prototype supports

[3] Rondo: a musical work that returns to its main theme a number of times (Melnik, Rahm and Bernstein 2003).

the execution of model management scripts that are written using high-level operators, which manipulate models and mappings as first-class objects. A script is a set of steps that may require the intervention of a human engineer; an example of script is the solution described in Section 2.4.3.1 for the propagation of changes due to the evolution problem in terms of the generic operators. Rondo implements all the operators described above except *ModelGen* and offers a graphical user interface for displaying and editing morphisms. A morphism is the simplest specification of mapping, shown as a set of lines connecting two elements of two models (Section 2.3.1 describes morphism in more detail). In the implementation, every intermediate result of a script can be examined and adjusted by a human engineer using a graphical tool. Rondo supports several schema languages, including relational and XML schemas; *morphisms*; and *selectors*, a set of schema elements that make it possible to directly apply operations, with the certainty that they will produce well-formed schemas (e.g., a table with its keys).

The central component of its architecture is an interpreter that executes scripts. It includes an interpreter that is run from the command line or invoked programmatically by external applications and tools. Its main task is to orchestrate the data flow between the operators. The operators can be defined either by providing a native implementation, or by means of schema management scripts. Schemas, morphisms and selectors are represented as structured objects in a common meta-meta-schema and can be stored in a DBMS or file system. The operators are defined in terms of transformations of these structured objects: schema, selector and morphism.

2.4.4.2 *Model-independent schema and data translation (MIDST)*

Atzeni et al., in Atzeni, Cappellari and Bernstein (2006), Atzeni, Cappellari and Gianforme (2007) and Atzeni, Cappellari, Torlone, Bernstein and Gianforme (2008) describe model-independent schema and data translation (MIDST), an implementation of the schema management operators, including *SchemaGen*[4], which translates schemas and their instances from one schema to another.

Schemas and mappings are represented in accordance with the MDM proposal (Atzeni and Torlone 1996). The basic idea is that "constructs" (e.g., class, association, etc) in metaschemas are rather similar. Therefore, in MDM, a metaschema is defined by a set of generic (i.e., model-independent) metaconstructs: *lexical*, *abstract*, *aggregation*, *generalization* and *function*. Each metaschema is defined by its constructs and the metaconstructs they refer to. For example, an abstract corresponds to a class in UML and to a table with its keys in a relational schema; likewise, a function from lexical to abstract corresponds to an attribute of a class in UML and to a column in a relational schema. MDM also introduces the concept of supermetaschema, a metaschema that has constructs corresponding to all the metaconstructs known to the system. Thus, each metaschema is a

[4] Called *ModelGen* in Atzeni, Cappellari & Bernstein (2006) and Atzeni, Cappellari and Gianforme (2007).

specialization of the supermetaschema and a schema in any metaschema is a schema in the supermetaschema. The translation of a schema from one metaschema to another is defined in terms of translations over the metaconstructs. The supermetaschema acts as a "pivot" metaschema, with each metaschema translated to and from the supermetaschema. Moreover, since every schema in any metaschema is an instance of the supermetaschema, a translation is performed by eliminating constructs not allowed in the target metaschema, and possibly introducing new constructs that are allowed.

MIDST includes the following features:

- A dictionary that includes three parts: (i) the meta level, which contains the description of metaschemas and the structure of the metalevel (shown in Figure 2.17) and includes three metaelements: construct, reference and property; (ii) the schema level, which contains the description of schemas; and (iii) the data level, which contains data for the various schemas.
- The elementary translations are also visible and independent of the engine that executes them. They are implemented by rules in a Datalog variant with Skolem functions.
- The translations at the data level are also written in Datalog and are generated almost automatically from the rules for schema translation.
- Mappings between source and target schemas and data are generated, as a by-product, by the materialization of Skolem functions in the dictionary.

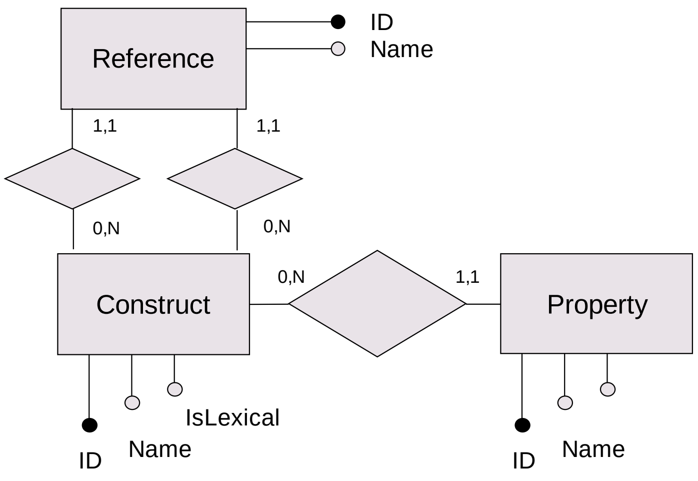


Figure 2.17 The structure of the metadictionary (from Atzeni, Capellari and Bernstein (2005))

2.4.4.3 *Bernstein, Melnik and Mork prototype for interactive schema and data translation*

Bernstein, Melnik and Mork (2005) demonstrate a prototype that translates schemas from a source metaschema to a target metaschema. The prototype is integrated with Microsoft Visual Studio 2005 to generate relational schemas from an object-oriented design.

The system translates schemas by first transforming the source metaschema S into a representation of S_0 in a universal metaschema, like the MDM introduced above (Atzeni, Torlone 1996). A sequence of rule-based transformations then eliminates from S_0 all modeling constructs that are absent from the target metaschema, producing in n steps a schema S_n. Each transformation takes, as input, the current snapshot S_i of the schema and produces, as output, schema S_{i+1} and the mapping m_{i+1} between S_i and S_{i+1}. The final mapping m between S_0 and S_n is obtained by composing the intermediate mappings as $m = m_1 \circ m_2 \circ \ldots \circ m_n$. Finally, S_n is cast into the target metaschema, thereby producing the output schema S'.

The prototype generates instance-level mappings, interactive editing, a general mechanism for dealing with inheritance, and integration with a commercial product featuring a high-quality user interface. Instance-level mappings are computed by composing (Melnik 2004) the elementary data transformations produced upon eliminating each successive modeling construct.

This prototype focuses on flexible mapping of inheritance hierarchies and the incremental regeneration of mappings after the source schema is modified. In constrast, MIDST, introduced in the previous subsection, is driven by a relational dictionary of schemas, models and translation rules.

2.4.4.4 *Papotti and Torlone prototype for translations through XML conversions*

Papotti and Torlone (2005) and Papotti and Torlone (2007), present an approach to the translation of web data between heterogeneous formats. The approach refers to an extension of the *SchemaGen* (*ModelGen*) operation in that it also translates schema instances. Translations operate over XML representations of schemas and instances and consist of a number of steps: (i) the source schema and its instances are converted to plain XML; (ii) the XML schema is translated into an instance of a supermetaschema expressed in an XML-based syntax, similar to the MDM introduced above (Atzeni and Torlone 1996); (iii) the supermetaschema is restructured by translating primitives used in the source metaschema that are not allowed in the target metaschemas; the output of this operation is a schema of the supermetaschema that only uses constructs allowed in the target metaschemas; accordingly, the set of instances are translated into the format coherent with the schema; and (iv) the schema is renamed using the syntax of the target metaschema and finally both the schema and the set of instances are deserialized and delivered to the target system.

Step (iii) is the crucial point of the translation procedure: it takes as input a set of instances and its schema and transforms them into a format suitable for the target metaschema. Since this operation occurs within the supermetaschema, where each primitive represents a class of constructs from different metaschemas, "generic" transformations that are independent of the particular pair of metaschemas can be applied. It follows that it is sufficient to predefine a number of basic transformation that can be composed to build complex translations.

Additionally, Papotti and Torlone (2007) define several properties that characterize the correctness, the consistency, the effectiveness and the efficiency of model translations.

2.4.4.5 *MOdel manageMENT (MOMENT)*

Boronat, Carsí and Ramos (2005a) and Boronat, Carsí and Ramos (2006), describe a framework, called MOMENT (MOdel manageMENT), that is embedded in the Eclipse platform and provides a set of generic operators for dealing with schemas through the Eclipse Modeling Framework (EMF) (Eclipse 2008). Algebra of schema management operators has been specified generically by using the Maude algebraic specification formalism. In MOMENT, a schema transformation can be applied to several source schemas, which may or may not conform the same metaschema. It generates one target schema and a set of traceability schemas. A traceability schema contains a set of traces that relate the elements of the source schema to the elements of the target schema, indicating which transformation rule has been applied to each source element. To apply a transformation to one or more schemas in MOMENT, two criteria must be met:

- The mappings between metaschemas must be defined as schemas of the QVT Relations metaschemas.
- The translation must be invoked by indicating the actual schemas to be transformed in the *SchemaGen* operator.

The same authors, in Boronat, Carsí and Ramos (2006), focus on the design, implementation and execution of the *SchemaGen* operator, which has two formal parameters: the symbol that represents the name of the translation and a polymorphic list of parameters for the translation. The result of the operator is a tuple consisting of the resulting target schema and traceability schemas. There is one traceability schema for each pair (*source schema*, *target schema*).

2.4.4.6 *The transformational approach to database engineering*

The transformational approach developed by Hainaut (2006) is also based on the ground that all transformations, included inter-model transformations, in the fragment of a single model may be studied. The approach defines the Generic Entity-Relationship Model (GER) which is an extended Entity-relationship model that includes, among others, the concepts of schema, entity type, domain attribute, relationship type, keys, as well as various constraints. In GER, a schema is a description of data structures.

Similar to previous approaches, a translation between two schemas involves, basically, the following: (i) the source schema is transformed into GER (the pivot model); (ii) the resulting schema is transformed through a set of rules; and (iii) the transformed schema obtained is expressed into the target model.

The approach defines the set of expressions to express the ER and Relational Models in GER. Additionally, it defines several families of GER transformations: mutation transformations, other elementary transformations, compound transformations, predicate-driven transformations and model driven transformations. About thirty operators have been implemented in DB-MAIN, a programmable CASE environment, which has proved sufficient to process schemas in a dozen operational models.

2.4.4.7 *An intermediate hypergraph data model*

Poulovassilis and McBrien (1998), McBrien and Poulovassilis (1999) and Boyd and McBrien (2005) describe a transformation approach using an special type of graphs, styled intermediate hypergraph data model (IHDM). The data aspects of a conceptual modeling language, such as ER, relational, UML or ORM are modeled as nodes, edges and some predefined constraints (inclusion, exclusion, union, mandatory, unique and reflexive) in HDMs. An example is shown in Figure 2.18.

The approach defines a set of mapping rules to exactly define how the higher level modeling languages are converted into these HDM. It also defines both-as-view (BAV) data integration rules to demonstrate, on the one hand, the equivalence between higher level schemas and, on the other hand, when there is any lost of information in the mapping process.

The work assumes that the schemas must have set-based semantics. The consideration of data types in the schemas has been left for future work, i.e., it is assumed that the data types match in the schemas being compared.

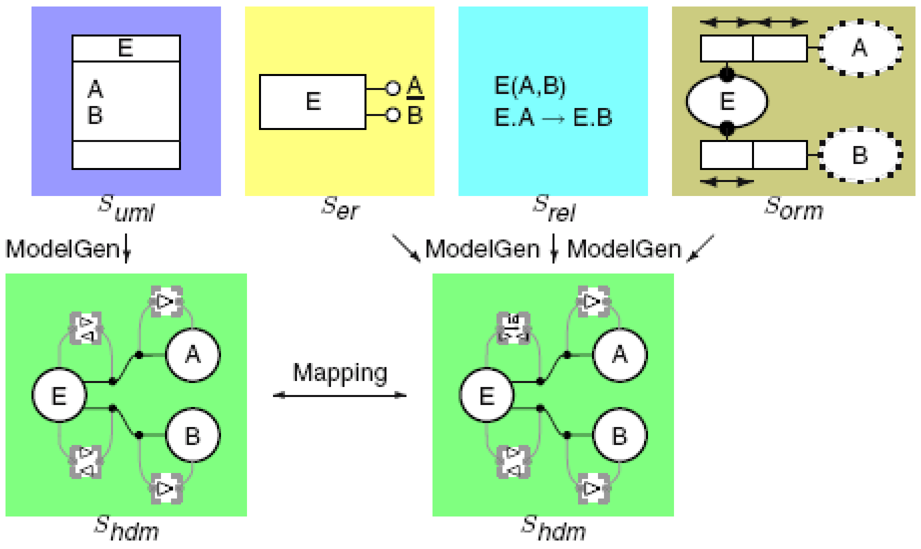


Figure 2.18 Conceptual modelling languages represented in HDM (from Boyd and McBrien (2005))

2.5 Conclusions

The problem of schema translation has been considered for decades in many different contexts. The growing number of languages and tools available to represent domains makes the problem much more complicated and makes it the difficult to find definitive solutions.

Moving away from specific translation tools towards more generic approaches has caused model management to become the alternative framework for solving the schema translation problem. It focuses on the generic description of problems and solutions in terms of generic model management operators. However, there are still some open issues to be solved in the model management framework, most of which are related to the schema mapping definition. These issues include the following:

- **Object-oriented context**. Most of the approaches, which define generic schema mappings, are described in the context of relational data bases. In the object-oriented context, some proposals define schema mappings as constraints. Other approaches use a specific mapping transformation language (such as QVT or ATL) to define schema mappings. In the former case, the definition of additional executable operations is needed to perform translations between schemas. In the latter case, most of the tools are still under development. Object-oriented alternatives that do not need an additional language for the definition of translation schema mappings should be explored.
- **Complex structures**. Schema mappings must be defined among complex structures of elements instead of just among simple elements. In the relational database context, some approaches propose the definition of a limited set of metaconstructs to handle more complex structures and define schema mappings in terms of these metaconstructs. However, the set of metaconstructs proposed is too limited for object-oriented metamodels that have very complex structures. Additional work is needed in the object-oriented context to propose an alternative capable of defining any type of complex structure and defining the mappings in terms of such structures.
- **Quality factors**. There is no uniform definition of the quality factors of translation schema mappings. Not all schemas contain the same knowledge and not all metaschemas cover the same semantic aspects, which makes it difficult to have a unique vision. Substantial work shall be required to explore how to define and prove the correctness, consistency and completeness, among other factors, of schema mappings.

The current literature does not propose an approach in the context of the object-oriented paradigm that solves all the aforementioned issues. The main goal of this research is to provide a generic object-oriented schema translation approach by solving these issues.

3 A generic object-oriented operation-based approach to the translation between MOF metaschemas

This chapter proposes a new approach to the schema translation problem. It deals with schemas whose metaschemas are instances of the OMG's Meta Object Facility (MOF). Most metaschemas can be defined as an instance of the MOF; therefore the approach is widely applicable. The well-known object-oriented concepts embedded in the MOF and its instances (object type, attribute, relationship type, operation, *IsA* hierarchies, refinements, invariants, pre-, postconditions, etc.) are leveraged to define metaschemas, schemas and their translations.

The main contribution of the approach is the extensive use of object-oriented concepts in the definition of translation mappings, particularly the use of operations (and their refinements) and invariants, both of which are formalized in OCL. The translation mappings can be used to check that two schemas are translation of each other, and to translate one into the other, in both directions. The translation mappings are declaratively defined by means of pre-, postconditions, and invariants, and they can be implemented in any suitable language. From an implementation point of view, by taking a MOF-based approach there are available a wide set of tools, including tools that execute OCL. As an example, the approach has been specified in the UML-based Specification Environment (USE) tool (Gogolla, Büttner and Richters 2007), already described in the first chapter.

The main aspects of this approach are as follows:

- Metaschemas are represented as instances of the OMG's MOF (Object Management Group 2006a). UML is, of course, an instance of the MOF, and almost all metaschemas can be defined as instances of it.
- Elementary translations are represented by means of operations hosted in object types, formalized in the OCL language (Object Management Group 2006b).

- The well known object-oriented concepts embedded in the MOF and its instances (object type, attribute, relationship type, operation, *IsA* hierarchies, refinements, invariants, pre-, postconditions, etc.) are leveraged to define metaschemas, schemas and their translations.

The rest of this chapter is structured as follows:

- Section 3.1 explains the main concept of the approach, the schema unit, and defines translation mappings as mappings between schema units.
- Section 3.2 explains how to define schema units in a MOF metaschema. This definition must be done only once per metaschema, since it is independent of the mapping translations.
- Section 3.3 explains how to define translation mapping expressions between any two MOF metaschemas and their use. These expressions can be defined by two invariants involving the relationships between the schema units of the two metaschemas. The two invariants are defined in OCL, and the relationships between schema units are defined by means of operations whose pre- and postconditions are formalized also in OCL. The sections ends describing how to use the operations defined in the previous sections in order to automatically translate between instances of the two metaschemas.

Throughout this chapter two small fragments of the ER and the Relational metaschemas are used as running example, similar to those used in (Gogolla 2005). The interested reader may find, in (Raventós 2008a), three simple examples of the complete application of the method: the one described in this chapter, the larger example of the UML and Relational metaschemas described in the QVT specification (Object Management Group 2007a) and the translation to ER of the large osCommerce[5] relational database.

3.1 Basic concepts

This section describes the concepts of schema, mapping and translation and explain the notation system used in this chapter. The example that is used throughout this chapter is also introduced.

3.1.1 Schema and mapping

A *schema S* is a valid instance of a metaschema *MS*. An instance of a (meta)schema *MS* is valid if it satisfies all the integrity constraints defined in *MS*. In turn, a metaschema *MS* is a valid instance of a meta metaschema *MMS* (Olivé 2007). In this paper, metaschemas are instances of the MOF, which is a meta-metaschema standardized by the OMG (Object Management Group 2006a). Therefore, an MOF *schema* is an instance of an MOF metaschema. Most metaschemas can be defined as an instances of the MOF. This chapter deals with UML, ER and the relational model, all of which can be defined as instances of

[5] www.oscommerce.com

the MOF. Figures 3.1(a) and 3.2(a) show, in UML, the fragments of the ER and relational metaschemas that will be used as examples.

Note that all the constraints included in the two metaschemas (e.g., uniqueness of names) have been formally specified in (Raventós 2008a).

In general, a *schema mapping* is a triple $M = (S_1, S_2, \Sigma)$ where S_1 is the source schema, S_2 is the target schema and Σ, called the *mapping expression,* is a set of constraints over S_1 and S_2. An instance of mapping M is a pair $\langle s_1, s_2 \rangle$ such that s_1 is an instance of S_1, s_2 is an instance of S_2 and the pair s_1, s_2 satisfies all the constraints Σ (Fagin et al. 2005).

This thesis is concerned with mappings between metaschemas; therefore, the mappings take the form $M = (MS_1, MS_2, \Sigma)$, where MS_1 is the source metaschema, MS_2 is the target metaschema and Σ is a set of constraints on MS_1 and MS_2. An instance of the mapping M is a pair $\langle S_1, S_2 \rangle$ such that S_1 is an schema that is an instance of MS_1, S_2 is a schema that is an instance of MS_2 and the pair S_1, S_2 satisfies all the constraints Σ. The mapping expression of most metaschema mappings is very long and complex. This thesis presents a new approach to the definition of mapping expressions that is based on the concept of a schema unit as defined below.

3.1.2 Schema units

Syntactically, a schema S is a valid set of instances of the entity types, relationship types and attributes defined in MS. This set of instances is called the *schema elements* of S. However, by focusing more on the semantics of the schemas than on their syntactical expression, the concept of a *schema unit* is defined. The schema units are the knowledge components of the schemas. A schema consists of a set $S = \{u_1, \dots, u_n\}$ of schema units u_i, such that the knowledge expressed by S is the set of knowledge components expressed by its schema units $u_1, \dots, u_n$.

In general, a schema unit is a concept type (entity or relationship type), a constraint or a derivation rule. For example, an entity type of an ER schema, a foreign key of a relational schema, and an OCL derivation rule of a derived attribute in a UML schema are three schema units (concept type, constraint and derivation rule, respectively). In some cases, a schema unit is an aggregation of concept types, constraints and/or derivation rules. For example, an association schema unit in a UML includes the cardinality constraints of its member ends.

Syntactically, a schema unit u is a set of schema elements such that:

- it can be added to a schema S when certain conditions are satisfied, and
- $S \cup \{u\}$ is a valid instance of MS.

The rationale behind this definition is that the knowledge expressed by a schema $S = \{u_1, \dots, u_i\}$ can be extended with a new schema unit u_{i+1}, for which $S' = \{u_1, \dots, u_i, u_{i+1}\}$ is obtained. In general, a schema unit can only be added to an existing schema if certain conditions are satisfied. For example, in an ER schema S, a relationship type (schema unit) can be added if the participant entity types are already part of S.

The idea of the schema unit is implicit or explicit in many schema translation approaches and, in some cases, a distinction is made between different kinds of schema units. For example, Boyd and McBrien (2005) distinguishes between nodal, link, link nodal and constraint. Languages with a rich set of predefined constraints have many kinds of schema units (Jarrar (2007) distinguishes about 30 kinds in the ORM – Description Logics mapping).

In this approach, it is required that each schema unit u of S be represented by an instance of some meta-entity type of MS. An instance of a meta-entity type of M can represent one schema unit at most, but not all instances of the meta-entity types of MS need represent schema units.

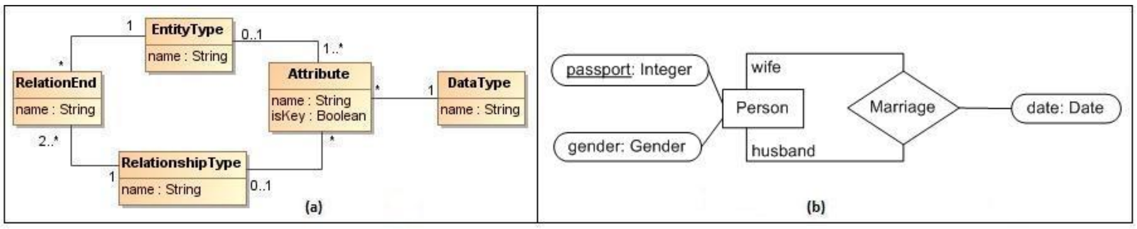


Figure 3.1 Fragment of the ER metaschema (a), and an example of one of its instances (b) (Gogolla 2005)

3.1.2.1 *Application to the ER metaschema*

In an ER schema, the schema units are entity types, relationship types, attributes and data types. These schema units are represented differently in different metaschemas. In UML, they may be represented as shown in Figure 3.1(a):

- Each ER entity type is represented by an instance of *EntityType*. The schema elements of an entity type named *e* are as follows: (1) an instance α of *EntityType*; (2) the instance of attribute *name* of α with value *e*; (3) the (one or more) instances of *Attribute* related to α whose *isKey* attribute has the value *True*; and for each of these attributes: (a) the instances of its attributes *name* and *isKey*; (b) the instance of its relationship with α; and (c) the instance of its relationship with the corresponding data type. If, for example, an entity type *e* has only one key attribute, then the schema elements of *e* makes up a set of seven instances. Note that an entity type and all its key attributes are grouped into a single schema unit.
- Each ER relationship type is represented by an instance of *RelationshipType*. The schema elements of a relationship type named *r* are as follows: (1) an instance β of *RelationshipType*; (2) the instance of attribute *name* of β with value *r*; (3) the (two or more) instances of *RelationEnd* related to β; and (4) for each of these relation ends: (a) the instances of its attribute *name*; (b) the instance of its relationship with β; and (c) the instance of its relationship with the corresponding entity type. Note that it is assumed that the cardinalities of the participants are unconstrained.

- Each ER data type is represented by an instance of *DataType*. The schema elements of a data type named *d* are as follows: (1) an instance λ of *DataType*; and (2) the instance of attribute *name* of λ wose value is *d*.
- Each ER attribute that is not a key of an entity type is represented by an instance of *Attribute* whose *isKey* attribute has the value *False*. The schema elements of an attribute named *a* are as follows: (1) an instance μ of *Attribute*; (2) the instance of attribute *name* of μ whose value is *a* and *isKey* whose value is *False*; (3) the instance of its relationship with an instance of *EntityType* or *RelationshipType*; and (4) the instance of its relationship with the corresponding data type.

In the ER schema example shown in Figure 3.1(b) there are seven schema units: three instances of *DataType*, one instance of *EntityType*, one instance of *RelationshipType*, and two instances of *Attribute*. These schema units are shown in the left part of Figure 3.3(b).

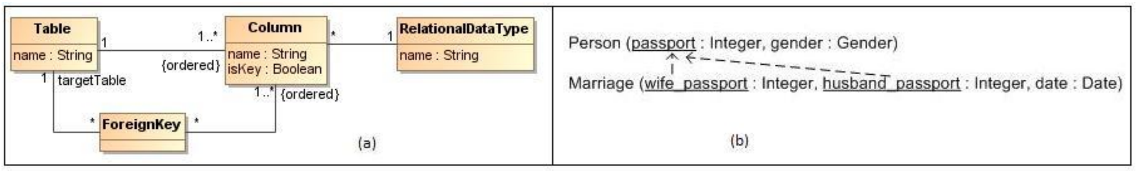


Figure 3.2 Fragment of the Relational metaschema (a), and an example of one of its instances (b) (Gogolla 2005)

3.1.2.2 *Application to the relational metaschema*

In a relational schema, the schema units are tables, columns, foreign keys and data types. These schema units are represented differently in different metaschemas. In UML, they may be represented as shown in Figure 3.2(a):

- Each relational table is represented by an instance of *Table*. The schema elements of a table named *t* are as follows: (1) an instance τ of Table; (2) the instance of attribute *name* of τ whose value is *t*; (3) the (one or more) instances of *Column* related to τ whose *isKey* attribute has the value *True*; and (4) for each of these columns: (a) the instances of its attributes *name* and *isKey*; (b) its relationship with τ; (c) and its relationship with the corresponding data type. Note that a table and all its key columns have been grouped into a single schema unit.
- Each relational data type is represented by an instance of *RelationalDataType*. The schema elements of a data type named *d* are as follows: (1) an instance γ of *RelationalDataType*; and (2) the instance of attribute *name* of γ whose value is *d*.
- Each relational column that is not a key of a table is represented by an instance of *Column* whose *isKey* attribute has the value *False*. The schema elements of a column named *c* are as follows: (1) an instance η of Column; (2) the instance of attribute *name* of η with value *c* and *isKey* with value *False*; (3) its relationship with an instance of *Table*; and (4) its relationship with the corresponding instance of *RelationalDataType*.

- Each foreign key is represented by an instance of *ForeignKey*. The schema elements of a foreign key *fk* are as follows: (1) an instance of *ForeignKey*; (2) the relationships of *fk* with *Column* that give the columns that comprise *fk*; and (3) the relationship of *fk* with the table referenced by the columns of *fk*.

In the relational example shown in Figure 3.2(b) there are nine schema units: three instances of *RelationalDataType*, two instances of *Table*, two instances of *Column*, and two instances of *ForeignKey*. These schema units are shown in the right part of Figure 3.3(b).

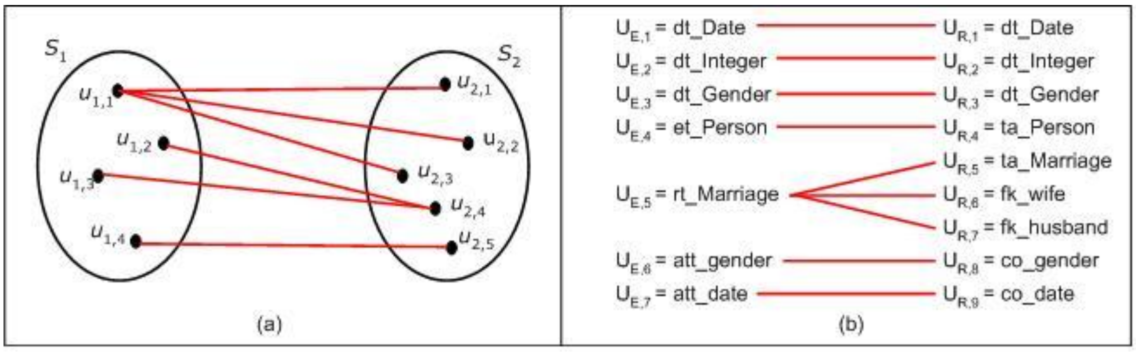


Figure 3.3 Abstract example of equivalences and inclusions (a), and their application to the schema examples (b)

3.1.3 Translation mapping

Let $S_1 = \{u_{1,1}, \dots, u_{1,n}\}$ and $S_2 = \{u_{2,1}, \dots, u_{2,m}\}$ be two schemas. S_1 and S_2 are translation of each other if the knowledge they express is the same. In other words, S_1 and S_2 are translations of each other if the knowledge expressed by their schema units $\{u_{1,1}, \dots, u_{1,n}\}$ and $\{u_{2,1}, \dots, u_{2,m}\}$ is the same. This means that there is a total and surjective relation $r \subseteq S_1 \times S_2$ that maps each schema unit of S_1 to its equivalent units in S_2, and the other way around.

In most cases the relation r satisfies the equivalence/inclusion constraint, which means that for each $r \in S_1$ at least one of the two following conditions hold (see Figure 3.3):

- $u_{1,i}$ is completely equivalent to a set $\{u_{2,1}, \dots, u_{2,k}\}$ of one or more schema units of S_2. This is, there is an equivalence mapping between $u_{1,i}$ and $\{u_{2,1}, \dots, u_{2,k}\}$.
- $u_{1,i}$ is completely included in a schema unit $u_{2,k}$ of S_2. In this case, there is an inclusion mapping between $u_{1,i}$ and $u_{2,k}$.

Formally,

$$\forall u_{1,i}, u_{1,j}, u_{2,k}, u_{2,l}\ ((u_{1,i}, u_{2,k}) \in r \wedge u_{1,j} \neq u_{1,i} \wedge u_{2,l} \neq u_{2,k} \rightarrow \neg\, ((u_{1,j}, u_{2,k}) \in r\ \wedge (u_{1,i}, u_{2,l}) \in r))$$

Note that if there is an equivalence mapping between $u_{1,i}$ and $\{u_{2,1}, \dots, u_{2,k}\}$ then there is an inclusion mapping between $u_{2,i}$ and $u_{1,i}$ for all $i = 1..k$.

In the abstract example in Figure 3.3(a), $u_{1,1}$ is completely equivalent to the set $\{u_{2,1}, u_{2,2}, u_{2,3}\}$; $u_{1,2}$ is completely included in $u_{2,4}$; $u_{1,3}$ is completely included in $u_{2,4}$; and $u_{1,4}$ is both completely equivalent to $u_{2,5}$ and completely included in $u_{2,5}$.

As a specific example, Figure 3.3(b) shows that the relationship type *Marriage* in the ER schema shown in Figure 3.1(b) is completely equivalent to the table *Marriage* and the two foreign keys of the relational schema shown in Figure 3.2(b).

The term "equivalent" has several meanings in the schema management field (Lie 1982, Hull 1986, Papotti and Torlone 2007), so the meaning must be specified in each case. In this approach, a schema unit $u_{1,i}$ is completely equivalent to a set $\{u_{2,1}, \dots, u_{2,k}\}$ when the following conditions hold:

- If $u_{1,i}$ may have instances (i.e., it is an entity or a relationship type) then the population of $u_{1,i}$ at any time can be obtained from the populations of $u_{2,1}, \dots, u_{2,k}$ at that time, and the other way round.
- If $u_{1,i}$ constrains the instances of S_1 then $u_{2,1}, \dots, u_{2,k}$ constrain the equivalent instances of S_2 in the same way.
- If $u_{1,i}$ derives the instances of S_1 then$u_{2,1}, \dots, u_{2,k}$ derive the equivalent instances of S_2 in the same way.

Let $M = (MS_1, MS_2, \Sigma)$ be a mapping. We say that M is a *translation mapping* when for any S_1 and S_2 such that $\langle S_1, S_2 \rangle$ is an instance of M then S_1 and S_2 are translation of each other. Therefore, in a translation mapping, the set of constraints Σ is satisfied only when the two schemas are translation of each other. In the next section we present an approach to defining translation mapping expressions that is based on the concept of a schema unit.

A translation mapping $M = (MS_1, MS_2, \Sigma)$ may be (Melnik 2004):

- Total: for all S_1 that are an instance of MS_1 there is at least one instance S_2 of MS_2 such that $\langle S_1, S_2 \rangle$ is an instance of M.
- Surjective: for all S_2 that are an instance of MS_2 there is at least one instance S_1 of MS_1 such that $\langle S_1, S_2 \rangle$ is an instance of M.
- Functional: for all S_1 that are an instance of MS_1 there is exactly one instance S_2 of MS_2 such that $\langle S_1, S_2 \rangle$ is an instance of M.
- Injective: for all S_2 that are an instance of MS_2 there is exactly one instance S_1 of MS_1 such that $\langle S_1, S_2 \rangle$ is an instance of M.
- Bijective: if it is total, surjective, functional, and injective.

The translation mapping between MS_1 = the ER metaschema in Figure 3.1(a) and MS_2 = the relational metaschema in Figure 3.2(a) may be total because for each ER schema there is at least one translation into an instance of MS_2. However, that mapping cannot be surjective because there are instances of MS_2 that cannot be adequately translated into an instance of MS_1. For example, MS_2 allows the representation of foreign keys that are not referential integrity constraints and cannot be represented in MS_1. Section 3.3.1 shows how, in this approach, that a schema unit cannot be translated into another metaschema is specified.

3.2 Defining the schema units of MOF schemas

The basic concept in the approach is the schema unit. This section explains how the schema units of MOF schemas should be defined. Let MS_i be an MOF metaschema. It is practical to assume that MS_i has a root entity type, called S_i*Element* such that all entity types of MS_i that may represent schema units are a direct or indirect subtype of S_i*Element*. The entity type S_i*Element* is derived by the union of its subtypes (also called abstract entity types in UML). Figures 3.4 and 3.5 show the definition of the root entity types (called *ErElement* and *RelationalElement*) in the ER and relational metaschemas in Figures 3.1 and 3.2, respectively.

S_i*Element* has two operations (*isSchemaUnit()* and *predecessors()*) that are used to define the schema units and their precedence relationships. Each schema unit is characterized by a special object, called a *schema unit characterization object*, which among other things defines the schema elements that make up a schema unit. The operations and the characterization objects are mapping-independent; therefore they are defined only once in a metaschema. In what follows it is showed how the operations and the characterization objects are defined. The explanations are illustrated by applying them to the ER and relational metaschemas.

3.2.1 isSchemaUnit() operation

In S_i*Element* is defined the query operation *isSchemaUnit*():*Boolean* whose value indicates whether or not an instance of S_i*Element* represents a schema unit. As stated above, the value of this operation is mapping-independent. In the context of S_i*Element* the operation can only give a default value (either *True* or *False*), and each subtype *ST* of S_i*Element* such that some or all of its instances represent schema units, redefines it (if necessary) to indicate whether or not the corresponding instance of *ST* represents a schema unit. It is not mandatory that all instances of *ST* have the same value for the operation *isSchemaUnit()*.

3.2.1.1 *Application to the ER metaschema*

In the ER metaschema of Figure 3.4, is defined[6]:

```
context ErElement::isSchemaUnit():Boolean
  body: True
```

[6] In UML, the body condition for an operation constrains the return result. The body condition differs from postconditions in that the body condition may be overridden when an operation is redefined, whereas postconditions can only be added during redefinition (Object Management Group 2006a, p. 107).

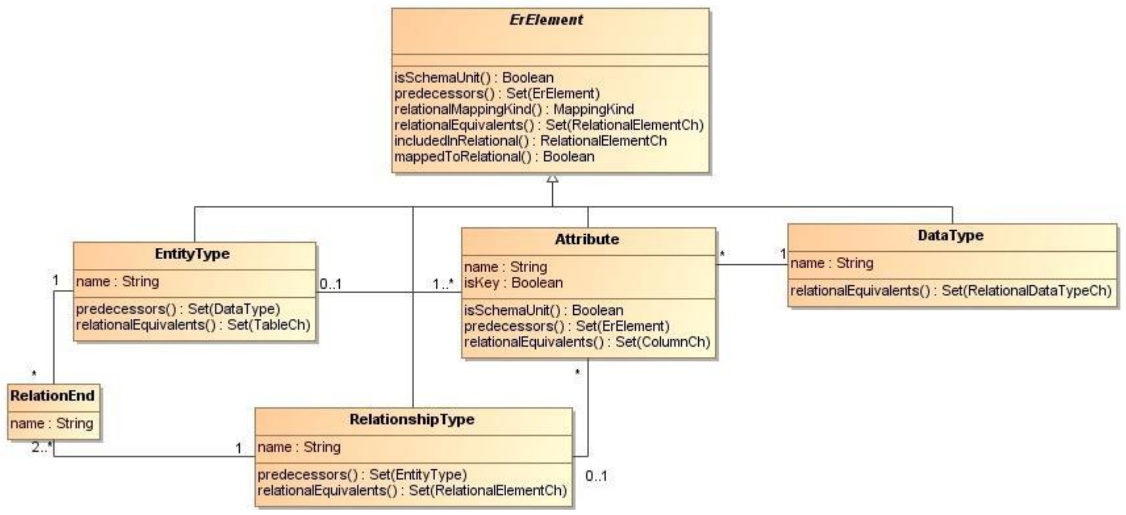

ErElement
isSchemaUnit() : Boolean
predecessors() : Set(ErElement)
relationalMappingKind() : MappingKind
relationalEquivalents() : Set(RelationalElementCh)
includedInRelational() : RelationalElementCh
mappedToRelational() : Boolean
EntityType
name : String
predecessors() : Set(DataType)
relationalEquivalents() : Set(TableCh)
Attribute
name : String
isKey : Boolean
isSchemaUnit() : Boolean
predecessors() : Set(ErElement)
relationalEquivalents() : Set(ColumnCh)
DataType
name : String
relationalEquivalents() : Set(RelationalDataTypeCh)
RelationEnd
name : String
RelationshipType
name : String
predecessors() : Set(EntityType)
relationalEquivalents() : Set(RelationalElementCh)
1
0..1
1..*
*
1
*
*
2..*
1
0..1

Figure 3.4 Definition of ErElement

This means that by default all (direct or indirect) instances of *ErElement* are schema units. There is an exception: not all instances of *Attribute* are schema units, but only those that are not keys. Therefore, the above operation is redefined as follows:

```
context Attribute::isSchemaUnit():Boolean
  body: not isKey
```

This is an example of an entity type in which some instances represent schema units and other do not. Note that *RelationEnd* (Figure 3.1) is not defined as a subtype of *ErElement* (Figure 3.4). This avoids to redefine that instances of *RelationEnd* are not schema units:

```
context RelationEnd::isSchemaUnit():Boolean
   body: False
```

3.2.1.2 *Application to the Relational metaschema*

Similarly, for the relational metaschema in Figure 3.5, is defined:

```
context RelationalElement::isSchemaUnit():Boolean
  body: True
```

This means that by default all (direct or indirect) instances of *RelationalElement* are schema units. There is an exception similar to the previous one: not all instances of *Column* are schema units, but only those that are not keys. Therefore, the above operation is redefined as follows:

```
context Column::isSchemaUnit():Boolean
   body: not isKey
```

3.2.2 Predecessors

A schema consists of a set $S = \{u_1, \dots, u_n\}$ of schema units u_i, but in general there are precedence relationships between them. Very often one can add a schema unit to a schema only when other units have already been defined. Those schema units that are direct predecessors of u_i, are called *predecessors* units. A schema unit cannot be a direct or indirect predecessor to itself. It is not difficult to define the predecessors of a

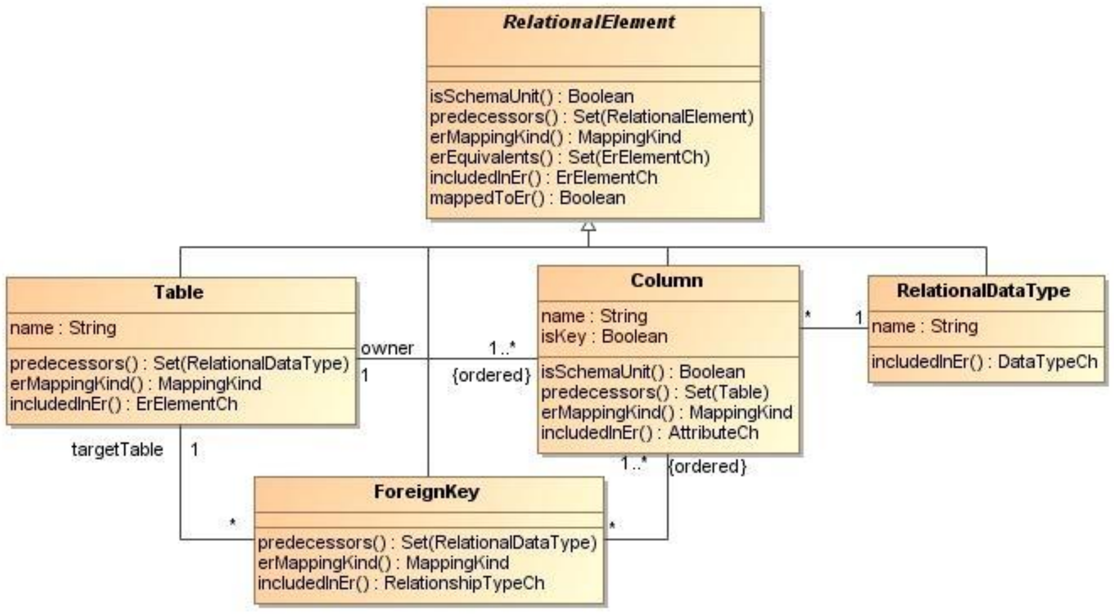

RelationalElement
isSchemaUnit() : Boolean
predecessors() : Set(RelationalElement)
erMappingKind() : MappingKind
erEquivalents() : Set(ErElementCh)
includedInEr() : ErElementCh
mappedToEr() : Boolean
Table
name : String
predecessors() : Set(RelationalDataType)
erMappingKind() : MappingKind
includedInEr() : ErElementCh
Column
name : String
isKey : Boolean
isSchemaUnit() : Boolean
predecessors() : Set(Table)
erMappingKind() : MappingKind
includedInEr() : AttributeCh
RelationalDataType
name : String
includedInEr() : DataTypeCh
ForeignKey
predecessors() : Set(RelationalDataType)
erMappingKind() : MappingKind
includedInEr() : RelationshipTypeCh
owner
1
1..*
{ordered}
*
1
targetTable
1
*
1..*
{ordered}
*

Figure 3.5 Definition of RelationalElement

schema unit, and they are very important in the specification of the translation process, as is shown in Section 3.3.6.

In the context of *S_iElement* the query operation *predecessors*() can only give a default value (the empty set):

```
context SᵢElement::predecessors():Set(SᵢElement)
  pre: isSchemaUnit()
  body: Set{}
```

However, each subtype *ST* of *S_iElement* such that some or all of its instances represent schema units redefines it (if necessary) to indicate its predecessor schema units. Note that the precondition specifies that *predecessors*() can be invoked (i.e., make sense) only in schema units.

3.2.2.1 *Application to the ER metaschema*

In the ER metaschema in Figure 3.4 is defined the following *predecessors()* operation:

```
context ErElement::predecessors():Set(ErElement)
  pre: isSchemaUnit()
  body: Set{}
```

This means that, by default, all schema units do not have predecessors. This is the case of *DataType*; therefore, there is no need to redefine the operation for it.

For *EntityType* the *predecessors()* operation is redefined as follows:

```
context EntityType::predecessors():Set(DataType)
  body: attribute -> select(isKey).dataType
```

This means that the predecessors of an entity type are the data types of its key attributes. For example, in the ER schema in Figure 4.1(b), the set of predecessors of the unit

$u_{E,4} = et\_Person$ is $\{u_{E,2} = dt_{Integer}\}$ (see Figure 4.3(b, left)). The predecessors of a relationship type are its participant entity types. This is formally defined as follows:

```
context RelationshipType::predecessors():Set(EntityType)
  body: relationEnd.entityType
```

Finally, the predecessors of a non-key attribute are its entity or relationship type and its data type. This is formally defined as follows:

```
context Attribute::predecessors():Set(ErElement)
  body: let type:ErElement =
          if entityType -> notEmpty() then entityType
          else relationshipType
          endif
          in Set{type,dataType}
```

3.2.2.2 *Application to the relational metaschema*

In the relational metaschema in Figure 3.5 is defined as follows:

```
context RelationalElement::predecessors():Set(RelationalElement)
  pre: isSchemaUnit()
  body: Set{}
```

This means that, by default, all schema units do not have predecessors. This is the case of *RelationalDataType*; therefore, there is no need to redefine the operation for it. For *Table* the *predecessors()* operation is redefined as follows:

```
context Table::predecessors():Set(RelationalDataType)
  body: column -> select(isKey).relationalDataType
```

This means that the predecessors of table are the data types of its key columns. The predecesssor of a non-key column is its table and its data type is formally defined as follows:

```
context Column::predecessors():Set(RelationalElement)
  body: Set{table, relationalDataType}
```

Finally, the predecessors of a foreign key are its non-key columns and the source and target tables are formally defined as follows:

```
context ForeignKey::predecessors():Set(RelationalElement)
  body: column -> select(not iskey)->asSet() ->
          union(column.table->asSet()) -> including{targetTable}
```

For example, in the relational schema in Figure 3.2(b), the set of predecessors of the unit $u_{R,6} = fk\_wife$ is $\{u_{R,5} = ta\_Marriage, u_{R,4} = ta\_Person\}$ (see Figure 3.3(b, right)).

3.2.3 Characterization objects

Translation mapping constraints are complex because they define relationships between two metaschemas that are usually themselves very complex. The relationships must take into account the details of how each schema unit is represented in its own metaschema. This thesis proposes an alternative that consists in using an indirection mechanism: each schema unit is characterized by an object (called a *characterization* object). The intuitive idea is to split a translation relationship between two sets of schema elements (one in each metaschema) that represent two schema units $u_{1,i}$ and $u_{2,k}$ into two simpler parts:

one between the schema elements of $u_{1,i}$ and the characterization object of $u_{2,k}$ and another between the characterization object of $u_{2,k}$ and its schema elements.

Characterization objects roughly correspond to the well-known value or domain value objects in the object-oriented design patterns field (Riehle 2006). In each metaschema, is defined a characterization object type for each subtype *ST* of S_i*Element* such that some or all of its instances represent schema units. For the sake of simplicity, the characterization object types are named by adding the suffix *Ch* to the corresponding name of the metaschema type. Each characterization object type includes a set of attributes that characterize the schema unit and two operations: *createUnit*() and *schemaUnit*(). The first operation creates a schema unit from its characterization object, and the second creates the schema unit corresponding to the characterization object, if it exists.

It is assumed that the operation *createUnit()* will only be invoked when the predecessors of the schema unit it creates have already been created. For example, the *createUnit()* of an instance of *RelationshipTypeCh* will only be invoked after the creation of the participant entity types.

The specification of the first operation is the same for all characterization object types; therefore, it is defined it in a general S_j *ElementCh*:

```
context SjElementCh::createUnit()
  post: schemaUnit()->notEmpty()
```

This operation ensures that in the schema there will be a schema unit whose characterization is *self*. It is assumed that all operations leave the schema in a state that satisfies all the integrity constraints defined in the metaschema. Note that only the postconditions of these operations can be specified in OCL. The method must be defined in an adequate imperative language. The examples in this thesis (and those reported in (Raventós 2008a)) use the procedural language included in USE.

The following illustrates the characterization object types and objects and their *schemaUnit()* operation by means of their application to the ER and relational metaschemas.

3.2.3.1 *Application to the ER metaschema*

Figure 3.6 shows the characterization object types of the ER metaschema.

The formal specification of the *schemaUnit()* operation in each case is as follows:

```
context DataTypeCh::schemaUnit():DataType
  body: DataType.allInstances() -> any(d:DataType|
          d.name = self.name)

context EntityTypeCh::schemaUnit():EntityType
  body: EntityType.allInstances() -> any(e:EntityType|
          e.name = self.name and
          self.keyAttribute -> collect(k:KeyAttribute|
            Tuple{n:k.name, t:k.type}) -> asSet =
          e.attribute->select(isKey) -> collect(ka:Attribute|
            Tuple{n:ka.name, t:ka.dataType.name})-> asSet)
```

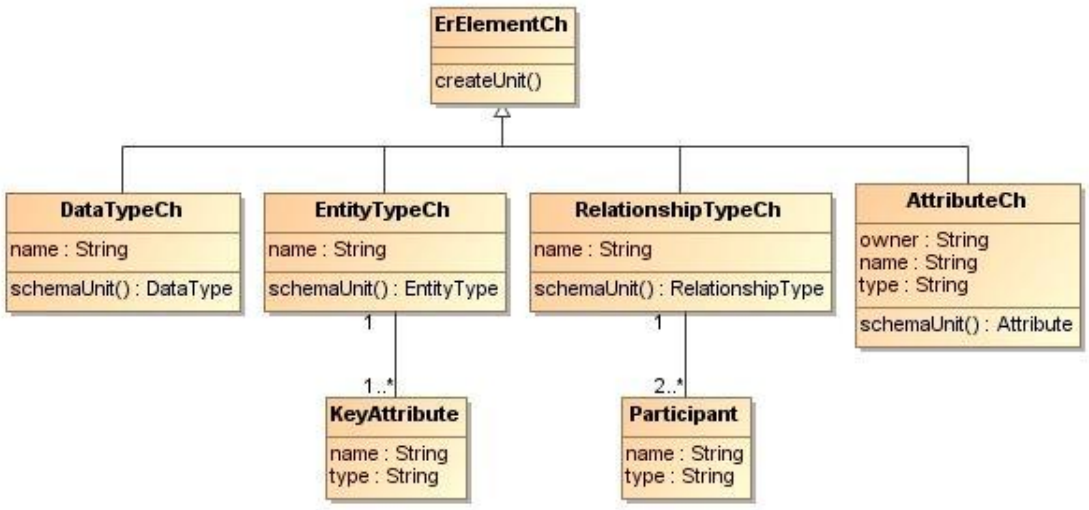


Figure 3.6 Characterization object types for the ER metaschema in Figure 3.3

```
context RelationshipTypeCh::schemaUnit():RelationshipType
  body: RelationshipType.allInstances()-> any(r:RelationshipType|
         r.name = self.name and
         self.participant -> collect(p:Participant|
           Tuple{n:p.name, t:p.type}) -> asSet =
         r.relationEnd -> collect(re:RelationEnd|
           Tuple{n:re.name, t:re.entityType.name}) -> asSet)

context AttributeCh::schemaUnit():Attribute
  body: Attribute.allInstances() -> any(a:Attribute|
           a.name = self.name and not a.isKey and
           a.dataType.name = self.type and
           (a.entityType.name = self.owner or
             a.relationshipType.name = owner))
```

3.2.3.2 *Application to the relational metaschema*

Figure 3.7 shows the characterization object types of the relational metaschema.

The formal specification of the *schemaUnit()* operation in each case is as follows:

```
context RelationalDataTypeCh::schemaUnit():RelationalDataType
  body: RelationalDataType.allInstances() -> any(
         d:RelationalDataType| d.name = self.name)

context TableCh::schemaUnit():Table
  body: Table.allInstances() -> any(t:Table|
           t.name = self.name and
           self.keyColumn -> collect(c:KeyColumn|
             Tuple{n:c.name, dt:c.type}) ->asSet() =
           t.column ->select(isKey) -> collect(co:Column|
           Tuple{n:co.name,co.relationalDataType.name})->asSet())
```

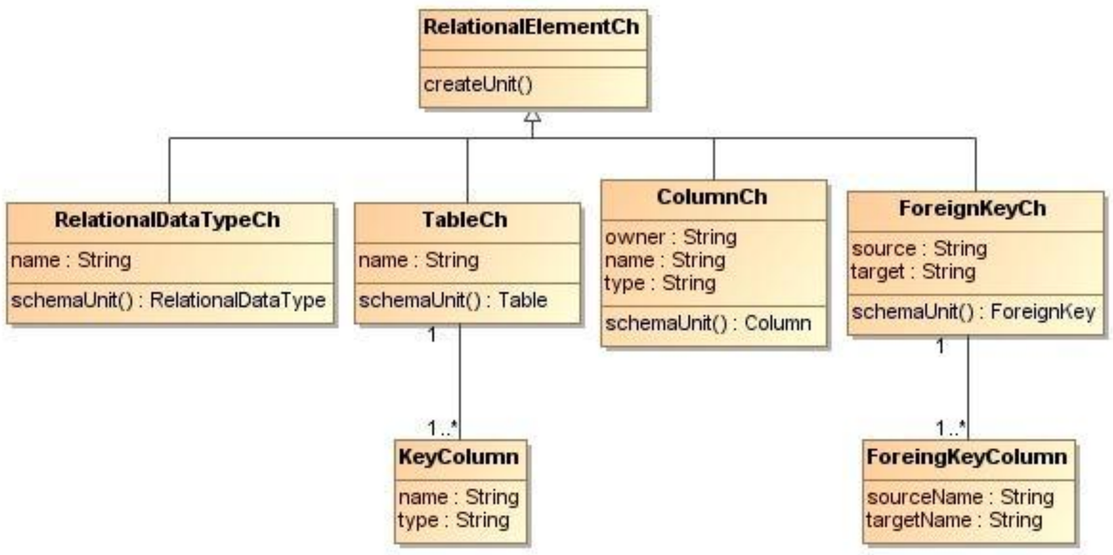


Figure 3.7. Characterization object types for the relational metaschema in Figure 3.5

```
context ColumnCh::schemaUnit():Column
  body: Column.allInstances() -> any(c:Column|
            c.name = self.name and not c.isKey and
            c.relationalDataType.name = self.type and
            c.table.name = self.owner)

context ForeignKeyCh::schemaUnit():ForeignKey
  body: ForeignKey.allInstances() -> any(f:ForeignKey|
            f.column -> any(true).table.name = self.source and
            f.targetTable.name = self.target and
            f.column -> any(true).table.name = self.source and
            f.targetTable.name = self.target and
            f.column -> forAll(co:Column|
              self.foreignKeyColumn -> exists(fkc|
                co.name = fkc.sourceName and
                f.targetTable.column -> select(isKey) ->
                collect(name) -> includes(fkc.targetName)))
```

3.3 Translation mapping expressions

Let $M = (MS_1, MS_2, \Sigma)$ be a translation mapping where MS_1 and MS_2 are instances of the MOF (Object Management Group 2006a). This section proposes an approach to defining the translation mapping constraints Σ that is based on using the core concepts of object-oriented languages, including operation, specialization/generalization and operation redefinition. These core concepts are part of the MOF and of its instances.

In S_i*Element*, the following four operations are defined: s_i*MappingKind()*, s_i*Equivalents()*, *includedInS*$_j$() and *mappedToS*$_j$(), which are used to specify the translation mapping constraints. In contrast to the operations defined in the previous section, these operations are mapping-dependent. In S_i*Element* these operations give a default result, but they can be redefined in the subtypes. In the following, each operation is defined in turn and then, Section 3.3.5 explains how they can be uses to define the mapping constraints.

3.3.1 s_iMappingKind

S_i*Element* includes the specification of the query operation s_j*MappingKind*(): *MappingKind*, whose value indicates how a schema unit of S_i is translated into S_j. The value of this operation is mapping-dependent. *MappingKind* is an enumeration data type whose values are as follows:

- *HasEquivalents*. A schema unit of S_i has this mapping kind when it is completely equivalent to a set $\{u_{j,1},\ldots,u_{j,k}\}$ of one or more schema units of S_j. The mapping kind of $u_{j,1},\ldots,u_{j,k}$ must be *IsIncluded*.
- *IsIncluded*. A schema unit of S_i has this mapping kind when it is included in a schema unit $u_{j,k}$ of S_j. The mapping kind of $u_{j,k}$ must be *HasEquivalents*.
- *Untranslatable*. A schema unit of S_i has this mapping kind when it cannot be translated into S_j. If a schema S_i contains one or more untranslatable schema units then its translation into S_j can only be partial. Note that in this approach it is easy to specify the schema units that cannot be translated in a given mapping.

When a schema unit of S_i has both an equivalence and an inclusion mapping with only one unit $u_{2,k}$ of S_j then the mapping kind of one of them is defined as *HasEquivalents* and the other as *IsIncluded*. There is an example in Figure 3.3(a), ($u_{1,4}$ and $u_{2,5}$), and several examples in Figure 3.3(b).

In the context of S_i*Element* the operation s_j*MappingKind*() can only give a default value, and each subtype *ST* of S_i*Element* such that some or all of its instances represent schema units redefines it (if necessary) to give the correct value. The value of the operation for the instances of *ST* that are not a schema unit is undefined. This may be enforced by means of a precondition, which in general can stated as follows:

```
context SiElement::SjMappingKind()
  pre: isSchemaUnit()
```

3.3.1.1 *Application to the Er-relational translation mapping: Er side*

In the ER metaschema example in Figure 3.4 is defined:

```
context ErElement::relationalMappingKind():MappingKind
  body: MappingKind::HasEquivalents
```

This means that by default all (direct or indirect) instances of *ErElement* that are schema units have an equivalence mapping, and that those instances that are not schema units have an undefined value for the operation. In this particular example, there is no need to redefine the operation in any subtype of *ErElement*.

3.3.1.2 *Application to the Er-Relational translation mapping: Relational side*

Similarly, in the relational metaschema example in Figure 3.5 is defined:

```
context RelationalElement::erMappingKind():MappingKind
  body: MappingKind::IsIncluded
```

This means that by default all (direct or indirect) instances of *RelationalElement* that are schema units have an inclusion mapping, and that those instances that are not schema unit have an undefined value for the operation.

In the relational metaschema example in Figure 3.5, most schema units may be defined using an inclusion mapping. There are exceptions due to the simplified ER metaschema used in this chapter. The exceptions are the instances of *Table* that cannot be translated as entity or relationship types, the columns of untranslatable tables and the instances of *ForeignKey* that do not correspond to referential integrity constraints. The exceptions can be defined as follows:

The *mappingKind* of *Table* is defined as follows:

```
context Table::erMappingKind():MappingKind
  body:   if translatableIntoEntityType or
             translateableIntoRelationshipType
          then MappingKind::IsIncluded
          else MappingKind::Untranslatable
          endif
```

where *translatableIntoEntityType* or *translatableIntoRelationshipType* are helper attributes defined as follows:

```
context Table:
   def: translatableIntoEntityType:Boolean = column ->
        select(isKey).foreignKey -> isEmpty()
   def: translatableIntoRelationshipType:Boolean =
           column -> select(isKey).foreignKey -> size() > 1 and
           column -> select(isKey) -> forAll(foreignKey ->size() = 1)
```

All instances of *Column* that are schema units and whose table are translatable have an inclusion mapping formally defined as follows:

```
context Column::erMappingKind():MappingKind
  body:if isSchemaUnit()
       then
          if table.erMappingKind() = MappingKind::IsIncluded
          then MappingKind::IsIncluded
          else MappingKind::Untranslatable
          endif
       else Set{} endif
```

Finally, the instances of *ForeignKey* whose source and reference tables are translatable and whose reference tables are translatable are translatable into an entity type. This is formally defined as follows:

```
context ForeignKey::erMappingKind():MappingKind
  body:if targetTable.erMappingKind() = MappingKind::IsIncluded and
            sourceTable.erMappingKind()= MappingKind::IsIncluded and
          targetTable.translatableIntoEntityType
       then MappingKind::IsIncluded
       else MappingKind::Untranslatable endif
```

where *sourceTable* is an auxiliary attribute defined as follows:

```
context ForeignKey:
 def: sourceTable:Table = column.table ->any(True)
```

In the above definition, it has been assumed that the relational metaschema inFigure 3.5 includes the constraint that all columns of a foreign key must belong to the same table. This could be enforced by the invariant formally defined as follows:

```
context ForeignKey inv allColumnsOfForeignKeyHaveSameTable:
  column.table -> size() = 1
```

3.3.2 s_jEquivalents

The s_j*Equivalents*() operation is defined in the context of S_i*Element.* The evaluation of this operation in a schema unit $u_{i,k}$ of S_i whose mapping kind is *HasEquivalents* gives the set of characterization objects of the schema units of schema S_j that are equivalent to $u_{i,k}$. The operation does not change either of the two schemas, but it creates one or more characterization objects. The signature and precondition of the operation in OCL is formally defined as follows:

```
context SiElement::sjEquivalents():Set(SjElementCh)
  pre: sjMappingKind() = MappingKind::HasEquivalents
  post atLeastOneCharacterizationObjectCreated:
       (SjElementCh.allInstances() - SjElementCh.allInstances@pre())
         -> notEmpty()
  post definingtheResult:
         result = SjElementCh.allInstances() –
                     SjElementCh.allInstances@pre()
```

The effect of the operation must be defined in the subtypes of S_i*Element* such that some or all of their instances represent schema units whose mapping kind is *HasEquivalents.* The effect can be procedurally defined by a method or declaratively by a postcondition. In this approach, it is defined a declarative specification from which a method can easily be derived. The USE tool used automatically checks that the effect of the method satisfies the postconditions and the integrity constraints defined in the metaschema.

3.3.2.1 *Application to the ER-relational translation mapping: ER side*

The adaptation of the above operation to the ER metaschema in Figure 3.4 is formally defined as follows:

```
context ErElement::relationalEquivalents():Set(RelationalElementCh)
  pre: relationalMappingKind() = MappingKind::HasEquivalents
  post definingtheResult:
       result = RelationalElementCh.allInstances() –
                  RelationalElementCh.allInstances@pre()
```

Given that the mapping kind of all ER schema units is *HasEquivalent*, it is necessary to redefine the operation *relationalEquivalents()* in each case, as shown below:

```
context DataType::relationalEquivalents():
          Set(RelationalDataTypeCh)
 post: rdt.oclIsNew() and rdt.oclIsTypeOf(RelationalDataTypeCh)
       and rdt.name = self.name
```

```
context EntityType::relationalEquivalents():Set(TableCh)
  post: t.oclIsNew() and t.oclIsTypeOf(TableCh) and
        t.name = self.name and self.attribute -> select(isKey) ->
        forAll(att| kc.oclIsNew() and kc.oclIsTypeOf(KeyColumn)
        and kc.name = att.name and kc.type = att.dataType.name and
        kc.tableCh = t)))

context RelationshipType::relationalEquivalents():
          Set(RelationalElementCh)
   post:  t.oclIsNew() and t.oclIsTypeOf(TableCh) and
          t.name = self.name and
          self.relationEnd -> forAll(re|
          re.entityType.attribute -> select(isKey) ->
          forAll(attkey| kc.oclIsNew() and kc.oclIsTypeOf(KeyColumn)
          and kc.name =  re.name.concat('_').concat(attkey.name)
          and kc.type = re.entityType.name and kc.tableCh = t)))
          and self.relationEnd -> forAll(re| fk.oclIsNew() and
          fk.oclIsTypeOf(ForeignKeyCh) and
          fk.sourceTable = t.name and
          fk.targetTable = re.entityType.name and
          re.entityType.attribute -> select(isKey) ->
          forAll(attkey| fkc.oclIsNew() and
          fkc.oclIsTypeOf(ForeignKeyColumn) and
          fkc.sourceName    = re.name.concat('_').concat(attkey.name)
          and fkc.targetName = attKey.name and
          fkc.foreignKeyCh = fk))

context Attribute::relationalEquivalents():Set(ColumnCh)
   post:  c.oclIsNew() and c.oclIsTypeOf(ColumnCh) and
          c.name = self.name and c.type = self.dataType.name) and
          c.owner = if self.entityType -> notEmpty()
                     then
                      self.entityType.name
                     else
                      self.relationshipType.name
                     endif
```

3.3.2.2 *Application to the Er-relational translation mapping: relational side*

The adaptation of the above operation to the relational metaschema in Figure 3.5 is formally defined as follows:

```
context RelationalElement::erEquivalents():Set(ErElementCh)
   pre: relationalMappingKind() = MappingKind::HasEquivalents
   post definingtheResult:
        result = ErElementCh.allInstances() -
                    ErElementCh.allInstances@pre()
```

Note that on the relational side there is no need to redefine the *erEquivalents()* operation because no schema units have a *HasEquivalents* mapping.

3.3.3 includedInS_j

It has been shown that the translation of a schema unit $u_{i,k}$ of S_i whose mapping kind is *HasEquivalents* is given by the result of the operation s_j*Equivalents*() invoked on $u_{i,k}$. The result is a non-empty set of instances of S_j*ElementCh* that are characterization objects of S_j

schema units. Similarly, each schema unit $u_{i,k}$ of S_i whose mapping kind is *IsIncluded* is translated into one and only one characterization object of a schema unit of S_j, which is given by the operation *includedInS$_j$* () invoked on $u_{i,k}$.

The operation is hosted in *S$_i$Element* and its formal definition in OCL is as follows:

```
context SiElement::includedInSj():SjElementCh
  pre: sjMappingKind() = MappingKind::IsIncluded
  post onlyOneCharacterizationObjectCreated:
         (SjElementCh.allInstances() -
           SjElementCh.allInstances@pre()) -> size()=1
  post definingtheResult:
       result = (SjElementCh.allInstances() -
                 SjElementCh.allInstances@pre()) ->any(True)
```

The effect of the operation must be defined in the subtypes of *S$_i$Element* such that some or all of their instances represent schema units whose mapping kind is *IsIncluded.* The effect can be procedurally defined by a method or declaratively by a postcondition. In this approach, a declarative specification is defined from which a method can easily be derived. The USE tool automatically checks that the effect of the method satisfies the postconditions and the integrity constraints defined in the metaschema.

3.3.3.1 *Application to the ER-relational translation mapping: ER side*

On the ER side there is no need to redefine the *includedInRelational()* operation because no schema units have an *IsIncluded* mapping.

3.3.3.2 *Application to the ER-relational translation mapping: relational side*

The specification of the above operation applied for the relational metaschema in Figure 3.5 is the following:

```
context RelationalDataType::includedInEr():ErElementCh
  post: d.oclIsNew() and d.oclIsTypeOf(DataTypeCh) and
        d.name = self.name

 context Table::includedInEr():ErElementCh
  post: if self.column -> select(isKey) -> forAll(
           foreignKey->isEmpty())
        then
           e.oclIsNew() and e.oclIsTypeOf(EntityTypeCh) and
           e.name = self.name and
           self.column -> select(isKey) -> forAll(co:Column|
              ka.oclIsNew() and ka.oclIsTypeOf(KeyAttribute) and
              ka.name = co.name and
              ka.type = co.relationalDataType.name and
              ka.entityTypeCh = e))
        else
           r.oclIsNew() and r.oclIsType(RelationshipTypeCh) and
           r.name = self.name and
           self.column -> select(isKey)-> forAll(co:Column|
              p.oclIsNew() and p.oclIsTypeOf(Participant) and
              p.name = co.name.substring(1,Set{1..co.name.size}->
              any(pos:Integer| co.name.substring(1,pos+1) =
              co.name.substring(1,pos).concat('_')) and
```

```
                p.type = co.relationalDataType.name))
            endif

    context Column::includedInEr():ErElementCh
     post: a.oclIsNew() and a.oclIsTypeOf(AttributeCh) and
           a.name = self.name and
           a.type = self.relationalDataType.name and
           a.owner = self.table.name

    context ForeignKey::includedInEr():ErElementCh
     post: r.oclIsNew() and r.oclIsTypeOf(RelationshipTypeCh) and
           r.name = self.column.table.name -> asSet() -> any(true)
           and self.column.table -> asSet() -> any(true).column ->
             select(isKey) -> forAll(co:Column| p.oclIsNew() and
               p.oclIsTypeOf(Participant) and
               p.name = co.name.substring(1,Set{1..co.name.size}
               -> any(pos:Integer| co.name.substring(1,pos+1) =
                 co.name.substring(1,pos).concat('_')) and
               p.type = co.relationalDataType.name))
```

3.3.4 mappedToS_j

The two previous sections have shown that the translation of a schema unit $u_{i,k}$ of S_i is given by the result of the operations *s_jEquivalents()* and *includedInS_j*() invoked on $u_{i,k}$. A schema unit is translated correctly if the results of these operations are consistent. The consistency condition is embodied in a single operation, called *mappedToS_j*(), which returns a *True* value if it is satisfied and a *False* value otherwise.

The formal specification in OCL is as follows:

```
  context SiElement::mappedToSj():Boolean
    pre: isSchemaUnit()
    body:
     if sjMappingKind() = MappingKind::HasEquivalents then
          self.sjEquivalents()->forAll(sj:SjElementCh|sj.schemaUnit()->
          notEmpty() and sj.schemaUnit().siMappingKind() =
          MappingKind::IsIncluded and
          sj.schemaUnit().includedInSj().schemaUnit() = self)
     else if sjMappingKind() = MappingKind::IsIncluded then
            self.includedInSj().schemaUnit()->notEmpty() and
            self.includedInSj().schemaUnit().siMappingKind() =
            MappingKind::HasEquivalents and
            self.includedInSj().schemaUnit().sjEquivalents().
              schemaUnit() -> includes(self)
          else
            False
          endif
     endif
```

This means that for each schema unit s_i of S_i, whose mapping kind is *HasEquivalents,* all schema units of S_j that are the equivalents of s_i, must have a mapping kind equals to *IsIncluded*, and the result of applying the *includedInS_i*() to each of them must be s_i. Moreover, for each schema unit s_i of S_i, whose mapping kind is *IsIncluded*, the result of applying *includedIn*S_i() to s_i, gives a s_j of S_j, whose equivalents are schema units including s_i.

The adaptation to the ER-relational mapping is straightforward. Here, the corresponding to the ER-side is shown below:

```
context ErElement::mappedToRelational():Boolean
  body:  if relationalMappingKind() = MappingKind::HasEquivalents
         then
         self.relationalEquivalents() -> forAll(
           re:RelationalElementCh| re.schemaUnit()->notEmpty() and
           re.schemaUnit().erMappingKind()=
             MappingKind::IsIncluded and
           re.schemaUnit().includedInEr().schemaUnit() = self)
         else
           if relationalMappingKind() = MappingKind::IsIncluded
           then
             self.includedInRelational().schemaUnit()->notEmpty()
             and self.includedInRelational().schemaUnit().
               erMappingKind() = MappingKind::HasEquivalents and
             self.includedInRelational().schemaUnit().
               erEquivalents().schemaUnit()->includes(self)
           else
             False
           endif
         endif
```

3.3.5 Translation mapping constraints

Let $M = (MS_1, MS_2, \Sigma)$ be a translation mapping where MS_1 and MS_2 are instances of the MOF (Object Management Group 2006a). The translation mapping constraints Σ consist of exactly two constraints, called *complete and consistent mapping to* S_2 and *complete and consistent mapping to* S_1.

In UML, these constraints can be formally defined by the following OCL invariants:

```
context S1Element inv completeAndConsistentMappingToS2:
  isSchemaUnit() and
  (s2mappingKind() = MappingKind::HasEquivalents or
    s2mappingKind() = MappingKind::IsIncluded)
  implies mappedToS2()

context S2Element inv completeAndConsistentMappingToS1:
  isSchemaUnit() and
  (s1mappingKind() = MappingKind::HasEquivalents or
    s1mappingKind() = MappingKind::IsIncluded)
  implies mappedToS1()
```

The intuitive meaning is that $\langle S_1, S_2 \rangle$ is an instance of translation mapping M if each translatable schema unit of S_1 is consistently mapped to S_2 and if each translatable schema unit of S_2 is consistently mapped to S_1. When both invariants hold, S_1 and S_2 are translations of each other. Note that the invariants exclude the schema units that are not translatable in the specified mapping.

The adaptation of the two constraints to the Er-relational mapping (see Figures 3.4 and 3.5) is straightforward, formally defined as follows:

```
context ErElement inv completeMappingToRelational:
  isSchemaUnit() and
  (relationalMappingKind() = MappingKind::HasEquivalents or
       relationalMappingKind()= MappingKind::IsIncluded)
  implies mappedToRelational()

context RelationalElement inv completeMappingToEr:
  isSchemaUnit() and
  (erMappingKind() = MappingKind::HasEquivalents or
       erMappingKind() = MappingKind::IsIncluded)
  implies mappedToEr()
```

3.3.6 Translating schemas

This section describes the use of the operations defined in the previous sections in the translation of schemas. Let $M = (MS_1, MS_2, \Sigma)$ be a mapping, and $S_1 = \{u_{1,1}, \ldots, u_{1,n}\}$ an instance of MS_1. The translation of S_1 into MS_2 is a schema $S_2 = \{u_{2,1}, \ldots, u_{2,m}\}$ such that $\langle S_1, S_2 \rangle$ is an instance of M. The translation of S_2 into MS_1 is similarly defined. The approach to the translation of a schema $S_1 = \{u_{1,1}, \ldots, u_{1,n}\}$ consists in translating each of its schema units $u_{i,j}$ following the order given by the operation *predecessors()*, starting with the units that have no predecessors. As it has been seen in Section 3.2.2 the computation of the predecessors is mapping-independent and straightforward.

The translation is done by applying an operation that we call *translateToS$_j$* () to the schema units. In what follows, the specification of the pre- and postconditions of the operation in OCL is given.

An instance $u_{i,k}$ of *S_iElement* can be translated into S_j if it represents a schema unit whose mapping kind is *HasEquivalents* or *IsIncluded*. The effect of the operation *translateToS$_j$* () must be that $u_{i,k}$ is mapped to S_j. This is captured by the simple following formal specification as follows:

```
context SiElement::translateToSj()
  pre:  isSchemaUnit() and
        (sjmappingKind() = MappingKind::HasEquivalents or
        sjmappingKind() = MappingKind::IsIncluded)
  post: mappedToSj()
```

There is no need to refine the specification of this operation in the subtypes of *S_iElement*. Concerning its implementation, the specification of *mappedToS$_j$* (explained in Section 3.3.4) suggests a straightforward implementation using the methods of the operations *createUnit()* (Section 3.2.3), *s_jEquivalents()* (Section 3.3.2) and *isIncludedInS$_j$()* (Section 3.3.3).

In (Raventós 2008a) it is described the implementation of the methods of *translateToEr()* and *translateToRelational()* in the procedural language described in USE (Gogolla, Büttner and Richters 2007) and the output obtained by their application to the example used in this paper. The same implementation is used in the translation of the osCommerce relational database to ER. In all cases, the time needed to translate and check its completeness and consistency (Section 3.3.5) can be considered satisfactory in a research environment in which research-oriented tools are used.

4 UML metaschema

In the field of software engineering, the Unified Modeling Language (UML) is a standardized specification language for object modeling. UML is officially defined in the Object Management Group (OMG) by the UML metamodel (Rumbaugh, Jacobson and Booch 2004). Like other MOF-based specifications, the UML metamodel and UML models may be serialized in XMI. UML was designed to specify, visualize, construct and document the artifacts of a software system (Rumbaugh, Jacobson and Booch 2004). For convenience, depending on the aspects of the systems being represented, UML divides concepts and constructs into views. The static view includes the elements that represent the concepts that are meaningful in a domain, their object structure and the relationships among them. *Unified Modeling Language: Superstructure, version 2.1* (Object Management Group 2007b) specifies the superstructure of the UML metamodel.

This chapter describes the subset of the static view of the UML metamodel considered to translate UML schemas to SBVR and vice versa. This subset is the part of the UML metamodel that is necessary to describe the structural schema of domains. To better illustrate the metaschema, an example of the schema, which is an instance of the metaschema, is included.

The UML metaschema, the definition of schema units (including the predecessors and characterization objects) and an example of instantiation have been specified in the UML-based Specification Environment (USE) tool. The detailed specifications are provided in the appendices.

The rest of this chapter is structured as follows:

- Section 4.1 describes an example of a schema that is an instance of the UML metaschema and which is used as a running example throughout this thesis.
- Section 4.2 defines, following the translation approach described in Chapter 3, the schema units of the UML metaschema, the precedence relationships between them, and the characterization objects of such schema units.

4.1 DBLP schema: an example of an instance of the UML metaschema

This section describes an example of a schema that will be used throughout this thesis to illustrate the translation between UML and SBVR. The example is based on the case study developed by Planas and Olivé (2006), with two additional association classes (*Editorship* and *Authorship*) and an additional attribute (*Gender*), which is an enumeration.

The structural schema presented in the case study deals with people (authors and editors) and their publications, which may be edited books or authored publications such as authored books, book chapters and journal papers. Book chapters and journal papers may or may not be conference papers.

Figure 4.1 shows the structural schema of DBLP.

The following constraints have been included in the case study for the translation to SBVR:

[1] Person: name

```
context Person inv nameIsKey:
Person.allInstances() -> isUnique(name)
```

[2] Book: isbn

```
context Book inv isbnIsKey:
Book.allInstances() -> isUnique(isbn)
```

[3] BookSeries: id

```
context BookSeries inv idIsKey:
BookSeries.allInstances() -> isUnique(id)
```

[4] Journal: issn

```
context Journal inv issnIsKey:
Journal.allInstances() -> isUnique(issn)
```

[5] Journal: title

```
context Journal inv titleIsKey:
Journal.allInstances() -> isUnique(title)
```

[6] ConferenceSeries: name

```
context ConferenceSeries inv nameIsKey:
ConferenceSeries.allInstances() -> isUnique(name)
```

[7] ConferenceEdition: title

```
context ConferenceEdition inv titleIsKey:
ConferenceEdition.allInstances() -> isUnique(title)
```

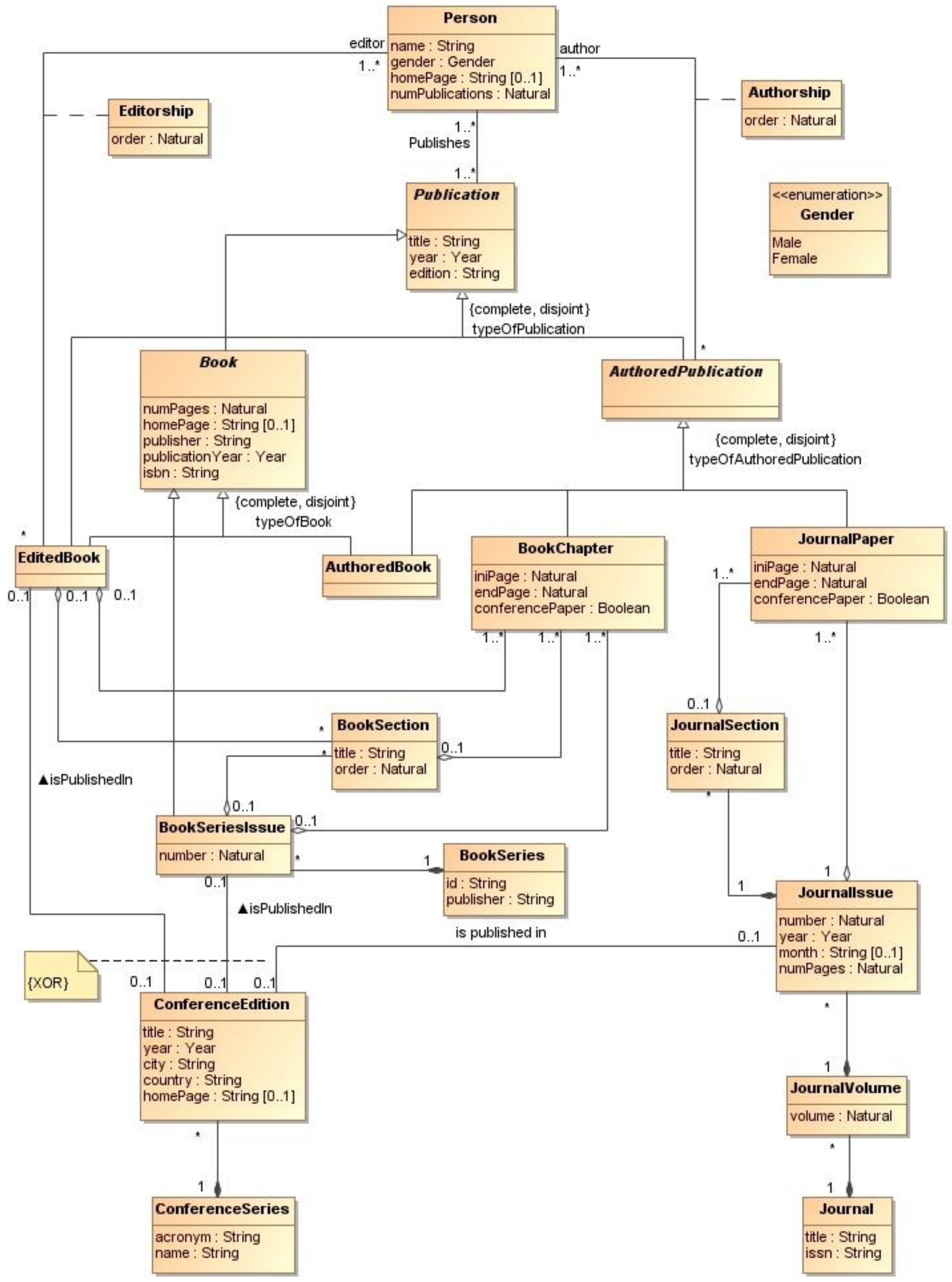


Figure 4.1 Structural schema of DBLP

4.2 Schema units of the UML metaschema

This section describes the fragment of the UML metamodel (Object Management Group 2007b) considered for the mapping to SBVR. The metaclasses included are those necessary to represent vocabularies and business rules. The following are the elements of the UML metamodel that have been excluded: *Package*, *PackageMerge*, *PackageableElement*, *Slot*, behavioral features including operations of the Kernel package, the Dependencies package, the Interfaces package, some derived information (i.e., some derived associations and attributes), the *visibility* attribute of *NamedElement*, the *isReadOnly* attribute of *Property,* the *defaultValue* association of *Property*, generalizations of associations, and associations with the value of *isAbstract* equal to true.

The fragment is described in terms of its schema units that is, its knowledge components, as defined in Chapter 3.

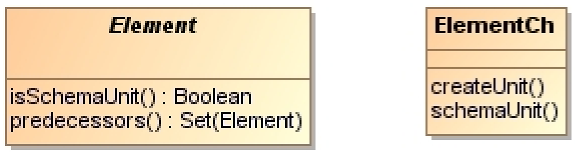


Figure 4.2 Definition of *Element* and *Element* characterization object

Since all UML metaclasses are subtypes of the abstract metaclass *Element*, in order to define the schema units, *Element* includes two operations (*isSchemaUnit()* and *predecessors()*). The specification of the *isSchemaUnit()* operation in the context of *Element* of Figure 4.2 is defined as follows:

```
 context Element::isSchemaUnit():Boolean
   body:   true
```

This means that, by default, all (direct or indirect) instances of *Element* are schema units. However, each subtype of *Element* such that some or all of its instances do not represent schema units redefines the *isSchemaUnit()* operation, as described below.

In the context of *Element,* the query operation *predecessors* give the empty set as default value:

```
 context Element::predecessors():Set(Element)
   pre:   isSchemaUnit()
   body:  Set{}
```

Each subtype of *Element* such that some or all of its instances represent schema units redefines the *predecessors() operation* (if needed).

Additionally, as described in Chapter 3, each schema unit is characterized by a schema unit characterization object type, which defines the schema elements comprised by the schema unit. The schema unit characterization object is defined for each subtype of *Element* such that some or all of its instances represent schema units. Each characterization object type includes a set of attributes that characterize the schema unit and two operations: *createUnit()* and *schemaUnit()*. The former creates a schema unit from its characterization object, and the latter gives the schema unit corresponding to the characterization object, if it exists. The operation *createUnit()* will only be invoked when the predecessors of the schema unit it creates have already been created. It is the same for all of the characterization object types and is defined in the general *ElementCh* (see Figure 4.2):

```
context ElementCh::createUnit()
  pre:   schemaUnit() -> notEmpty()
```

The *schemaUnit()* operation is redefined in each subtype of *ElementCh*.

In a UML schema, the schema units are classes, data types, enumerations, attributes, associations, association classes, generalizations, generalization sets and constraints.

The following subsections define the schema units in terms of their schema elements. They provide, for each schema unit, a generic description of it, its abstract syntax, the specifications of the *isSchemaUnit()* and *predecessors()* operations used to define it, and its schema unit characterization object.

4.2.1 Class schema unit

Generic description

A class schema unit describes a set of objects that share the same specifications of features, constraints and semantics. A class schema unit may be defined as *abstract* that is, the instances of the class may be derived by union of the subtypes of a partition of the class.

The DBLP example shown in Figure 4.1 has 17 class schema units represented by instances of *Class* named *Person*, *Publication*, *Book*, *AuthoredPublication*, *EditedBook*, *AuthoredBook*, *BookChapter*, *JournalPaper*, *BookSection*, *BookSeriesIssue*, *BookSeries*, *JournalSection*, *JournalIssue*, *ConferenceEdition*, *ConferenceSeries*, *JournalVolume* and *Journal*. *Publication*, *Book* and *AuthoredPublication* have classes defined as *abstract*. The instances of *Publication* may be derived by union of the instances of *EditedBook* and *AuthoredPublication*. The instances of *Book* may be derived by union of the instances of *EditedBook* and *AuthoredBook*. And finally, the instances of *AuthoredPublication* may be derived by union of *AuthoredBook*, *BookChapter* and *JournalPaper*.

Abstract syntax

Each class schema unit is represented by an instance of *Class*. The schema elements of a class, named *c*, are as follows: (1) the instance β of *Class*; (2) the instance of attribute *name* of β with value *c*; and (3) the instance of attribute *isAbstract* of β with value *True* or *False*.

Figure 4.3 shows the abstract syntax of the class schema unit. Note that the *isSchemaUnit()* and *predecessors() operation*s are not redefined in *Class*, meaning that all instances of *Class* are schema units and do not have predecessors.

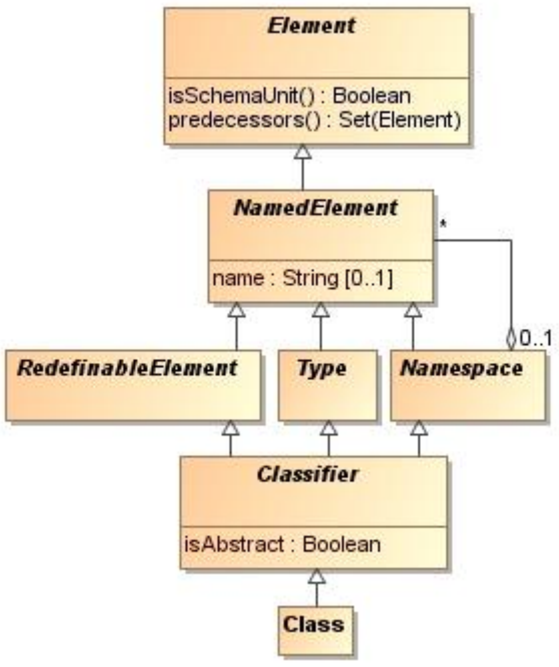


Figure 4.3 Class schema unit

Characterization object

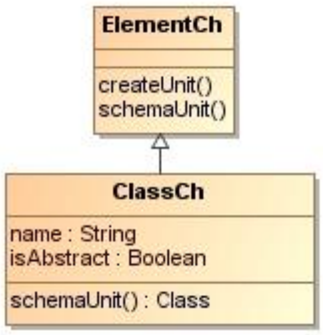


Figure 4.4 Class schema unit characterization object *ClassCh*

Figure 4.4 shows the characterization object type for the class schema unit, *ClassCh*. The *schemaUnit() operation* is defined, formally, as follows:

```
 context ClassCh::schemaUnit():Class
   body: Class.allInstances() -> any(c:Class| c.name = self.name and
          c.isAbstract = self.isAbstract)
```

This means that the *schemaUnit() operation* of *ClassCh* is a query that gives the instance of *Class* whose attributes *name* and *isAbstract* have the same values as the ones given in the attributes *name* and *isAbstract* of *ClassCh*, respectively.

4.2.2 Data type schema unit

Generic description

A data type schema unit is a type whose instances are identified only by their value. Additionaly, UML includes some predefined data types, called primitive types, which are as follows: *Boolean*, *Integer*, *UnlimitedNatural* and *String*.

The DBLP example shown in Figure 4.1 has two data type schema units represented by two instances of *DataType* named *Natural* and *Year*, respectively. It also includes two data types represented by two instances of *PrimitiveType* named *Boolean* and *String*, respectively.

Abstract syntax

Each data type schema unit that is not an enumeration is represented by an instance of *DataType* that is not an instance of *Enumeration*. The schema elements of a data type named *d* are as follows: (1) an instance η of *DataType* (or *PrimitiveType*); (2) the instance of attribute *name* of η with value *d*; and (3) the instance of attribute *isAbstract* of η with value *False*.

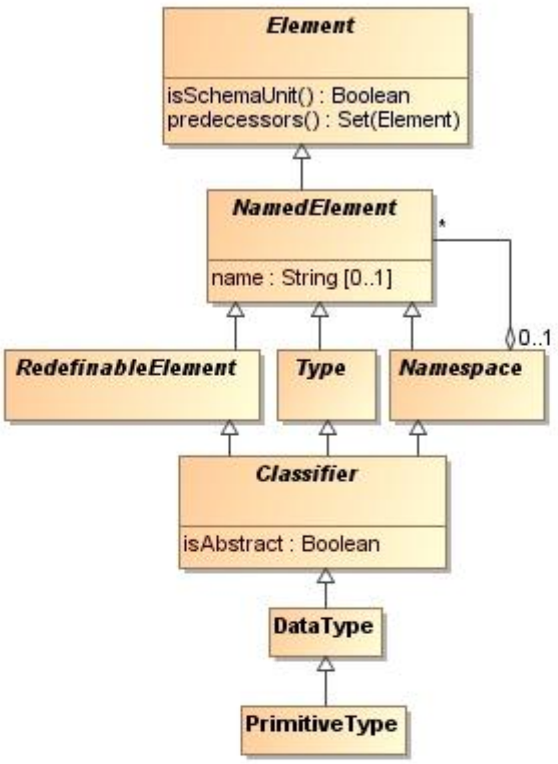


Figure 4.5 Data type and primitive type schema units

Figure 4.5 shows the abstract syntax of a data type schema unit. The *isSchemaUnit()* and *predecessors()* operations are not redefined in *DataType*.

Characterization object

For the data type characterization object *DataTypeCh* (see Figure 4.6), the *schemaUnit()* operation is defined, formally, as follows:

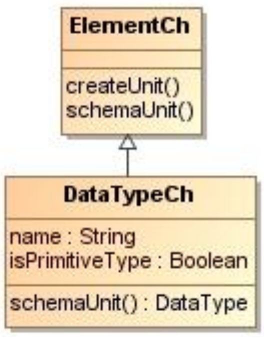


Figure 4.6 Data type schema unit characterization object *DataTypeCh*

```
 context DataTypeCh::schemaUnit():DataType
   body: DataType.allInstances() -> any(d:DataType|
         d.name = self.name and d.isAbstract = false and
         self.isPrimitiveType implies d.oclIsTypeOf(PrimitiveType))
```

This means that the *schemaUnit() operation* of *DataTypeCh* is a query that gives the instance of *DataType* whose attribute *name* has the same value as the one given in the attribute *name* of *DataTypeCh* and whose *isAbstract* attribute has a value equal to *False.* Moreover, if the attribute *isPrimitiveType* is *true* then the instance of *DataType* is also an instance of *PrimitiveType*.

4.2.3 Enumeration schema unit

Generic description

An enumeration schema unit is a special type of data type whose values are enumerated in the model as enumeration literals.

The DBLP example shown in Figure 4.1 has one enumeration schema unit represented by an instance of *Enumeration* named *Gender*. The two instances of *EnumerationLiteral* related to said instance of *Enumeration* are named *Male* and *Female*, respectively.

Abstract syntax

Each enumeration is represented by an instance of *Enumeration*. The schema elements of an enumeration *e* are as follows: (1) the instance γ of *Enumeration*; (2) the instance of attribute *name* of γ with value *e*; (3) the instance of attribute *isAbstract* of γ with value *False*; (4) the instances of *EnumerationLiteral* related to γ; and (5) for each of these enumeration literals, the instance of its attribute *name* and its relationship with *e* in a given order*.*

Figure 4.7 shows the abstract syntax of an enumeration schema unit. Note that the *isSchemaUnit()* and *predecessors() operation*s are not redefined in *Enumeration*, meaning that all instances of *Enumeration* are schema units and they do not have predecessors. The *isSchemaUnit()* operation is redefined in *InstanceSpecification*:

```
  context InstanceSpecification::isSchemaUnit():Boolean
   body:  false
```

This means that the instances of *InstanceSpecification* are not schema units.

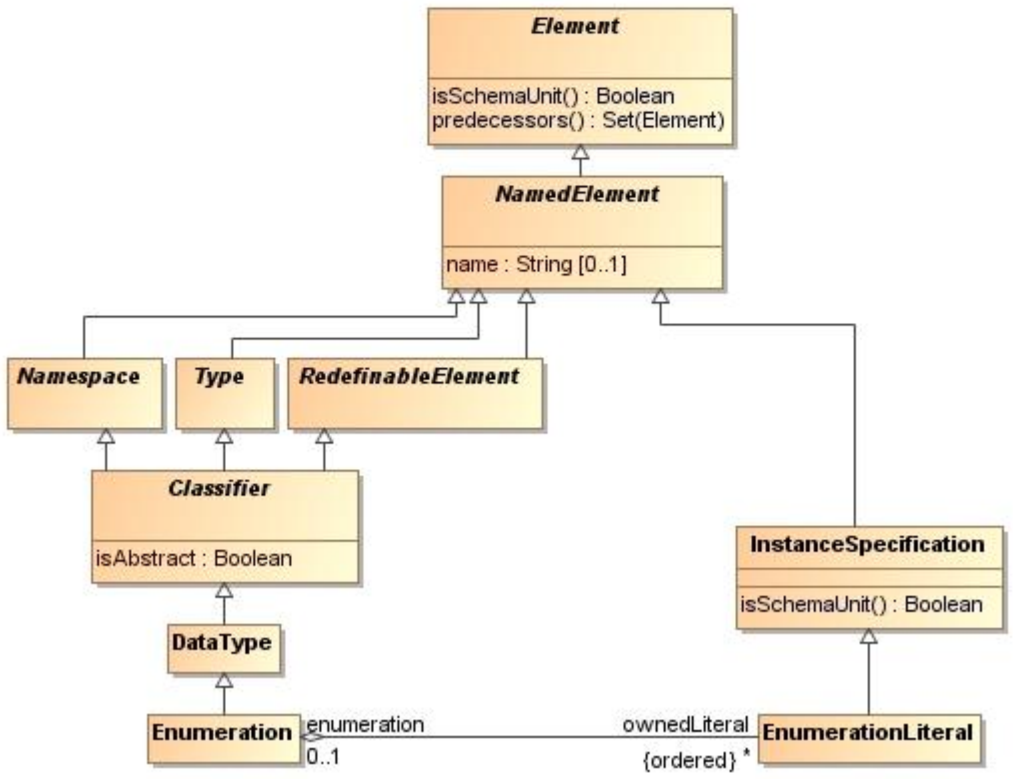


Figure 4.7 Enumeration schema unit

Characterization object

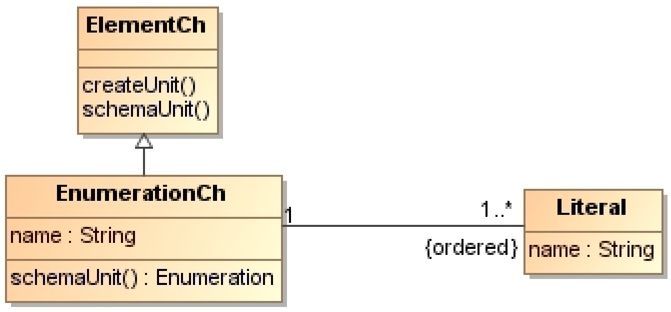


Figure 4.8 Enumeration schema unit characterization object *EnumerationCh*

For the enumeration characterization object *EnumerationCh* (see Figure 4.8), the *schemaUnit() operation* is defined, formally, as follows:

```
context EnumerationCh::schemaUnit():Enumeration
  body: Enumeration.allInstances() -> any(e:Enumeration|
          e.name = self.name and
          e.ownedLiteral -> collect(name) =
          self.literal -> collect(name))
```

This means that the *schemaUnit() operation* of *EnumerationCh* is a query that gives the instance of *Enumeration* whose attribute *name* has the same value as the one given in the attribute *name* of *EnumerationCh* and for which the ordered sequence of names of its *ownedLiterals* is equal to the ordered sequence of names of the *literals* of *EnumerationCh*.

4.2.4 Attribute schema unit

Generic description

An attribute schema unit is a structural feature that relates the instance of the class that owns the attribute to a value or collection of values of the type of the attribute.

The DBLP example shown in Figure 4.1 has 41 attribute schema units represented by instances of *Property*: *name*, *gender*, *homePage* and *numPublications* of *Person*; *title*, *year* and *edition* of *Publication*; *order* of *Editorship*; *order* of *Authorship*; *numPages, homePage*, *publisher*, *publication* and *isbn* of *Book*; *iniPage*, *endPage* and *conferencePaper* of *BookChapter*; *iniPage*, *endPage* and *conferencePaper* of *JournalPaper*; *title* and *order* of *JournalSection*; *title* and *order* of *BookSection*; *number* of *BookSeriesIssue*; *id* and *publisher* of *BookSeries; title, year*, *city*, *country* and *homepage* of *ConferenceEdition*; *acronym* and *name* of *ConferenceSeries*; *number*, *year*, *month* and *numPages* of *JournalIssue*; *volume* of *JournalVolume*; and *title* and *issn* of *Journal*.

Abstract syntax

Each attribute is represented by an instance of *Property* that is owned by a *Class* or a *DataType*. The schema elements of an attribute named *at* are as follows: (1) the instance π of *Property*; (2) the instance of attribute *name* of π with value *at*; (3) the instances of its Boolean attributes *isDerived* and *isDerivedUnion* and the instance of its *aggregation* attribute; (4) the instance of its relationship with an instance of *Class* or *DataType*; (5) an instance of a subtype of *LiteralSpecification* (usually *LiteralInteger*) with the instance of its attribute *value* and the relationship to π (for the *lowerValue*); (6) an instance of a subtype of *LiteralSpecification* (usually *LiteralInteger* or *LiteralUnlimitedNatural*) with the instance of its attribute *value* and the relationship to π (for the *upperValue*); and (7) the instance of its relationship with the corresponding type.

Figure 4.9 shows the abstract syntax of an attribute schema unit. The *isSchemaUnit()* query operation is redefined in the *Property* metaclass. Not all instances of *Property* are schema units. Only those that represent an attribute that is, the properties that are not related to any instance of *Association* by *memberEnd* or its specializations are schema units. Therefore, the query operation is formally redefined in *Property* as follows:

```
context Property::isSchemaUnit():Boolean
body:   self.association -> isEmpty()
```

The *isSchemaUnit()* query operation is also redefined in the *LiteralSpecification* abstract metaclass. No instances of either metaclass are schema unit.

```
context LiteralSpecification::isSchemaUnit():Boolean
body:   false
```

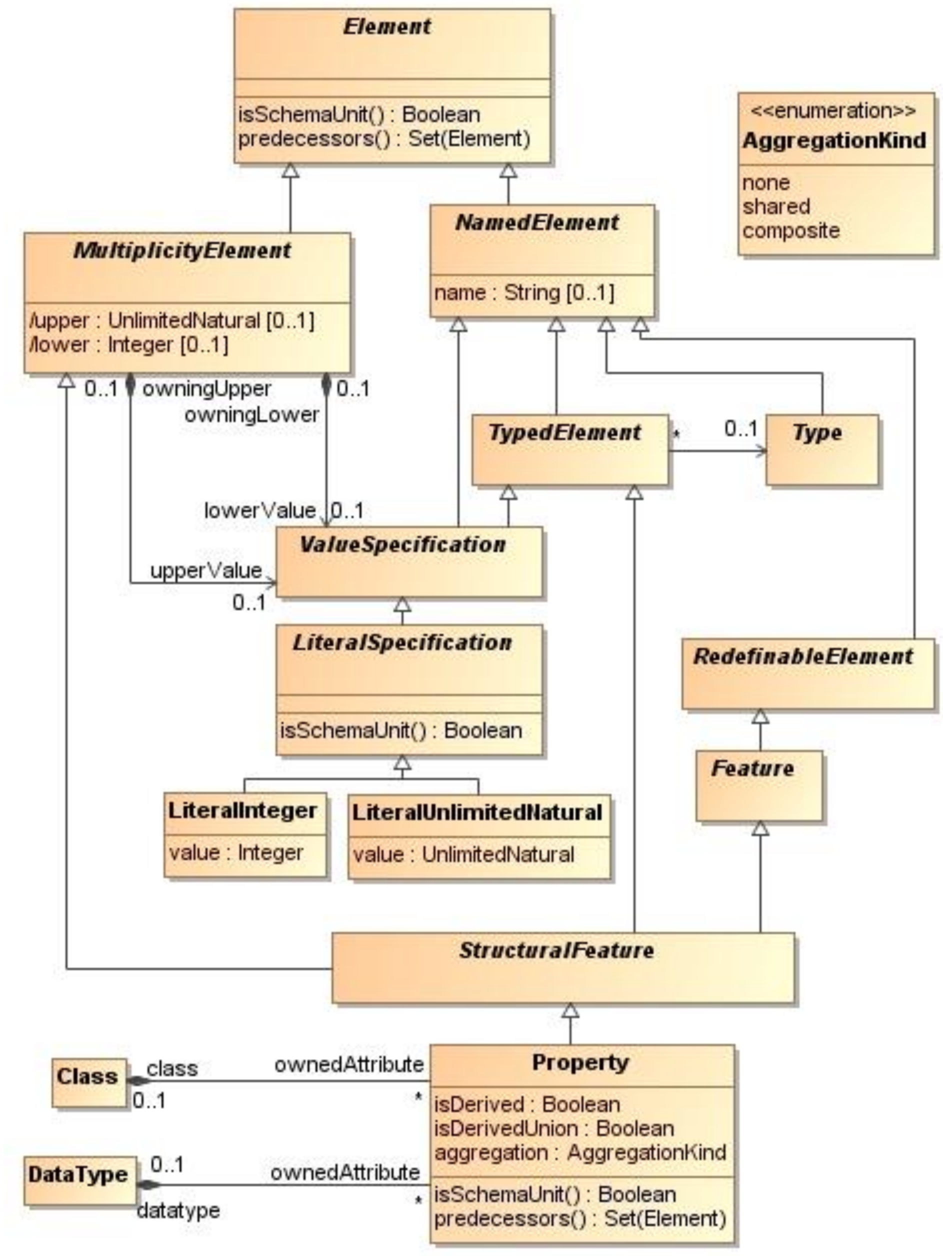


Figure 4.9 Attribute schema unit

The *predecessors()* operation of *Property* is specified as follos:

```
 context Property::predecessors():Set(Element)
  body: Element.allInstances() -> select(el:Element |
          (el.oclIsTypeOf(Class) and
          (el.oclAsType(Class) = self.class or
         el.oclAsType(Class) = self.type)) or
          (el.oclIsKindOf(DataType) and
          (el.oclAsType(DataType) = self.dataType or
         el.oclAsType(DataType = self.type)))
```

This means that the predecessors of a property that represents an attribute are its owning class or owning data type and its type.

Characterization object

For the attribute schema unit characterization object *PropertyCh* (see Figure 4.10), the *schemaUnit() operation* is defined, formally, as follows:

```
 context PropertyCh::schemaUnit():Property
   body: Property.allInstances() -> any(p:Property|
           p.association -> isEmpty()
           p.name = self.name and p.type.name = self.type and
           self.ownerClassName -> notEmpty() implies
             p.class.name = self.ownerClassName and
           self.ownerDataTypeName -> notEmpty() implies
             p.dataType.name = self.ownerDataTypeName and
           p.isDerived = self.isDerived and
           p.isDerivedUnion = self.isDerivedUnion and
           p.aggregation = self.aggregation_ and
           p.lowerValue.oclAsType(LiteralInteger).value =
           self.lowerValue and
           if self.upperValue -> notEmpty() then
             p.upperValue.oclAsType(LiteralInteger).value =
             self.upperValue
           else
             p.upperValue.oclIsTypeOf(LiteralUnlimitedNatural)
           endif)
```

This means that the *schemaUnit() operation* of *PropertyCh* is a query that gives the instance of *Property* whose owner is a class or data type with the attribute *name* with the same value as the one given in the attribute *name*; the *isDerived*, *isDerivedUnion* and *aggregation* attributes have the same value as the ones with the same names given in *PropertyCh*; the multiplicity constraints of the property are given in the *lowerValue* and *upperValue* attributes. Note that the *upperValue* attribute of *PropertyCh* has a value when the type of the *upperValue* specification is of type *LiteralInteger*.

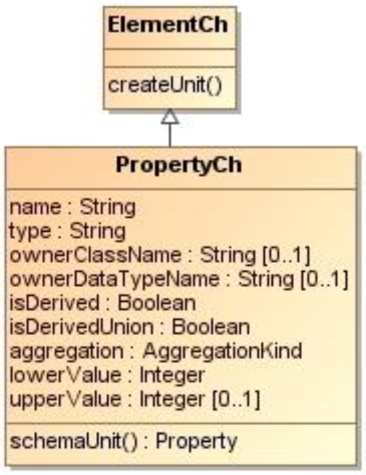


Figure 4.10 Attribute schema unit characterization object *PropertyCh*

4.2.5 Association schema unit

Generic description

An association schema unit specifies a semantic relationship that can occur between class schema units. The instances of association schema units are the identifiable individual relationships between instances of the classes that have the relationship. The association schema unit includes the following: the association with its name, if any; the member ends with their respective aggregation kinds; and the cardinality constraints between the participants, i.e., the multiplicities of the member ends of the association.

The DBLP example shown in Figure 4.1 has 15 association schema units represented by instances of *Association*: *R_1(bookChapter, bookSection), R_2(bookChapter, bookSeriesIssue), R_3(bookChapter, editedBook), R_4(bookSection, editedBook), R_5(journalPaper, journalIssue), R_6(journalPaper, journalSection), R_7(bookSeries, bookSeriesIssue), R_8(conferenceSeries, conferenceEdition), R_9(journal, journalVolume), R_{10}(journalIssue, journalSection), R_{11}(journalVolume, journalIssue), IsPublishedIn(conferenceEdition, bookSeriesIssue), IsPublishedIn(conferenceEdition, EditedBook), Publishes(person, publication)* and *IsPublishedIn(conferenceEdition, journalIssue).*

Note that association classes are considered different schema units than the association schema unit.

Abstract syntax

Each association schema unit is represented by an instance of *Association*. The schema elements of an association that may be named *as* are as follows: (1) the instance ρ of *Association*; (2) if it has a name, the instance of attribute *name* of ρ with value *as*; (3) the instances of its attribute *isAbstract*; (4) the instances of *Property* that are member ends of ρ; and (5) for each of these properties: (a) the instance of its relationship to its *Type*; (b) the instances of its attributes *isDerived*, *isDerivedUnion* and *aggregation*; (c) the instance of its relationship with ρ, in a given order; (d) an instance of a subtype of *LiteralSpecification* (usually *LiteralInteger)* with the instance of its attribute *value* and the relationship to the property (for the *lowerValue*); and (e) an instance of a subtype of *LiteralSpecification* (usually *LiteralInteger* or *LiteralUnlimitedNatural)* with the instance of its attribute *value* and the relationship to the property (for the *upperValue*).

Figure 4.11 shows the abstract syntax of an association schema unit. All instances of *Association* are schema units. Therefore, the *isSchemaUnit()* query is not redefined in *Association*. The *predecessors()* operation of *Association* is specified as follows:

```
context Association::predecessors():Set(Element)
  body: Property.allInstances() -> select(p:Property |
         p.association = self).type -> asSet()
```

This means that the predecessors of an association are the types of the member ends of the association.

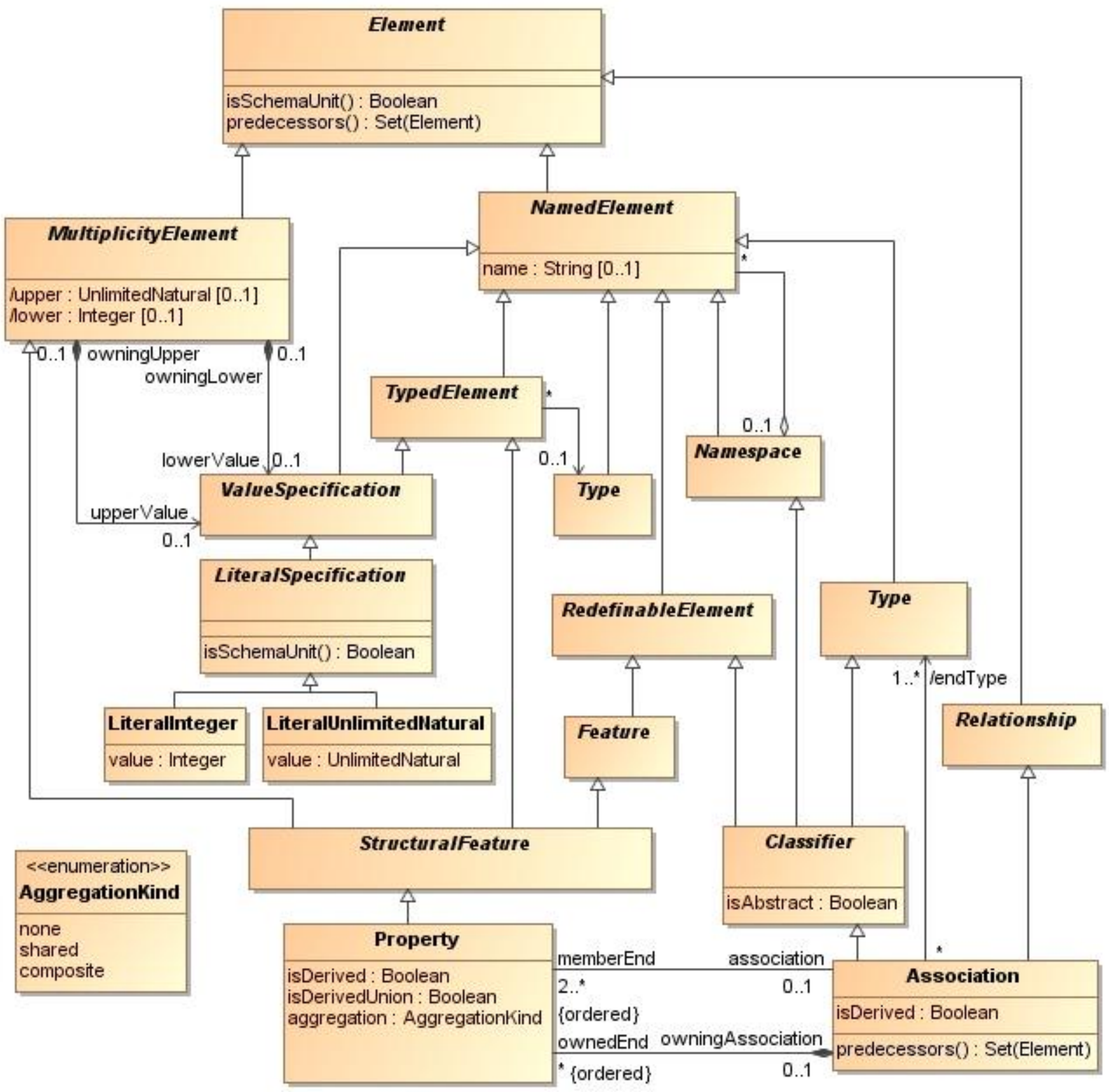


Figure 4.11 Association schema unit

Characterization object

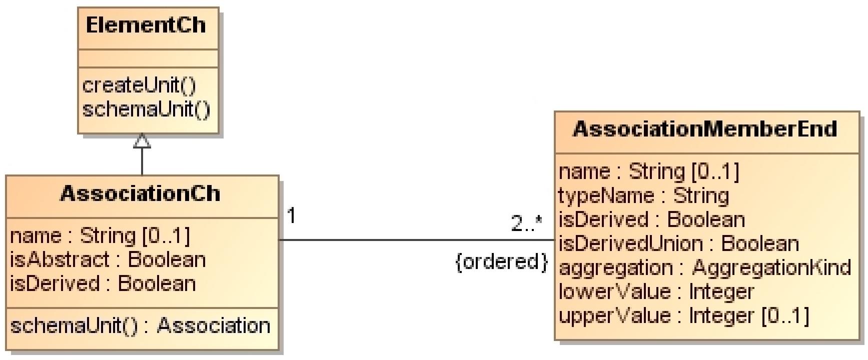


Figure 4.12 Association schema unit characterization object *AssociationCh*

For the association schema unit characterization object *AssociationCh* (see Figure 4.12), the *schemaUnit() operation* is defined, formally, as follows:

```
 context AssociationCh::schemaUnit():Association
  body: Association.allInstances() -> any(a:Association|
         self.name ->notEmpty() implies a.name = self.name and
         a.isAbstract = self.isAbstract and
         a.isDerived = self.isDerived and
         a.memberEnd -> collect(m:Property|
           Tuple{n:m.name, id:m.isDerived,
             idu:m.isDerivedUnion, ag:m.aggregation,
             l:m.lowerValue.oclAsType(LiteralInteger).value,
             u:m.upperValue.oclAsType(LiteralInteger).value,
             un:m.upperValue.oclIsTypeOf(LiteralUnlimitedNatural)})
         =
         self.associationMemberEnd ->collect(m:AssociationMemberEnd|
           Tuple{n:m.name, id:m.isDerived,
             idu:m.isDerivedUnion, ag:m.aggregation,
             l:m.lowerValue, u:m.upperValue,
             un:m.upperValue->isEmpty()}))
```

This means that the *schemaUnit() operation* of *AssociationCh* is a query that gives the instance of *Association* whose *name* attribute is empty or whose value is equal to the value of the *name* attribute of *AssociationCh*; the *isAbstract* and *isDerived* attributes have the same value as the *isAbstract* and *isDerived* attributes of *AssociationCh*, respectively; and the ordered sequence of values of the *name* attributes of *memberEnds* is equal to the ordered sequence of values of the *name* attributes of the *associationMemberEnds* of *AssociationCh*.

4.2.6 Association class schema unit

Generic description

An association class schema unit is a schema unit that is both an association schema unit and a class schema unit. The association class schema unit includes the structural elements of a class schema unit and of an association class schema unit.

The DBLP example shown in Figure 4.1 has two association class schema units represented by instances of *AssociationClass* and named *Editorship* and *Authorship*, respectively.

Abstract syntax

Each association class is represented by an instance of *AssociationClass*. The schema elements of an association class, named *ac,* are as follows: (1) the instance ξ of *AssociationClass*; (2) the instance of attribute *name* of ξ with value *ac*; (3) the instances of its attributes *isDerived* and *isAbstract*; (4) the instances of *Property* that are member ends ξ (such as *owningAssociation*); and (5) for each of these properties: (a) the instance of its relationship to its *Type*; (b) the instances of its attributes *isDerived*, *isDerivedUnion* and *aggregation*; (c) the instance of its relationship with ξ; (d) an instance of a subtype of *LiteralSpecification* (usually *LiteralInteger)* with the instance of its attribute *value* and the relationship to the property (for the *lowerValue*); and (e) an instance of a subtype of

LiteralSpecification (usually *LiteralInteger* or *LiteralUnlimitedNatural)* with the instance of its attribute *value* and the relationship to the property (for the *upperValue*).

Figure 4.13 shows the abstract syntax of an association class schema unit. As with the association and class schema units, all instances of *AssociationClass* are schema units. Therefore, the *isSchemaUnit()* query is not redefined in *AssociationClass*.

The *predecessors()* operation of *AssociationClass* is specified as follows:

```
context AssociationClass::predecessors():Set(Element)
  body: Property.allInstances() -> select(p:Property |
          p.association = self).type -> asSet()
```

This means that the predecessors of an association class are the types of the member ends of the association class.

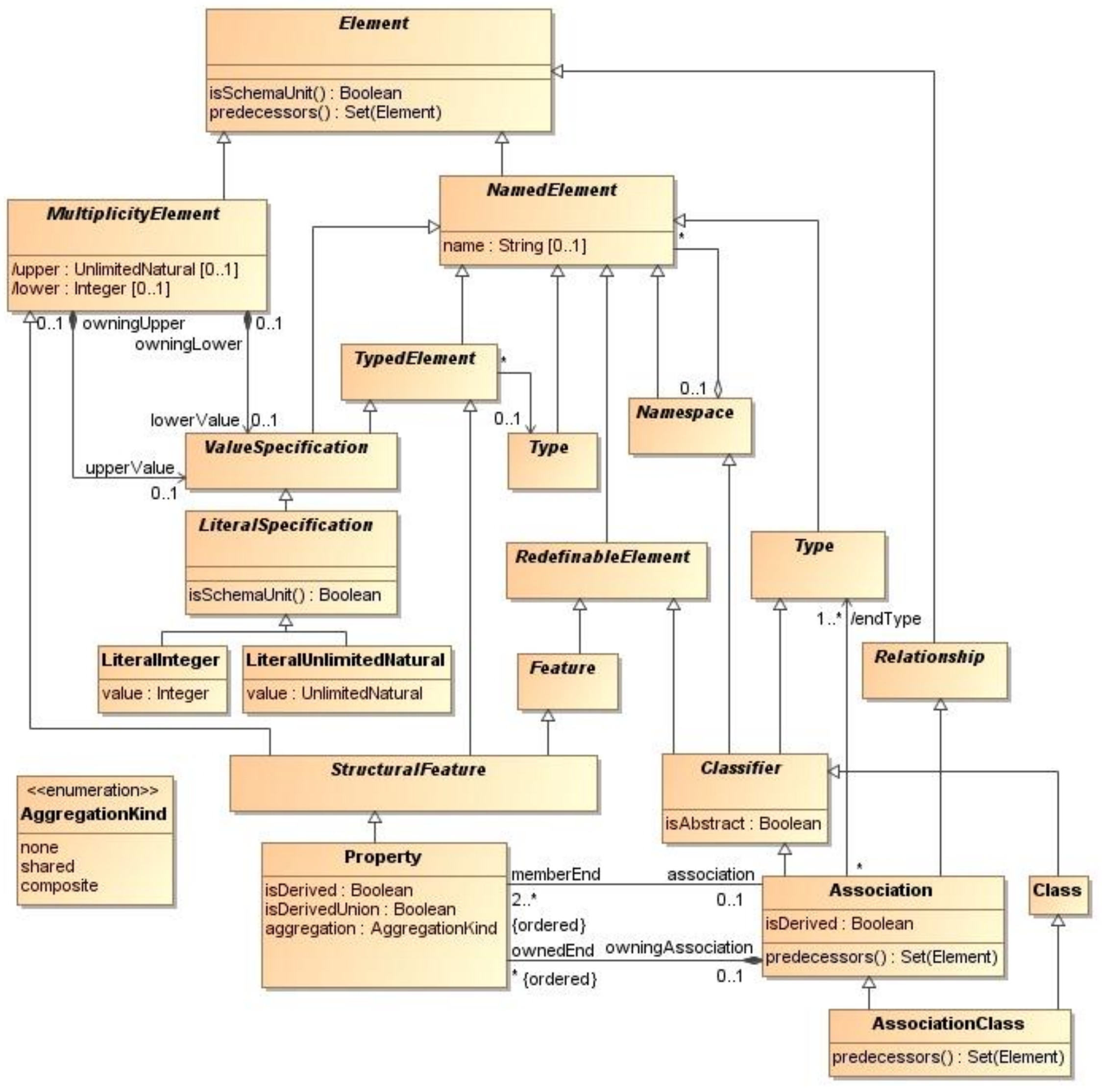


Figure 4.13 Association class schema unit

Characterization object

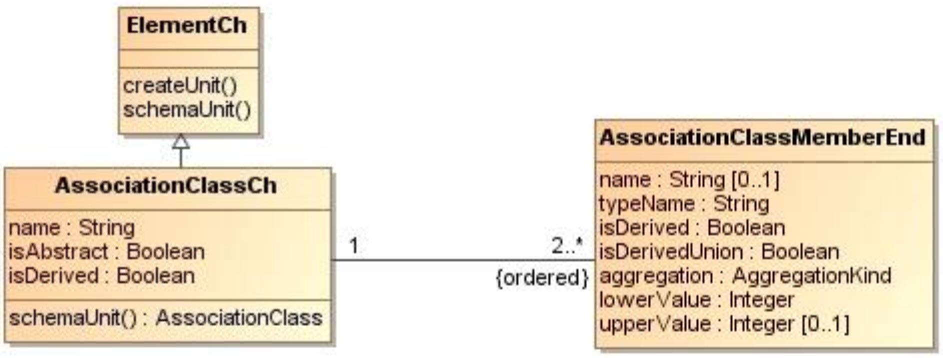


Figure 4.14 Association class schema unit characterization object *AssociationClassCh*

For the association class schema unit characterization object *AssociationClassCh* (see Figure 4.14), the *schemaUnit()* operation is defined, formally, as follows:

```
context AssociationClassCh::schemaUnit():AssociationClass
  body: AssociationClass.allInstances() ->
        any(a:AssociationClass| a.name = self.name and
        a.isAbstract = self.isAbstract and
        a.isDerived = self.isDerived and
        a.memberEnd -> collect(m:Property|
           Tuple{n:m.name, id:m.isDerived,
              idu:m.isDerivedUnion, ag:m.aggregation,
              l:m.lowerValue.oclAsType(LiteralInteger).value,
              u:m.upperValue. oclAsType(LiteralInteger).value,
              un:m.upperValue.oclIsTypeOf(LiteralUnlimitedNatural)})
        =
        self.associationClassMemberEnd ->
           collect(m:AssociationClassMemberEnd|
           Tuple{n:m.name, id:m.isDerived,
              idu:m.isDerivedUnion, ag:m.aggregation,
              l:m.lowerValue, u:m.upperValue,
              un:m.upperValue -> isEmpty()}))
```

This means that the *schemaUnit() operation* of *AssociationClassCh* is a query that gives the instance of *AssociationClass* whose value of the *name* attribute is equal to the value of the *name* attribute of *AssociationClassCh*; the *isAbstract* and *isDerived* attributes have the same value as the *isAbstract* and *isDerived* attributes of *AssociationClassCh*, respectively; and the ordered sequence of values of the *name* attributes of *memberEnds* is equal to the ordered sequence of the values of the *name* attributes of the *AssociationClassMemberEnd*(s) of *AssociationClassCh*.

4.2.7 Generalization schema unit

Generic description

A generalization schema unit is a taxonomic relationship *(IsA* relationship*)* between two classes' schema units.

The DBLP example shown in Figure 4.1 has nine generalization schema units represented by instances of *Generalization*: *EditedBook IsA Publication, Book IsA Publication,* *AuthoredPublication IsA Publication, EditedBook IsA Book, AuthoredBook IsA Book, AuthoredBook IsA AuthoredPublication, BookChapter IsA AuthoredPublication, JournalPaper IsA AuthoredPublication* and *BookSeriesIssue IsA Book.*

Abstract syntax

Each generalization is represented by an instance of *Generalization*, which is associated, through the general association, to a *Class*. The schema elements of a generalization are as follows: (1) an instance of *Generalization* σ; (2) the instance of the relationship between an instance of *Class* (general) and σ; and (3) the instance of the relationship between an instance of *Class* (specific) and σ.

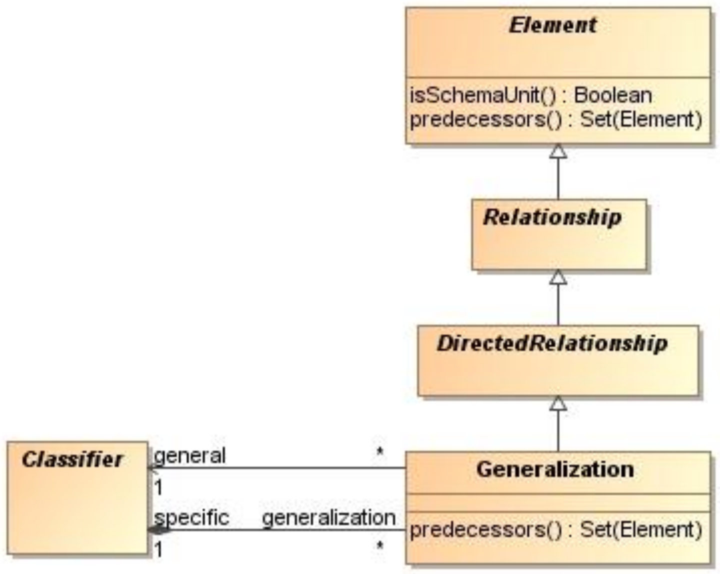


Figure 4.15 Generalization schema unit

Figure 4.15 shows the abstract syntax of a generalization schema unit. All instances of *Generalization* are schema units. Therefore, the *isSchemaUnit()* query operation is not redefined in *Generalization*. The *predecessors()* operation of *Generalization* is specified as follows:

```
context Generalization::predecessors():Set(Element)
  body: Element.allInstances() -> select (el:Element|
        el.oclIsTypeOf(Classifier) and
        (el.oclAsType(Classifier) = self.specific or
        el.oclAsType(Classifier) = self.general))
```

This means that the predecessors of a generalization are the classifiers that correspond to the general or specific element of the generalization.

Characterization object

For the generalization schema unit characterization object *GeneralizationCh* (see Figure 4.16), the *schemaUnit() operation* is defined, formally, as follows:

```
context GeneralizationCh::schemaUnit():Generalization
  body: Generalization.allInstances() -> any(g:Generalization|
        g.general.name = self.generalClassName and
        g.specific.name = self.specificClassName)
```

This means that the *schemaUnit() operation* of *GeneralizationCh* is a query that gives the instance of *Generalization* whose *name* attribute of its *genera*l class and its *specifi*c class has the same values as the *generalClassName* and *specificClassNam*e attributes of *GeneralizationCh*, respectively.

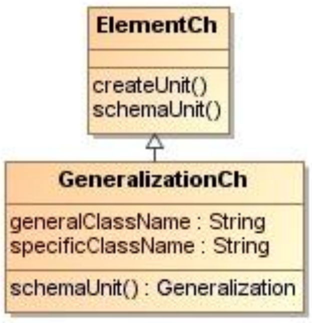

ElementCh
createUnit()
schemaUnit()
GeneralizationCh
generalClassName : String
specificClassName : String
schemaUnit() : Generalization

Figure 4.16 Generalization schema unit characterization object *GeneralizationCh*

4.2.8 Generalization set schema unit

Generic description

A generalization set schema unit defines a particular set of generalization schema units that describe the way in which a general class schema unit may be divided using specific class schema units. The generalization set schema unit includes the disjunction and covering constraints.

The DBLP example shown in Figure 4.1 has three generalization set schema units represented by instances of *GeneralizationSet* and named *typeOfPublication*, *typeOfBook* and *typeOfAuthoredPublication*. *TypeOfPublication* partitions *Publication* into *EditedBook* and *AuthoredPublication*; *TypeOfBook* partitions *Book* into *EditedBook* and *AuthoredBook*; and *TypeOfAuthoredPublication* partitions *AuthoredPublication* into *AuthoredBook*, *BookChapter* and *JournalPaper*. The three generalization sets are covering and disjoint.

Abstract syntax

Each generalization set is represented by an instance of *GeneralizationSet*. The schema elements of a generalization set that may be named *gs* are as follows: (1) the instance of *GeneralizationSet* ω; (2) if it has a name, the instance of attribute *name* of ω with value *gs*; (3) if the generalization set corresponds to a powertype extent, the instance of its relationship to an instance of *Classifier* (the powertype); (4) the instances of the two or more relationships between an instance of *Generalization* and ω; and (5) the instances of attributes *isCovering* and *isDisjoint* of ω.

Figure 4.17 shows the abstract syntax of a generalization set schema unit. All instances of *GeneralizationSet* are schema units. Therefore, the *isSchemaUnit()* query operation is not redefined in *GeneralizationSet*.

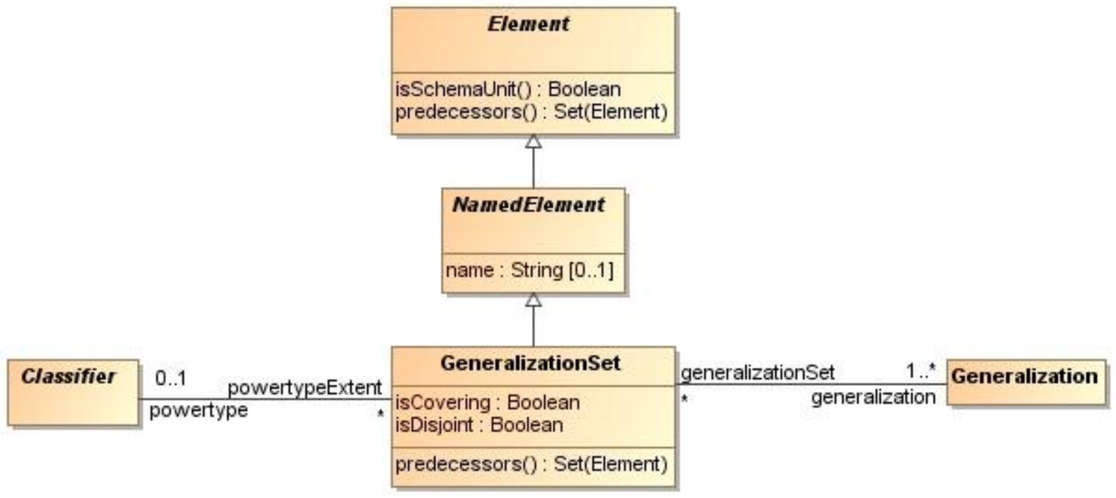


Figure 4.17 Generalization set schema unit

The *predecessors()* operation of *GeneralizationSet* is specified as follows:

```
 context Generalization::predecessors():Set(Element)
   body: Element.allInstances() -> select (el:Element|
          el.oclIsTypeOf(Generalization) and
          el.oclAsType(Generalization).generalizationSet ->
            includes(self)))
```

This means that the predecessors of a generalization set are the generalizations associated with such a generalization set.

Characterization object

For the generalization set schema unit characterization object *GeneralizationSetCh* (see Figure 4.18), the *schemaUnit() operation* is defined, formally, as follows:

```
 context GeneralizationSetCh::schemaUnit():GeneralizationSet
   body: GeneralizationSet.allInstances() ->
          any(gs:GeneralizationSet|
          gs.name -> notEmpty() implies gs.name = self.name and
          gs.powertype ->notEmpty() implies
            gs.powertype.name = self.powertype and
          gs.isCovering = self.isCovering and
          gs.isDisjoint = self.isDisjoint and
          gs.generalization -> collect(g:Generalization|
          Tuple{gc:g.general.name, sp:g.specific.name}) ->asSet() =
            self.participant -> collect(p:Participant|
          Tuple{gc:p.generalClassName, sp:p.specificClassName}) ->
            asSet())
```

This means that the *schemaUnit() operation* of *GeneralizationSetCh* is a query that gives the instance of *GeneralizationSet* whose *name* attribute of its powertype is equal to the one given in the *powertype* attribute of *GeneralizationSetCh*; the *isCovering* and *isDisjoint* attributes have values equal to the *isDisjoint* and *isCovering* values of *GeneralizationSetCh*; and each of its generalizations has the values of the names of its *genera*l class and its *specifi*c class as the values of *generalClassName* and *specificClassName* of the *participants* of the *GeneralizationSetCh,* respectively.

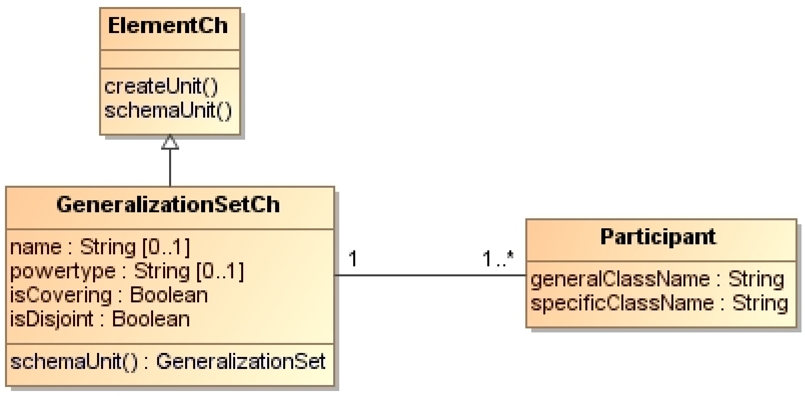


Figure 4.18 Generalization set schema unit characterization object *GeneralizationSetCh*

4.2.9 Constraint schema unit

Generic description

A constraint schema unit is a condition or restriction for the purpose of declaring some of the semantics of one or more schema units. UML has a few predefined static constraints with an associated graphic symbol. Cardinality, aggregation, composition, disjointness and covering constraints have already been included in other schema units. Therefore, a constraint schema unit is an *XOR* constraint or a constraint specified as invariant. An invariant is a constraint that is linked to a class schema unit. An invariant constraint consists of an OCL expression of type Boolean, which must be true for each instance of the class schema unit at any time.

The DBLP example shown in Figure 4.1 has eight constraint schema units represented by instances of *Constraint*. One is the *XOR* constraint shown in the figure and the other seven are the ones specified as invariants in OCL.

Abstract syntax

Each constraint is represented by an instance of *Constraint*. The schema elements of a constraint that may be named *co* are as follows: (1) the instance ψ of *Constraint*; (2) if it has a name, the instance of attribute *name* of ψ with value *co*; (3) the instances of its relationship with the instances of *Element* (*constrainedElement*); (4) the instance of its relationship with an instance of *Namespace* (context); and either (a) an instance of *Expression* with the instance of its attribute *symbol* and the instance of its relationship with ψ (the *specification*); or (b) an instance of *OpaqueExpression* with the instances of its attributes *body* and *language* and the instance of its relationship with ψ (the *specification*).

Figure 4.19 shows the abstract syntax of a constraint schema unit. All of the instances of *Constraint* are schema units. Therefore, the *isSchemaUnit()* query operation is not redefined in *Constraint*. However, the instances of *Expression* and *OpaqueExpression* are not schema units and the *isSchemaUnit()* query operation is specified as follows:

```
context Expression::isSchemaUnit():Boolean
body:  false
```

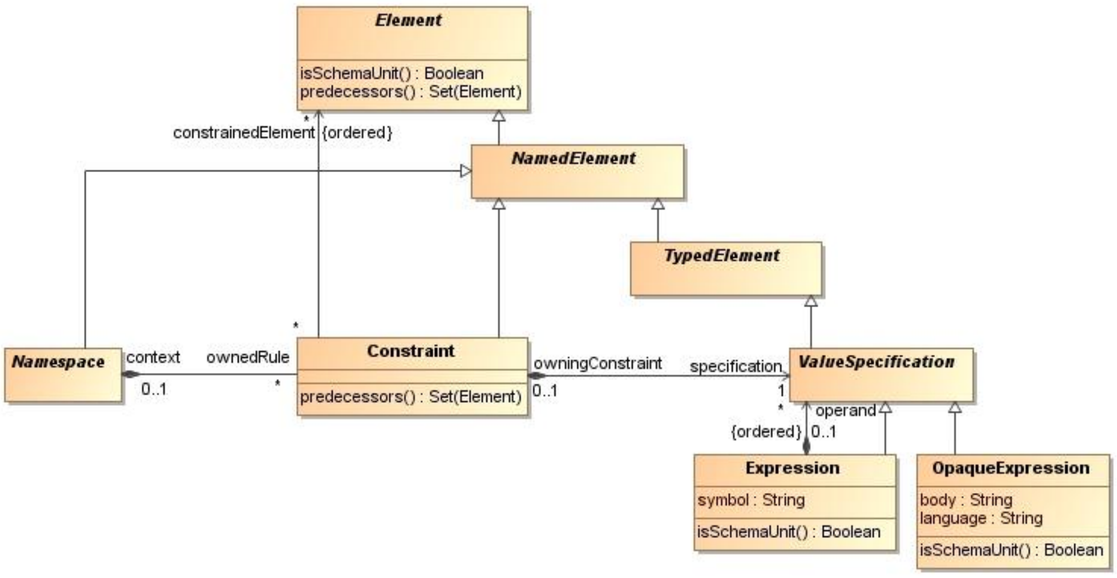


Figure 4.19 Constraint schema unit

```
context OpaqueExpression::isSchemaUnit():Boolean
body:  false
```

The *predecessors()* operation of *Constraint* is specified as follows:

```
context Constraint::predecessors():Set(Element)
  body: Element.allInstances() -> select (el:Element|
         (el.oclIsTypeOf(Namespace) and
         el.oclAsType(Namespace).ownedRule ->includes(self))
         or el.constraint -> includes(self))
```

This means that the predecessors of a constraint are the context of the constraint and the elements constrained by it.

Characterization object

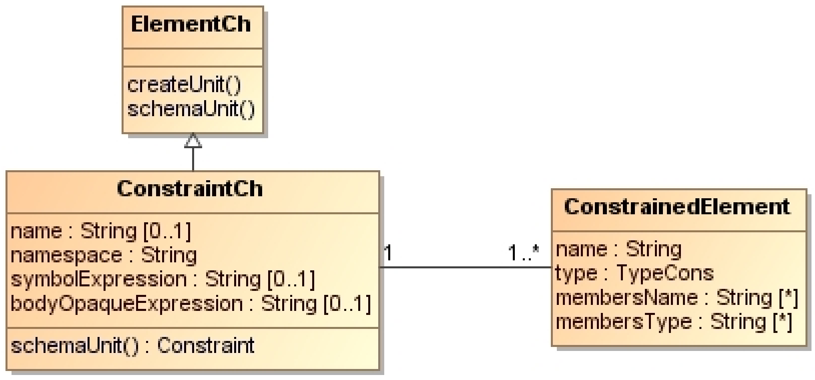


Figure 4.20 Constraint schema unit characterization object *ConstraintCh*

Figure 4.20 shows the constraint schema unit characterization object *ConstraintCh,* limited to the type of constraints found in the DBLP example. For the constraint schema unit characterization object *ConstraintCh*, the *schemaUnit() operation* is defined, formally, as follows:

```
context ConstraintCh::schemaUnit():Constraint
  body: Constraint.allInstances() -> any(c:Constraint|
          self.name ->notEmpty() implies
            c.name = self.name and
          self.namespace ->notEmpty() implies
            c.context.name = self.namespace and
          self.constrainedElement -> forAll( co|
          if co.typeCon = TypeElement::Class then
            c.constrainedElement.oclAsType(Class).name->includes(
              co.name)
          else
            if co.type = TypeElement::property then
             c.constrainedElement.oclAsType(Property).
              name->includes(co.name)
            else
              c.constrainedElement->
              collect(oclAsType(Association).name)->
              includes(co.name) and
              c.constrainedElement ->
                collect(oclAsType(Association).memberEnd) ->
                asSet() -> exists(me | co.membersName ->
                  includesAll(me.name)) and
              c.constrainedElement ->
                collect(oclAsType(Association).memberEnd.type) ->
                asSet() -> exists(me | co.membersType ->
                  includesAll(me.name))
            endif
          endif) and
          self.symbolExpression -> notEmpty() implies
            c.specification.oclAsType(Expression).symbol =
            self.symbolExpression and
          self.bodyOpaqueExpression -> notEmpty() implies
            c.specification.oclAsType(OpaqueExpression).body =
            self.bodyOpaqueExpression )
```

This means that the *schemaUnit() operation* of *ConstraintCh* is a query that gives the instance of *Constraint* whose *name* attribute is equal to the one given in the *name* attribute of *ConstraintCh*; the *name* attribute of the *context* has a value equal to the *namespace* attribute of *ConstraintCh*; each of its constrained elements has the *name* attribute that is the one given in the name of *ConstrainedElement* associated with *ConstraintCh*; and the *symbol* or *body* attributes of the specification of the constraint has the same value as the one given in the *symbolExpression* or *bodyOpaqueExpression*, respectively, of *ConstraintCh*.

The complete specification in the USE tool of the UML metaschema can be found in Appendix A. Appendix B shows a representative fragment of the instances defined in USE to specify the DBLP structural schema. The methods for creating schema units of the characterization objects are also provided in Appendix C.

5 SBVR meanings metaschema

Since the 1960s, many formal languages have been developed to allow software engineers to specify conceptual models. A few of these, such as UML (Rumbaugh, Jacobson and Booch 2004) and XML schema (Harold 2001), have become widely used standards.

These languages have been designed for use by software engineers, whose ultimate goal is to design software artifacts. Consequently, they employ notations and concepts that are not readily understood by "domain experts" (e.g., healthcare experts, finance experts, transportation experts, business managers, etc.) who understand the actual problem domain and are responsible for finding the solutions to problems.

Because of this, domain experts initiate discussions with software engineers to express their concerns and transcribe them into languages that only software engineers can read and write. Consequently, much of the business knowledge needed to operate an organization and deal with its environment is captured only in languages that business experts can neither read nor write.

Moreover, businesses change constantly and new decisions must be made accordingly in the business environment. The process of incorporating new decisions into the operating software that supports the affected business functions is error-prone, partly because business experts cannot actually read what the software engineer has written and verify that is consistent with the intentions, and partly because although other business elements involved in the decision are captured in software engineering languages, the software engineer who encounters them may not be aware of their relevance to the decision at hand.

For these reasons, OMG developed the Semantics of Business Vocabulary and Rules (SBVR) specification, which was published as an OMG Available Specification in February 2008. This specification was the first step in providing standard support for the "business vocabulary management" and "business rules management" tools that have recently appeared in the marketplace. These tools capture the business concepts and business rules in languages that are close enough to ordinary language so that business experts can read and write them, and at the same time formal enough to capture the intended semantics and present it in a form that is suitable for engineering the automation of rules. The specification provides a metamodel for the concepts used in capturing vocabulary and business rules.

For various reasons, the SBVR specification did not include normative specification of a language to be used by business-people to express their vocabulary and rules, partly because some of the tool builders involved used proprietary "business languages" in their tooling. There was also disagreement about using English-like text versus texts similar to other natural languages or graphical representations.

SBVR itself was written in "Structured English," a language that is defined in the specification, but is not normative and may not actually be supported by any tooling. Moreover, in June 2008, OMG submitted a request for a proposal to define at least one standard language in which business experts can express their vocabulary and rules (Object Management Group 2008b). For this reason, Chapter 7 of this thesis proposes a metamodel of this, non-normative SBVR Structured English language and provides a set of operations to derive the "Structured English representations" from SBVR meanings.

The rest of this chapter is structured as follows:

- Section 5.1 gives an overview of the meanings and the structure of some of the meanings described in the SBVR specification.
- Section 5.2 defines, following the translation approach described in Chapter 3, the schema units of the SBVR metaschema, the precedence relationships between them and the characterization objects of such schema units.

5.1 Overview of SBVR meanings

The Object Management Group (OMG) published *Semantics of Business Vocabulary and Business Rules (SBVR), v.1.0* (Object Management Group 2008a) as an *Available Specification* in February 2008. This document defines the vocabulary and rules for documenting the semantics of business vocabulary, business facts and business rules. The specification is applicable to the domain of business vocabularies and business rules for all kinds of business activities in all kinds of organizations. It is conceptualized optimally for business-people rather than for automated rules processing, and is designed to be used for business purposes, regardless of information system designs.

SBVR was initially developed by the Business Rules Group (Object Management Group 2004), which has been working exclusively in this area since the late 1980s. Key notions of the SBVR approach are presented succinctly in the BRG's Business Rule Manifesto (Business Rule Group 2003).

SBVR is based on the idea that the purpose of systems for the management of business vocabulary and rules is to capture and maintain the expression of business meanings. Meanings exist only in business decision makers, and SBVR divides such meanings into two categories (see Figure 5.1):

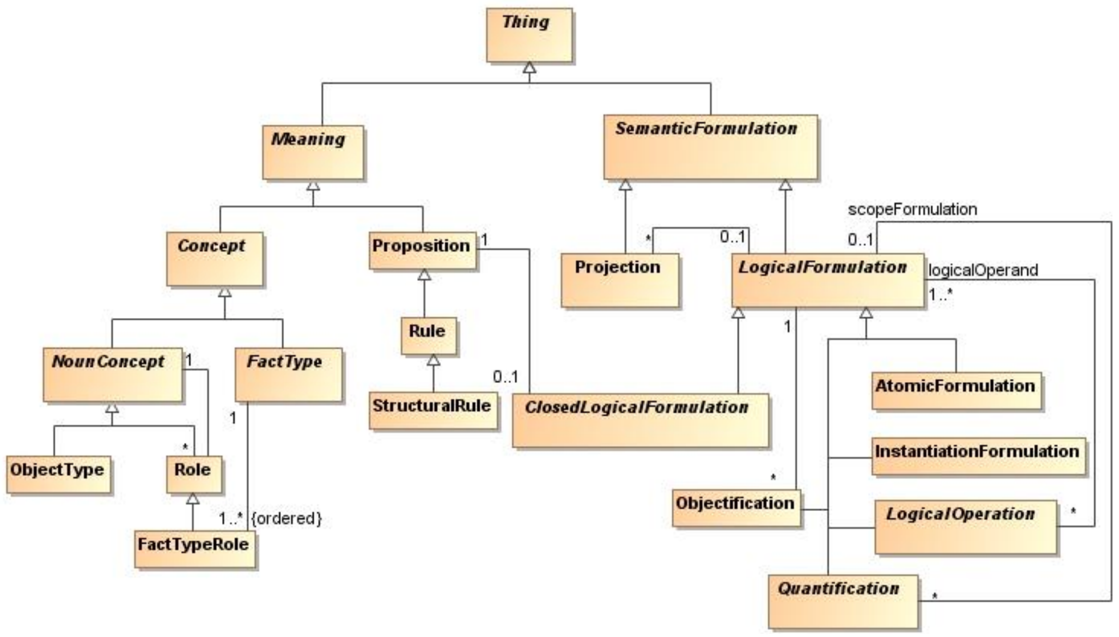


Figure 5.1 Fragment of the abstract syntax of the SBVR metamodel

- Concepts: classifiers of things (noun concepts) and classifiers of states and actions (verb concepts or fact types).
- Propositions: meanings of statements ("complete thoughts").

For example, an instance of a noun concept may be represented by a designation or name as "edited book" or "person." Instances of fact types may be represented by fact type forms. For example, "editor has edited book" may represent an instance of an associative fact type between the "editor" and "edited book" fact type roles. SBVR also defines a number of more specialized categories of concept. Additional details about each category are given in the next section.

A rule (also called element of guidance) is a proposition that guides the conduct of business. SBVR further divides the meaning of "rule" into the following subcategories:

- Structural rules, which are statements of necessity, stating properties that are fundamental to the concepts involved.
- Operational rules (not shown in Figure 5.1), which state intents and requirements for the business operation.
- Pieces of advice (not shown in Figure 5.1), which state facts that clarify the scope of rules.

SBVR includes constructs called semantic formulations that structure the meaning of rules or the definition of concepts. There are two kinds of semantic formulations: logical formulations and projections. Logical formulations are further specialized into logical operations, quantifications, atomic formulations based on fact types, and other formulations for special purposes, such as objectification. Logical formulations are recursive. Several kinds of logical formulations embed other logical formulations, and some of them introduce logical variables (not shown in Figure 5.1).

In SBVR, any rule is constructed by applying a modal operator (necessity, obligation, possibility and permissibility) to a logical formulation. Since obligation, possibility and permissibility formulations may not be represented in UML structural schemas, they have been left out of the scope of this thesis.

For example, “It is necessary that each edited book has at least one editor” is a proposition based on a necessity formulation (structural rule), which is structured as follows:

```
The structural rule embeds a universal quantification.
. The universal quantification introduces a first variable.
. . The first variable ranges over the concept "edited book."
. The universal quantification scopes over an existential quantification.
. . The existential quantification has a minimum cardinality of 1.
. . The existential quantification introduces a second variable.
. . . The second variable ranges over the concept "editor."
. . The existential quantification scopes over an atomic formulation.
. . . The atomic formulation is based on the fact type "edited book has editor."
. . . . The atomic formulation has a role binding.
. . . . . The role binding is of the role "edited book" of the fact type.
. . . . . The role binding binds to the first variable.
. . . .The atomic formulation has a second role binding.
. . . . . The second role binding is of the role "editor" of the fact type.
. . . . . The second role binding binds to the second variable.
```

The indentation in the example shows a hierarchical structure in which a logical formulation at one level operates on, or quantifies over, one or more logical formulations at the next highest level. Each kind of logical formulation, including quantification and logical operations, can be embedded in another logical formulation at any depth and in almost any combination. Note also that each line in the example corresponds to an instance of an element of the SBVR metamodel.

Figure 5.2 shows the same structural rule as an instance of the SBVR metamodel. Complete representations of structural rules as instances of the SBVR metamodel are quite cumbersome, as shown in Figure 5.2. Therefore, simplified versions of representations of rules, as shown in Figure 5.3, are used in the rest of the thesis.

Figure 5.3, combines in a single tree node, the kind of SBVR instance and schema element referenced by the node. That is, a node representing an instance of *Variable* or *FactTypeRole* includes the name of the concept that it ranges over; a node representing an instance of a subtype of *QuantificationFormulation* includes the cardinality values; and a node representing an instance of *AtomicFormulation* includes the expression of the fact type that it is based on. Additional details of other elements are introduced when they arise.

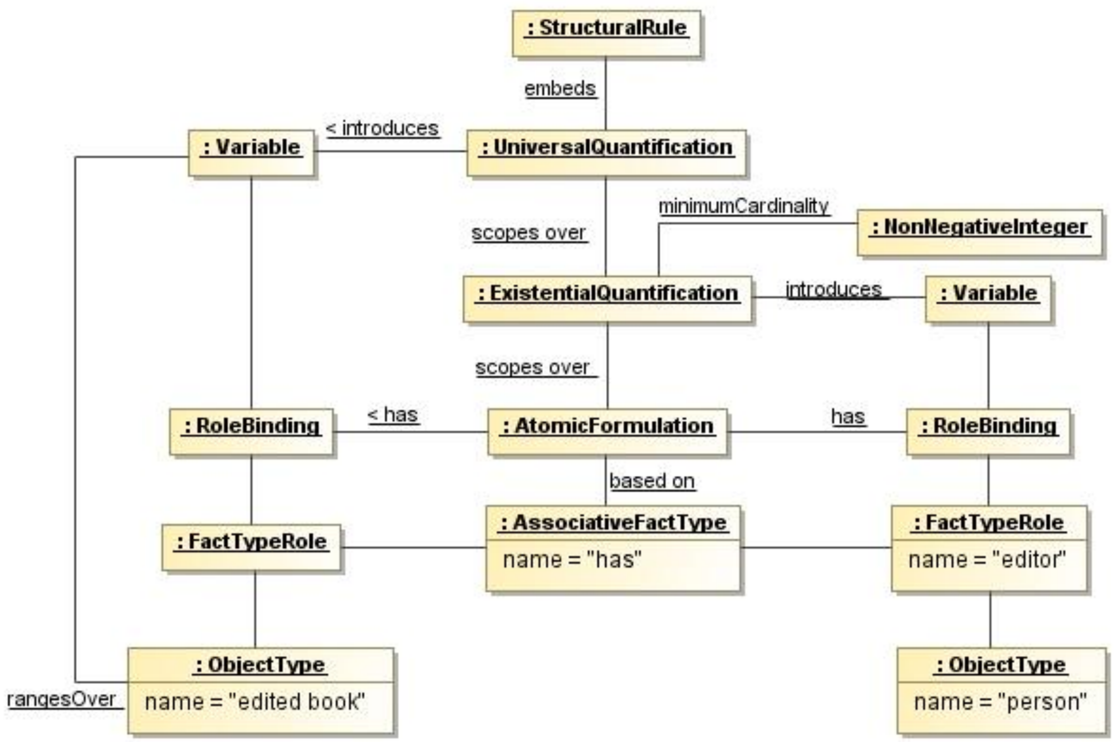


Figure 5.2 Example of structural rule as an instance of the SBVR metamodel

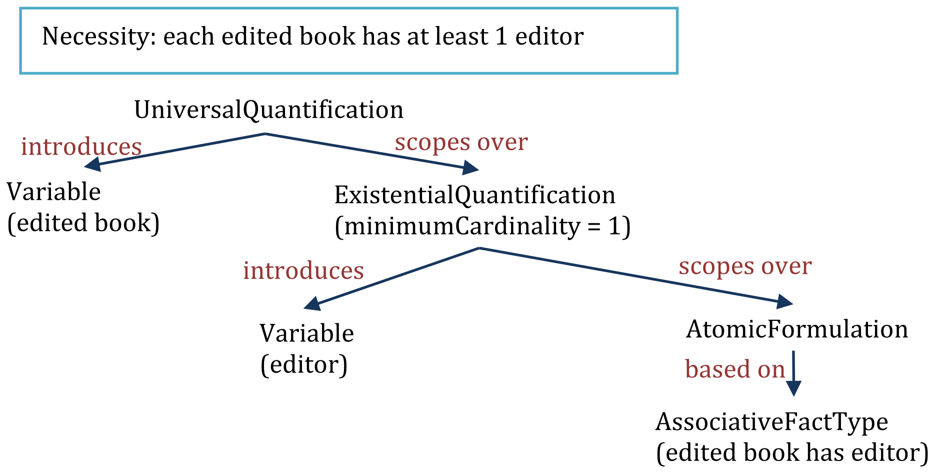

Necessity: each edited book has at least 1 editor
UniversalQuantification
introduces
scopes over
Variable (edited book)
ExistentialQuantification (minimumCardinality = 1)
introduces
scopes over
Variable (editor)
AtomicFormulation
based on
AssociativeFactType (edited book has editor)

Figure 5.3 Simplified version of the structure of a structural rule

In addition to logical formulation, SBVR specifies another type of structure of meanings (semantic formulation) called projection. Projections are used to formulate definitions of meanings. For example, a noun concept may be defined by the set of things (instances) that exist in the domain at any time. In the DBLP example, the object type "authored publication" can be defined as the disjunction of the object types named "authored book," "book chapter" and "journal paper."

More details about the various types of concepts, structural rules and other types of semantic formulation are given in the next section.

5.2 Schema units of the SBVR metaschema

This section describes the subset of the SBVR metamodel that is necessary to describe conceptual schemas as a combination of concepts and facts as defined in SBVR. In order to translate the subset of UML described in Chapter 4 to SBVR, only the subset of SBVR that describes meanings (concepts and propositions) and semantics formulations is necessary. Based on the approach to translating MOF metaschemas described in Chapter 3, this chapter provides the definition of (i) schema units, (ii) the relation of precedence between them, and (iii) the objects that characterize such schema units (characterization objects).

All elements concerning representations and business statements, rather than meanings, have been excluded from the SBVR meanings metamodel. Moreover, elements concerning meanings that have no equivalents in a UML conceptual schema have also been excluded:

- Questions: meanings that are interrogatories.
- Uniform Resource Identifiers (URI) vocabulary.
- Modal logics different than modal formulations of necessity (i.e., operational rules and pieces of advice).

As in the previous chapter, the fragment is described in terms of its schema units that is, its knowledge components.

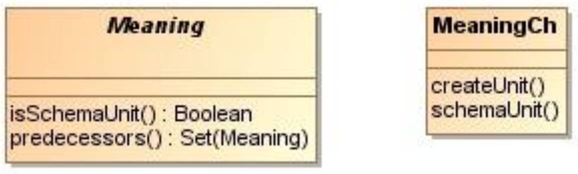


Figure 5.4 Definition of *Meaning* and its characterization object *MeaningCh*

In this case, all SBVR metaclasses for which some instances are schema units are subtypes of the abstract metaclass *Meaning*. In order to define the schema units, *Meaning* includes two operations (*isSchemaUnit()* and *predecessors()*), as shown in Figure 5.4.

The query operation *isSchemaUnit()* is defined formally as follows:

```
context Meaning::isSchemaUnit():Boolean
body:  false
```

The query operation is redefined in all subtypes that are schema units and are not abstract.

In the *Meaning* metaclass of Figure 5.4, the *predecessors()* operation is specified as follows:

```
context Meaning::predecessors:Set(Meaning)
  pre: isSchemaUnit()
  body: Set{}
```

This means that, by default, no schema units have predecessors.

As explained in Chapter 3, characterization objects are used to characterize schema units. In the SBVR metaschema, there is a characterization object type for each subtype of *Meaning* such that some or all of its instances represent schema units. Each characterization object type includes the set of attributes that characterize the schema unit and two operations: *createUnit* and *schemaUnit*. The former creates a schema unit from its characterization object, and the latter gives the schema unit corresponding to the characterization object.

The specification of the *createUnit* operation is the same for all characterization object types, and therefore is specified in *MeaningCh* as follows:

```
context MeaningCh::createUnit()
  post: schemaUnit() -> notEmpty()
```

The *schemaUnit() operation* is redefined in each subtype of *MeaningCh*.

In an SBVR schema, the schema units are object types, individual concepts, value types, characteristics, associative fact types, is-property-of fact types, partitive fact types, categorization fact types, reference schemes and structural rules.

The following subsections define each schema unit in terms of its schema elements. They provide, for each schema unit, a generic description of it, its abstract syntax, the specifications of the *isSchemaUnit()* and *predecessors()* operations used to define it, and its schema unit characterization object.

5.2.1 Object type schema unit

Generic description

As stated in the previous section, SBVR describes two types of meanings: concepts and propositions. Concepts are further subdivided into noun concepts and fact types. Noun concepts are defined as classifiers of things. A subtype of a noun concept is an object type.

In SBVR, an object type schema unit is defined as a noun concept that classifies things based on their common properties. The meaning of an object type is equivalent to the meaning of "entity type" in conceptual modeling. Olivé captures the essence of the meaning of entity type by providing the following definition: "An entity type is a concept whose instances at a given time are identifiable individual objects that are considered to exist in the domain at that time" (Olivé 2007).

An object type schema unit may also include a definition that describes its meaning through constraints that satisfy a set of things. For example, the instances of *Book* may be defined as the union of the instances of *EditedBook* and *AuthoredBook*. In this case, the object type schema unit includes the elements needed to structure such a definition. However, in order to make this chapter easier to understand, definitions will not be included in object type schema units from this section to Section 5.2.8. Such definitions will be described in Section 5.2.9.

The DBLP example, as an instance of the SBVR metaschema, includes 19 object type schema units: *Person*, *Publication*, *Book*, *AuthoredPublication*, *EditedBook*, *AuthoredBook*, *BookChapter*, *JournalPaper*, *BookSection*, *BookSeriesIssue*, *BookSeries*, *JournalSection*, *JournalIssue*, *ConferenceEdition*, *ConferenceSeries*, *JournalVolume* and *Journal* (those whose equivalent meaning in UML is represented by a class), as well as *Authorship* and *Editorship* (those whose equivalent meaning in UML is represented by an association class).

Abstract syntax

Each object type is represented by an instance of *ObjectType*. The schema elements of an object type named *o* are as follows: (1) the instance β of *ObjectType*; and (2) the instance of attribute *name* of β with value *o*.

Figure 5.5 shows the abstract syntax of the object type schema unit and the value type schema unit (described in the next section). The *isSchemaUnit()* query operation is redefined as follows:

```
context ObjectType::isSchemaUnit():Boolean
body:  true
```

This means that all instances of *ObjectType* are schema units. In constrast, here, object type has no predecessors. The operation predecessors is specified in Section 5.2.9, when projections are introduced.

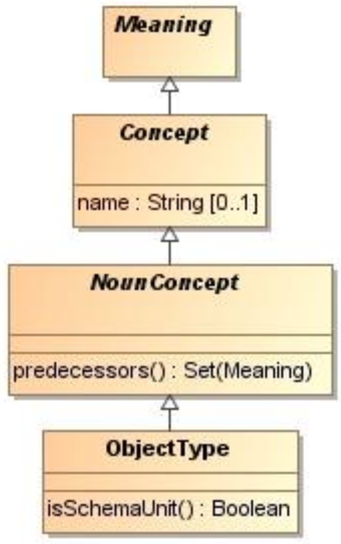


Figure 5.5 Object type schema unit

Characterization object

Figure 5.6 shows the characterization object for the object type and value type schema unit *NounConceptCh*.

The *schemaUnit() operation* is formally defined, as follows:

```
context NounConceptCh::schemaUnit():NounConcept
  body:  NounConcept.allInstances() -> any(c:NounConcept|
         c.name = self.name and
         if isValueType then c.oclIsTypeOf(ValueType)
         else c.oclIsTypeOf(ObjectType) endif)
```

This means that the *schemaUnit() operation* of *NounConceptCh* is a query that gives the instance of *ObjectType* or *ValueType* (depending on the *isValueType* attribute of *NounConceptCh*) whose attribute *name* has the same value as the one given in the attribute *name* of *NounConceptCh*.

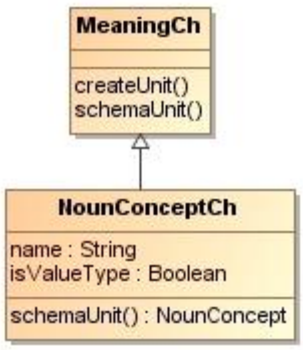


Figure 5.6 Object type and value type schema unit characterization object *NounConceptCh*

5.2.2 Value type schema unit

Generic description

SBVR does not distinguish between entity types and lexical entity types (entity types whose instances are words). However, this distinction is necessary in order to consistently translate UML data types to SBVR and vice versa. Therefore, *ValueType*, a special subtype of *NounConcept* not included in the SBVR specification has been created. Note that the name "*value type"* has been taken from Object-Role Modeling (ORM) (Halpin, 2008), a fact-oriented language defined by one of the authors of SBVR.

A value type schema unit is defined as a noun concept whose instances are words in the language used in the domain.

As an instance of the SBVR metaschema, the DBLP example includes four value type schema units: one named "string" and whose meaning is equivalent to the meaning of the *String PrimitiveType* of UML; one named "*natural"* and whose meaning is equivalent to the meaning of the *Natural data type* of UML; one named "date" and whose meaning is equivalent to the data type named *Date* of UML; and one named "gender" whose meaning is equivalent to the *Gender* enumeration of UML.

Abstract syntax

Each value type is represented by an instance of *ValueType*. The schema elements of a value type named *v* are as follows: (1) the instance β of *ValueType*; and (2) the instance of attribute *name* of β with value *v*.

Figure 5.7 shows the abstract syntax of the value type schema unit. The *isSchemaUnit()* query operation was defined in the previous section.

Characterization object

Figure 5.6 showed the characterization object of each value type schema unit. The *schemaUnit()* operation was defined in the previous section and the *predecessors()* operation will be defined in Section 5.2.9.

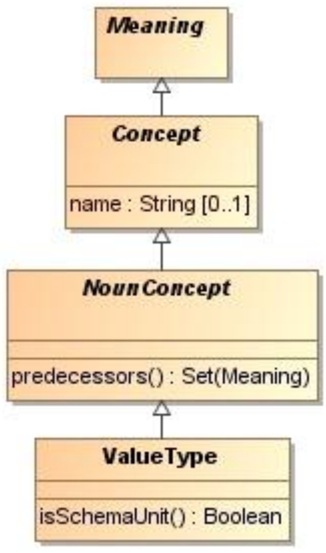


Figure 5.7 Value type schema unit.

5.2.3 Individual concept schema unit

Generic description

"Individual concept" is a special type of noun concept. In SBVR, it is a noun concept that corresponds to only one object. For example, in DBLP, the individual concept *Female* corresponds to a noun concept whose one instance is the individual gender of women.

The SBVR document defines an individual concept as a noun concept, but its use is confused with "instance" throughout the document. For example, the definition of instantiation formulation, which is used to bind an instance of a concept with the concept, includes as an example the binding of an individual concept (rather than its instance) with a concept.

To avoid further confusion, this thesis considers that an individual concept is a noun concept and that the instantiation formulation binds instances of concepts to concepts.

Abstract syntax

Each individual concept is represented by an instance of *IndividualConcept*. The schema elements of an individual concept named *in* are as follows: (1) the instance δ of *IndividualConcept*; and (2) the instance of attribute *name* of δ with value *in*.

Figure 5.8 shows the abstract syntax of the individual concept schema unit. The *isSchemaUnit()* query operation is redefined as follows:

```
context IndividualConcept::isSchemaUnit():Boolean
body:  true
```

This means that all instances of *IndividualConcept* are schema units. Additionally, the individual concept schema unit has no predecessors.

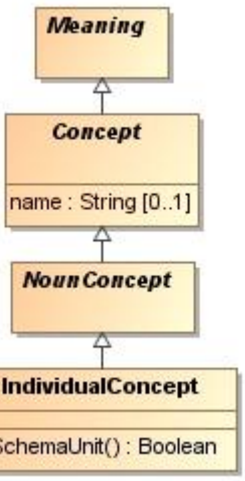


Figure 5.8 Individual concept schema unit

Characterization object

Figure 5.9 shows the individual concept characterization object *IndividualConceptCh.*

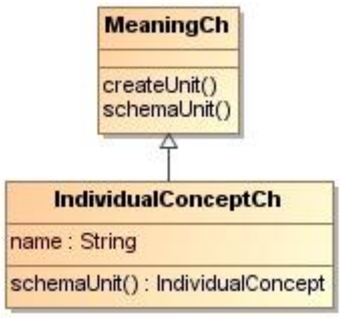


Figure 5.9 Individual concept schema unit characterization object *IndividualConceptCh*

The *schemaUnit() operation* is formally defined as follows:

```
context IndividualConceptCh::schemaUnit():IndividualConcept
  body:  IndividualConcept.allInstances() ->
            any(c:IndividualConcept| c.name = self.name)
```

This means that the *schemaUnit() operation* of *IndividualConceptCh* is a query that gives the instance of *IndividualConcept* whose attribute *name* has the same value as the one given in the attribute *name* of *IndividualConceptCh.*

5.2.4 Characteristic schema unit

Generic description

As stated above, concepts are subdivided into noun concepts and fact types. A fact type (also called verb concept) is a concept that is the meaning of a verb involving one or more noun concepts (fact type roles). An instance of a fact type is an event, activity, situation or circumstance that occurs in the actual domain, and for each role of the fact type there is an instance of said role involved in the instance of the fact type. Each fact type has at least one role. Depending on the number of roles participating in a fact type (arity of the fact type), a fact type is further divided into characteristic, binary fact type or *n-ary* fact type where *n* > 2.

A characteristic or unary fact type schema unit is a fact type that has exactly one role. The DBLP example, as an instance of the SBVR metaschema, includes two characteristic schema units. The first one has a fact type role that ranges over the object type named *BookChapter* and is named "*being ConferencePaper.*" The second characteristic is a fact type role that ranges over the object type named *JournalPaper* and is also named "*being ConferencePaper.*" "Book chapter being conference paper" and "journal paper being conference paper" represent the facts that a book chapter is a conference paper or a journal paper is a conference paper, respectively.

Abstract syntax

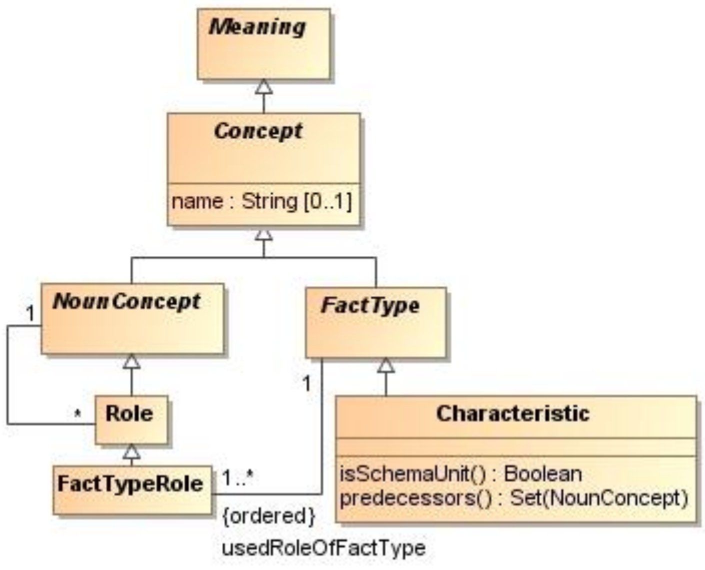


Figure 5.10 Characteristic schema unit

Each characteristic is represented by an instance of *Characteristic*. The schema elements of a characteristic named *ch* are as follows: (1) the instance of β of *Characteristic*; (2) the instance of attribute *name* of β with value *as*; (3) the instance of *FactTypeRole* that is a role of β; (4) if the *FactTypeRole* has a name, the instance of the attribute *name* with its value; (5) the instance of the relationship of the *FactTypeRole* to the *NounConcept* concept that the role ranges over; and (6) the instance of the relationship of the *FactTypeRole* with β.

Figure 5.10 shows the abstract syntax of the characteristic schema unit. The *isSchemaUnit()* query operation is redefined in *Characteristic* as follows:

```
context Characteristic::isSchemaUnit():Boolean
body:  true
```

The *predecessors()* operation of *Characteristic* is specified as follows:

```
context Characteristic::predecessors():Set(NounConcept)
  body: self.factTypeRole -> collect(nounConcept) -> asSet()
```

This means that the predecessor of a characteristic is the noun concept that the fact type role of the characteristic ranges over.

Characterization object

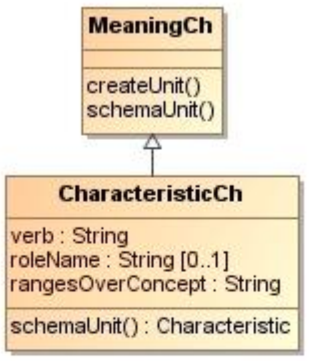


Figure 5.11 Characteristic schema unit characterization object *CharacteristicCh*

For the characteristic characterization object *CharacteristicCh* (see Figure 5.11), the schema unit operation is formally defined as follows:

```
 context CharacteristicCh::schemaUnit():Characteristic
  body:  Characteristic.allInstances() -> any(ch:Characteristic |
         ch.name = self.verb and ch.factTypeRole -> collect(ft|
           Tuple{n:ft.name,c:ft.nounConcept.name}) =
           Tuple{n:self.roleName, c:self.rangesOverConcept} )
```

This means that the *schemaUnit()* operation of *CharacteristicCh* is a query that gives the instance of a *Characteristic* whose attribute *name* has the same value as the one given in the attribute *verb* of *CharacteristicCh*; the *name* attribute of its *factTypeRole* and the name of the concept which its *factTypeRole* ranges over have the same values as the ones given in the attribute *roleName* and *rangesOver* of *Characteristic*.

5.2.5 Associative and categorization fact type schema units

Generic description

Fact types with more than one role are classified, based on the semantic nature of the fact type, into associative fact type or specialization fact type.

An associative fact type is a fact type that has more than one role, for which there is a non-hierarchical relationship among the participants involved in the fact type. Additionally, there are two particular kinds of binary associative fact types: partitive fact types and is-property-of fact types. A partitive fact type is a binary associative fact type that means that a given part (i.e., the instance of the concept that plays one of the roles in the fact type) is in the composition of a whole (i.e., the instance of the concept that plays the other role in the fact type). An is-property-of fact type means that the instance of the concept that plays the first role in the fact type constitutes an essential quality of the instance of the concept that plays the second role in the fact type.

A specialization fact type indicates hierarchical relationships among concepts and is futher divided, in SBVR, as a categorization fact type and a contextualization fact type. A categorization fact type indicates that the instance of the concept that plays the first role in the fact type is also an instance of the concept that plays the second role in the fact type (i.e., the category). A contextualized fact type means a hierarchical relationship from a particular perspective or viewpoint or in a certain situation. The semantics of contextualized fact types is not covered explicitly in UML. Therefore, for the purpose of this thesis, only categorization fact types have been considered.

The DBLP example, as an instance of the SBVR metaschema, includes 12 associative fact type schema units that may be expressed in ordinary language as follows:

"author has authored publication,"
"book chapter is part of book section,"
"book chapter is part of book series issue,"
"book chapter is part of edited book,"
"conference edition is published in book series issue,"
"conference edition is published in edited book,"
"conference edition is published in journal issue,"
"book section is part of edited book,"
"editor has edited book,"
"journal paper is part of journal issue,"
"journal paper is part of journal section,"
"person publishes publication."

DBLP includes five partitive fact type schema units expressed in natural language as follows:

"book series includes book series issue,"
"conference series includes conference edition,"
"journal includes journal volume,"
"journal issue includes journal section,"
"journal volume includes journal issue."

It also includes nine categorization fact type schema units that may be expressed in ordinary language as follows:

"book is a category of publication,"
"edited bok is a category of publication,"
"authored publication is a category of publication,"
"edited book is a category of book,"
"authored book is a category of book,"
"authored book is a category of authored publication,"
"book chapter is a category of authored publication,"
"jounral paper is a category of authored publication,"
"book series issue is a category of book."

Finally, DBLP includes 29 is-property-of fact type schema units that may represented in ordinary language as follows:

"authorship has order,"
"book has home page,"
"book has isbn,"
"book has num pages,"
"book has publication year,"
"book has publisher,"
"book chapter has end page,"
"book chapter has ini page,"
"book section has order,"
"book section has title,"
"book series has id,"
"book series has publisher,"

"book series issue has number,"
"conference edition has city,"
"conference edition has country,"
"conference edition has home page,"
"conference edition has title,"
"conference edition has year,"
"conference series has acronym,"
"conference series has name,"
"editorship has order,"
"journal has issn,"
"journal has title,"
"journal issue has month,"
"journal issue has num pages,"
"journal issue has number,"
"journal issue has year,"
"journal paper has end page,"
"journal paper has ini page,"
"journal section has order,"
"journal section has title,"
"journal volume has volume,"
"person has gender,"
"person has home page,"
"person has name,"
"person has num publications,"
"publication has edition,"
"publication has title,"
"publication has year."

Abstract syntax

Each associative fact type that is neither an is-property-of fact type nor a partitive fact type is represented by an instance of *AssociativeFactType*. The schema elements of an associative fact type named *as* are as follows: (1) the instance ρ of *AssociativeFactType*; (2) the instance of attribute *name* of ρ with value *as*; (3) the ordered instances of *FactTypeRole* that are roles of ρ; and (4) for each of these fact type roles: (a) the instance of its attribute *name* with its value (if it has a name); (b) the instance of its relationship to the *NounConcept* concept that the role ranges over; and (c) the instance of its relationship with ρ in a given order.

Each is-property-of fact type is represented by an instance of *IsPropertyOfFactType*. The schema elements of an is-property-of fact type named *is* are as follows: (1) the instance η of *IsPropertyOfFactType*; (2) the instance of attribute *name* of η with value *is*; (3) the ordered instances of *FactTypeRole* that are roles of η; and (4) for each of these fact type roles: (a) the instance of its attribute *name* with its value (if it has a name); (b) the instance of its relationship to the *NounConcept* concept that the role ranges over; and (c) the instance of its relationship with η in a given order.

Each partitive fact type is represented by an instance of *PartitiveFactType*. The schema elements of a partitive fact type named *pa* are as follows: (1) the instance ψ of *PartitiveFactType*; (2) the instance of attribute *name* of ψ with value *pa*; (3) the ordered instances of *FactTypeRole* that are roles of ψ; and (4) for each of these fact type roles: (a) the instance of its attribute *name* with its value (if it has a name); (b) the instance of its relationship to the *NounConcept* concept that the role scopes over; and (c) the instance of its relationship with ψ in a given order.

Each categorization fact type is represented by an instance of *CategorizationFactType*. The schema elements of a categorization fact type named *ca* are as follows: (1) the instance γ of *CategorizationFactType*; (2) the instance of attribute *name* of γ with value *ca*; (3) the ordered instances of *FactTypeRole* that are roles of γ; and (4) for each of these fact type roles: (a) the instance of its attribute *name* with its value (if it has a name); (b) the instance of it relationship to the *NounConcept* concept that the role scopes over; and (c) the instance of its relationship with γ in a given order.

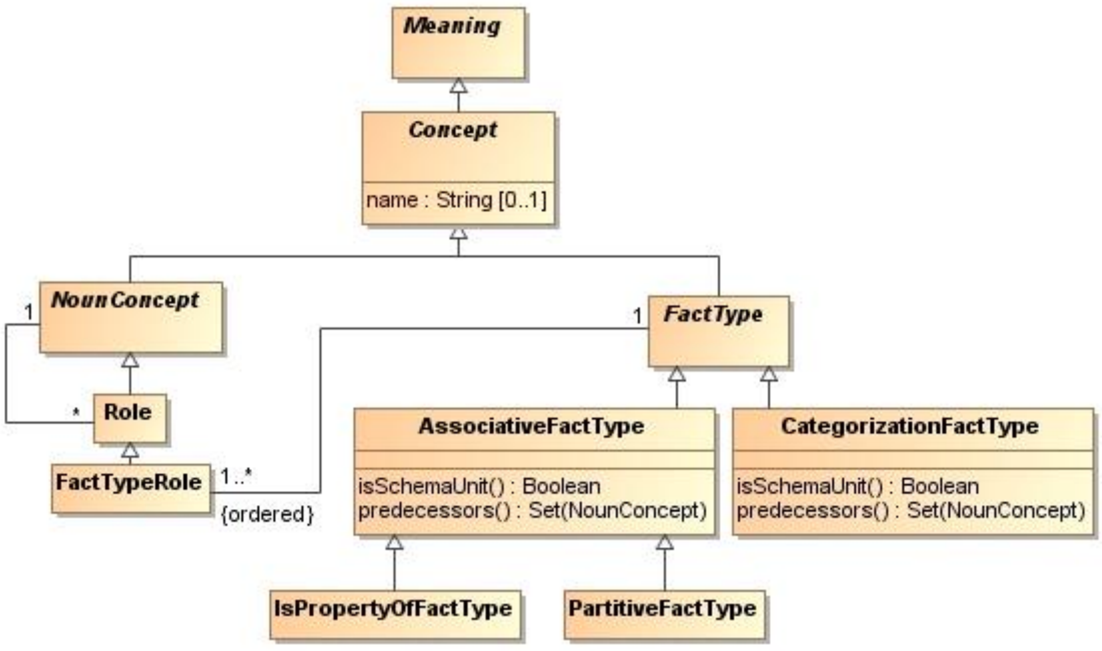


Figure 5.12 Associative and categorization fact type schema units

Figure 5.12 shows the abstract syntax of associative and categorization fact type schema units. The *isSchemaUnit()* query operation of *AssociativeFactType* and *CategorizationFactType* are redefined as follows:

```
context AssociativeFactType::isSchemaUnit():Boolean
body:  true

context CategorizationFactType::isSchemaUnit():Boolean
body:  true
```

The *predecessors()* operations of *AssociativeFactType* and CategorizationFactType are specified as follows:

```
context AssociativeFactType::predecessors():Set(NounConcept)
  body: self.factTypeRole -> collect(nounConcept) -> asSet()

context CategorizationFactType::predecessors():Set(NounConcept)
  body: self.factTypeRole -> collect(nounConcept) -> asSet()
```

This means that the predecessors of an associative or categorization fact type are the noun concepts that the fact type roles of the fact type range over.

Characterization object

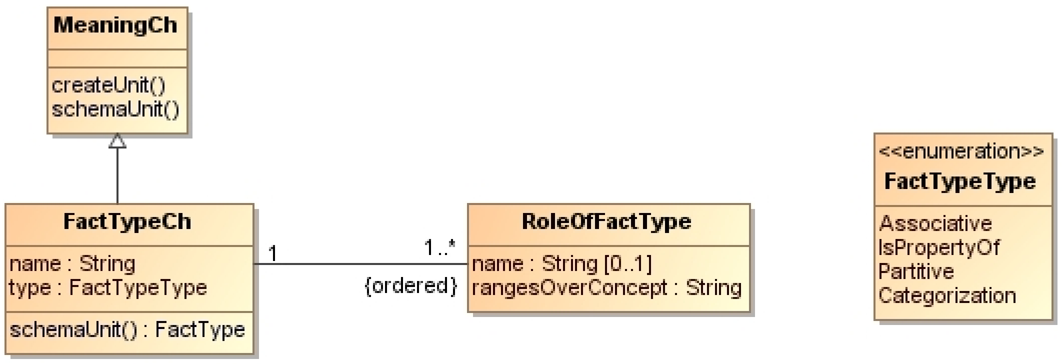


Figure 5.13 Fact type schema unit characterization object *FactTypeCh*

Figure 5.13 shows *FactTypeCh*, the characterization object of the associative fact type, is-property-of fact type, partitive fact type and categorization fact type schema units. The *schemaUnit()* operation is formally defined as follows:

```
context FactTypeCh::schemaUnit():FactType
  body:  FactType.allInstances() -> any(ft:FactType |
            ft.name = self.name     and
            self.type = FactTypeType::Associative implies
              ft.oclIsTypeOf(AssociativeFactType) and
            self.type = FactTypeType::IsPropertyOf implies
              ft.oclIsTypeOf(IsPropertyOfFactType) and
            self.type = FactTypeType::Partitive implies
              ft.oclIsTypeOf(PartitiveFactType) and
            self.type = FactTypeType::Categorization implies
              ft.oclIsTypeOf(CategorizationFactType) and
            ft.factTypeRole -> collect(ft|
              Tuple{n:ft.name,c:ft.nounConcept.name}) =
            self.roleOfFactType -> collect(rf|
              Tuple{n:rf.name, c:rf.rangesOverConcept}))
```

This means that the *schemaUnit() operation* of *FactTypeCh* is a query that gives the instance of a *FactType* that is of the subtype indicated in the *type* attribute of *FactTypeCh*, whose attribute *name* has the same value as the one given in the attribute *name* of *FactTypeCh*; it has the *factTypeRoles* whose *name* and the name of the concept which it ranges over are the same as the ones given in the attribute *name* and *rangesOver* of *RoleOfFactType*.

5.2.6 Categorization scheme and segmentation schema units

Generic description

SBVR defines a particular kind of object type, called *categorization scheme*, for partitioning things in the categorization fact type dimension. The extension (set of instances) of a given general concept is partitioned into the extensions of the categories of said general concept.

Moreover, a segmentation is a particular kind of categorization scheme whose contained category or categories are complete (total) and disjoint with respect to the general concept that has the categorization scheme.

The DBLP example, as an instance of the SBVR metaschema, includes three segmentations, named "type of publication," "type of book" and "type of authored publication," which partition the concepts named "publication," "book" and "authored publication," respectively.

The segmentation "type of publication" for "publication" contains the categories named "edited book" and "authored publication."

The segmentation "type of book" for "book" contains the categories named "edited book" and "authored book."

The segmentation "type of authored publication" for "authored publication" contains the categories named "authored book," "book chapter" and "journal paper."

Abstract syntax

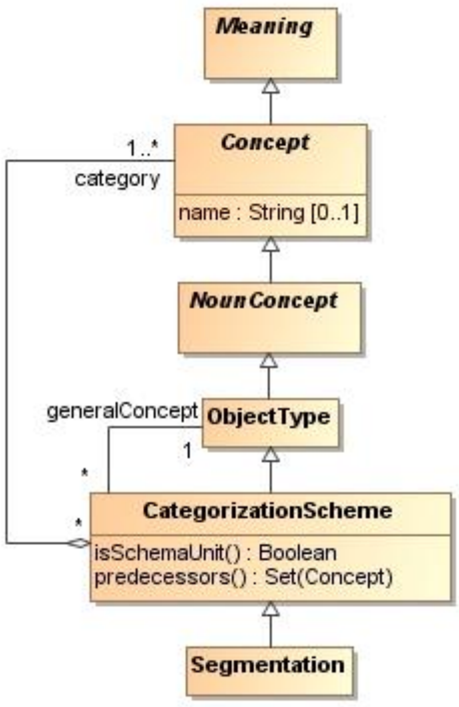


Figure 5.14 Categorization scheme and segmentation schema unit

Each categorization scheme or segmentation is represented by an instance of *CategorizationScheme* or *Segmentation*, respectively. The schema elements of a categorization scheme named *cs* are as follows: (1) the instance υ of *CategorizationScheme* or *Segmentation*; (2) the instance of attribute *name* of υ with value *cs*; (3) the instance of the relationship to *ObjectType* (the general concept); and (4) the instances of the relationships to *Concept* (the categories).

Figure 5.14 shows the abstract syntax of the categorization scheme and segmentation schema units. The *isSchemaUnit()* query operation is specified in *CategorizationScheme* as follows:

```
context CategorizationScheme::isSchemaUnit():Boolean
body:  true
```

The *predecessors()* operation of *CategorizationScheme* is specified as follows:

```
context CategorizationScheme::predecessors():Set(Concept)
  body: Concept.allInstances() -> select(c: Concept|
        self.category -> includes(c) or
        self.generalConcept ->
          includes(c.oclAsType(ObjectType)))
```

This means that the predecessors of a categorization scheme (and a segmentation) are the general concepts and the categories that define the categorization scheme.

Characterization object

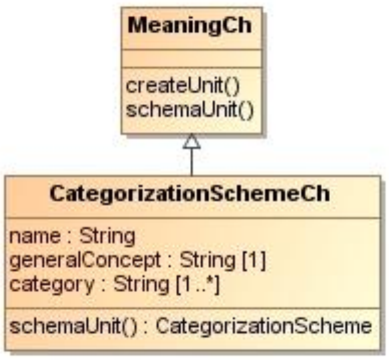


Figure 5.15 Categorization scheme and segmentation schema unit characterization object *CategorizationSchemeCh*

For the categorization scheme and segmentation schema units characterization object *CategorizationSchemeCh* (see Figure 5.15), the *schemaUnit() operation* is formally defined as follows:

```
context CategorizationSchemeCh::schemaUnit():
        CategorizationScheme
  body: CategorizationScheme.allInstances() ->
        any(ct:CategorizationScheme| ct.name = self.name and
        self.isSegmentation implies
          ct.oclIsTypeOf(Segmentation) and
        ct.generalConcept -> any(name) = self.generalConcept
        and ct.category -> collect(name) = self.category)
```

This means that the *schemaUnit() operation* of *CategorizationSchemeCh* is a query that gives the instance of *CategorizationScheme* or *Segmentation* whose attribute *name* has the same value as the one given in the attribute *name* of *CategorizationSchemeCh* or *Segmentation*, respectively; it is associated with the general concepts (*ObjectType*(s)) and *categories* (*Concept*(s)) whose *name* attributes have the same values as the ones given in the attributes *generalConcept* and *category* of *CategorizationSchemeCh*.

5.2.7 Reference scheme schema unit

Generic description

A reference scheme schema unit is a particular form of business rule (constraint). For a given concept, a reference scheme identifies one or more properties (fact type roles or characteristics) of the corresponding objects (instances of the given concept) used to distinguish one instance from another. A reference scheme applies to all instances of the concept. More than one reference scheme can be used simultaneously for instances of the same concept. Some concepts have no agreed-upon reference scheme.

The DBLP example, as an instance of the SBVR metaschema, includes seven reference schemes to indicate the following constraints: *name* identifies *person*, *isbn* identifies *book*, *id* identifies *book series*, *issn* identifies *journal*, *title* identifies *journal*, *name* identifies *conference series* and *title* identifies *conference edition*.

Abstract syntax

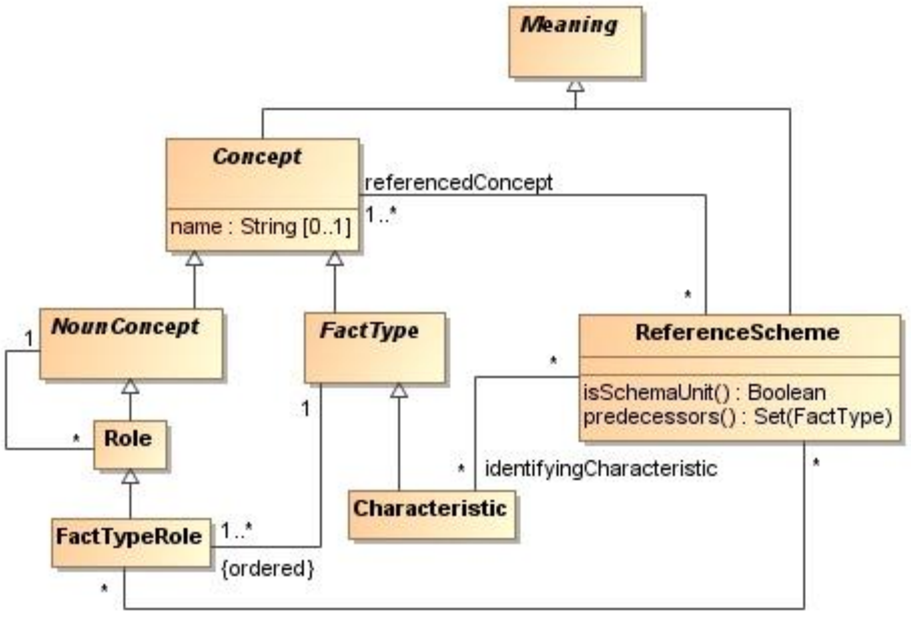


Figure 5.16 Reference scheme schema unit

Each reference scheme is represented by an instance of *ReferenceScheme.* The schema elements of a reference scheme that may be named *re* are as follows: (1) the instance τ of *ReferenceScheme*; (2) if it has a name, the instance of attribute *name* with value *re*; (3) the instances of its relationships with the *Concept(s)* that can be identified using this reference scheme; (4) the instances of its relationships to the *FactTypeRole(s)* that the reference scheme simply uses; and (5) the instances of its relationship to the *Characteristic(s)* that are identifying characteristic(s).

Figure 5.16 shows the abstract syntax of the reference scheme schema unit. The *isSchemaUnit()* query operation is redefined as follows:

```
context ReferenceScheme::isSchemaUnit():Boolean
body:  true
```

The *predecessors() operation* of *ReferenceScheme* is specified as follows:

```
context ReferenceScheme::predecessors():Set(FactType)
  body: self.factTypeRole.factType->union(
         self.identifyingCharacteristic.factType
```

This means that the predecessors of a reference scheme are the fact types in which the fact type roles and the identifying characteristics are the properties used by the reference scheme.

Characterization object

Figure 5.17 shows the reference scheme schema unit characterization object *ReferenceSchemeCh*.

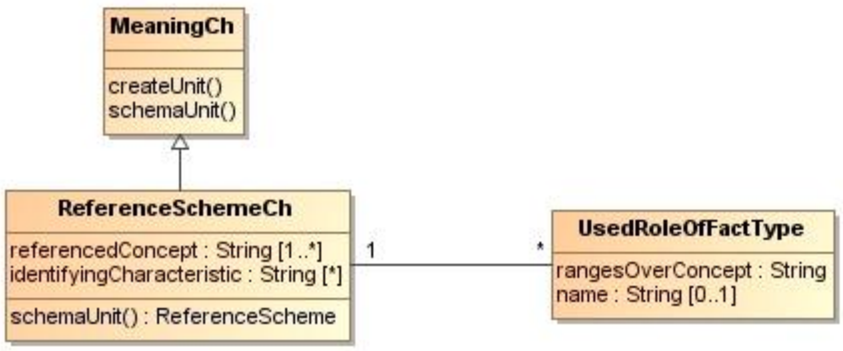

MeaningCh
createUnit()
schemaUnit()
ReferenceSchemeCh
referencedConcept : String [1..*]
identifyingCharacteristic : String [*]
schemaUnit() : ReferenceScheme
1
*
UsedRoleOfFactType
rangesOverConcept : String
name : String [0..1]

Figure 5.17 Reference scheme schema unit characterization object *ReferenceSchemeCh*

For the reference scheme schema unit characterization object *ReferenceSchemeCh*, the *schemaUnit() operation* is formally defined as follows:

```
 context ReferenceSchemeCh::schemaUnit():ReferenceScheme
   body: ReferenceScheme.allInstances() ->
          any(r:ReferenceScheme|
          r.referencedConcept -> collect(name) =
          self.referencedConcept and
          r.identifyingCharacteristic -> collect(name) =
          self.identifyingCharacteristic and
          self.usedRoleOfFactType -> notEmpty() implies
          (r.factTypeRole -> collect(nounConcept.name) =
          self.usedRoleOfFactType->
            collect(s|s.rangesOverConcept) and
          self.usedRoleOfFactType -> forAll(ro|
            ro.name -> notEmpty() implies
            r.factTypeRole.name -> includes(ro.name))))
```

This means that the *schemaUnit() operation* of *ReferenceSchemeCh* is a query that gives the instance of *ReferenceScheme* which identifies the *Concept*(s) whose *name* attribute has the value of the *identifiedConcepts* attribute of *ReferenceSchemeCh*; which uses the *Characteristic*(s) whose *name* attribute has the value of the *usedCharacteristic* attribute of *ReferenceSchemeCh*; which uses the *FacTypeRoles* whose *name* and the *name* of the concept that it ranges over have the same values as the ones given in the *name* and *rangedOverConcept* of the *UsedFactType*(s).

5.2.8 Structural rule schema unit

Generic description

As stated in Section 5.1, structural rules are statements of necessity that state the conditions that must be satisfied by the concepts involved. Structural rules are structured by necessity logical formulations. In SBVR, a necessity logical formulation is a logical formulation that formulates the meaning that another formulation (embedded formulation) is always true. The various kinds of modal formulation are not covered in UML structural schemas. In a UML structural schema, all constraints describe conditions that must be satisfied in the domain. Therefore, the distinction between the various types of modal formulations has been omitted. All logical formulations have been considered necessity formulations.

The particular kinds of logical formulation considered are the following:

- Atomic formulation (Figure 5.24 shows its abstract syntax). This is a logical formulation that is based on a fact type and has a role binding for each role of the fact type. Each role binding of the atomic formulation is a connection between one of the fact type roles of the fact type and a bindable target (a variable, expression or individual concept). The atomic formulation formulates the following meaning: there is an event, activity, situation or circumstance that occurs in the actual world that puts each referent of each role binding in its respective fact type role. Section 5.1 showed an example of atomic formulation.
- Instantiation formulation (Figure 5.25 shows its abstract syntax). This is a logical formulation that considers a concept and binds to a bindable target, and that formulates the following meaning: the thing to which the bindable target refers is an instance of the concept.
- Logical operation (Figure 5.26 shows its abstract syntax). This is a logical formulation that operates on logical operands, which in turn are also logical formulations. Logical operations are further classified into logical negation and binary logical operation.
    - A logical negation has exactly one operand and formulates that the meaning of the logical operand is false.
    - A binary logical operation has exactly two operands and is further classified into:
        - Conjunction. This formulates that the meanings of both logical operands are true.
        - Disjunction. This formulates that the meaning of at least one of the operands is true.
        - Equivalence. This formulates that the meanings of its logical operands are either all true or all false.
        - Exclusive disjunction. This formulates that the meaning of one logical operand is true and the meaning of the other logical operand is false.
        - Implication. This operates on an antecedent (first logical operand) and a consequent (second logical operand) and formulates that the meaning of the consequent is true if the meaning of the antecedent is true.

    - Nand formulation. This formulates that the meaning of at least one of its logical operands is false.
    - Nor formulation. This formulates that the meaning of each of its logical operands is false.
    - Whether-or-not formulation. This operates on a consequent (first logical operand) and an inconsequent (second logical operand) and formulates that the meaning of the consequent is true regardless of the meaning of the inconsequent.

- Quantification (Figure 5.27 shows its abstract syntax). This is a logical formulation that introduces a variable (a reference to an element of a set, whose referent may vary). A variable may range over a concept, which means that the set of referents of the variable (the possible values that it may take) is limited to the instances of said concept. Additionally, a logical formulation may restrict a variable, which means that the set of referents of the variable is further limited to those things for which the meaning formulated by that logical formulation is true when the thing is substituted for each occurrence of the variable in the formulation. A quantification may scope over another logical formulation. Quantifications are further classified into different kinds of quantification, and the following are the meanings of each particular kind:
    - Universal quantification. This is a quantification that scopes over another logical formulation and has the following meaning: for each referent of the variable introduced by the quantification, the meaning formulated by the logical formulation for the referent is true.
    - At-least-n quantification. This is a quantification that has a minimum cardinality and has the following meaning: the number of distinct referents of the variable introduced by the quantification that exist and that satisfy a scope formulation, if there is one, is no less than the minimum cardinality. For example "each tennis match has a least two sets." Existential quantification, a particular kind of at-least-n quantification, has a minimum cardinality of 1.
    - At-most-n quantification. This is a quantification that has a maximum cardinality and has the following meaning: the number of distinct referents of the variable introduced by the quantification that exist and that satisfy a scope formulation, if there is one, is no greater than the maximum cardinality. At-most-one quantification, a particular kind of at-most-n quantification, has a maximum cardinality of 1.
    - Exactly-n quantification. This is a quantification that has a cardinality and has the following meaning: the number of distinct referents of the variable introduced by the quantification that exist and that satisfy a scope formulation, if there is one, equals the cardinality. Exactly-one

quantification, a particular kind of exactly-n quantification, has the cardinality of 1.

    - Numeric range quantification. This is a quantification that has a minimum cardinality and a maximum cardinality greater than the minimum cardinality and has the following meaning: the number of referents of the variable introduced by the quantification that exist and that satisfy a scope formulation, if there is one, is not less that the minimum cardinality and is not greater than the maximum cardinality.

- Objectification (Figure 5.28 shows its abstract syntax). This is a logical formulation that involves a bindable target (i.e., a variable, expression or individual concept) and a considered logical formulation, and that formulates the following meaning: the thing to which the bindable target refers is an event, activity, situation or circumstance that occurs in the actual world that corresponds to the meaning of the considered logical formulation. This thesis only uses the objectification formulation to nominalize fact types.

The DBLP example, as an instance of the SBVR metaschema, includes 73 structural rule schema units.

Of these 73 structural rules, 45 are structured by a closed universal quantification that scopes over an exactly-one quantification. The exactly-one quantification scopes over an atomic formulation based on one of the is-property-of, associative or partitive fact types described above (Section 5.2.5). An example is shown in Figure 5.18.

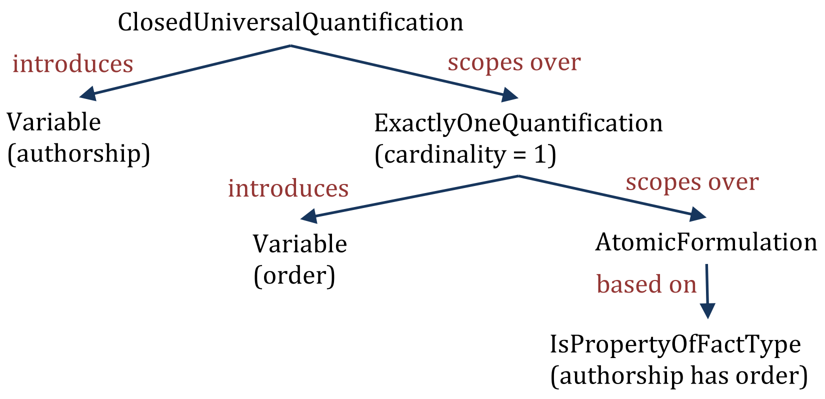


Figure 5.18 Simplified version of the structure of the "each authorship has exactly one order" structural rule

In UML, the meanings of these rules are represented by cardinality constraints of attributes (i.e., no graphical symbol is shown in the diagram) or member ends of association (i.e., a multiplicity symbol, 1, is shown in the diagram). They can be represented in ordinary language as follows:

"each authorship has exactly one order,"
"each book has exactly one isbn,"
"each book has exactly one num pages,"
"each book has exactly one publication year,"

"each book has exactly one publisher,"
"each book chapter has exactly one end page,"
"each book chapter has exactly one ini page,"
"each book section has exactly one order,"
"each book section has exactly one title,"
"each book series has exactly one id,"
"each book series has exactly one publisher,"
"each book series issue has exactly one book series,"
"each book series issue has exactly one number,"
"each conference edition has exactly one city,"
"each conference edition has exactly one country,"
"each conference edition has exactly one title,"
"each conference edition has exactly one year,"
"each book section has at least one book chapter,"
"each authored publication has at least one author,"
"each book series issue has at least one book chapter,"
"each edited book has at least one book chapter,"
"each conference series has exactly one acronym,"
"each conference series has exactly one name,"
"each conference edition has exactly one conference series,"
"each journal issue has exactly one num pages,"
"each journal issue has exactly one number,"
"each journal issue has exactly one year,"
"each journal section has exactly one journal issue,"
"each journal paper has exactly one end page,"
"each journal paper has exactly one ini page,"
"each editorship has exactly one order,"
"each journal has exactly one issn,"
"each journal has exactly one title,"
"each journal volume has exactly one journal,"
"each journal section has exactly one order,"
"each journal section has exactly one title,"
"each journal volume has exactly one volume,"
"each journal issue has exactly one journal volume,"
"each person has exactly one gender,"
"each person has exactly one name,"
"each publication has exactly one edition,"
"each publication has exactly one title,"
"each publication has exactly one year,"
"each person has exactly one num publications,"
"each journal paper is part of exactly one journal issue,"

Of the 73 structural rules, 14 are structured by a closed universal quantification that scopes over an at-most-one quantification. The at-most-one quantification scopes over an atomic formulation based on one of the is-property-of, associative or partitive fact types described above (Section 5.2.5). The difference between these rules and those described above is that the closed universal quantification scopes over an at-most-one quantification instead of an exactly-one quantification. The rest of the structure is the same. In UML, the meanings of these rules are represented by a multiplicity symbol at one member end of the association (1..*). They can be represented in ordinary language as follows:

"each book has at most one home page,"
"each book chapter is part of at most one book section,"
"each book chapter is part of at most one book series issue,"
"each book chapter is part of at most one edited book,"
"each book section is part of at most one edited book,"

"each conference edition has at most one home page,"
"each book series issue has at most one conference edition,"
"each conference edition is published in at most one book series issue,"
"each conference edition is published in at most one edited book,"
"each edited book has at most one conference edition,"
"each conference edition is published in at most one journal issue,"
"each journal issue has at most one conference edition,"
"each journal issue has at most one month,"
"each person has at most one home page."

Of the 73 structural rules, six are structured by a closed universal quantification that scopes over an existential quantification. The existential quantification scopes over an atomic formulation based on one of the associative fact types described above (Section 5.2.5). Now, the closed universal quantification scopes over an existential quantification. In UML, the meanings of these rules are represented by a multiplicity symbol of an attribute ([0..1]) or a multiplicity symbol of a member end of an association (0..1). They can be represented in ordinary language as follows:

"each edited book has at least one editor,"
"each journal issue has at least one journal paper,"
"each journal paper is part of at most one journal section,"
"each journal section has at least one journal paper,"
"each person publishes at least one publication,"
"each publication has at least one person."

The 65 structural rules mentioned above correspond to cardinality constraints. They all have a similar structure; the only difference among them is the kind of quantification that the closed universal quantification scopes over.

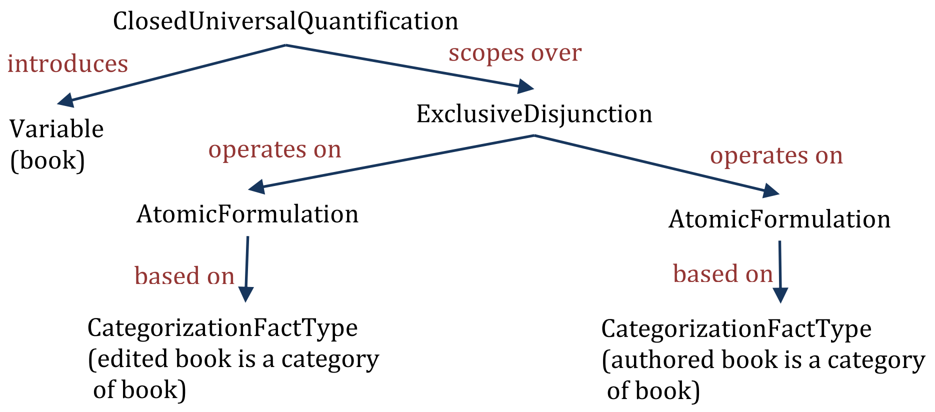


Figure 5.19 Simplified version of the structure of the "each book is an edited book or is an authored book but not both" structural rule

Of the 73 structural rules, two are structured by a closed universal quantification that scopes over an exclusive-disjunction binary logical operation. Each operand of the exclusive-disjunction is an atomic formulation that scopes over a categorization fact type. Figure 5.19 shows the simplified version of the "each book is an edited book or is an authored book but not both" structural rule.

In UML, the meaning of both rules is represented by the {disjoint,complete} symbol of a generalization set with two generalizations. They can be represented in ordinary language as follows:

"each book is a edited book or is a authored book but not both,"
"each publication is a edited book or is a authored publication but not both."

Of the 73 structural rules, one is structured by a closed universal quantification that scopes over a disjunction binary logical operation. Its first operand is an atomic formulation based on a categorization fact type; its second operand is a disjunction binary logical operations whose operands are atomic formulations, each based on a different categorization fact type. In UML, the meaning of this constraint is represented by the {complete} symbol of a generalization set with three generalizations. It can be represented in ordinary language as follows:

"each book is an authored book or a book chapter or a journal paper."

Figure 5.20 shows the simplified version of the structure of such a rule.

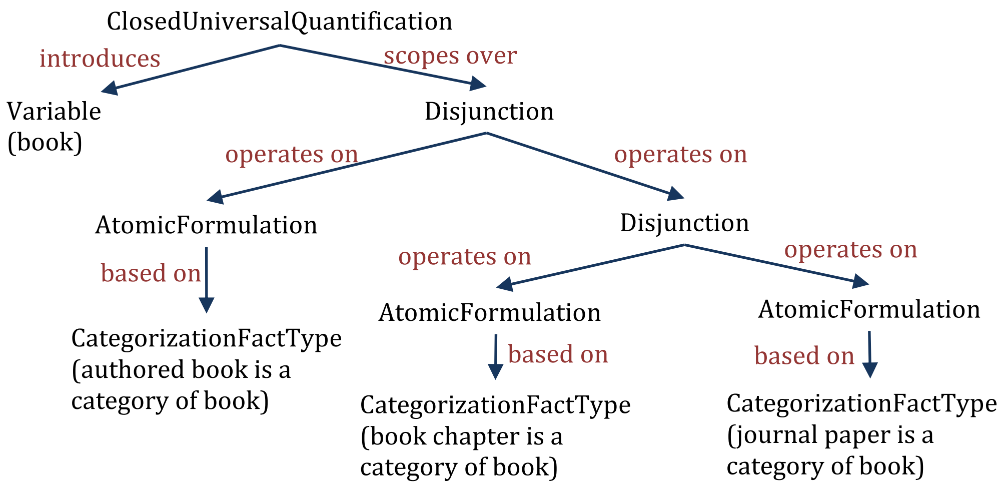

ClosedUniversalQuantification
introduces
scopes over
Variable (book)
Disjunction
operates on
operates on
AtomicFormulation
based on
CategorizationFactType (authored book is a category of book)
Disjunction
operates on
operates on
AtomicFormulation
based on
CategorizationFactType (book chapter is a category of book)
AtomicFormulation
based on
CategorizationFactType (journal paper is a category of book)

Figure 5.20 Simplified version of the structure of the "each book is an authored book or a book chapter or a journal paper" structural rule

Of the 73 structural rules, two are structured by a closed universal quantification that introduces a variable restricted by an atomic formulation. This atomic formulation is based on a categorization fact type. The closed universal quantification scopes over a nor logical operation. Both operands of the nor logical operation are atomic formulations based on categorization fact types. In UML, the meaning of these two rules, together, is represented by the {disjoint} symbol of a generalization set with three generalizations. It can be represented in natural language as follows:

"each authored publication that is an authored book is neither a book chapter nor a journal paper,"
"each authored publication that is a book chapter is neither an authored book nor a journal paper."

Figure 5.21 shows the simplified version of the structure of the first rule.

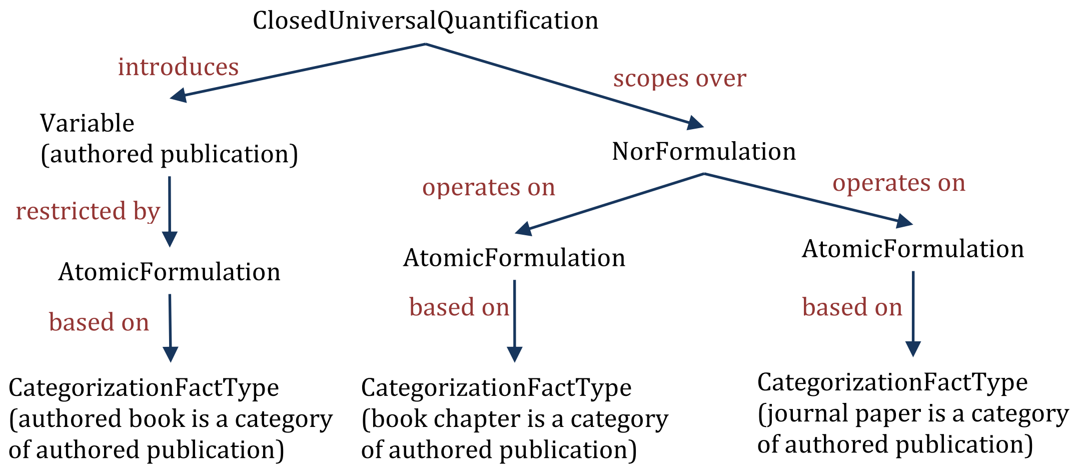


Figure 5.21 Simplified version of the structure of "each authored publication that is an authored book neither is a book chapter nor a journal paper" structural rule

Of the 73 structural rules, three are structured by a closed universal quantification that introduces a variable restricted by an atomic formulation. This atomic formulation is based on an associative fact type. The closed universal quantification scopes over a nor formulation logical operation. Both operands of the exclusive-disjunction binary logical operation are atomic formulations based on associative fact types. In UML, the meaning of these three rules, together, is represented by the {XOR} symbol between three associations. This can be represented in natural language as follows:

"each conference edition that is published in a book series issue is published neither in an edited book nor in a journal issue,"
"each conference edition that is published in an edited book is published neither in a book series issue nor in a journal issue,"
"each conference edition that is published in a journal issue is published neither in an edited book nor in a book series issue."

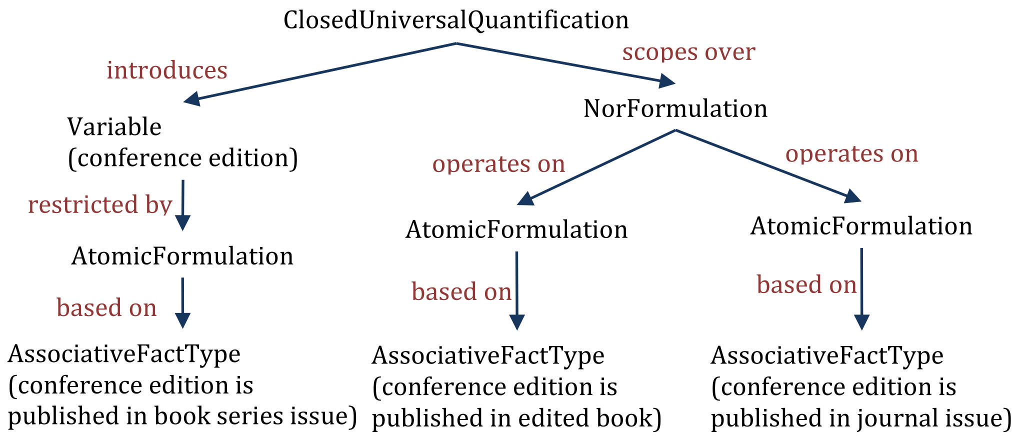


Figure 5.22 Simplified version of the structure of the "each conference edition that is published in a book series issue neither is published in an edited book nor in a journal issue" structural rule

Figure 5.22 shows the simplified version of the structure of the first rule presented above. The figure represents the following structure:

The structural rule embeds a closed universal quantification.
. The closed universal quantification introduces a first variable.
. . The first variable ranges over the concept "conference edition."
. . . The first variable is restricted by an atomic formulation.
. . . . The atomic formulation is based on the fact type "conference edition is published in book series issue."
. . . . The atomic formulation introduces a free variable
. . . . . The free variable ranges over the concept "book series issue."
. . . . . The atomic formulation has a role binding.
. . . . . . The role binding is of the role "conference edition" of the fact type.
. . . . . . The role binding binds to the first variable.
. . . . . The atomic formulation has a second role binding.
. . . . . . The second role binding is of the role "book series issue" of the fact type.
. . . . . . The second role binding binds to the free variable.
. . The universal quantification scopes over a nor formulation.
. . . The nor formulation has an atomic formulation as its first operand.
. . . The atomic formulation introduces a second free variable.
. . . . The second free variable ranges over the concept "edited book."
. . . The atomic formulation is based on the fact type "conference edition is published in edited book."
. . . . The atomic formulation has a role binding.
. . . . . The role binding is of the role "conference edition" of the fact type.
. . . . . The role binding binds to the first variable.
. . . . The atomic formulation has a second role binding.
. . . . . The second role binding is of the role "edited book" of the fact type.
. . . . . The second role binding binds to the second free variable.
. . . The nor formulation has an atomic formulation as its second operand.
. . . The atomic formulation introduces a third free variable.
. . . . The third free variable ranges over the concept "journal issue."
. . . The atomic formulation is based on the fact type "conference edition is published in journal issue."
. . . . The atomic formulation has a role binding.
. . . . . The role binding is of the role "conference edition" of the fact type.
. . . . . The role binding binds to the first variable.
. . . . The atomic formulation has a second role binding.
. . . . . The second role binding is of the role "journal issue" of the fact type.
. . . . . The second role binding binds to the third free variable.

Abstract syntax

Figure 5.23 to 5.28 show the abstract syntax of the complex structural rule schema unit.

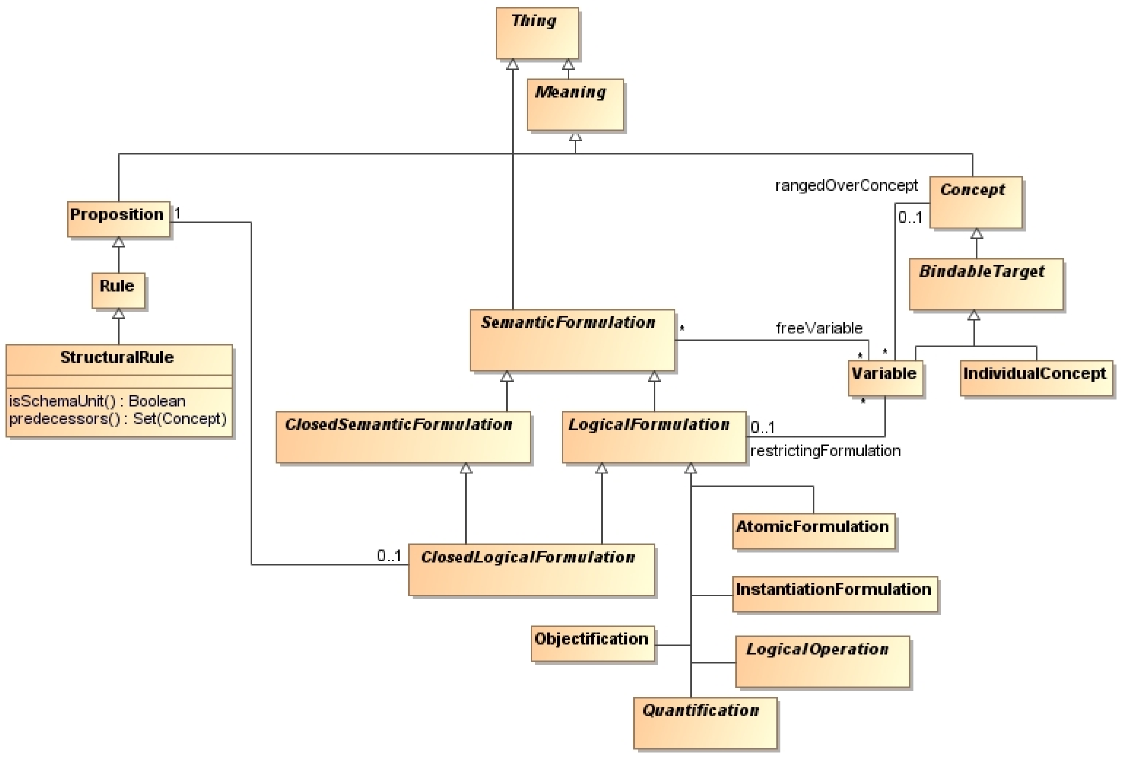


Figure 5.23 *StructuralRule* schema unit

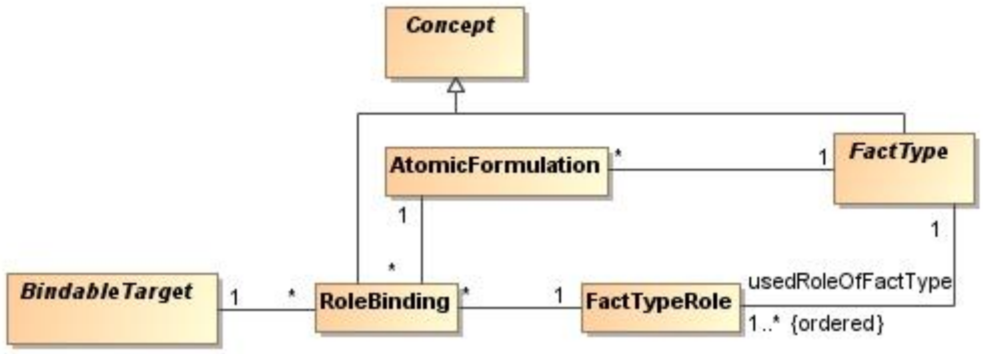


Figure 5.24 Atomic formulation

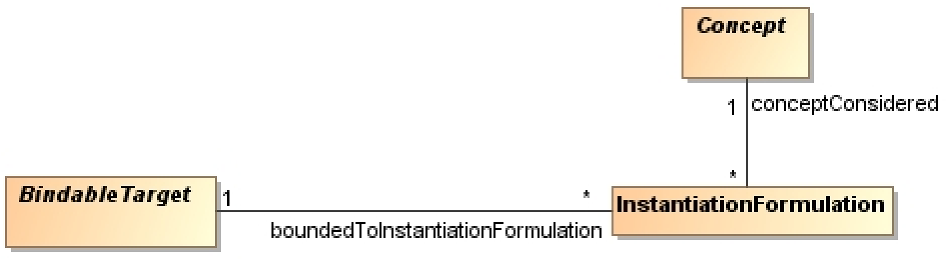


Figure 5.25 Instantiation formulation

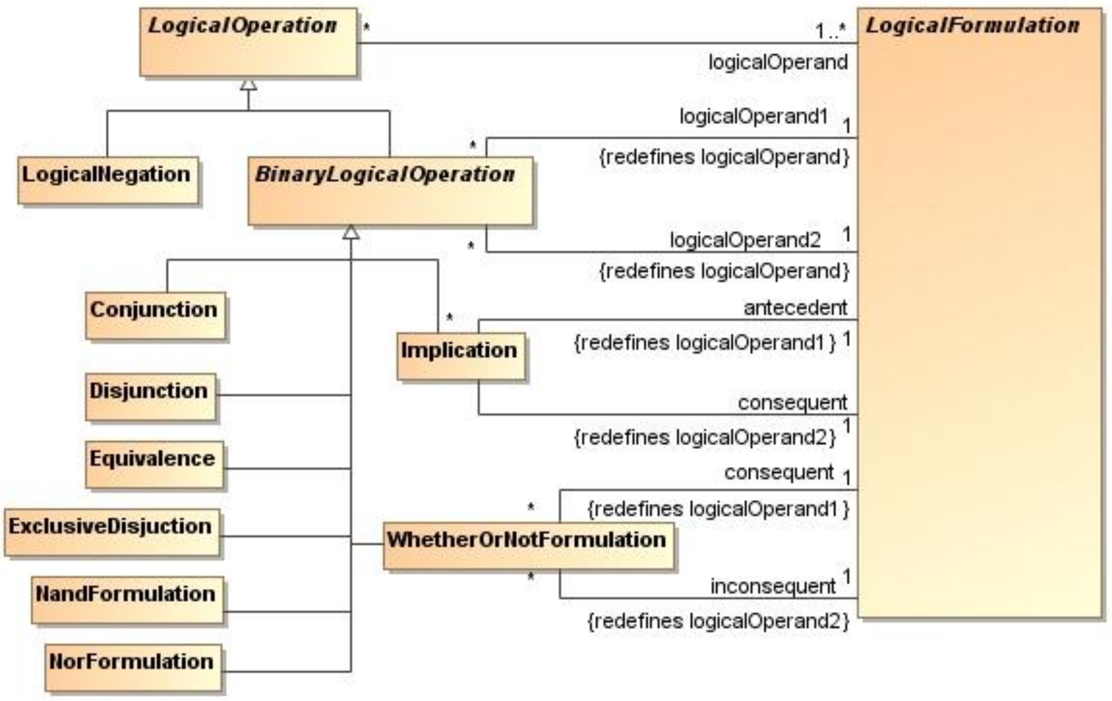


Figure 5.26 Logical operation

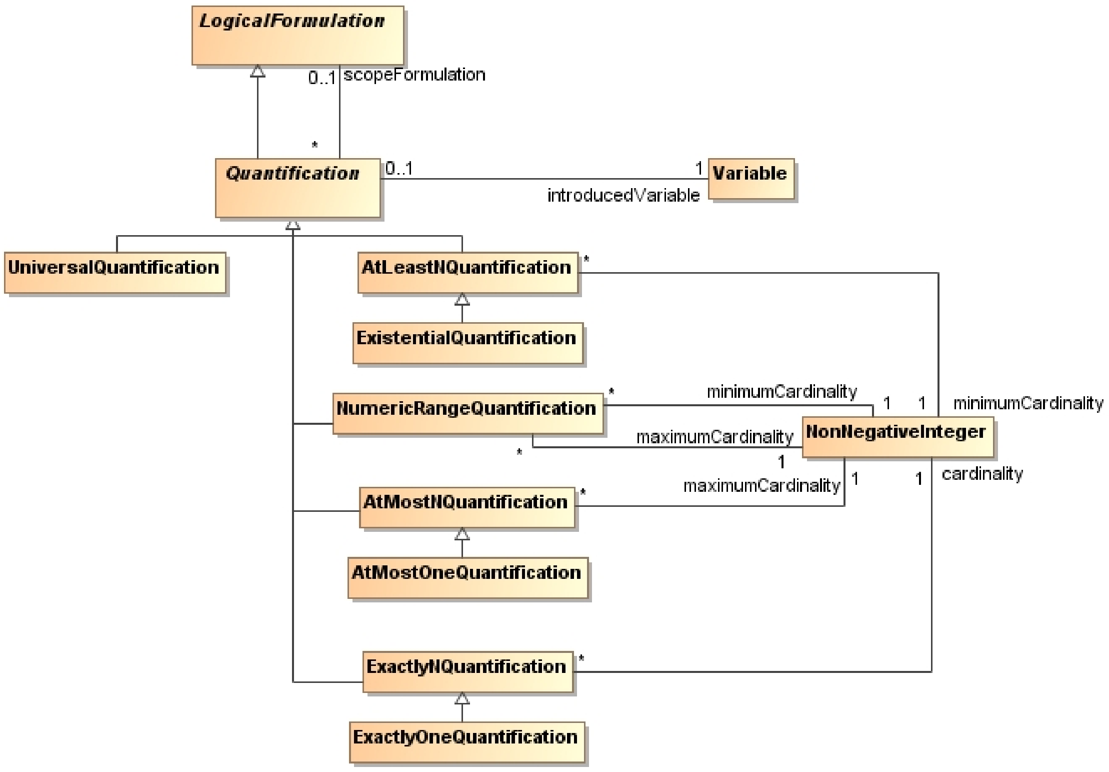


Figure 5.27 Quantification

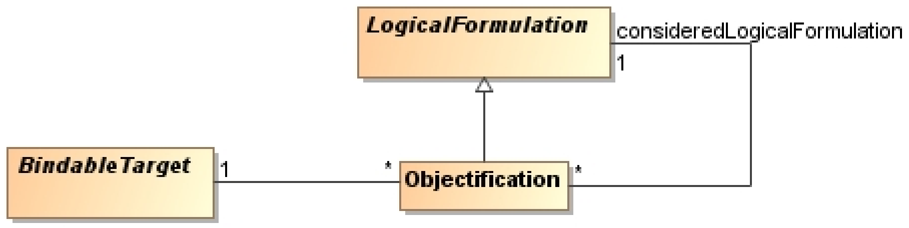


Figure 5.28 Objectification

Each structural rule schema unit is represented by an instance of *StructuralRule*. The schema elements of a structural rule that may have a name *st* are as follows: (1) the instance of ε of *StructuralRule*; (2) if it has a name, the instance of the attribute *name* with value *st*; (3) the instance of the attribute *isTrue* with value *True*; (4) the instance of *ClosedUniversalQuantification* that structures ε; (5) the instance of relationship between the *ClosedUniversalQuantification* and ε; (6) the instance of *Variable* introduced by the closed universal quantification; (7) the instance of relationship between the *Variable* and the *ClosedUniversalQuantification*; (8) the instance of relationship between the *Variable* and the instance of *Concept* that the variable ranges over; (9) the instance of a subtype of *LogicalFormulation* that the quantification scopes over; (10) the instance of relationship between the *LogicalFormulation* and the *ClosedUniversalQuantification*; (11) the instances of *Variable* that are defined as free variables of the *LogicalFormulation* and the instance of relationship between them; (12) the instance of relationship between an instance of *Variable* and the *LogicalFormulation* (restrictingFormulation); and (13) depending on the type of formulation, the following instances:

- Instantiation formulation: (i) the instance of *LogicalFormulation* that is an instance of *InstantiationFormulation*; (ii) the instance of relationship to a *BindableTarget*; and (iii) the instance of relationship to a *Concept*.
- Atomic formulation: (i) the instance of *LogicalFormulation* that is an instance of *AtomicFormulation*; (ii) the instance of relationship between the *AtomicFormulation* and the *FactType* that is based on the *AtomicFormulation*; and (iii) the instances of *RoleBinding* that occur in the *AtomicFormulation*. For each *RoleBinding*: (i) the instance the relationship between the *BindableTarget* and the *RoleBinding*; and (ii) the instance of relationship between the *FactTypeRole* and the *RoleBinding*.
- Logical negation: (i) the instance of *LogicalFormulation* that is an instance of *LogicalNegation*; (ii) the instance of *LogicalFormulation* that is the operand of the logical negation; and (iii) the instance of the relationship between the *LogicalNegation* and the *LogicalFormulation*. The set of instances defined in point 11 to describe the *LogicalFormulation* that is the operand.
- Binary logical formulation: (i) the instance of *LogicalFormulation* that is an instance of a subtype of *BinaryLogicalOperation* (*Conjunction*, *Disjunction*, *Equivalence*, *ExclusiveDisjunction*, *Implication*, *NandFormulation*, *NorFormulation* or *WhetherOrNotFormulation*); (ii) the instance of *LogicalFormulation* that is the first

operand of the *BinaryLogicalOperation* and the instance of the relationship between the two; and (iii) the instance of *LogicalFormulation* that is the second operand of the *BinaryLogicalOperation* and the instance of the relationship between the two. The set of instances defined in point 11 to describe the *LogicalFormulation* of both operands.

- Quantification: (i) the instance of *LogicalFormulation* that is an instance of a subtype of *Quantification*; (ii) the instance of *Variable* that introduces the instance of relationship between the *Variable* and the subtype of *Quantification*; (iii) the instance of *LogicalFormulation* that the subtype of *Quantification* scopes over and the relationship between the two; (iv) the set of instances defined in point 11 to describe the *LogicalFormulation* that the quantification scopes over; and (v) if the subtype of *Quantification* is:
    - At-least-n quantification or existential quantification: the instance of *NonNegativeInteger* and the instance of relationship (minimum cardinality) between the quantification and the *NonNegativeInteger*.
    - Numeric range quantification: the two instances of *NonNegativeInteger* and the two instances of relationship (minimum cardinality and maximum cardinality) between the quantification and the *NonNegativeInteger*.
    - At-most-n quantification or at-most-one quantification: the instance of *NonNegativeInteger* and the instance of relationship (maximum cardinality) between the quantification and the *NonNegativeInteger*.
    - Exactly-n quantification and exactly-one quantification: the instance of *NonNegativeInteger* and the instance of relationship (cardinality) between the quantification and the *NonNegativeInteger*.
- Objectification: (i) the instance of *Objectification*; (ii) the instance of the relationship between a *BindableTarget* and the *Objectification*; (iii) the instance of *LogicalFormulation* that the *Objectification* considers; and (iv) the instance of relationship between the *Objectification* and the *LogicalFormulation* considered. The set of instances defined in point 11 to describe the *LogicalFormulation* that is considered by the *Objectification*.

The *isSchemaUnit()* query operation of *StructuralRule* is specified as follows:

```
context StructuralRule::isSchemaUnit():Boolean
body:  true
```

The *predecessors() operation* of *StructuralRule* is specified as follows:

```
context StructuralRule::predecessors():Set(Concept)
  body:  self.closedLogicalFormulation.conceptsUsed()
```

This means that the predecessors of a structural rule are the concepts used by the logical formulation that structures the structural rule. *ConceptsUsed* is a query operation, defined in *LogicalFormulation* and redefined in its subtypes, that gives the concepts that are used in the logical formulations. Formally, this is defined as follows:

```
context LogicalFormulation::conceptsUsed():Set(Concept)
  body:  Set{}

context AtomicFormulation::conceptsUsed():Set(Concept)
  body:  Concept.allInstances() -> select(c:Concept|
         self.freeVariable -> collect(rangedOverConcept)->
            includes(c) or
         self.factType = c.oclAsType(FactType) or
         self.roleBinding -> collect(bindableTarget) ->
            includes(c.oclAsType(BindableTarget)) or
         self.roleBinding -> collect(factTypeRole)->
            includes(c.oclAsType(FactTypeRole)))

context InstantiationFormulation::conceptsUsed():Set(Concept)
  body:  Concept.allInstances() -> select(c:Concept|
         self.freeVariable -> collect(rangedOverConcept)->
            includes(c) or
         self.conceptConsidered = c or
         self.bindableTarget->
            includes(c.oclAsType(BindableTarget)))

context LogicalNegation::conceptsUsed():Set(Concept)
  body:  self.freeVariable -> collect(rangedOverConcept) ->
         asSet() -> union(self.logicalOperand.conceptsUsed())

context BinaryLogicalOperation::conceptsUsed():Set(Concept)
  body:  self.freeVariable -> collect(rangedOverConcept) ->
         asSet() -> union(self.logicalOperand1.conceptsUsed()->
            union(self.logicalOperand2.conceptsUsed()))

context Quantification::conceptsUsed():Set(Concept)
  body:  self.freeVariable -> collect(rangedOverConcept) ->
         asSet() -> union(self.scopeFormulation.conceptsUsed())

context Objectification::conceptsUsed():Set(Concept)
  body:  Concept.allInstances() -> select(c:Concept|
         self.freeVariable -> collect(rangedOverConcept)->
            includes(c) or
         self.bindableTarget.oclAsType(Variable).rangedOverConcept = c
         or self.bindableTarget.oclAsType(IndividualConcept) =
         c.oclAsType(IndividualConcept)) -> union(
         self.consideredLogicalFormulation.conceptsUsed())
```

Characterization object

Figure 5.29 shows the structural rule schema unit characterization object, *StructuralRuleCh*. The *schemaUnit() operation* is formally defined as follows:

```
context StructuralRuleCh::schemaUnit():StructuralRule
  body:  StructuralRule.allInstances() ->
         any(st:StructuralRule|
            st.closedLogicalFormulation -> notEmpty() and
            self.formulation.existsFormulation(
               st.closedLogicalFormulation))
```

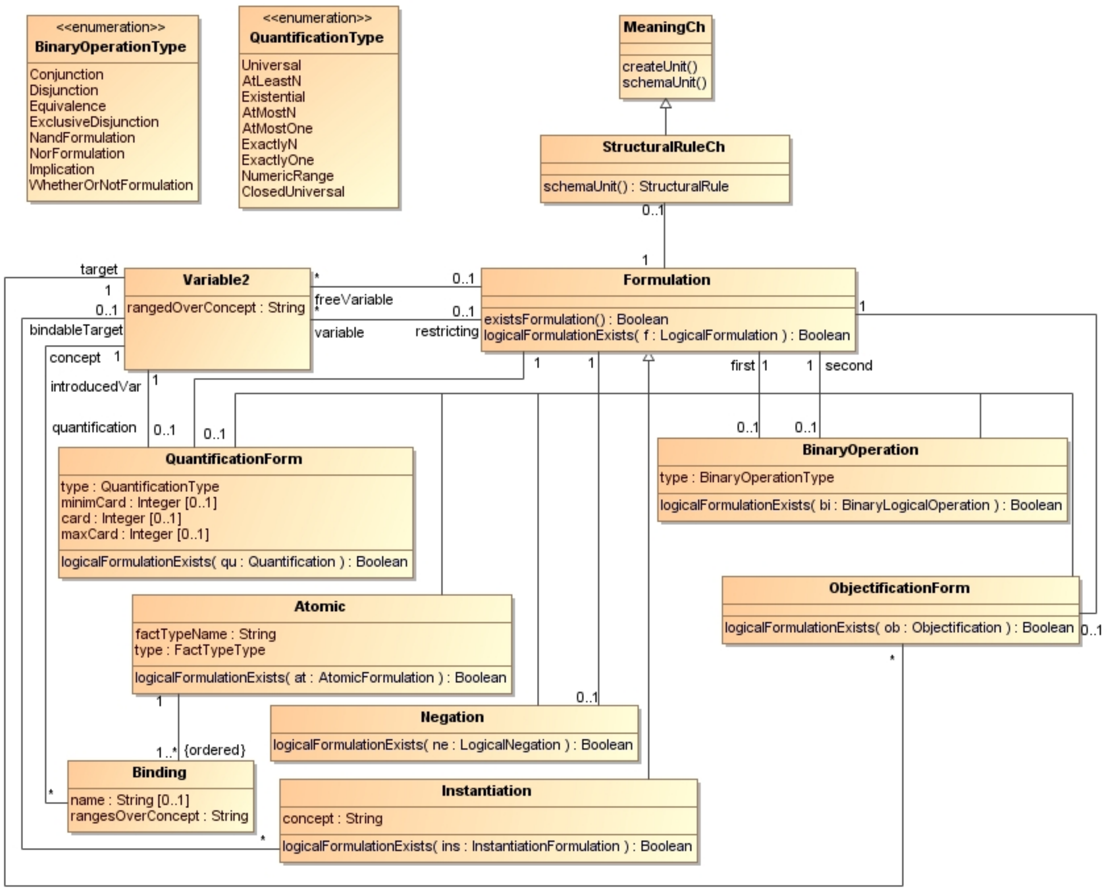


Figure 5.29 Structural rule schema unit characterization object *StructuralRuleCh*

This means that the *schemaUnit()* operation of *StructuralRuleCh* is a query that gives the instance of *StructuralRule* associated with a *ClosedLogicalFormulation*.

The *existsFormulation()* operation is formally defined as follows:

```
context Formulation::
        existsFormulation(sf:LogicalFormulation):Boolean
  body:   self.freeVariable -> notEmpty()
          implies
          (self.freeVariable -> forAll(fv|
            sf.freeVariable -> exists(va|
              va.rangedOverConcept.name =
              fv.rangedOverConcept and
              fv.restricting -> notEmpty()
              implies fv.restricting.existsFormulation(
                    va.restrictingFormulation)))) and
          self.logicalFormulationExists(sf)
```

This means that the *existsFormulation()* operation returns *True* if, for each free variable of *Formulation*, the logical formulation given by the parameter is associated with a free variable whose ranged over concept has the attribute *name* with the same value as the *rangedOverConcept* attribute. Additionally, if the variable is restricted by another *Formulation*, there is an instance of *LogicalFormulation* that is characterized by such *Formulation*.

The operation *logicalFormulationExists* is defined in *Formulation* as follows:

```
context Formulation::
  logicalFormulationExists(f:LogicalFormulation):Boolean
  body:  (abstract)
```

This means that the operation is defined as *abstract* in *Formulation* and redefined in its subtypes. In each subtype, the operation *logicalFormulationExists()* checks whether there is an instance of a subtype of *LogicalFormulation* that corresponds to that defined in each subtype.

```
context Instantiation::logicalFormulationExists
        (ins:InstantiationFormulation):Boolean
  body:  ins.bindableTarget.oclAsType(IndividualConcept).name
         = self.bindableTarget.rangedOverConcept and
         ins.conceptConsidered.oclAsType(Variable).
         rangedOverConcept.oclAsType(NounConcept)name = self.concept
```

The *logicalFormulationExists* operation, in the context of *Instantiation*, returns *True* if the *InstantiationFormulation* is binded to an *IndividualConcept* whose *name* has the same value as the name of the *bindableTarget* associated to *Instantiation*, and the name of the concept considered in the *InstantiationFormulation* has the same value as the *concept* attribute of *Instantiation*.

```
context Atomic::logicalFormulationExists
        (at:AtomicFormulation):Boolean
  body:  at.factType.name = self.factTypeName and
         self.type = FactTypeType::Categorization implies
           at.factType.oclIsTypeOf(CategorizationFactType) and
         self.type = FactTypeType::IsPropertyOf implies
           at.factType.oclIsTypeOf(IsPropertyOfFactType) and
         self.type = FactTypeType::Associative implies
           at.factType.oclIsTypeOf(AssociativeFactType) and
         self.type = FactTypeType::Partitive implies
           at.factType.oclIsTypeOf(PartitiveFactType) and
         self.binding -> collect(ro|
           Tuple{n:ro.name, c:ro.rangesOverConcept}) =
         at.factType.factTypeRole -> collect(fr |
           Tuple{n:fr.name, c:fr.nounConcept.name}) and
         self.binding -> forAll(bi:Binding|
           at.roleBinding -> exists(rb|
             if bi.name -> notEmpty()
             then rb.factTypeRole.name = bi.name
             else rb.factTypeRole.nounConcept.name =
              bi.rangesOverConcept
             endif and
           rb.bindableTarget.oclAsType(Variable).
             rangedOverConcept.name =
           bi.concept.rangedOverConcept))
```

The *logicalFormulationExists* operation, in the context of *Atomic*, returns *True* if the *AtomicFormulation* is based on a *FactType* whose type is the same as the *type* attribute of *Atomic* and it has the same bindings as the ones given in the *Binding*(s) associated to *Atomic*.

```
context Negation::logicalFormulationExists
        (ne:LogicalNegation):Boolean
  body:  self.formulation.existsFormulation(ne.logicalOperand)
```

The *logicalFormulationExists* operation, in the context of *Negation*, returns *True* if the *logicalOperand* of the *LogicalNegation* also exists.

```
context QuantificationForm::logicalFormulationExists
        (qu:Quantification):Boolean
  body: self.introducedVar->notEmpty() implies
        qu.introducedVariable.rangedOverConcept.name =
        self.introducedVar.rangedOverConcept and

        self.introducedVar.restricting->notEmpty() implies
        self.introducedVar.restricting.
        existsFormulation(qu.introducedVariable.
        restrictingFormulation) and

        self.type = QuantificationType::Universal implies
           (qu.oclIsTypeOf(UniversalQuantification) and
        self.formulation.existsFormulation(qu.oclAsType(
          UniversalQuantification).scopeFormulation)) and

        self.type = QuantificationType::AtLeastN implies
        (qu.oclIsTypeOf(AtLeastNQuantification) and
          qu.oclAsType(AtLeastNQuantification).
          minimumCardinality.value = self.minimCard and
          self.formulation.existsFormulation(qu.oclAsType(
            AtLeastNQuantification).scopeFormulation)) and

        self.type = QuantificationType::Existential implies
        (qu.oclIsTypeOf(ExistentialQuantification) and
          qu.oclAsType(ExistentialQuantification).
            minimumCardinality.value = self.minimCard and
          self.formulation.existsFormulation(qu.oclAsType(
          ExistentialQuantification).scopeFormulation)) and

        self.type = QuantificationType::AtMostN implies
        (qu.oclIsTypeOf(AtMostNQuantification) and
          qu.oclAsType(AtMostNQuantification).
            maximumCardinality.value = self.maxCard and
          self.formulation.existsFormulation(qu.oclAsType(
            AtMostNQuantification).scopeFormulation)) and

        self.type = QuantificationType::AtMostOne implies
        (qu.oclIsTypeOf(AtMostOneQuantification) and
          qu.oclAsType(AtMostOneQuantification).
            maximumCardinality.value = self.maxCard and
          self.formulation.existsFormulation(qu.oclAsType(
            AtMostOneQuantification).scopeFormulation)) and

        self.type = QuantificationType::ExactlyN implies
        (qu.oclIsTypeOf(ExactlyNQuantification) and
          qu.oclAsType(ExactlyNQuantification).
            cardinality.value = self.card and
          self.formulation.existsFormulation(qu.oclAsType(
            ExactlyNQuantification).scopeFormulation)) and

        self.type = QuantificationType::ExactlyOne implies
        (qu.oclIsTypeOf(ExactlyOneQuantification) and
          qu.oclAsType(ExactlyOneQuantification).
            cardinality.value = self.card and
```

```
              self.formulation.existsFormulation(qu.oclAsType(
               ExactlyOneQuantification).scopeFormulation)) and

            self.type = QuantificationType::NumericRange implies
            (qu.oclIsTypeOf(NumericRangeQuantification) and
              qu.oclAsType(NumericRangeQuantification).
                minimumCardinality.value = self.minimCard and
              qu.oclAsType(NumericRangeQuantification).
                maximumCardinality.value = self.maxCard and
              self.formulation.existsFormulation(qu.oclAsType(
                NumericRangeQuantification).scopeFormulation))
```

The *logicalFormulationExists()* operation, in the context of *Quantification*, returns *True* if the subtype of *QuantificationFormulation* corresponds to the one indicated in the *type* attribute of *Quantification* and the cardinalities are the ones indicated in the *minimCard*, *card* and *maxCard* attributes.

```
 context BinaryOperation::logicalFormulationExists
        (bi:BinaryLogicalOperation):Boolean
   body:  if self.type = BinaryOperationType::Implication
          then
            bi.oclIsTypeOf(Implication) and
            self.first.existsFormulation(bi.oclAsType(
              Implication).antecedent) and
            self.second.existsFormulation(bi.oclAsType(
              Implication).consequent) and
          else
            if self.type =
              BinaryOperationType::WhetherOrNotFormulation
            then
              bi.oclIsTypeOf(WhetherOrNotFormulation) and
              self.first.existsFormulation(bi.oclAsType(
                WhetherOrNotFormulation).consequent) and
              self.second.existsFormulation(bi.oclAsType(
                WhetherOrNotFormulation).inconsequent) and
            else
              self.first.existsFormulation(bi.logicalOperand1) and
              self.second.existsFormulation(bi.logicalOperand2) and

              self.type = BinaryOperationType::Conjunction
              implies bi.oclIsTypeOf(Conjunction) and

              self.type = BinaryOperationType::Disjunction
              implies bi.oclIsTypeOf(Disjunction) and

              self.type = BinaryOperationType::Equivalence
              implies bi.oclIsTypeOf(Equivalence) and

              self.type = BinaryOperationType::ExclusiveDisjunction
              implies bi.oclIsTypeOf(ExclusiveDisjunction) and

              self.type = BinaryOperationType::NandFormulation
              implies bi.oclIsTypeOf(NandFormulation) and

              self.type = BinaryOperationType::NorFormulation
              implies bi.oclIsTypeOf(NorFormulation)
            endif
          endif
```

The *logicalFormulationExists()* operation, in the context of *Binary*, returns *True* if the subtype of *BinaryFormulation* corresponds to the one indicated in the *type* attribute of *Binary* and their respective logical operands exist.

```
 context ObjectificationForm::logicalFormulationExists(
         ob:Objectification):Boolean
   body: ob.bindableTarget.oclAsType(Variable).
            rangedOverConcept.name =
         self.target.rangedOverConcept and
         self.formulation.existsFormulation(
            ob.consideredLogicalFormulation)
```

The *logicalFormulationExists()* operation, in the context of *ObjectificationForm*, returns *True* if the *Objectification* binds to a *concept* whose *name* attribute has the same value as the *rangedOverConcept* attribute of the *target* of the *ObjectificationForm* and the *consideredFormulation* of the *Objectification* also exists.

5.2.9 Object Type or Value Type schema unit with a definition

Generic description

As stated in Section 5.1, a projection is a semantic formulation that may be used to structure a definition of a meaning. A closed projection thas is, a projection that includes no variable without binding may define a noun concept, a fact type or a question. This thesis only considers closed projections that define noun concepts. A closed projection that defines a noun concept introduces exactly one variable, called a projection variable, which may have a constraining formulation variables. It formulates a set of properties, called incorporated characteristics, that are sufficient to determine the noun concept. The set of properties (incorporated characteristics) that may be included to define a noun concept are as follows: (i) characteristics of the ranged-over concept of the projection variable; (ii) if a logical formulation restricts the projection variable, the meaning of that formulation with respect to such variable; and (iii) the meaning of the constraining formulation with respect to the projection variable, if there is one. More details about incorporated characteristics may be found in the specification document (Object Management Group 2008a).

The DBLP example, as an instance of the SBVR metaschema, includes five object types that include a closed projection to structure the definition of each object type: "*publication*," "*book*," "*authored publication*," "*editorship*" and "*authorship*." Moreover, the value type named *"gender"* also includes a definition structured by a closed projection.

The object type named "publication" is defined as the union of "edited book" and "authored publication." The definition is structured by a closed projection that has a projection variable that ranges over the concept named "publication." The projection constrains a disjunction whose operands are atomic formulations. The first atomic formulation is based on the categorization fact type "edited book is a category of publication" and the second atomic formulation is based on the categorization fact type "authored publication is a category of publication."

The object type named "book" is defined as the union of "edited book" and "authored book." The definition is structured by a closed projection that has a projection variable that ranges over the concept named "book." The projection constrains a disjunction whose operands are atomic formulations. The first atomic formulation is based on the categorization fact type "edited book is a category of book" and the second atomic formulation is based on the categorization fact type "authored book is a category of book." The object type named "authored publication" is defined as the union of "authored book," "book chapter" and "journal paper."

The definition is structured by a closed projection that has a projection variable that ranges over the concept named "authored publication." The projection constrains a disjunction. The first operand of the disjunction is an atomic formulation based on the categorization fact type "authored book is a category of authored publication." The second operand is a second disjunction whose operands are atomic formulations. The first atomic formulation is based on the categorization fact type of "book chapter is a category of authored publication" and the second atomic formulation is based on the categorization fact type "journal paper is a category of authored publication." Figure 5.30 shows a simplified version of the instantiation of the "authored publication" schema unit.

The object type named "editorship" is defined as an objectification (reification) of the fact type "edited book has editor." The definition is structured by a closed projection that has a projection variable which ranges over the concept named "actuality." Actuality is a concept that means an event, activity, situation or circumstance that occurs in the actual world. The closed projection constrains an objectification formulation. The objectification also has two free variables, one ranging over the concept "edited book" and the other ranging over the concept "person." The objectification considers an atomic formulation based on the associative fact type "edited book has editor." The atomic formulation has two role bindings that bind each role of the fact type to the corresponding free variables. The meaning is "editorship is an actuality that an edited book has an editor."

Similarly to "editorship," "authorship" is defined as an objectification of the fact type "authored publication has author."

The value type named "gender" is defined as the union of the individual concepts "male" and "female." The definition is structured by a closed projection that has a projection variable that ranges over the concept named "gender." The projection constrains a disjunction whose both operands are instantiation formulations. The instantiation formulations bind the projection variable to the individual concepts "female" and "male" respectively meaning that the union of instances of the individual concepts define the "gender" concept.

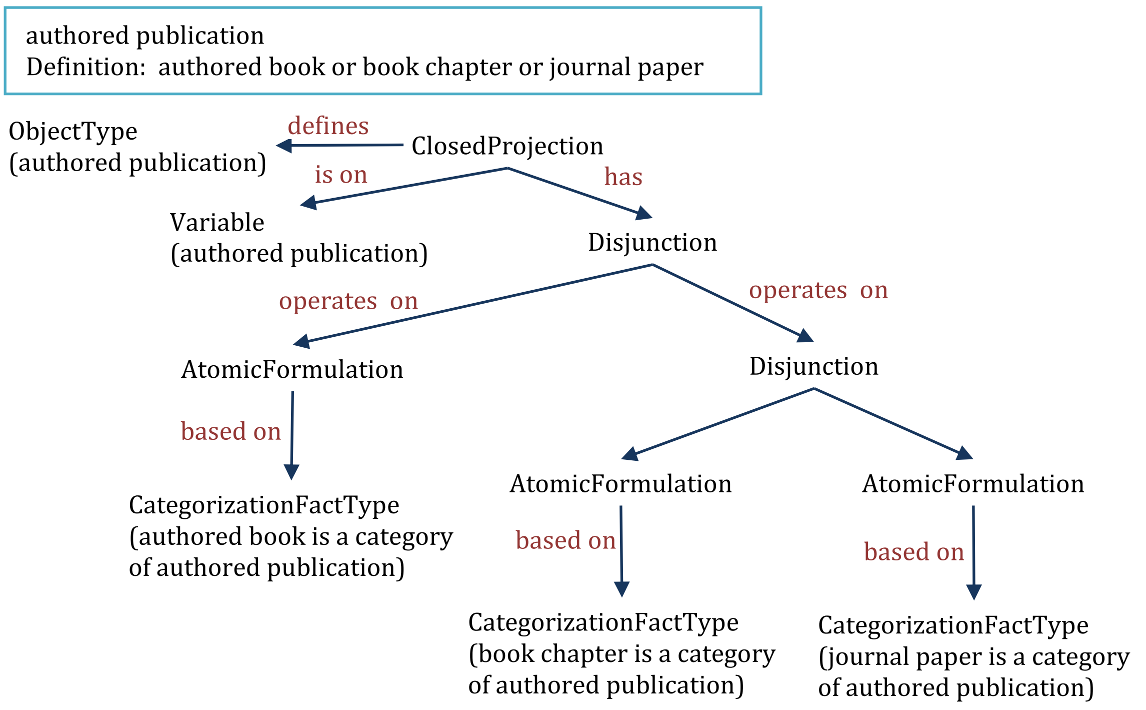

authored publication
Definition: authored book or book chapter or journal paper
ObjectType (authored publication)
defines
ClosedProjection
is on
has
Variable (authored publication)
Disjunction
operates on
operates on
AtomicFormulation
Disjunction
based on
AtomicFormulation
AtomicFormulation
CategorizationFactType (authored book is a category of authored publication)
based on
based on
CategorizationFactType (book chapter is a category of authored publication)
CategorizationFactType (journal paper is a category of authored publication)

Figure 5.30 Simplified version of the object type 'authored publication'

Abstract syntax

Each object type or value type named *e* is represented by an instance of *ObjectType* or *ValueType*, respectively. The schema elements are as follows: (1) the instance β of *ObjectType* or *ValueType*, respectively; (2) the instance of attribute *name* of β with value *e*; (3) the instance of ϕ of *ClosedProjection*; (4) the instance of relationship between *ClosedProjection* and the *NounConcept* that it defines; (5) the instance(s) of *Variable* that the projection introduces, and for each *Variable*, the instance of its relationship to the *Projection*; (6) the instance of *LogicalFormulation*; (7) the instance of relationship between the *LogicalFormulation* and the *ClosedProjection*; and (8) the set of instances defined in point 11 of Section 5.2.8 to describe the *LogicalFormulation*.

Figure 5.31 shows the abstract syntax of object type and value type with a closed projection schema units.

Now, the *predecessors() operation* of *NounConcept* is specified as follows:

```
context NounConcept::predecessors():Set(Concept)
  body:  if self.closedProjection -> notEmpty()
         then
           self.closedProjection.projectionVariable ->
             collect(factTypeRole) ->
             includes(c.oclAsType(FactTypeRole)) -> union(
               self.logicalFormulation.conceptsUsed()))
         else Set{}
         endif
```

This means that the predecessors of an object type or value type associated with a closed projection are the concepts used by the logical formulation that constrains the closed projection.

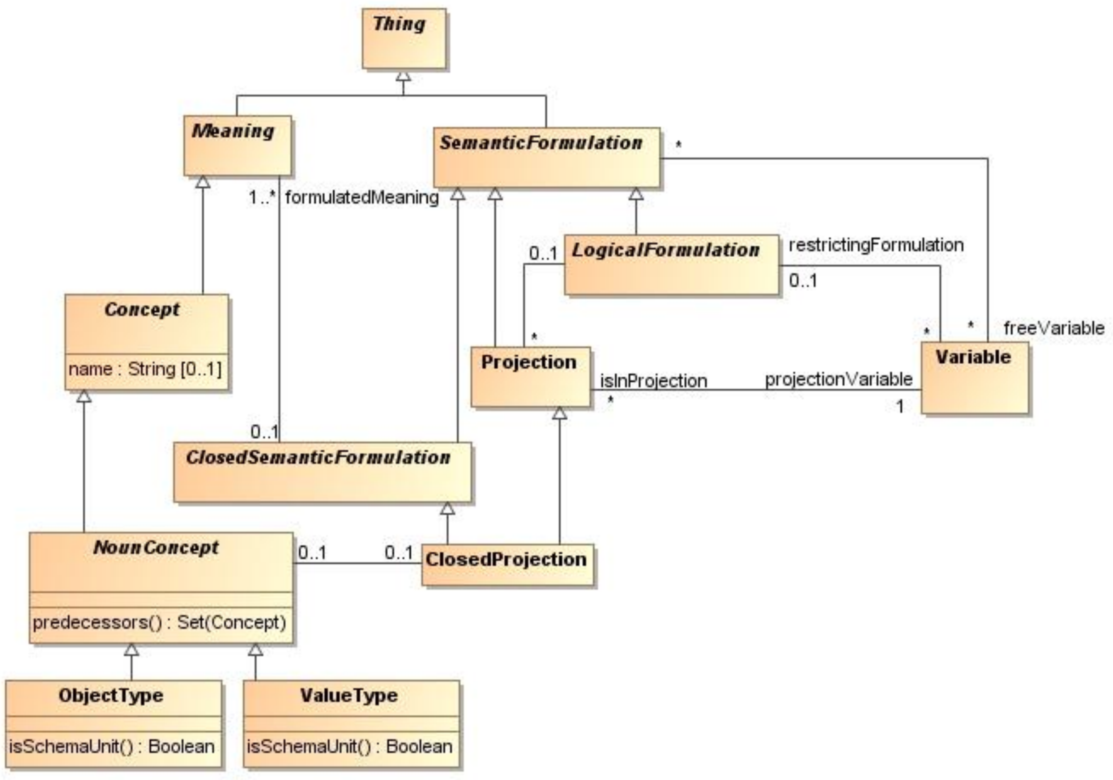


Figure 5.31 *ObjectType* and *ValueType* with closed projection schema units

Abstract syntax

Figure 5.32 shows the object type and value type schema units characterization object *NounConceptCh*.

For the object type and value type characterization object *NounConceptCh*, the *schemaUnit() operation* is formally defined as follows:

```
context NounConceptCh::schemaUnit():NounConcept
  body:  NounConcept.allInstances() -> any(c:NounConcept|
            c.name = self.name and
            if self.isValueType
            then c.oclIsTypeOf(ValueType)
            else c.oclIsTypeOf(ObjectType)
            endif and
            self.formulation -> notEmpty()implies
            ClosedProjection.allInstances() ->
               exists(cp:ClosedProjection| cp.nounConcept = c and
                  self.projectionVariable.rangedOverConcept =
                  cp.projectionVariable.rangedOverConcept.name and
                  self.formulation.existsFormulation(
                     cp.logicalFormulation)))
```

This means that the *schemaUnit() operation* of *NounConceptCh* is a query that gives the instance of *ObjectType* or *ValueType*, as explained in Section 5.2.2. Additionally, if *NounConceptCh* is associated with a *Formulation*, the instance of *ObjectType* or *ValueType* is associated with a *ClosedProjection*. The closed projection may be associated with the same class *Formulation* defined above (see Section 5.2.8).

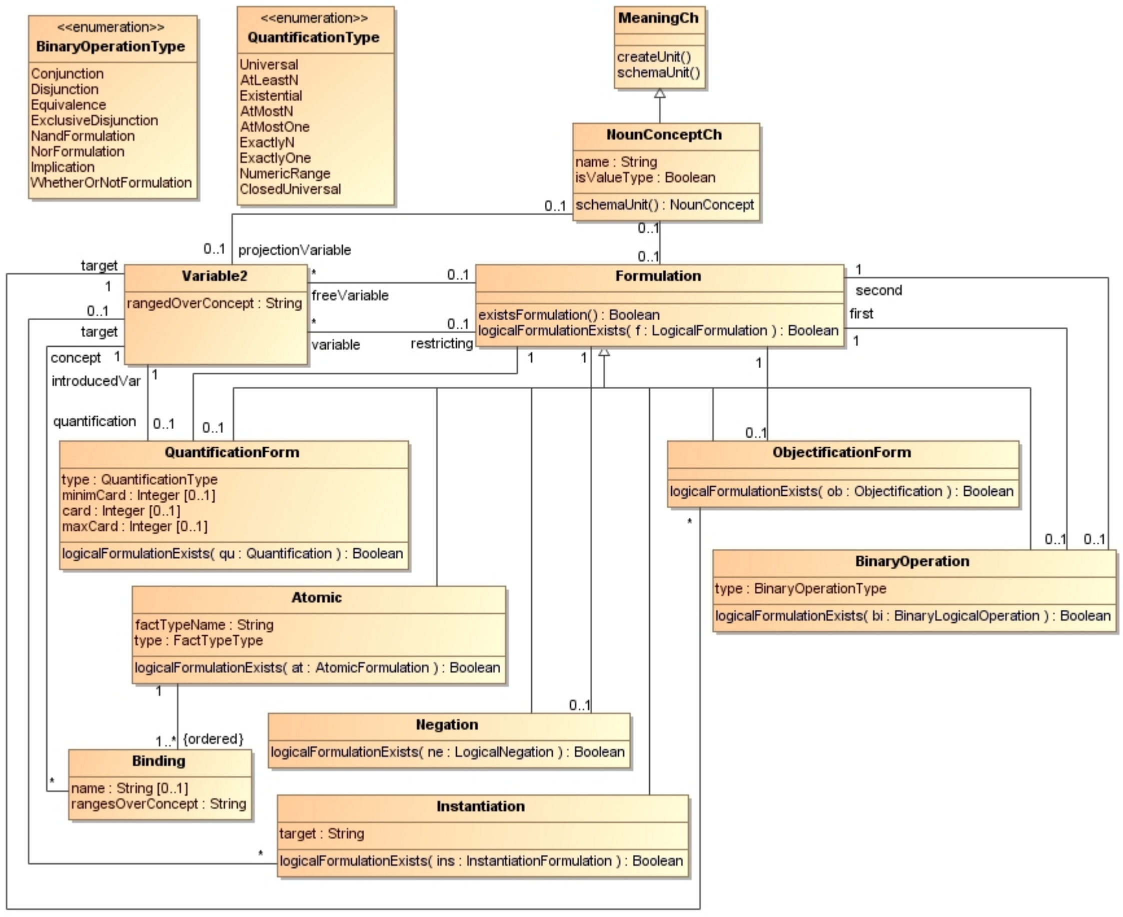


Figure 5.32 Object type and value type schema units characterization object *NounConceptCh*

The complete specification in the USE tool of the SBVR metaschema can be found in Appendix D. Appendix E shows a representative fragment of the instances defined in USE to specify the DBLP schema as an instance of the SBVR metaschema. The methods for creating schema units of the characterization objects are provided in Appendix F.

6 Translation mapping expressions between UML and SBVR meanings

The schema translation problem was described, in Chapter 3 as follows: given a (source) metaschema MS_1, a (source) schema S_1 (instance of MS_1) and a (target) metaschema MS_2, obtain a schema S_2, an instance of MS_2, that suitably corresponds to S_1.

Chapter 3 also described how to define and use translation mapping expressions between any two MOF metaschemas. Such expressions are defined by two invariants involving the relationships between the schema units of the two metaschemas. The two invariants are defined in OCL and the relationships between schema units are defined by means of operations whose pre- and postconditions are also formalized in OCL. The end of Chapter 3 described how to automatically translate between instances of the two metaschemas.

This chapter applies the translation approach proposed in Chapter 3 to the UML and SBVR meanings metaschemas described in Chapter 4 and Chapter 5, respectively. In particular, it describes the set of all operations that are mapping-dependent between the two languages.

This chapter is structured as follows:

- Section 6.1 defines the query operations that indicate how schema units of UML are translated to SBVR and vice versa. This is the mapping kind (*HasEquivalents*, *IsIncluded* or *Untraslatable*) relationship among schema units of the two metaschemas.
- Section 6.2 defines the *sbvrEquivalents()* operations on the UML schema units whose mapping kinds are *HasEquivalents*.
- Section 6.3 defines the *includedInUml()* operations on the SBVR schema units whose mapping kinds are *IsIncluded*.
- Section 6.4 specifies the translation mapping constraints, defined formally in OCL, as two invariants called *completeAndConsistentMappingToUML* and *completeAndConsistentMappingToSBVR*.
- Section 6.5 describes the *translateToUml* and *translateToSbvr* operations for automatically translating from UML to SBVR and vice versa.

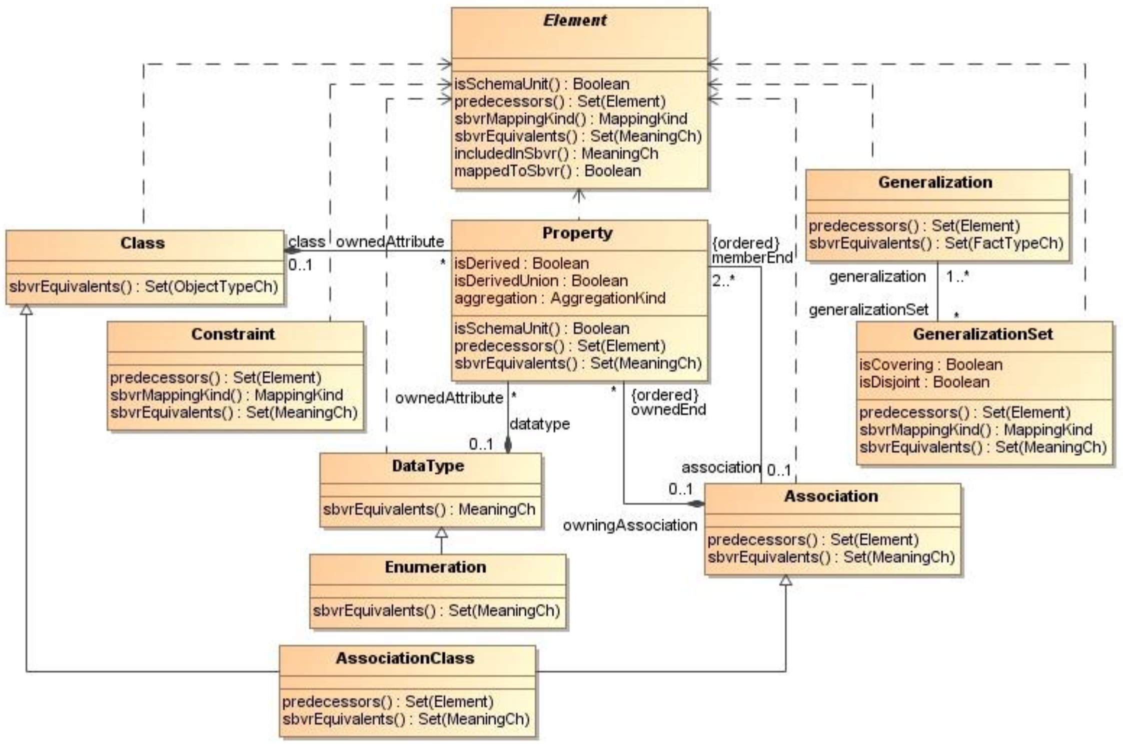

Element
isSchemaUnit() : Boolean
predecessors() : Set(Element)
sbvrMappingKind() : MappingKind
sbvrEquivalents() : Set(MeaningCh)
includedInSbvr() : MeaningCh
mappedToSbvr() : Boolean
Class
sbvrEquivalents() : Set(ObjectTypeCh)
class
0..1
ownedAttribute
*
Property
isDerived : Boolean
isDerivedUnion : Boolean
aggregation : AggregationKind
isSchemaUnit() : Boolean
predecessors() : Set(Element)
sbvrEquivalents() : Set(MeaningCh)
{ordered} memberEnd
2..*
Generalization
predecessors() : Set(Element)
sbvrEquivalents() : Set(FactTypeCh)
generalization
1..*
generalizationSet
*
GeneralizationSet
isCovering : Boolean
isDisjoint : Boolean
predecessors() : Set(Element)
sbvrMappingKind() : MappingKind
sbvrEquivalents() : Set(MeaningCh)
Constraint
predecessors() : Set(Element)
sbvrMappingKind() : MappingKind
sbvrEquivalents() : Set(MeaningCh)
ownedAttribute
*
datatype
0..1
*
{ordered} ownedEnd
association
0..1
0..1
owningAssociation
DataType
sbvrEquivalents() : MeaningCh
Association
predecessors() : Set(Element)
sbvrEquivalents() : Set(MeaningCh)
Enumeration
sbvrEquivalents() : Set(MeaningCh)
AssociationClass
predecessors() : Set(Element)
sbvrEquivalents() : Set(MeaningCh)

Figure 6.1 Definition of UML schema units including SBVR mapping-dependent operations

Figure 6.1 shows the definition of UML schema units, which now includes the set of SBVR mapping-dependent operations. The dashed lines in the figure show that the subtypes of *Element* are indirect subtypes rather than direct subtypes.

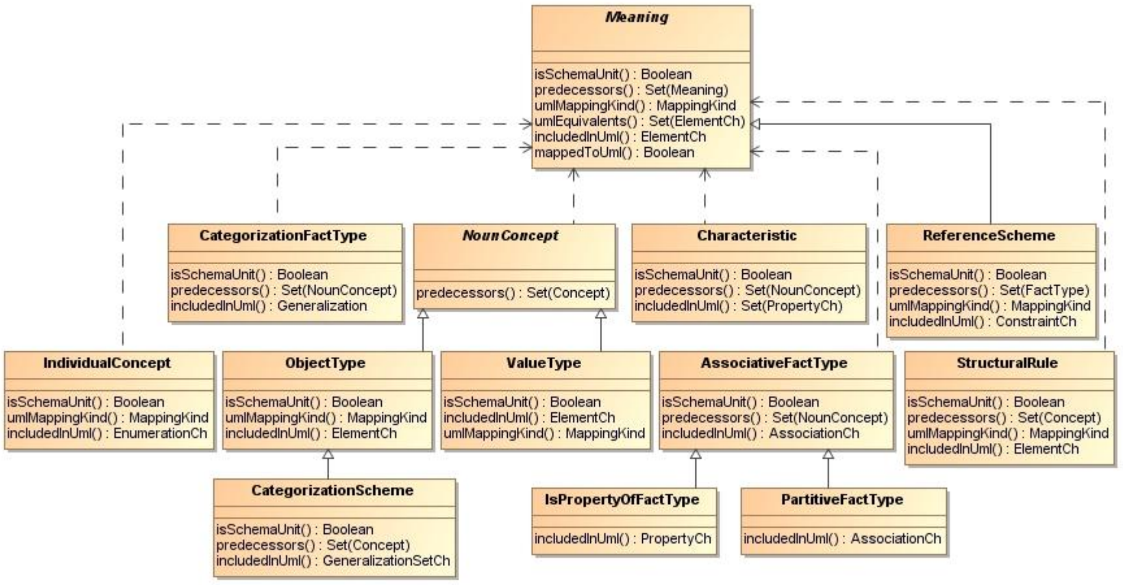

Meaning
isSchemaUnit() : Boolean
predecessors() : Set(Meaning)
umlMappingKind() : MappingKind
umlEquivalents() : Set(ElementCh)
includedInUml() : ElementCh
mappedToUml() : Boolean
CategorizationFactType
isSchemaUnit() : Boolean
predecessors() : Set(NounConcept)
includedInUml() : Generalization
NounConcept
predecessors() : Set(Concept)
Characteristic
isSchemaUnit() : Boolean
predecessors() : Set(NounConcept)
includedInUml() : Set(PropertyCh)
ReferenceScheme
isSchemaUnit() : Boolean
predecessors() : Set(FactType)
umlMappingKind() : MappingKind
includedInUml() : ConstraintCh
IndividualConcept
isSchemaUnit() : Boolean
umlMappingKind() : MappingKind
includedInUml() : EnumerationCh
ObjectType
isSchemaUnit() : Boolean
umlMappingKind() : MappingKind
includedInUml() : ElementCh
ValueType
isSchemaUnit() : Boolean
includedInUml() : ElementCh
umlMappingKind() : MappingKind
AssociativeFactType
isSchemaUnit() : Boolean
predecessors() : Set(NounConcept)
includedInUml() : AssociationCh
StructuralRule
isSchemaUnit() : Boolean
predecessors() : Set(Concept)
umlMappingKind() : MappingKind
includedInUml() : ElementCh
CategorizationScheme
isSchemaUnit() : Boolean
predecessors() : Set(Concept)
includedInUml() : GeneralizationSetCh
IsPropertyOfFactType
includedInUml() : PropertyCh
PartitiveFactType
includedInUml() : AssociationCh

Figure 6.2 Definition of SBVR schema units including UML mapping dependent operations.

Figure 6.2 shows the definition of SBVR meanings schema units, which now includes the set of UML mapping dependent operations. The dashed lines in the figure show that the subtypes of *Meaning* are indirect subtypes rather than direct subtypes.

6.1 *umlMappingKind()* and *sbvrMappingKind()* operations

This section describes two query operations: (i) the *sbvrMappingKind()* query operation, in the context of *Element*, whose value indicates how a UML schema unit is translated into SBVR; and (ii) the *umlMappingKind()* query operation, in the context of *Meaning*, whose value indicates how an SBVR schema unit is translated into UML. As explained in Chapter 3, the value of both operations is an enumeration data type whose values are *HasEquivalents*, *IsIncluded* and *Untranslatable*.

To ensure that both operations are only defined in instances of schema units of both schemas, the following preconditions are defined:

```
context Element::sbvrMappingKind():MappingKind
  pre: isSchemaUnit()

context Meaning::umlMappingKind():MappingKind
  pre: isSchemaUnit()
```

6.1.1 UML side

On the UML side, the operation *sbvrMappingKind()* is defined, in *Element*, as follows:

```
context Element::sbvrMappingKind():MappingKind
  body: MappingKind::HasEquivalents
```

This means that, by default, all (direct or indirect) instances of *Element* that are schema units have an equivalence mapping, and that those instances that are not schema units have an undefined value for the operation. There are some exceptions, and therefore the operation is redefined in certain subtypes of *Element*, as shown in Figure 6.1.

In UML, generalization sets may or may not have a name. In SBVR, there is no concept without a name, so generalization sets must always have a name in order to have an equivalent in SBVR. This is formally defined as follows:

```
context GeneralizationSet::sbvrMappingKind():MappingKind
  body:  if self.name -> isEmpty()
         then MappingKind::Untranslatable
         else MappingKind::HasEquivalents
         endif
```

As stated in Chapter 4, this thesis limits the types of constraints considered for translation to the predefined static constraint *XOR* and certain uniqueness constraints: those that indicate that one attribute is a key of the class that contains said attribute (i.e., those specified as invariants in the DBLP example). The whole OCL metaschema would need to be included in order to translate other types of constraints, so this has been left for further work. Therefore, the *sbvrMappingKind* operation of *Constraint* is defined as follows:

```
context Constraint::sbvrMappingKind():MappingKind
  body: if self.specification.oclAsType(Expression).symbol = 'XOR'
         or self.constrainedElement->exists(e1,e2|
           e1.oclIsTypeOf(Class) and e2.oclIsTypeOf(Property) and
           e2.oclAsType(Property).class = e1.oclAsType(Class) and
           self.specification.oclAsType(OpaqueExpression).body =
           e1.oclAsType(Class).name.concat('.allInstances->
             isUnique('.concat(e2.oclAsType(Property).name.
                concat(')'))))
          then MappingKind:: HasEquivalents
          else MappingKind:: Untranslatable
          endif
```

This means that the constraint has equivalents in SBVR if it corresponds to the predefined *XOR* constraint or if it constrains a class and an attribute of the class and the specification of the express constraint that the attribute is a key of the class.

6.1.2 SBVR meanings side

On the SBVR meanings side, the operation *umlMappingKind()* is defined, in *Meaning*, as follows:

```
context Meaning::umlMappingKind():MappingKind
  body: MappingKind::IsIncluded
```

This means that, by default, all (direct or indirect) instances of *Meaning* that are schema units have an inclusion mapping, and that those instances that are not schema units have an undefined value for the operation. There are some exceptions, and therefore the operation is redefined in certain subtypes of *Meaning*, as shown in Figure 6.2.

SBVR allows *IndividualConcept* to be defined as part of the schema. Here, only those which are included in the definition of another concept have UML equivalents (in UML, each *IndividualConcept* corresponds to an *EnumerationLiteral*). This is formalized as follows:

```
context IndividualConcept::umlMappingKind():MappingKind
  body: if self.variable.projection->notEmpty()
        then  MappingKind::IsIncluded
        else MappingKind:: Untranslatable
        endif
```

An SBVR *ReferenceScheme* indicates which roles and characteristics identify concepts. For translation into UML, this thesis only considers the reference schemes that are equivalent to the identifier constraint—that is, the *ReferenceScheme* that references object types (not fact types). This is formally specified as follows:

```
context ReferenceScheme::umlMappingKind():MappingKind
  body: if self.referencedConcept -> forAll(oclIsType(ObjectType))
        then  MappingKind::IsIncluded
        else MappingKind::Untranslatable
        endif
```

In order to translate any type of structural rule, it would be necessary to include the entire OCL metaschema in the UML metaschema. Since this inclusion has been left for further work, only a limited set of SBVR is translatable into UML. In particular, the structural rules considered translatable to UML are those that represent graphical UML constraints (multiplicities, XOR, and disjointness and completeness of generalization sets) and the identifier constraint mentioned above.

Figure 6.3 shows the general form of a structural rule representing the multiplicity constraint between two or more concepts, *Card(concept$_1$, ... concept$_{n-1}$;concept$_n$;associativeFactType) = (min,max)*. The rule is structured by a universal quantification that introduces a variable for each concept *∈(concept$_1$, ... concept$_{n-1}$,)* and scopes over a quantification (e.g., existential, exactly, at-least-n, etc.). The quantification introduces a variable that ranges over the concept *concept$_n$* and scopes over an atomic formulation based on the *associativeFactType* that relates all of the previous concepts. An example was shown in Figure 5.18 of Chapter 5.

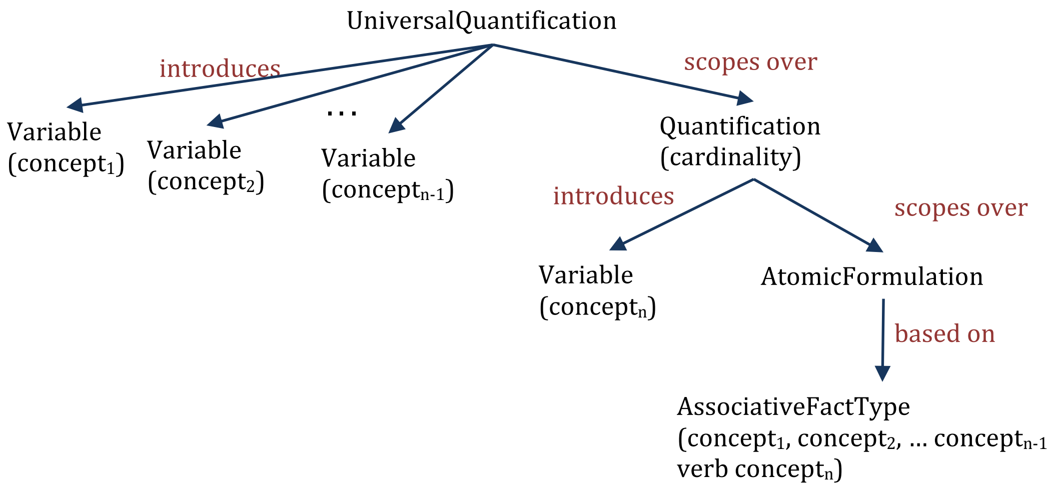


Figure 6.3 General form of structural rule representing a multiplicity constraint

In order to check whether a structural rule corresponds to a multiplicity constraint, the following operation is defined in the *StructuralRule* context.

The *isMultiplicity()* operation returns a Boolean whose value is *true* if the meaning of the structural rule (self) corresponds to the meaning of a UML cardinality constraint.

```
context StructuralRule : isMultiplicity() : Boolean
  body:   self.closedLogicalFormulation.
          oclAsType(ClosedUniversalQuantification).
          scopeFormulation.oclAsType(Quantification).
          scopeFormulation.isAtomicOfAssociativeFactType()
```

The *isAtomicOfAssociativeFactType()* operation is defined in the context of a *LogicalFormulation* and returns *true* if the logical formulation corresponds to an atomic formulation based on an associative fact type (or an is-property-of fact type).

```
context LogicalFormulation::isAtomicOfAssociativeFactType() :
          Boolean
  body:   self.oclIsTypeOf(AtomicFormulation) and
          (self.oclAsType(AtomicFormulation).factType.
            oclIsTypeOf(AssociativeFactType) or
          self.oclAsType(AtomicFormulation).factType.
            oclIsTypeOf(IsPropertyOfFactType))
```

The disjointness and covering constraint of a generalization set may be specified with a unique rule if the generalization set has exactly two generalizations. Figure 6.4 shows the general form of such a structural rule. The figure shows the structure of a rule representing "each generalConcept is a category$_1$ or is a category$_2$ but not both." The rule is structured by a closed universal quantification (each) that introduces a variable that ranges over the general concept. The quantification scopes over an exclusive disjunction (or… but not both). Each operand of the exclusive disjunction is an atomic formulation based on the categorization fact type (i.e., $category_1$ is a category of generalConcept and $category_2$ is a category of generalConcept). An example was shown in Figure 5.19 of Chapter 5.

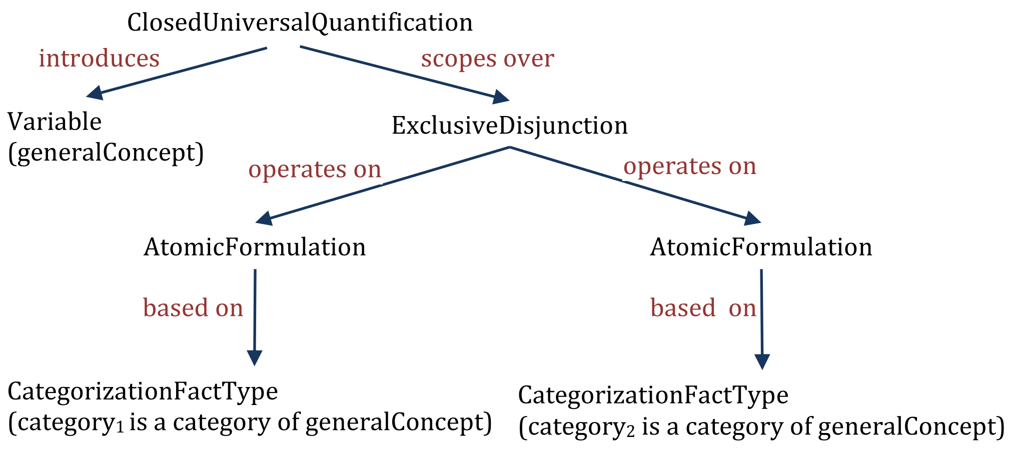

ClosedUniversalQuantification
introduces
scopes over
Variable (generalConcept)
ExclusiveDisjunction
operates on
operates on
AtomicFormulation
AtomicFormulation
based on
based on
CategorizationFactType (category1 is a category of generalConcept)
CategorizationFactType (category2 is a category of generalConcept)

Figure 6.4 General form of structural rule representing covering and disjointness of a generalization set with two generalizations

The *isDisjointAndCovering()* operation returns a Boolean whose value is *true* if the meaning of the structural rule (self) corresponds to the meaning of a UML {disjoint,complete} constraint of a generalization set with two generalizations.

```
context StructuralRule::isDisjointAndCovering() : Boolean
  body:  self.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         scopeFormulation.oclAsType(ExclusiveDisjunction).
         logicalOperand1.oclAsType(AtomicFormulation) ->
           notEmpty() and self.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         scopeFormulation.oclAsType(ExclusiveDisjunction).
         logicalOperand2.oclAsType(AtomicFormulation) ->
           notEmpty()
```

In the case of a generalization set with three or more generalizations, the covering constraint is represented by a unique rule, and the disjointness constraint is represented by a different set of rules.

The covering rule is structured by a closed universal quantification that introduces a variable that ranges over the general concept of the generalization set. The quantification scopes over a tree of *n-1* disjunctions, where *n* is the number of generalizations that compound the generalization set, as shown in Figure 6.5. The structure of the covering rule also serves in the case of a generalization set with two generalizations.

The *isCovering()* operation returns a Boolean whose value is *true* if the meaning of the structural rule (self) corresponds to the meaning of a UML {complete} constraint of a generalization set. The rule is structured as shown in Figure 6.5.

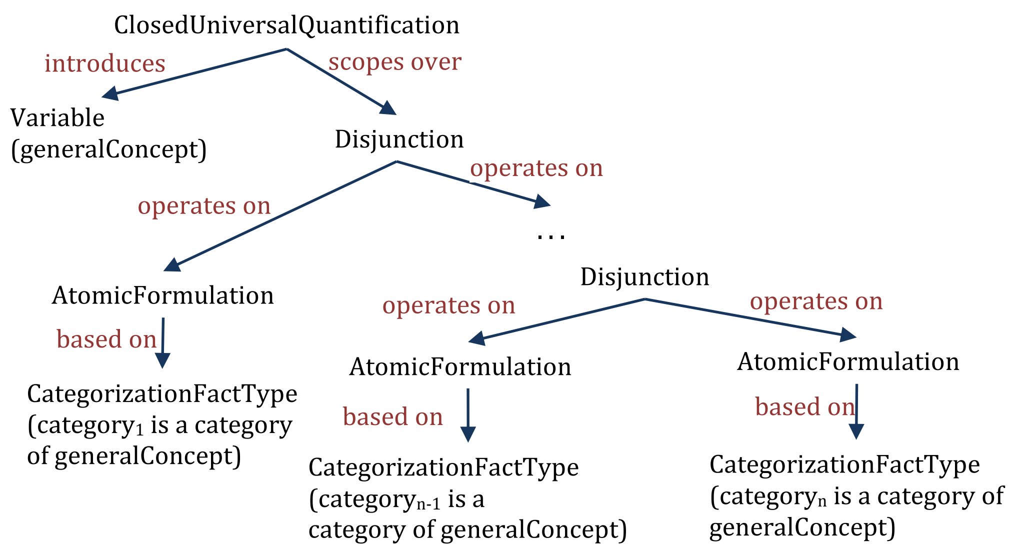

ClosedUniversalQuantification
introduces
scopes over
Variable (generalConcept)
Disjunction
operates on
operates on
...
Disjunction
operates on
operates on
AtomicFormulation
based on
CategorizationFactType (category1 is a category of generalConcept)
AtomicFormulation
based on
CategorizationFactType (categoryn-1 is a category of generalConcept)
AtomicFormulation
based on
CategorizationFactType (categoryn is a category of generalConcept)

Figure 6.5 General form of the structural rule representing the covering constraint of a generalization set

```
context StructuralRule::isCovering() : Boolean
  body:  self.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         scopeFormulation.isDisjunctionOfCoveringRule()
```

The *isDisjunctionOfCoveringRule()* operation is defined in the context of a *LogicalFormulation* and returns *true* if the logical formulation corresponds to a disjunction where the first operand is an atomic formulation based on a categorization fact type and the second operand also corresponds to an atomic formulation based on a categorization fact type or to a second disjunction structured like the previous one.

```
context LogicalFormulation::isDisjunctionOfCoveringRule() : Boolean
  body:  self.oclIsTypeOf(Disjunction) and
         self.oclAsType(Disjunction).logicalOperand1.
           isAtomicOfCategorization() and
         (self.oclAsType(Disjunction).logicalOperand2.
           isAtomicOfCategorization() or
         self.oclAsType(Disjunction).logicalOperand2.
           isDisjunctionOfCoveringRule())
```

The *isAtomicOfCategorization()* operation is defined in the context of a *LogicalFormulation* and returns *true* if the logical formulation corresponds to an atomic formulation based on a categorization fact type.

```
context LogicalFormulation::isAtomicOfCategorization() : Boolean
  body:  self.oclIsTypeOf(AtomicFormulation) and
         self.oclAsType(AtomicFormulation).factType.
         oclIsTypeOf(CategorizationFactType)
```

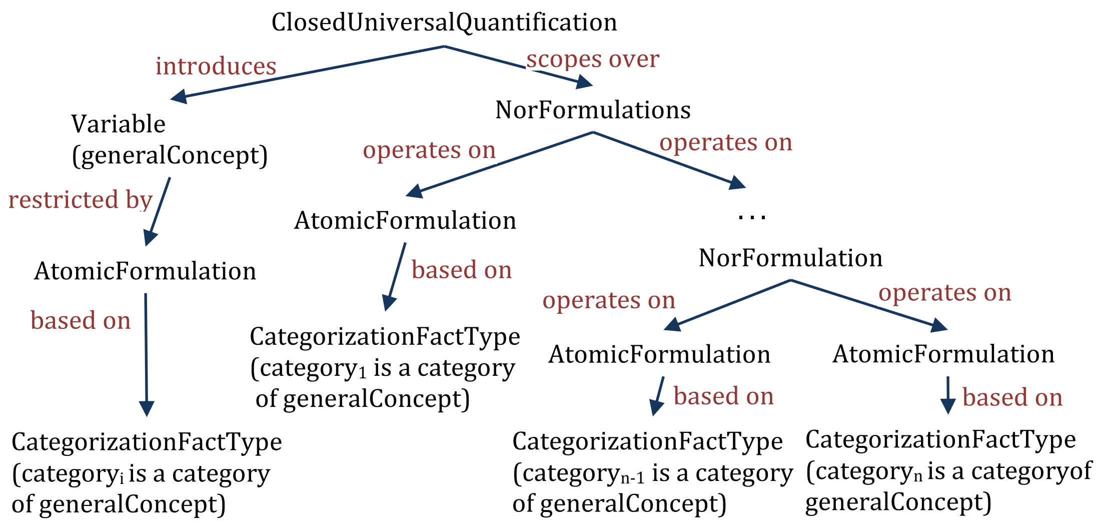

ClosedUniversalQuantification
introduces
scopes over
Variable (generalConcept)
NorFormulations
restricted by
operates on
operates on
AtomicFormulation
AtomicFormulation
...
based on
based on
NorFormulation
CategorizationFactType (category$_1$ is a category of generalConcept)
operates on
operates on
AtomicFormulation
AtomicFormulation
based on
based on
CategorizationFactType (category$_i$ is a category of generalConcept)
CategorizationFactType (category$_{n-1}$ is a category of generalConcept)
CategorizationFactType (category$_n$ is a categoryof generalConcept)

Figure 6.6 General form of the structural rule partially representing the disjointness constraint of a generalization set

In order to define the disjointness constraint of a generalization set with $n > 3$ generalizations, it is necessary to define *n-1* structural rules. Each rule means that if an instance of the general concept is an instance of one of the categories, then this instance cannot be an instance of any of the other categories. Figure 6.6 shows the general form of the structural rule.

The *isDisjoint()* operation is a Boolean whose value is *true* if the meaning of the structural rule (self) corresponds to the meaning of a UML {disjoint} constraint of a generalization set.

```
context StructuralRule::isDisjoint() : Boolean =
  body:  self.closedLogicalFormulation.
           oclAsType(ClosedUniversalQuantification).
           introducedVariable.restrictingFormulation.
           isAtomicOfCategorization() and
         self.closedLogicalFormulation.
           oclAsType(ClosedUniversalQuantification).
           scopeFormulation.isNorFormulationOfDisjointRule()
```

The *isNorFormulationOfDisjointRule()* operation is defined in the context of a *LogicalFormulation* and returns *true* if the logical formulation corresponds to a nor formulation where the first operand is an atomic formulation based on a categorization fact type and the second operand is also an atomic formulation based on a categorization fact type or a second nor formulation structured like the previous one.

```
context LogicalFormulation::isNorFormulationOfDisjointRule() :
         Boolean
  body:  self.oclIsTypeOf(NorFormulation) and
         self.oclAsType(NorFormulation).logicalOperand1.
           isAtomicOfCategorization() and
         (self.oclAsType(NorFormulation).logicalOperand2.
           isAtomicOfCategorization() or
         self.oclAsType(NorFormulation).logicalOperand2.
           isNorFormulationOfDisjointRule())
```

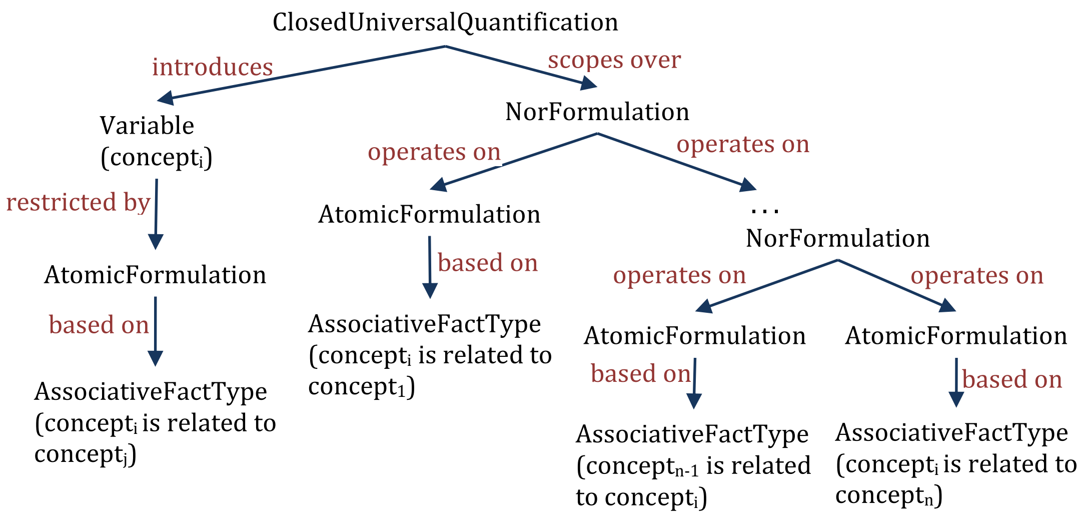

ClosedUniversalQuantification
introduces
scopes over
Variable (concept$_i$)
NorFormulation
restricted by
operates on
operates on
AtomicFormulation
...
NorFormulation
based on
AtomicFormulation
based on
operates on
operates on
AssociativeFactType (concept$_i$ is related to concept$_1$)
AtomicFormulation
AtomicFormulation
AssociativeFactType (concept$_i$ is related to concept$_j$)
based on
based on
AssociativeFactType (concept$_{n-1}$ is related to concept$_i$)
AssociativeFactType (concept$_i$ is related to concept$_n$)

Figure 6.7 General form of the structural rule partially representing the *XOR* constraint

In order to define the *XOR* constraint among $n \geq 2$ associations, it is necessary to define *n-1* structural rules. Each rule means that if an instance of the concept is associated with an instance of a second concept, then the first instance cannot be associated with an instance of any of the other concepts participating in the *XOR* relationship. Figure 6.7 shows the general form of the structural rule.

The *isXOR()* operation is a Boolean whose value is *true* if the meaning of the structural rule (self) corresponds to the meaning of a UML {XOR} constraint.

```
context StructuralRule::isXOR() : Boolean
  body:  self.closedLogicalFormulation.
           oclAsType(ClosedUniversalQuantification).
           introducedVariable.restrictingFormulation.
           isAtomicOfAssociativeFactType() and
         self.closedLogicalFormulation.
           oclAsType(ClosedUniversalQuantification).
           scopeFormulation.isNorFormulationOfXORRule()
```

The *isNorFormulationOfXORRule()* operation is defined in the context of a *LogicalFormulation* and returns *true* if the logical formulation corresponds to a nor formulation where the first operand is an atomic formulation based on an associative fact type and the second operand is also an atomic formulation based on an associative fact type or a second nor formulation structured like the previous one.

```
context LogicalFormulation::isNorFormulationOfXORRule() :
         Boolean
  body:  self.oclIsTypeOf(NorFormulation) and
         self.oclAsType(NorFormulation).logicalOperand1.
           isAtomicOfAssociativeFactType() and
         (self.oclAsType(NorFormulation).logicalOperand2.
           isAtomicOfAssociativeFactType() or
         self.oclAsType(NorFormulation).logicalOperand2.
           isNorFormulationOfXORRule())
```

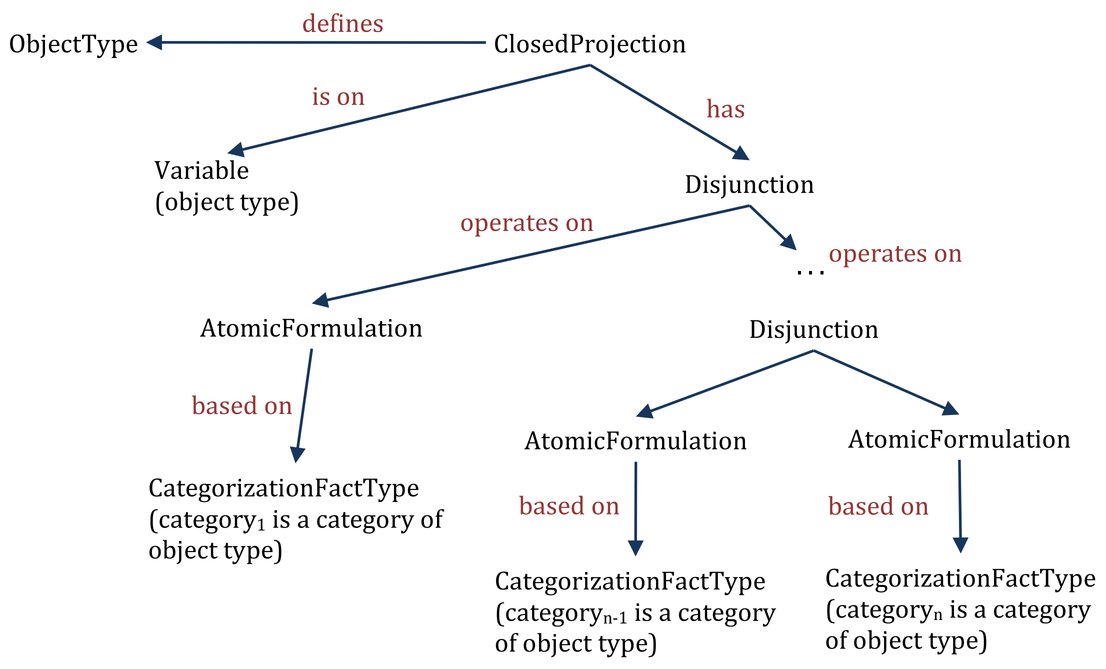

ObjectType
defines
ClosedProjection
is on
has
Variable (object type)
Disjunction
operates on
operates on
...
AtomicFormulation
Disjunction
based on
AtomicFormulation
AtomicFormulation
CategorizationFactType (category1 is a category of object type)
based on
based on
CategorizationFactType (categoryn-1 is a category of object type)
CategorizationFactType (categoryn is a category of object type)

Figure 6.8 General form of an object type whose extension is defined as the union of the instances of other object types

Using the operations defined above, the *umlMappingKind()* operation of *StructuralRule* is defined as follows:

```
context StructuralRule::umlMappingKind():MappingKind
  body:  if isMultiplicity() or isDisjointAndCovering() or
            isCovering() or isXOR() or isDisjoint()
         then MappingKind::IsIncluded
         else MappingKind::Untranslatable
         endif
```

Closed projections are used to formalize the definition of a concept. As in the case of structural rules, the whole OCL metamodel would be necessary in order to automate the translation of all possible definitions. Here, the concepts considered translatable are those whose closed projections represent graphical UML definitions, such as definitions of enumerations, definitions of abstract classes and definitions of association classes.

The meaning of an abstract class with $n > 1$ subclasses, in SBVR, is defined as an object type defined by a closed projection that has *n-1* disjunctions structured as a tree, as shown in Figure 6.8. Figure 5.30 in Chapter 5 was an example of this.

The *isAbstract()* operation returns a Boolean whose value is *true* if the meaning of the closed projection (self) corresponds to the meaning of a UML class *abstract* that has a partition of two subclasses. The structure of the disjunction is the same as that of the covering rule described above.

```
context ClosedProjection::isAbstract() : Boolean
  body: self.logicalFormulation.isDisjunctionOfCoveringRule()
```

The *isObjectification()* operation returns a Boolean whose value is *true* if the meaning of the closed projection (self) corresponds to the meaning of the reification of an associative fact type.

```
    context ClosedProjection::isObjectification() : Boolean
      body:  self.logicalFormulation.oclIsTypeOf(Objectification)
```

Therefore, the *umlMappingKind()* operation of *ObjectType* is redefined as follows:

```
    context ObjectType::umlMappingKind() : MappingKind
      body:  if self.closedProjection -> isEmpty()
             then MappingKind::IsIncluded
             else if self.closedProjection.isAbstract() or
                        self.closedProjection.isObjectification()
                  then MappingKind::IsIncluded
                  else MappingKind::Untranslatable
                  endif
             endif
```

An enumeration with $n > 1$ literals, in SBVR, is defined as a value type defined by a closed projection that has *n-1* disjunctions structured as a tree, as shown in Figure 6.9.

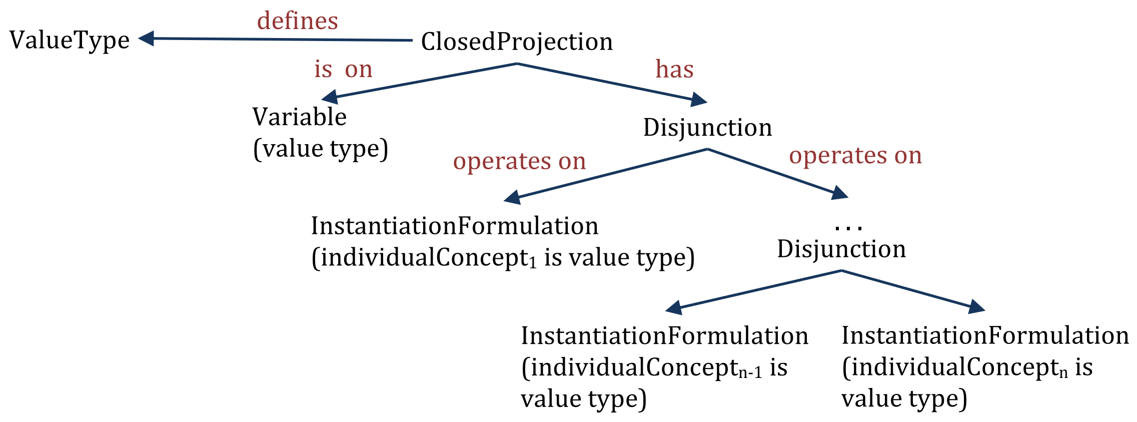

ValueType
defines
ClosedProjection
is on
has
Variable (value type)
Disjunction
operates on
operates on
InstantiationFormulation (individualConcept$_1$ is value type)
...
Disjunction
InstantiationFormulation (individualConcept$_{n-1}$ is value type)
InstantiationFormulation (individualConcept$_n$ is value type)

Figure 6.9 General form of a value type whose extension is defined as the union of the instances of individual concepts

The *isEnumeration()* operation returns a Boolean whose value is *true* if the meaning of the closed projection (self) corresponds to the meaning of a UML enumeration.

```
    context ClosedProjection::isEnumeration() : Boolean
      body:  self.logicalFormulation.isDisjunctionOfIndividuals
```

The *isDisjunctionOfIndividuals()* operation, defined in the context of a logical formulation, returns a Boolean whose value is *true* if the logical formulation is structured as shown in Figure 6.9.

```
    context LogicalFormulation::isDisjunctionOfIndividuals() : Boolean
      body:  self.oclIsTypeOf(Disjunction) and
             self.oclAsType(Disjunction).logicalOperand1.
             oclIsTypeOf(InstantiationFormulation) and
             (self.oclAsType(Disjunction).logicalOperand2.
             oclIsTypeOf(InstantiationFormulation) or
             self.oclAsType(Disjunction).logicalOperand2.
             isDisjunctionOfIndividuals())
```

Therefore, the *umlMappingKind* operation of *ValueType* is redefined as follows:

```
    context ValueType::umlMappingKind() : MappingKind
      body:  if self.closedProjection -> isEmpty()
             then MappingKind::IsIncluded
             else if self.closedProjection.isEnumeration()
                  then MappingKind::IsIncluded
                  else MappingKind::Untranslatable
                  endif
             endif
```

6.2 *sbvrEquivalents()* and *umlEquivalents()* operations

This section describes the *sbvrEquivalents()* and *umlEquivalents* operations in the context of *Element* and *Meaning*, respectively.

In this section the *hasSent* ('^') operator is used to invoke operations within a postcondition (Object Management Group 2006b, pàg. 29). The use of this operator allows to better structure the postconditions. However, note that the USE tool does not allow the use of such operator. Therefore, the implementation, in USE, of a postcondition with an invocation to an operation includes, within the postcondition, the fragment corresponding to the invoked operation.

6.2.1 UML side

The evaluation of the *sbvrEquivalents()* operation on a UML schema unit whose mapping kind is *HasEquivalents* gives the set of SBVR characterization objects that are equivalent to the UML schema unit. The signature and precondition of the operation in OCL is as follows:

```
    context Element::sbvrEquivalents():Set(MeaningCh)
      pre: sbvrMappingKind() = MappingKind::HasEquivalents
      post definingTheResult:
           result = MeaningCh.allInstances() -
                      MeaningCh.allInstances@pre()
```

The effect of the operation is redefined in the subtypes of *Element* such that some or all of their instances represent schema units whose mapping kind is *HasEquivalents*.

The following subsections give, for each *sbvrEquivalents()* operation redefined, examples that illustrate the mapping between elements of the two schemas, the general description of the operation and the formal specification of the operation in OCL.

6.2.1.1 *sbvrEquivalents() of data type*

Examples

The data type schema units represented by instances of *DataType* named "Natural" and "Year" are represented, in SBVR, as two instances of *ValueType* with the same name, respectively.

General description

A *DataType* schema unit maps to a characterization object of *ValueType*. The *name* attribute of *ValueTypeCh* is the same as the *DataType* name.

Formal specification in OCL

```
    context DataType::sbvrEquivalents():Set(NounConceptCh)
      post NounConceptChCreated:
           nc.oclIsNew() and nc.oclIsTypeOf(NounConceptCh) and
           nc.name = self.name and nc.isValueType = true
```

6.2.1.2 *sbvrEquivalents() of class*

Examples

One example of a class schema unit included in the DBLP schema (see Figure 6.10) is the class schema unit "Person" represented by the instance of *Class* whose *name* attribute has the value "Person." The equivalent meaning, in SBVR, is represented by an object type schema unit represented by the instance of *ObjectType* whose *name* attribute has the value "person."

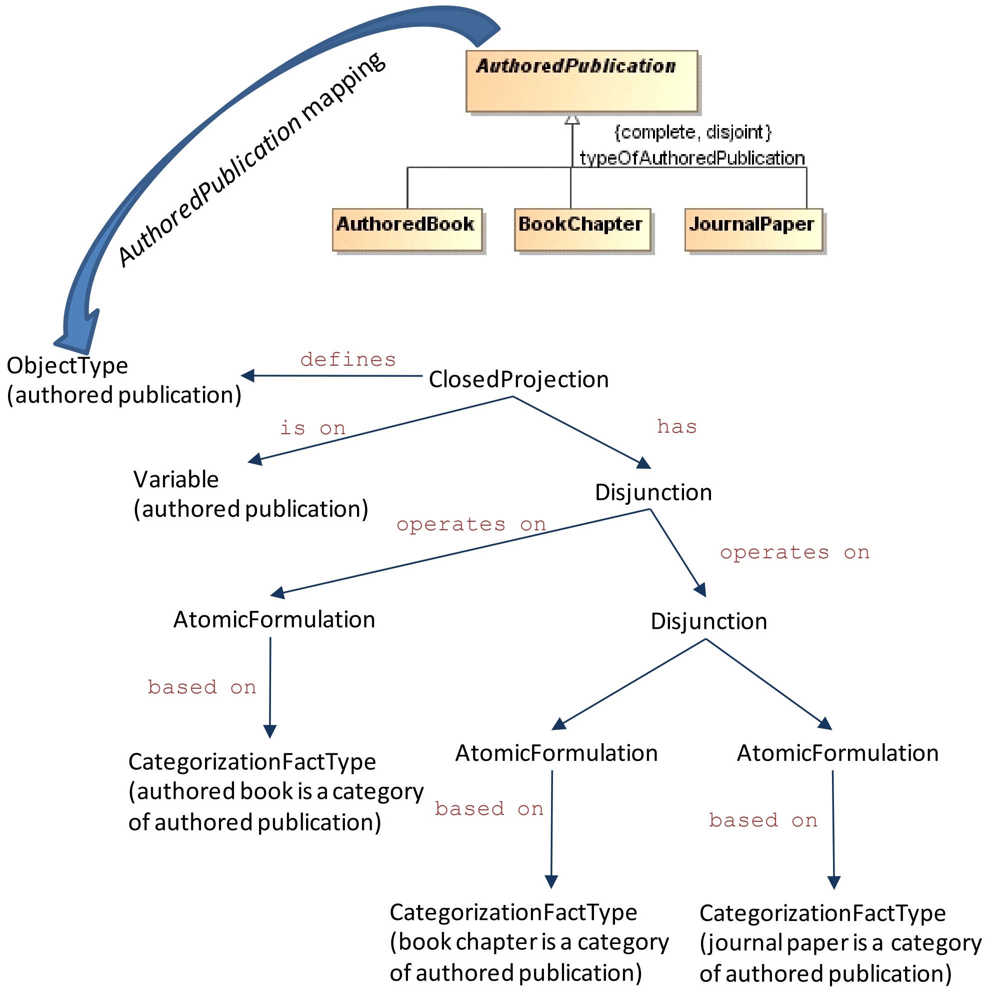


Figure 6.10 Example of mapping the abstract class "AuthoredPublication" to SBVR

The DBLP example also includes a class schema unit "Publication" represented by the instance of *Class* whose *name* and *isAbstract* attributes have the values "AuthoredPublication" and *True*, respectively. In this case, the "AuthoredPublication" class schema unit is equivalent, in SBVR, to an "authored publication" object type schema unit represented by the instance of *ObjectType* whose *name* attribute has the value

“AuthoredPublication” (see Figure 6.10). Additionally, since the class is abstract, the object type is defined by a closed projection whose meaning defines the extension of the “AuthoredPublication” object type as the union of the extension of the “AuthoredBook,” “BookChapter” and “JournalPaper” object types.

General description

The *sbvrEquivalents()* operation of a class named *c* gives an instance of a characterization object of an *ObjectType*. The instance of *NounConceptCh* has the attributes *name* and *isValueType* with the values *c* (the same name as the *Class*) and *false*, respectively.

Additionally, if the value of the *isAbstract* attribute of the *Class* is *True*, the characterization object of *ObjectType* is associated with an instance of *Formulation*. *FormulationCh* is associated with a *projectionVariable* whose *rangedOverConcept* attribute has the value *c*. The formulation is structured such that the concept is the union of the other concepts, as shown in Figure 6.8.

Formal specification in OCL

The *sbvrEquivalents()* operation of *Class* is formally defined in OCL as follows:

```
context Class::sbvrEquivalents():Set(NounConceptCh)
  post NounConceptChCreated:
       let categories = self.generalization -> asSequence ->
              collect(specific).name
       in
       ob.oclIsNew() and ob.oclIsTypeOf(NounConceptCh) and
       ob.name = self.name and ob.isValueType = false and

       -- if the class is abstract the object type includes a
       -- definition structured by a closed projection
       if self.isAbstract then
         v1.oclIsNew() and v1.oclIsTypeOf(Variable2) and
         v1.nounConceptCh = ob and
         v1.rangedOverConcept = self.name and

       -- the projection is structured as a tree of disjunctions
         self^binaryOfAtomicCategorizations(
           BinaryOperationType::Disjunction, ob,v1, categories,
           self.name)
       else
         true
       endif
```

The postcondition includes the invocation of the *binaryOfAtomicOfCategorizations* operation through the '^' OCL operator (Object Management Group 2006b, pàg. 29)[7]. Such operation constrains the structure of a formulation that corresponds to a tree of binary operations where the leaves are atomic formulations of categorizations as showed in Figure 6.10.

[7] The USE tool does not allow the use of the '^' operator. Therefore, the binaryOfAtomicCategorizations() is included in postconditions that invoke it.

```
context Element::binaryOfAtomicOfCategorizations(
         typ:BinaryOperationType, mch:MeaningCh, v1:Variable2,
         cates:Sequence(String), gen:String):Set(MeaningCh)

  post BinaryChCreated:
       -- for each category there is an atomic formulation based
       -- on the categorization fact type of the category
       categs -> forAll(cat:String |
         at1.oclIsNew() and at1.oclIsTypeOf(Atomic) and
         at1.factTypeName = 'is a category of'  and
         at1.type = FactTypeType::Categorization and
         bi1.oclIsNew() and bi1.oclIsTypeOf(Binding) and
         at1.binding->indexOf(bi1) = 1 and bi1.atomic = at1 and
         bi1.variable = v1 and
         bi1.rangesOverConcept = cat and
         bi2.oclIsNew() and bi2.oclIsTypeOf(Binding) and
         at1.binding->indexOf(bi2) = 2 and bi2.atomic = at1 and
         bi2.variable = v1 and
         bi2.rangesOverConcept = gen and

         -- if the category is not the last one then the atomic
         -- formulation is a left leaf of a binary formulation
         if cat <> categs->last() then
           bi.oclIsNew() and bi.oclIsTypeOf(BinaryOperation) and
           bi.type = typ and bi.first = at1 and
           if cat = categs->first() then mch.formulation = bi
           else BinaryOperation.allInstances() ->
                 exists(bio : BinaryOperation|
                    categs->at(categs -> indexOf(cat))-1) =
                    bio.first.oclAsType(Atomic).binding ->
                          first.name and bio.second = bi)
           endif
           -- if the category is the last one, the atomic
           -- formulation is a right leaf of a binary formulation
         else
           BinaryOperation.allInstances() -> exists(bio|
             categs->at(categs->indexOf(cat)-1) =
             bio.first.oclAsType(Atomic).binding -> first().name
             and bio.second = at1)
         endif )
```

6.2.1.3 *sbvrEquivalents() of enumeration*

Example

The DBLP example includes an enumeration schema unit represented by the instance of *Enumeration* whose *name* attribute has the value *"*Gender."

The equivalent knowledge is represented in SBVR by the following schema units (see Figure 6.11):

- The value type schema unit represented by the instance of *ValueType* whose *name* attribute has the value "Gender"; the object type defined by a *ClosedProjection* whose meaning defines that the "Gender" object type is the exclusive disjunction of the "Male" and "Female" individual concepts; and

- Two individual schema units represented by two instances of *IndividualConcept* whose *name* attributes have the valuee "Male" and "Female," respectively.

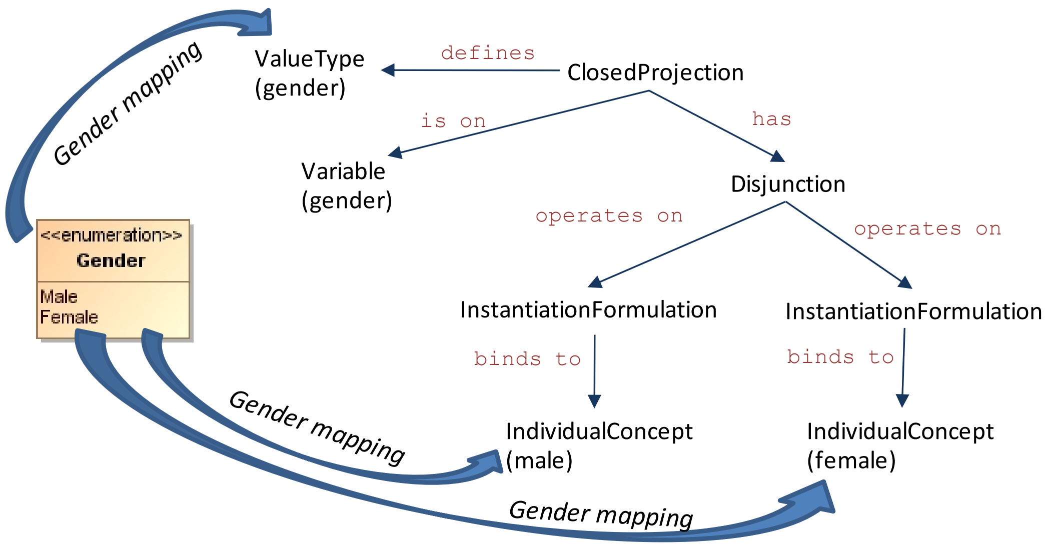


Figure 6.11 Example of mapping the enumeration "Gender" to SBVR

General description

An *Enumeration* named *e* maps to several characterization objects of *Meaning*:

- One characterization object of a *ValueType, NounConceptCh,* with the attribute *name*, the value of which is the same as that of the attribute *name* of the *Enumeration*; and the *valueType* attribute is *True*;
- For each instance of *EnumerationLiteral* of the enumeration, one characterization object of *IndividualConcept* with the attribute *name*, the value of which is the same value as the attribute *name* of the *EnumerationLiteral*; and
- The characterization object of *ValueType* that is associated with an instance of *Formulation*. The *Formulation* is associated with a *projectionVariable* whose *rangedOverConcept* attribute has a value of *e*. The formulation is structured such that the concept is the union of instantiation formulations that binds to the individual concepts mentioned above.

Formal specification in OCL

```
context Enumeration::sbvrEquivalents():Set(MeaningCh)
  post NounConceptChAndIndividualChsCreated:
       let liters = ownedLiteral.name in
       ob.oclIsNew() and ob.oclIsTypeOf(NounConceptCh) and
       ob.name = self.name and ob.isValueType = true and

       v1.oclIsNew() and v1.oclIsTypeOf(Variable2) and
       v1.NounConceptCh = ob and
       v1.rangedOverConcept = self.name and

       -- for each literal there is an individual concept
       liters -> forAll(lit:String |
         ind.oclIsNew() and ind.oclIsTypeOf(IndividualConceptCh)
         and ind.name = lit  and
-- for each literal there is an instantiation
int1.oclIsNew() and int1.oclIsTypeOf(Instantiation) and
int1.concept = v1 and int1.target = lit and

-- if the literal is not the last one then the
-- instantiation formulation is a left leaf of a
-- disjunction
if lit <> liters->last() then
  bi.oclIsNew() and bi.oclIsTypeOf(BinaryOperation) and
  bi.type = BinaryOperationType::Disjunction and
  bi.first = int1 and
  if lit = liters->first then ob.formulation = bi
  else BinaryOperation.allInstances() ->
          exists(bio : BinaryOperation|
                liters->at(liters -> indexOf(lit))-1) =
                bio.first.oclAsType(Instantiation).target
       and bio.second = bi)
  endif

  -- if the literal is the last one, the instantiation
  -- formulation is a right leaf of a disjunction
else
  BinaryOperation.allInstances() -> exists(bio|
    liters->at(liters->indexOf(cat)-1) =
    bio.first.oclAsType(Instantiation).target and
    bio.second = int1)
endif )
```

6.2.1.4 *sbvrEquivalents() of attribute*

Example

In DBLP, the *conferencePaper* attribute schema unit of the *JournalPaper* class is represented by the instance of *Property* whose attribute *name* has the value "conferencePaper," is of Boolean type and has a cardinality of 1.

The equivalent meaning, in SBVR (see Figure 6.12), is represented by:

- A *characteristic* schema unit that describes "journal paper being conference paper."

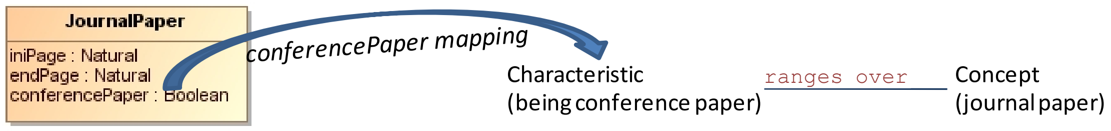


Figure 6.12 Example of mapping the attribute "conferencePaper" to SBVR

Moreover, in DBLP, the *acronym* attribute schema unit of the *ConferenceSeries* class is represented by the instance of *Property* whose attribute *name* has the value "acronym" and has a cardinality of 1.

The equivalent meaning, in SBVR (see Figure 6.13), is represented by:

- An *is-property-of fact type* schema unit that describes that the "conference series has acronym"; and

- A structural rule schema unit represented by an instance of *StructuralRule* whose meaning states that "each conference series has exactly one acronym."

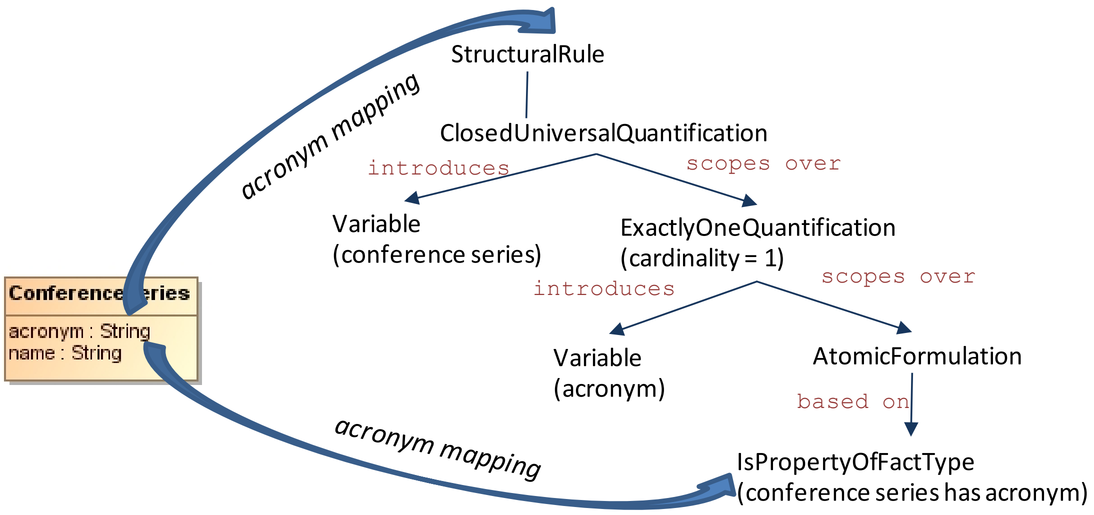


Figure 6.13 Example of mapping the attribute "acronym" to SBVR

General description

A Boolean attribute, represented in UML as a *Property* of *Boolean* type, maps to a characterization object of *Characteristic* (unary fact type). The *verb* attribute of *CharacteristicCh* is the concatenation of the string "being" with the same value as the attribute *name* of *Property*. The *rangesOverConcept* attribute has the same value as the name of the *Class* that owns the *Property*.

A non-Boolean attribute represented by a *Property* related to a class by *ownedAttribute* maps to the following characterization objects of *Meaning*:

- One characterization object of an IsPropertyOfFactType, FactTypeCh, whose type attribute is equal to FactTypeType::IsPropertyOf, whose attribute name has the value "has," and which has two fact type roles. The first role ranges over an object type whose attribute name has the same value as the attribute name of the Class that owns the Property. The second role has the name of the Property and its rangesOverConcept attribute has the same value as the attribute name of the same type as the Property; and
- A characterization object of StructuralRule that corresponds to the meaning of the cardinality constraint (if the multiplicity value of the Property is different from "0..*").

Formal specification in OCL

An additional operation that creates a cardinality structural rule characterization object is defined.

The *CardinalityCh(names, verb, factType, type, minCar, card, maxCar)* operation constrains the structural rule characterization object that corresponds to a cardinality constraint, as shown in Figure 6.13. *StructuralRuleCh* is associated with a *QuantificationForm* of type *ClosedUniversal* that introduces a *Variable2* whose attribute *rangesOverConcept* has the value *name1*. The *QuantificationForm* is associated with a second *QuantificationForm* whose type (e.g., at least, at most, exactly, etc.) has the value *type*. The *QuantificationForm* also introduces a *Variable2* whose attribute *rangesOverConcept* has a value equivalent to the last string of *names.* The meaning of the structural rule is a cardinality constraint between a certain number of *names*, each one of its type *type*, which are related through a fact type named *verb*. The meaning is that, given *n* concepts (strings) in *names*, the first *n-1 names* are related between *minCar* and *maxCar* to the last concept (string) of *names*.

```
context Element::CardinalityCh(
          names:Sequence(String), verb:String,
          factType:FactTypeType, type:QuantificationType,
          minCar:Integer, maxCar:Integer[0..1]):Set(MeaningCh)
  post:   st.oclIsNew() and st.oclIsTypeOf(StructuralRuleCh) and
          fo.oclIsNew()  and  fo.oclIsTypeOf(QuantificationForm)  and
          fo.structuralRuleCh = st and
          fo.type = QuantificationType::ClosedUniversal and
          qu.oclIsNew() and qu.oclIsTypeOf(QuantificationForm) and
          qu.quantificationForm = fo and
          qu.type = typeOfQuantification(minCar, maxCar) and
          qu.card = minCar and qu.minimCard = minCar and
          qu.maxCard = maxCar    and
          at.oclIsNew() and at.oclIsTypeOf(Atomic) and
          at.quantificationForm = qu and at.factTypeName = verb and
          at.type = factType and
          names -> forAll(na|   v1.oclIsNew() and
            v1.oclIsTypeOf(Variable2) and
            if names->last <> na then
              v1.quantification = fo
            else v1.quantification = qu
            endif and v1.rangedOverConcept = na and
            bi1.oclIsNew() and bi1.oclIsTypeOf(Binding) and
            at.binding -> indexOf(bi1) = names -> indexOf(na) and
            bi1.atomic = at and bi1.variable = v1 and
            bi1.rangesOverConcept = na)
```

where *TypeOfQuantification(min, max)*, defined in the context of *Element*, gives the type of the equivalent SBVR quantification depending on the *min* and *max* values:

```
context Element::typeOfQuantification(min:Integer,
          max:Integer[0..1]):QuantificationType
 body: if max -> notEmpty() and min = max and min = 1
       then QuantificationType::ExactlyOne
       else
          if max -> notEmpty() and min = max and min <> 1
          then QuantificationType::ExactlyN
          else
            if min = 1 and max -> isEmpty()
            then QuantificationType::Existential
            else
              if min > 1 and max -> isEmpty()
              then QuantificationType::AtLeastN
              else
                if min = 0 and max -> notEmpty()
                then QuantificationType::AtMostN
                else
                    if min = 0 and max -> notEmpty() and max = 1
                    then QuantificationType::AtMostOne
                    else QuantificationType::NumericRange
                    endif
                endif
            endif
        endif
    endif
endif
```

The specification of the *sbvrEquivalents()* operation is as follows:

```
context Property::sbvrEquivalents():Set(MeaningCh)
  post CharacteristicChOrIsPropertyFactTypeChAndRuleChCreated:
      let name1:String = self.class.name in
      let name2:String = self.name in
      let names:Sequence(String) = Sequence{name1, name2} in
      let verb:String = 'has' in
      let factType:FactTypeType = FactTypeType::IsPropertyOf in

      -- equivalent of a boolean attribute
      if self.type.name = 'Boolean' then
        ch.oclIsNew() and ch.oclIsTypeOf(CharacteristicCh) and
        ch.verb = 'being '.concat(self.name) and
        ch.rangesOverConcept = self.class.name
      else
      -- equivalent of a non-boolean attribute
      -- there is an instance of fact type characterization object
        fa.oclIsNew() and fa.oclIsTypeOf(FactTypeCh) and
        fa.name = verb and fa.type= FactTypeType::IsPropertyOf
        and ro1.oclIsNew() and ro1.oclIsTypeOf(RoleOfFactType)
        and ro1.factTypeCh = fa and ro1.rangesOverConcept = name1
        and fa.roleOfFactType -> indexOf(ro1) = 1 and
        ro2.oclIsNew() and ro2.oclIsTypeOf(RoleOfFactType) and
        ro2.factTypeCh = fa and ro2.name = name2 and
        ro2.rangesOverConcept = self.type.name and
        fa.roleOfFactType -> indexOf(ro2)and

      -- there is an instance of structural rule characterization
      -- object
        (self.lower() <> 0 or
        self.upperValue.oclIsTypeOf(LiteralInteger)) implies
          self^cardinalityCh(names, verb, factType,
          self.lowerValue.oclAsType(LiteralInteger).value,
          self.upperValue.oclAsType(LiteralInteger).value)
      endif
```

6.2.1.5 *sbvrEquivalents() of association*

Example

DBLP includes an association schema unit named “Publishes” that relates the classes named “Person” and “Publication.” The association schema unit is represented by the *Association* whose *name* attribute has the value “Publishes” and whose member ends are the instances of *Class* named “Person” and “Publication,” respectively. Additionally, as a multiplicity element, the association includes the cardinality constraints between "Person" and "Publication."

The equivalent meaning is represented in SBVR with the following schema units (see Figure 6.14):

- An associative fact type schema unit describing that "person publishes publication";
- A structural rule schema unit meaning that "each person publishes at least one publication"; and
- A structural rule schema unit meaning that "each publication has at least one person."

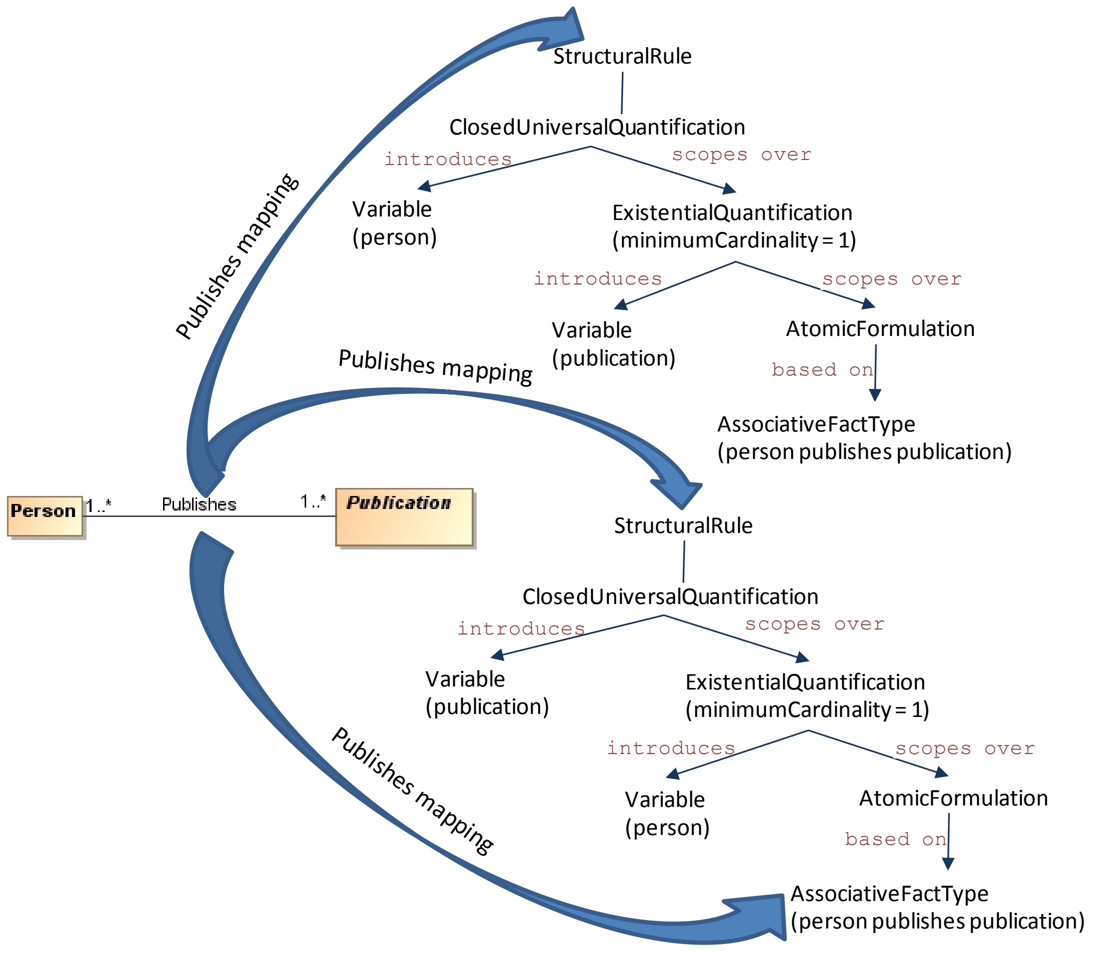


Figure 6.14 Example of mapping the association 'publishes' to SBVR

General description

In UML, an association represented by an instance of *Association* related, by *memberEnd*, to *n* > 1 instances of *Class* maps to the following characterization objects of *Meaning*:

- A characterization object of an AssociativeFactType, FactTypeCh, whose type attribute is equal to Associative (or Partitive if the association corresponds to a composition). The attribute name may have different values: if the association has a name, the value is the name of the association; if the association corresponds to a composition, the value is “includes”; if the association corresponds to an aggregation, the value is “is part of”; and if none of the other cases apply, the value is “has.” The instance of FactTypeCh has n fact type roles, and the value of the rangesOverConcept attribute of each role is the name of the type of a member end of the Association;

- A characterization object of StructuralRule (for each member end whose multiplicity is different than “0..*”). StructuralRuleCh is associated with a QuantificationForm of type ClosedUniversal that introduces n-1 instances of Variable2 whose attribute rangesOverConcept has the same value as the name of the type of one of the opposite members. The QuantificationForm is associated with a second QuantificationForm whose type depends on the multiplicity values and which also introduces a Variable2 whose attribute rangesOverConcept has the same value as the name of the type of the member end that has the multiplicity constraint.

Formal specification in OCL

The *associationType()* operation, defined in the context of *Association*, returns whether the association is a composition, an aggregation or neither:

```
context Association::associationType():AggregationKind
  body:  if self.memberEnd -> exists(pr|
            pr.aggregation = AggregationKind::composite)
         then
            AggregationKind::composite
         else
            if self.memberEnd -> exists(pr|
               pr.aggregation = AggregationKind::shared)
            then AggregationKind::shared
            else AggregationKind::none
            endif
         endif
```

The *associationName()* operation, defined in the context of *Association*, returns the *name* that is given to the relationship. The *name* may have different values: if the association has a name, the value is the name of the association; if the association corresponds to a composition, the value is “includes”; if the association corresponds to an aggregation, the value is “is part of”; and if none of the other cases apply, the value is “has.”

```
context Association::associationName():String
  body:  if self.memberEnd -> exists(pr|
               pr.aggregation = AggregationKind::composite)
         then if self.name -> notEmpty() then 'includes'
               else self.name
               endif and
         else if self.memberEnd -> exists(pr|
                  pr.aggregation = AggregationKind::shared)
               then 'is part of'
            else if self.name -> notEmpty() then self.name
                  else 'has'
                  endif
            endif
         endif
```

The *sbvrEquivalents()* operation of *Association* is formally defined as follows:

```
context Association::sbvrEquivalents():Set(MeaningCh)
  post AssociativeFactTypeChAndStructuralRulesChCreated:

       -- there is a fact type characterization object
       fa.oclIsNew() and fa.oclIsTypeOf(FactTypeCh) and
       fa.name = associationName() and fa.type = typeOfFactType()
       and self.memberEnd -> forAll(me | ro.oclIsNew() and
       ro.oclIsTypeOf(RoleOfFactType) and ro.factTypeCh = fa and
       ro.name = me.name and ro.rangesOverConcept = me.type.name
       and self.memberEnd->indexOf(me) = fa.roleOfFactType->
         indexOf(ro)) and

       -- for each cardinality constraint there is a structural
       -- rule characterization object
       self.memberEnd -> forAll(me|
       (me.lower() <> 0 or
         not me.upperValue.oclIsTypeOf(LiteralUnlimitedValue))
       implies
         self^cardinalityCh(
           self.memberEnd -> excludes(me)->union(me)->
             collect(name),self.factTypeName(),
           if associationType() = AggregationKind::composite
           then FactTypeType::Partitive
           else FactTypeType::Associative
           endif,
           me.lower(),me.upperValue.oclAsType(LiteralInteger))
```

Note that the postcondition includes the invocation to the *cardinalityCh*() operation.

6.2.1.6 *sbvrEquivalents() of association class*

Example

The DBLP example includes an association class schema unit represented by an instance of *AssociationClass* whose *name* attribute has the value “Editorship” and relates “EditedBook” with “editor.”

The equivalent meaning is represented, in SBVR, by the following schema units (see Figure 6.15):

- An object type schema unit represented by an instance of ObjectType whose name attribute has the value “editorship”; the object type is defined by an instance of ClosedProjection whose meaning defines that “editorship” is an “actuality that an editor has an edited book”;
- An associative fact type schema unit represented by an instance of AssociativeFactType describing that “editor has edited book.” The associative fact type includes the fact type role named “editor” that ranges over the object type “person”; and

- A structural rule schema unit represented by an instance of StructuralRule expressing the meaning that “each edited book has at least one editor.”

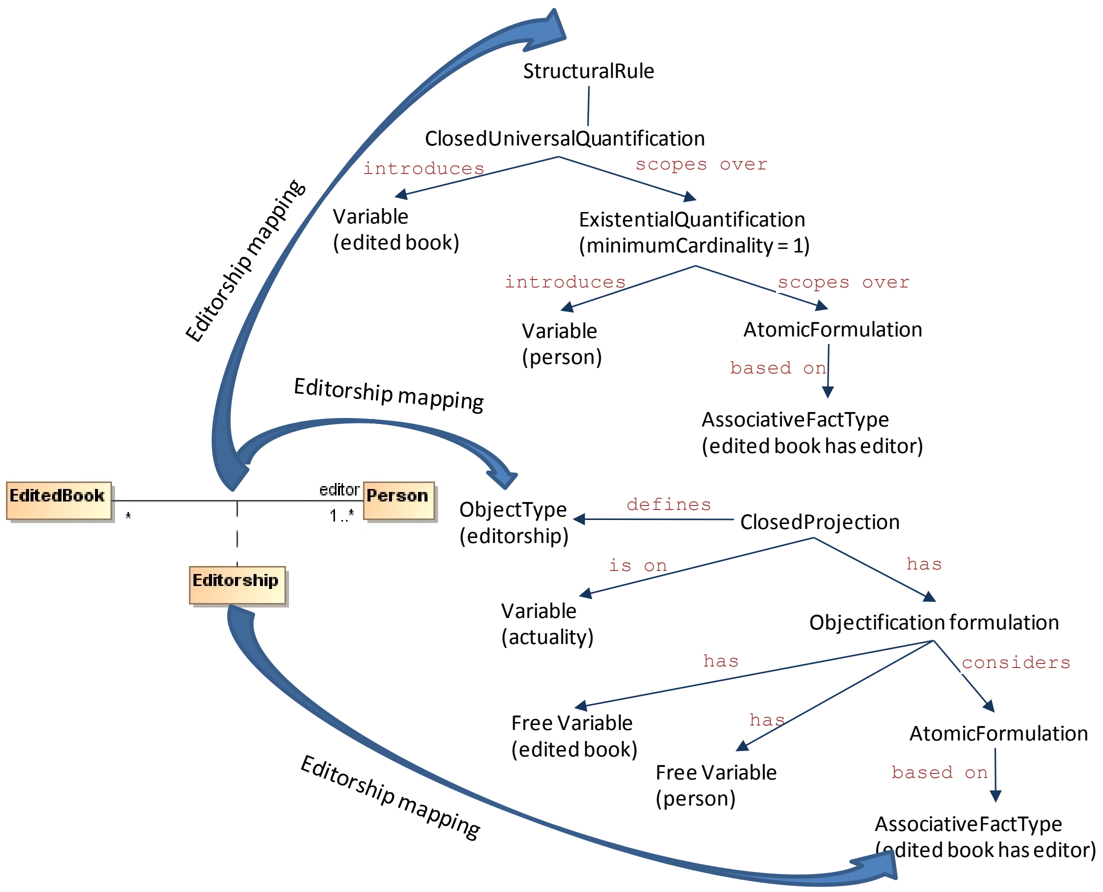


Figure 6.15. Example of mapping the association class "Editorship"

General description

An association class represented by an instance of *AssociationClass* maps to the equivalents of class defined in Section 6.2.1.2 and to the equivalents of association defined in Section 6.2.1.5. Additionally, the object type characterization object is associated with a *formulation*. The *formulation* is related to a variable whose *rangedOverConcept* attribute has the value “actuality.” It is also related to two additional variables whose *rangedOverConcept* has the value of the *name* of the member ends of the association class. The *formulation* is associated with an *ObjectificationForm* whose target is the first variable defined, and with an instance of *Atomic* for the associative fact type resulting from the association class.

Formal specification in OCL

```
context AssociationClass::sbvrEquivalents():Set(MeaningCh)
  post AdditionalDefinitionFormulationCreated:
       -- the definition introduces a variable that ranges over
       -- 'actuality'
       v1.oclIsNew() and v1.oclIsTypeOf(Variable2) and
       v1.nounConceptCh.name = self.name and
       v1.rangedOverConcept = 'actuality' and
       -- it is structured by an objectificationForm
```

```
ob.oclIsNew() and ob.oclIsTypeOf(ObjectificationForm) and
ob.nounConceptCh.name = self.name and ob.target = v1 and

-- there is an atomic formulation based on the fact type of
-- the association
at.oclIsNew() and at.oclIsTypeOf(Atomic) and
ob.formulation = at and
at.factTypeName = self.associationName() and
at.type = typeOfFactType() and
self.memberEnd -> forAll(me|
  -- for each member end there is an free variable
  v2.oclIsNew() and v2.oclIsTypeOf(Variable2) and
  if me.name->notEmpty() then
       v2.rangedOverConcept = me.name
  else v2.rangedOverConcept = me.type.name
  endif and
  v2.formulation = ob and
  -- and there is a binding on the atomic formulation
  bi1.oclIsNew() and bi1.oclIsTypeOf(Binding) and
  at.binding -> indexOf(bi1) = self.memberEnd ->
  indexOf(me) and bi1.atomic = at and bi1.variable = v2)
```

6.2.1.7 *sbvrEquivalents() of generalization*

Example

The DBLP example includes a generalization schema unit represented by the instance of *Generalization* that relates the general class named "Book" and the specific class named "EditedBook."

The same meaning is represented, in SBVR, as a categorization fact type schema unit describing that "edited book is a category of book," as shown in Figure 6.16.

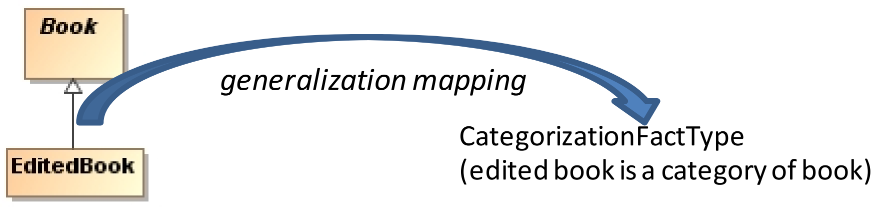


Figure 6.16. Example of mapping the generalization relationship between "Book" and "EditedBook"

General description

A generalization, represented by an instance of *Generalization* related to two instances of *Class* by *specific* and *general* associations, maps to a characterization object of *FactType*: the *FactTypeCh* whose *type* attribute is equal to *FactTypeType::Categorization* and whose *name* attribute has a value equal to "is a category of." The *FactTypeCh* is associated with two instances of *RoleOfFactType*. The first one has no name and the *rangesOverConcept* attribute has the same value as the name of the specific *Class* of the *Generalization*. The second one has no name and the *rangesOverConcept* attribute has the same value as the name of the general *Class* of the *Generalization*.

Formal specification in OCL

```
    context Generalization::sbvrEquivalents():Set(FactTypeCh)
      post FactTypeCh:
           fa.oclIsNew() and fa.oclIsTypeOf(FactTypeCh) and
           fa.name = 'is a category of' and
           fa.type= FactTypeType::Categorization and
           ro1.oclIsNew() and ro1.oclIsTypeOf(RoleOfFactType) and
           ro1.factTypeCh = fa and
           ro1.rangesOverConcept = self.specific.name and
           fa.roleOfFactType -> indexOf(ro1) = 1 and ro2.oclIsNew() and
           ro2.oclIsTypeOf(RoleOfFactType) and
           ro2.factTypeCh = fa and ro2.name = self.name and
           ro2.rangesOverConcept = self.general.name and
           ro2.roleOfFactType -> indexOf(ro2) = 2
```

6.2.1.8 *sbvrEquivalents() of generalization set*

Example

DBLP includes a generalization set schema unit represented by the instance of *GeneralizationSet* named “typeOfBook.”

In SBVR, the equivalent meaning is represented by the following schema units (see Figure 6.17):

- A segmentation schema unit represented by an instance of Segmentation whose name attribute has the value “type of book.” The segmentation is related to the “book” object type as a general concept and to the “edited book” and “authored book” as categories.
- A structural rule schema unit meaning that “each book is an edited book or is an authored book but not both.”

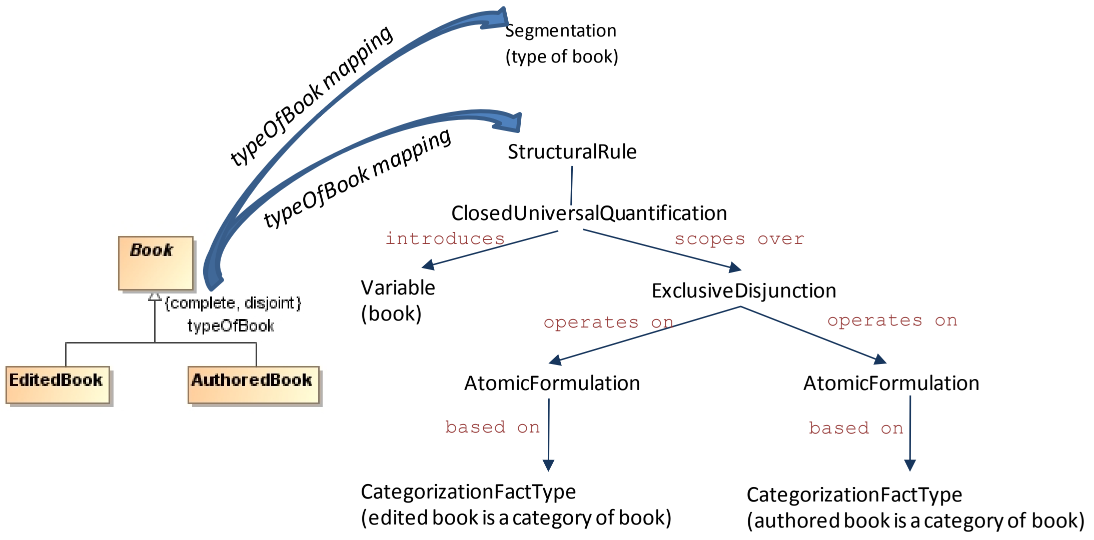


Figure 6.17. Example of mapping the "typeOfBook" generalization set

Note that the example is a partition of a concept into two other concepts. The closed universal quantification scopes over an exclusive disjunction. However, the DBLP example also includes a generalization set schema unit represented by the instance of *GeneralizationSet* named “typeOfAuthoredPublication.”

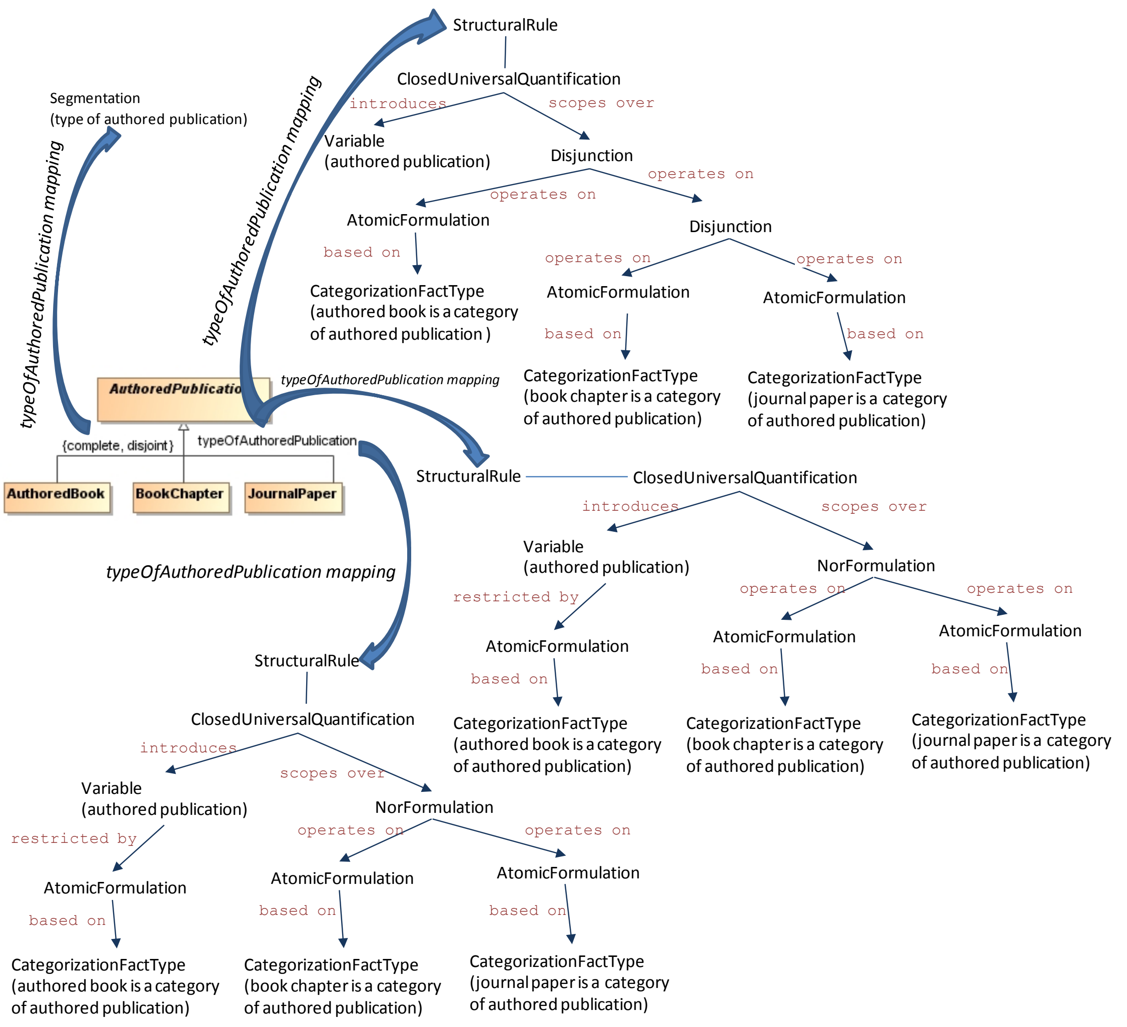

StructuralRule
ClosedUniversalQuantification
introduces
scopes over
Variable (authored publication)
Disjunction
operates on
AtomicFormulation
based on
CategorizationFactType (authored book is a category of authored publication)
Disjunction
CategorizationFactType (book chapter is a category of authored publication)
CategorizationFactType (journal paper is a category of authored publication)
Segmentation (type of authored publication)
typeOfAuthoredPublication mapping
AuthoredPublication
{complete, disjoint}
typeOfAuthoredPublication
AuthoredBook
BookChapter
JournalPaper
NorFormulation
restricted by

Figure 6.18. Example of mapping the "typeOfAuthoredPublication" generalization set

In SBVR, the equivalent meaning is represented by the following schema units (see Figure 6.18):

- A segmentation schema unit represented by an instance of Segmentation whose name attribute has the value “type of publication.” The segmentation is related to the “authored publication” object type as a general concept and to “authored book,” “book chapter” and “journal paper” as categories.
- A structural rule schema unit whose meaning is the covering constraint of the generalization set—that is, “each authored publication is an authored book or a book chapter or a journal paper.”
- Two structural rule schema units whose meanings are the disjointness constraints of the generalization set—that is, “each authored book that is an authored publication is neither a book chapter nor a journal paper” and “each book chapter that is an authored publication is neither an authored book nor a journal

paper."Note that if the number of generalizations that form a generalization set is greater than two, it is not possible to represent, in SBVR, the disjointness and covering constraints with only one structural rule.

If the number of generalizations that form a generalization set is greater than two, it is not possible to represent, in SBVR, the disjointness and covering constraints with only one structural rule.

General description

A generalization set, represented by an instance of *GeneralizationSet* having a partition of *n* instances of *Generalization*, maps to the following characterization object of *Meaning*:

- A characterization object of CategorizationScheme. The CategorizationSchemeCh has the attribute name with the same value as the attribute name of the GeneralizationSet. Its generalConcept attribute has the same value as the name of the general class of any of the generalizations of the generalization set. Its category attribute has the same value as the set of names of the different specific classes of the generalizations of the generalization set. Its isSegmentation attribute is true if the generalization set is covering and disjoint.
- If the generalization set is covering and disjoint and has only two generalizations, a characterization object of StructuralRule that represents both properties. The StructuralRuleCh is associated with a QuantificationForm whose attribute is ClosedUniversal and which introduces a variable whose rangedOverConcept has the same value as the name of the general concept of any generalization of the generalization set. The quantificationForm is associated with a BinaryOperation whose type attribute is ExclusiveDisjunction. The BinaryOperation is associated with two instances of Atomic, one for each generalization.
- If the generalization set is covering and disjoint and has more than two generalizations, there are n-1 different characterization objects of StructuralRule. One StructuralRuleCh represents the isCovering property of the generalization set, and the others represent the disjointness property of the generalization set.

Formal specification in OCL

For a better structuring of the *sbvrEquivalents()* operation, some additional operations are created in the context of Class.

The *coveringAndDisjointCh(gen, subs)* operation constrains the structure of a structural rule characterization object whose meaning is a disjointness and covering constraint of the partition of the concept named *gen* and which has the concepts named in the sequence *subs*.

```
context Class::CoveringAndDisjointCh(
        gen:String, subs:Sequence(String)):Set(MeaningCh)
  post: st.oclIsNew() and st.oclIsTypeOf(StructuralRuleCh) and
        fo.oclIsNew() and fo.oclIsTypeOf(QuantificationForm) and
        fo.structuralRuleCh = st and
        fo.type = QuantificationType::ClosedUniversal and
        v1.oclIsNew() and v1.oclIsTypeOf(Variable2) and
```

```
          v1.quantification = fo and
          v1.rangedOverConcept = gen and
          bi.oclIsNew() and bi.oclIsTypeOf(BinaryOperation) and
          bi.type = BinaryOperationType::ExclusiveDisjunction and
          bi.quantificationForm = fo and
          subs -> forAll(su|
            at.oclIsNew() and at.oclIsTypeOf(Atomic) and
            if su = subs -> first() then bi.first = at
            else bi.second = at
            endif and
            at.factTypeName = 'is a category of' and
            at.type = FactTypeType::Categorization and
            bi1.oclIsNew() and bi1.oclIsTypeOf(Binding) and
            at.binding -> first() = bi1 and bi1.variable = v1 and
            bi1.rangesOverConcept = su and
            bi2.oclIsNew() and bi2.oclIsTypeOf(Binding) and
            at.binding -> last() = bi1 and bi2.variable = v1
            and bi2.rangesOverConcept = gen)
```

The *CoveringCh(gen, subs)* operation constrains the structure of a structural rule characterization object. The meaning of the structural rule is a covering constraint of the partition of *gen* that has the concepts named in the sequence *subs*. Note that the structural rule has the same structure as a projection that defines an abstract class, as described in Section 6.2.1.2.

```
context Element::coveringCh(gen:String, subs:Sequence(String):
            Set(MeaningCh)
  post:   st.oclIsNew() and st.oclIsTypeOf(StructuralRuleCh) and
          fo.oclIsNew() and fo.oclIsTypeOf(QuantificationForm) and
          fo.structuralRuleCh = st and
          fo.type = QuantificationType::ClosedUniversal and
          v1.oclIsNew() and v1.oclIsTypeOf(Variable2) and
          v1.quantification = fo and
          v1.rangedOverConcept = gen and

          -- the structural rule is structured as a tree of
          -- disjunctions as described in Section 6.2.1.2
          self.binaryOfAtomicCategorizations(
            BinaryOperationType::Disjunction, st, v1, subs, gen)
```

Given a class with *n* subclasses, the *disjointnessCh(gen, subs)* operation constrains *n-1* structural rule characterization objects. The meaning of the structural rules, taken together, is a covering constraint of the partition of *gen* that has *subs* subclasses.

```
context Element::disjointessCh(gen:String, subs:Sequence(String)):
            Set(MeaningCh)
  post:   subs -> forAll(su | su <> subs->last and
          st.oclIsNew() and st.oclIsTypeOf(StructuralRuleCh) and
          fo.oclIsNew() and fo.oclIsTypeOf(QuantificationForm) and
          fo.structuralRuleCh = st and

          -- the structural rule is structured by a closed
          -- universal quantification and introduces a variable
          -- that ranges over the generic concept
          fo.type = QuantificationType::ClosedUniversal and
          v2.oclIsNew() and v2.oclIsTypeOf(Variable2) and
          v2.quantification = fo and v2.rangedOverConcept = gen
          and

          -- the variable is restricted by an atomic formulation
          at.oclIsNew() and at.oclIsTypeOf(Atomic) and
          v2.restricting = at and
          at.factTypeName = 'is a category of' and
          at.type = FactTypeType::Categorization and
          bi1.oclIsNew() and bi1.oclIsTypeOf(Binding) and
          bi1.atomic = at and bi1.variable = v2 and
          bi1.rangesOverConcept = su and
          bi2.oclIsNew() and bi2.oclIsTypeOf(Binding) and
          bi2.atomic = at and bi2.variable = v2 and
          bi2.rangesOverConcept = gen and

          -- the quantification is structured as a tree of
          -- norFormulations
          self.binaryOfAtomicCategorizations(
            BinaryOperationType::NorFormulation, st,v2, subs,
            gen))
```

The *sbvrEquivalents()* operation of *GeneralizationSet* is specified as follows:

```
    context GeneralizationSet::sbvrEquivalents():Set(MeaningCh)
      post CategorizationSchemeAndStructuralRulesChCreated:
           let generalizations = self.generalization->asSequence in
           let gen = self.generalization-> collect(general.name)->
                     any(true) in
           let subs = generalizations -> collect(specific.name) in

           cs.oclIsNew() and cs.oclIsTypeOf(CategorizationScheme) and
           cs.name = self.name and cs.generalConcept = gen and
           cs.category = subs and
           if self.isDisjoint and self.isCovering then
             cs.isSegmentation  and
             self.generalization->size()= 2 implies
               self^coveringAndDisjointCh(gen, subs)
           else
             self.isDijoint implies self^disjointessCh(gen, subs) and
             self.isCovering implies self^coveringCh(gen, subs)
           endif
```

6.2.1.9 *sbvrEquivalents() of constraint*

Examples

The DBLP example includes a constraint represented by an instance of *Constraint* named "XOR."

The knowledge meant by this constraint is represented, in SBVR, by three structural rule schema units, as shown in Figure 6.19:

- One represented by an instance of StructuralRule whose meaning is that "each conference edition that is published in a book series issue is published neither in an edited book nor in a journal issue";

- A second one represented by an instance of StructuralRule whose meaning is that "each conference edition that is published in an edited book is published neither in a book series issue nor in a journal issue"; and
- A third one represented by an instance of StructuralRule whose meaning is that "each conference edition that is published in a journal issue is published neither in an edited book nor in a book series issue."

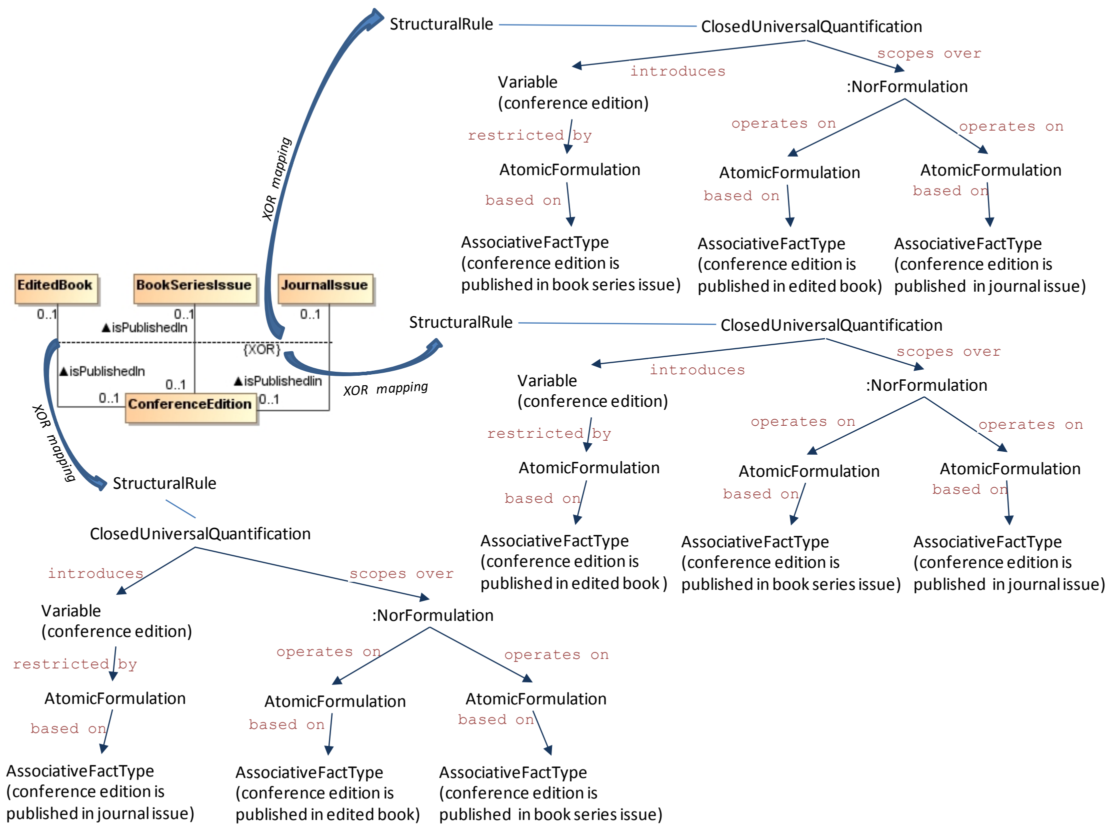

StructuralRule
ClosedUniversalQuantification
introduces
scopes over
Variable (conference edition)
:NorFormulation
restricted by
operates on
operates on
AtomicFormulation
AtomicFormulation
AtomicFormulation
based on
based on
based on
AssociativeFactType (conference edition is published in book series issue)
AssociativeFactType (conference edition is published in edited book)
AssociativeFactType (conference edition is published in journal issue)
XOR mapping
EditedBook
BookSeriesIssue
JournalIssue
ConferenceEdition
isPublishedIn
0..1
{XOR}
StructuralRule
ClosedUniversalQuantification
introduces
scopes over
Variable (conference edition)
:NorFormulation
AssociativeFactType (conference edition is published in edited book)
AssociativeFactType (conference edition is published in book series issue)
AssociativeFactType (conference edition is published in journal issue)
XOR mapping
StructuralRule
ClosedUniversalQuantification
AssociativeFactType (conference edition is published in journal issue)
AssociativeFactType (conference edition is published in edited book)
AssociativeFactType (conference edition is published in book series issue)

Figure 6.19. Example of mapping a "XOR" constraint

The DBLP example also includes a constraint schema unit represented by an instance of *Constraint* whose invariant body states that the *name* attribute is unique for instances of *Person*.

The equivalent knowledge is represented, in SBVR, as a reference scheme schema unit represented by an instance of *ReferenceScheme* (see Figure 6.20). The *ReferenceScheme* has, as a referenced concept, the object type named "Person" and, as the used role of fact type, the fact type role of the is-property-of fact type meaning "person has name."

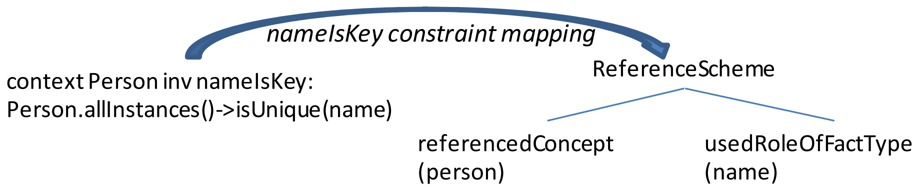

nameIsKey constraint mapping
context Person inv nameIsKey:
Person.allInstances()->isUnique(name)
ReferenceScheme
referencedConcept (person)
usedRoleOfFactType (name)

Figure 6.20. Example of mapping the "nameIsKey" constraint

General description

A constraint, represented by an instance of *Constraint*, depending on the type of constraint, maps to different characterization objects of *Meaning.* As stated above, this thesis considers, for the purposes of mapping to SBVR, two types of constraints: constraints expressing that the values of an *attribute* of a *Class* are unique, and predefined *XOR* constraints.

In the first case, the constraint is mapped to a reference scheme characterization object, *ReferenceSchemeCh*. The *referencedConcept* attribute of *ReferenceSchemeCh* has the same value as the *name* attribute of the context of the constraint. The *ReferenceSchemeCh* is associated with a *UsedRoleOfFactType* whose *rangesOverConcept* has the same value as the name of the constrained element (i.e., the property that is unique within the instances of *Class)*.

An *XOR* constraint between *n* associations, where n>1, is mapped to *n* structural rule characterization objects, *StructuralRuleCh*. Each *StructuralRuleCh* is associated with a *QuantificationForm* of type *ClosedUniversal* that introduces a variable whose *rangedOverConcept* attribute has the same value as the name of the context of the constraint. The *QuantificationForm* has a second *Variable2* (*freeVariable*) that restricts an *Atomic* for one of the associative fact types. The *QuantificationForm* scopes over a *BinaryOperation* whose type is *NorFormulation*. The *NorFormulation* has two operands, each of which is an *Atomic* for one of the other two associative fact types.

Formal specification in OCL

Given an *XOR* constraining *n* associations, the *XORStructuralRuleCh(base:String, roles:Sequence(TupleType{verb,end}))* operation constrains *n* structural rule characterization objects. The meaning of each structural rule is equivalent to "each *base* that *verb*$_1$ *end*$_1$ neither *verb*$_2$ *end*$_2$ nor *verb*$_3$ *end*$_3$*...nor verb*$_n$ *end*$_n$," where each tuple of *verb*$_j$ and *end*$_j$ is one of the tuples included in the *roles* sequence. Note that there are *n* rules, each of which is restricted by an atomic formulation based on one different associative fact type, as shown in Figure 6.19. The meaning of the structural rules, taken together, is the *XOR* constraint.

```
context Element::xorStructuralRuleCh(base:String,
          roles:Sequence(Tuple{verb,end})):
          Set(MeaningCh)
  post:   roles -> forAll(ro |
            st.oclIsNew() and st.oclIsTypeOf(StructuralRuleCh) and
            fo.oclIsNew() and fo.oclIsTypeOf(QuantificationForm)
            and  fo.structuralRuleCh = st and

            -- the structural rule is structured by a closed
            -- universal quantification and introduces a variable
            -- that ranges over the base concept
            fo.type = QuantificationType::ClosedUniversal and
            v2.oclIsNew() and v2.oclIsTypeOf(Variable2) and
            v2.quantification = fo and v2.rangedOverConcept = base
            and
```

```
                -- the variable is restricted by an atomic formulation
                at.oclIsNew() and at.oclIsTypeOf(Atomic) and
                v2.restricting = at and at.factTypeName = ro.verb and
                at.type = FactTypeType::AssociativeFactType and
                bi1.oclIsNew() and bi1.oclIsTypeOf(Binding) and
                bi1.atomic = at and bi1.variable = v2 and
                bi1.rangesOverConcept = base and
                bi2.oclIsNew() and bi2.oclIsTypeOf(Binding) and
                bi2.atomic = at and bi2.variable = v2 and
                bi2.rangesOverConcept = ro.end and

                -- the quantification is structured as a tree of
                -- norFormulations
                self^binaryOfAtomicOfAssociativeFactTypes(
                  BinaryOperationType::NorFormulation, st,v1,
                    roles->excluding(ro), base))
```

The *binaryOfAtomicOfAssociativeFactTypes()* operation constrains the structure of a formulation that corresponds to the tree of binary operations where the leaves are atomic formulations of associative fact types, as shown in Figure 6.19.

```
context Element::binaryOfAtomicOfAssociativeFactTypes(
         typ:BinaryOperationType, mch:MeaningCh, v1:Variable2,
         roles:Sequence(TupleType(verb:String,end:String)),
         base:String):Set(MeaningCh)

  post BinaryChCreated:
 -- for each tuple there is an atomic formulation based
 -- on an associative fact type where the base is a role,
 -- and the verb and the other role are one of the roles
       roles -> forAll(rol:String |
         at1.oclIsNew() and at1.oclIsTypeOf(Atomic) and
         at1.factTypeName = rol.verb  and
         at1.type = FactTypeType::AssociativeFacType and
         bi1.oclIsNew() and bi1.oclIsTypeOf(Binding) and
         bi1.order = 1 and bi1.atomic = at1 and
         bi1.variable = v1 and
         bi1.rangesOverConcept = base and
         bi2.oclIsNew() and bi2.oclIsTypeOf(Binding) and
         bi2.order = 2 and bi2.atomic = at1 and
         bi2.variable = v1 and
         bi2.rangesOverConcept = rol.end and

         -- if the role is not the last one then the atomic
         -- formulation is a left leaf of a binary formulation
         if rol <> roles->last then
           bi.oclIsNew() and bi.oclIsTypeOf(BinaryOperation) and
           bi.type = typ and bi.first = at1 and
           if rol = roles->first then mch.formulation = bi
           else
             BinaryOperation.allInstances() ->
               exists(bio : BinaryOperation|
                  roles->at(roles -> indexOf(rol))-1) =
                        bio.first.oclAsType(Atomic).binding ->
                  last.name and bio.second = bi)
           endif
```

```
          -- if the rol is the last one, the atomic
          -- formulation is a right leaf of a binary formulation
          else
            BinaryOperation.allInstances() -> exists(bio|
              roles->at(roles->indexOf(rol)-1) =
              bio.first.oclAsType(Atomic).binding -> last.name
              and bio.second = at1)
          endif
```

The formal specification of the *sbvrEquivalents()* operation of *Constraint* is as follows:

```
context Constraint::sbvrEquivalents():Set(MeaningCh)
  post MeaningsChCreated:
      let IsUniqueConstraint = self.constrainedElement -> size()=1
          and self.specification.oclAsType(Expression).symbol =
          self.context_.name.concat(' .allInstances() ->
            isUnique(').concat(self.constrainedElement->first().
            oclAsType(Property).name).concat(')') in
      let XORConstraint = self.valueSpecification.
          oclAsType(Expression).symbol = 'XOR' in
      let XORAssociations =
          self.constrainedElement.oclAsType(Association) in
      let base = self.context.name in
      let roles = XORAssociations -> collect( ass |
          Tuple{ass.name,ass.memberEnd-> any(me| me.type.name <>
          base).type.name)
      in

      IsUniqueConstraint implies
        (re.oclIsNew() and re.oclIsTypeOf(ReferenceSchemeCh) and
        re.referencedConcept -> includes(self.context.name) and
        self.constrainedElement -> forAll(cel|
          us.oclIsNew() and us.oclIsTypeOf(UsedRoleOfFactType)
          and us.referenceSchemeCh = re and
          us.rangesOverConcept = cel.oclAsType(Property).name)
          and
      XORConstraint implies
        self^xorStructuralRuleCh(base, roles)
```

6.2.2 SBVR meanings side

The evaluation of the *umlEquivalents()* operation on an SBVR schema unit whose mapping kind is *HasEquivalents* gives the set of UML characterization objects that are equivalent to the SBVR schema unit. The signature and precondition of the operation in OCL is as follows:

```
context Meaning::umlEquivalents():Set(ElementCh)
  pre: umlMappingKind() = MappingKind::HasEquivalents
  post definingTheResult:
      result = ElementCh.allInstances() -
               ElementCh.allInstances@pre()
```

The effect of the operation is not redefined in the subtypes of *Meaning* because all of the instances of the subtypes that represent schema units have the mapping kind *isIncluded*.

6.3 *includedInUml()* operations

This section describes the *includedInUml()* and *includedInSBVR* operations in the context of *Meaning* and *Element*, respectively.

6.3.1 UML side

The evaluation of the *includedInSbvr()* operation, in the context of *Element,* on a UML schema unit whose mapping kind is *IsIncluded* gives an SBVR characterization object. The signature and precondition of the operation in OCL are as follows:

```
context Element::includedInSbvr():MeaningCh
  pre: sbvrMappingKind() = MappingKind::IsIncluded
  post definingTheResult:
         result = (MeaningCh.allInstances() –
              MeaningCh.allInstances@pre()) -> any(True)
```

The effect of the operation is defined in the subtypes of *Element* such that some or all of their instances represent schema units whose mapping kind is *IsIncluded*.

The effect of the operation is not redefined in the subtypes of *Element* because all of the instances of the subtypes that represent schema units have the mapping kind *hasEquivalents*.

6.3.2 SBVR side

The evaluation of the *includedInUml()* operation, in the context of the *Meaning* operation on an SBVR schema unit whose mapping kind is *IsIncluded,* gives a UML characterization object. The signature and precondition of the operation in OCL are as follows:

```
context Meaning::includedInUml():ElementCh
  pre: umlMappingKind() = MappingKind::IsIncluded
  post definingTheResult:
         result = (ElementCh.allInstances() –
              ElementCh.allInstances@pre()) -> any(True)
```

The effect of the operation is defined in the subtypes of *Meaning* such that some or all of their instances represent schema units whose mapping kind is *IsIncluded*.

In order to facilitate working with the cardinalities of the different subtypes of *Quantification*, two additional operators are defined in *Quantification*: *lowerValue()* and *upperValue()*. They return the minimum cardinality and maximum cardinality of each type of quantification, respectively. Their specifications are defined as abstract and they are redefined in the subtypes of *Quantification* as follows:

```
context UniversalQuantification::lowerValue():Integer
  body:  0

context UniversalQuantification::upperValue():UnlimitedNatural
  body:  UnlimitedNatural

context AtLeastNQuantification::lowerValue():Integer
  body:  self.minimumCardinality.value

context AtLeastNQuantification::upperValue():UnlimitedNatural
```

```
  body:  UnlimitedNatural

context NumericRangeQuantification::lowerValue():Integer
  body:  self.minimumCardinality.value

context NumericRangeQuantification::upperValue():Integer
  body:  self.maximumQuantification.value

context AtMostNQuantification::lowerValue():Integer
  body:  0

context AtMostNQuantification::upperValue():Integer
  body:  self.maximumCardinality.value

context ExactlyNQuantification::lowerValue():Integer
  body:  self.cardinality.value

context ExactlyNQuantification::upperValue():Integer
  body:  self.cardinality.value
```

Moreover, given a structural rule whose meaning is a multiplicity constraint, the operation *quantificationType()* returns the subtype of *Quantification* that the universal quantification of the structural rule scopes over.

```
context StructuralRule::quantificationType():Quantification
  body:  self.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         scopeFormulation.oclAsType(Quantification)
```

6.3.2.1 *includedInUml() of value type*

General description

A *ValueType* maps to a characterization object of a *DataType* whose attribute *name* has the same value as the attribute *name* of the *ValueType*. It maps to an *Enumeration* if the value type is defined as the union of individual concepts.

Formal specification in OCL

```
context ValueType::includedInUml():MeaningCh
  post DataTypeOrEnumerationChCreated:
      dt.oclIsNew() and
      if not ClosedProjection.allInstances() -> exists(cp|
          cp.nounConcept = self and cp.isEnumeration())
      then
        dt.oclIsTypeOf(DataTypeCh) and dt.name = self.name and
        if self.name = 'String' or self.name = 'Integer' then
          dt.isPrimitiveType = true
        else dt.isPrimitiveType = false
        endif
      else
        dt.oclIsTypeOf(EnumerationCh) and dt.name = self.name and
        ClosedProjection.allInstances() -> any(cp |
          cp.nounConcept = self and cp.isEnumeration()).
          logicalFormulation.sequenceOfLiterals() -> forAll(lit|
          li.oclIsNew() and li.oclIsTypeOf(Literal) and
          li.enumerationCh = dt and li.name = lit)
      endif
```

Given a logical formulation that structures a closed projection as a tree of disjunctions of instantiation formulations, as shown in Figure 6.9, the *sequenceOfLiterals()* operation returns the sequence of individual concepts bound in the instantiation formulations of the leaves of the tree (i.e., the names of the literals).

```
context LogicalFormulation::sequenceOfLiterals:Sequence(String)
  body: self.oclAsType(Disjunction).logicalOperand1.
         oclAsType(InstantiationFormulation).bindableTarget.name->
         union(
         if self.oclAsType(Disjunction).logicalOperand2.
           oclIsTypeOf(InstantiationFormulation)
         then
           self.oclAsType(Disjunction).logicalOperand2.
           oclAsType(InstantiationFormulation).bindableTarget.name
         else
           self.oclAsType(Disjunction).logicalOperand2.
             sequenceOfLiterals()
         endif
```

6.3.2.2 *includedInUml() of object type*

General description

An *ObjectType* maps to a characterization object of a *Class* or of an *AssociativeClass* with the attribute *name* that has the same value as the attribute *name* of the *ObjectType*. It maps to an *AssociativeClass* if the object type is defined as the objectification of an associative fact type. In this case, the elements of the *AssociativeClassCh* created are the union of the elements in the *Class* and the elements created in the *includedInUml()* of an associative fact type (see Section 6.3.2.5, below).

Formal specification in OCL

```
context ObjectType::includedInUml():MeaningCh
  post ClassOrAssociationClassChCreated:
       let str1:StructuralRule = StructuralRule.allInstances() ->
           any(st| st.isMultiplicity() and
           st.closedLogicalFormulation.
           oclAsType(ClosedUniversalQuantification).
           introducedVariable.rangedOverConcept =
           self.closedProjection.logicalFormulation.
           oclAsType(Objectification).
           consideredLogicalFormulation.
           oclAsType(AtomicFormulation).factType.factTypeRole ->
            last().nounConcept) in

       let str2:StructuralRule = StructuralRule.allInstances() ->
           any(st| st.isMultiplicity() and
           st.closedLogicalFormulation.
           oclAsType(ClosedUniversalQuantification).
           introducedVariable.rangedOverConcept =
           self.closedProjection.logicalFormulation.
           oclAsType(Objectification).
           consideredLogicalFormulation.
           oclAsType(AtomicFormulation).factType.factTypeRole ->
           first().nounConcept) in
```

```
let asft:AssociativeFactType = self.closedProjection.
     logicalFormulation.oclAsType(Objectification).
     consideredLogicalFormulation.
     oclAsType(AtomicFormulation).factType.
     oclAsType(AssociativeFactType) in

if ClosedProjection.allInstances() -> exists(cp|
   cp.nounConcept = self and cp.isObjectification()
then
as.oclIsNew() and as.oclIsTypeOf(AssociationClassCh) and
as.name = self.name and asft.factTypeRole -> forAll(ro|
     me.oclIsNew() and
     me.oclIsTypeOf(AssociationClassMemberEnd) and
     me.associationClassCh = as and me.name = ro.name and
     me.typeName = ro.nounConcept.name and
     me.isDerived = false and me.isDerivedUnion = false and
     me.aggregation_ = AggregationKind::none and
     (if ro.order = 1
      then str2 -> isEmpty() implies
        (me.lowerValue = 0 and
           me.upperValue.oclIsTypeOf(UnlimitedNatural)) and
           str2 -> notEmpty() implies
           (me.lowerValue =
               str2.quantificationType().lowerValue() and
            me.upperValue =
               str2.quantificationType().upperValue())
      else
        str1 -> is Empty() implies
           (me.lowerValue = 0 and
           me.upperValue.oclIsTypeOf(UnlimitedNatural)) and
        str2 -> notEmpty() implies
           (me.lowerValue =
               str1.quantificationType().lowerValue() and
            me.upperValue =
               str1.quantificationType().upperValue())
     endif
else
   cl.oclIsNew() and cl.oclIsTypeOf(ClassCh) and
   cl.name = self.name and
   if ClosedProjection.allInstances() -> exists(cp|
     cp.nounConcept = self and cp.isAbstract
   then cl.isAbstract
   else cl.isAbstract = false
   endif
endif
```

6.3.2.3 *includedInUml() of individual concept*

General description

An *IndividualConcept* that is a schema unit maps to a characterization object of an *Enumeration*. The attribute *name* of the *Enumeration* has the same value as the attribute *name* of the *ObjectType* that uses this *IndividualConcept* in its definition.

Formal specification in OCL

```
context IndividualConcept::includedInUml():EnumerationCh
```

```
  post EnumerationChCreated:
       let cp:ClosedProjection =
         self.variable.isInProjection.oclAsType(ClosedProjection)
       in
       en.oclIsNew() and en.oclIsTypeOf(EnumerationCh) and
       en.name = cp.nounConcept.name and
       cp.logicalFormulation.sequenceOfLiterals()-> forAll (lit |
         li.oclIsNew() and li.oclIsTypeOf(Literal) and
         li.enumerationCh = en and li.name = lit)
```

6.3.2.4 *includedInUml() of characteristic*

General description

A *Characteristic* maps to a characterization object of a *Property.* The *PropertyCh* has the concatenation of 'being ' and *name* attribute as the name of the *Characteristic* and the attribute *type* has the same value 'Boolean'. The *ownerClassName* attribute has the same value as the name of the concept that the role ranges over. The *lowerValue* and *upperValue* attributes have 1 as values.

Formal specification in OCL

```
context Characteristic::includedInUml():PropertyCh
  post PropertyChCreated:
       pr.oclIsNew() and pr.oclIsTypeOf(PropertyCh) and
       'being '.concat(pr.name) = self.name and
       pr.ownerClassName = self.factTypeRole ->
         first().nounConcept.name and
       pr.type = 'Boolean' and
       pr.isDerived = false and pr.isDerivedUnion = false and
       pr.aggregation_ = AggregationKind::none and
       pr.lowerValue = 1 and pr.upperValue = 1
```

6.3.2.5 *includedInUml() of is-property-of fact type*

General description

An *IsPropertyOfFactType* maps to a characterization object of a *Property.* The *name* attribute of the *PropertyCh* is the name of the second role of the *IsPropertyOfFactType*, while its *type* attribute has the same value as the name of the concept that the second role ranges over. The *ownerClassName* attribute has the same value as the name of the concept that the first role ranges over. The *lowerValue* and *upperValue* attributes are determined by the type of quantification formulation that defines the multiplicity constraint.

Formal specification in OCL

```
context IsPropertyOfFactType::includedInUml():PropertyCh
  post PropertyChCreated:
       let str:StructuralRule = StructuralRule.allInstances() ->
         any(st| st.isMultiplicity() and
           st.closedLogicalFormulation.
           oclAsType(ClosedUniversalQuantification).
           introducedVariable.rangedOverConcept.name =
           self.factTypeRole-> first().nounConcept.name and
           st.closedLogicalFormulation.
           oclAsType(ClosedUniversalQuantification).
           scopeFormulation.oclAsType(Quantification).
```

```
          introducedVariable.rangedOverConcept.name =
          self.factTypeRole -> last().name)
      in
      pr.oclIsNew() and pr.oclIsTypeOf(PropertyCh) and
      pr.name = self.factTypeRole -> last().nounConcept.name and
      pr.ownerClassName = self.factTypeRole ->
         first().nounConcept.name and
      pr.type = self.factTypeRole-> last().name and
      pr.isDerived = false and pr.isDerivedUnion = false and
      pr.aggregation_ = AggregationKind::none and

      str -> notEmpty() implies
       (pr.lowerValue = str.quantificationType().lowerValue() and
        pr.upperValue = str.quantificationType().upperValue())
```

6.3.2.6 *includedInUml() of associative or partitive fact type*

General description

In general, an *AssociativeFactType* and a *PartitiveFactType* map to a characterization object of an *Association.* However, if there is an object type defined as the objectification of the *AssociativeFactType* or the *PartitiveFactType*, the *FactType* maps to a characterization object of an *AssociationClass*.

In the first case, the *name* attribute of the *AssociationCh* is the name of the *AssociativeFactType*. The *AssociationCh* has two instances of *AssociationMemberEnd*. The first one, if the first fact type role has a name, has that name as its *name* attribute. The *typeName*e attribute has the same value as the name of the object type that the role scopes over. The *aggregation* attribute has the value *AggregationKind::composite* if the name of the *AssociativeFactType* is "is included in." The *aggregation* attribute has the value *AggregationKind::shared* if the name of the *AssociativeFactType* is "is part of." Otherwise, the *aggregation* attribute has the value *AggregationKind::none*. The values of the *lowerValue* and *upperValue* attributes are determined by the existence of a structural rule that restricts the multiplicity of the second role with respect to this first role. Similarly, the values of the second *AssociationMemberEnd* are defined from the values of the second fact type role.

In the second case, the *name* attribute of the *AssociationClassCh* is the name of the *ObjectType* of the objectification. The *AssociationClassCh* has two instances of *AssociationClassMemberEnd*. Both have the same values as those defined for the *AssociationMemberEnd* of the *AssociationCh*.

Formal specification in OCL

```
context AssociativeFactType::includedInUml():MeaningCh
  post AssociationChorAssociationClassCreated:
      -- multiplicity structural rule 1
      let str1:StructuralRule = StructuralRule.allInstances->
          any(st| st.isMultiplicity and
            st.closedLogicalFormulation.
            oclAsType(ClosedUniversalQuantification).
            introducedVariable.rangedOverConcept =
            self.factTypeRole -> last().nounConcept)  in
```

```
-- multiplicity structural rule 2
let str2:StructuralRule = StructuralRule.allInstances->
     any(st| st.isMultiplicity and
       st.closedLogicalFormulation.
       oclAsType(ClosedUniversalQuantification).
       introducedVariable.rangedOverConcept =
       self.factTypeRole -> first().nounConcept) in

if not ClosedProjection.allInstances() -> exists(cp|
     cp.isObjectification and
     cp.logicalFormulation.oclAsType(Objectification).
     consideredLogicalFormulation.
     oclAsType(AtomicFormulation).factType.
     oclAsType(AssociativeFactType) = self)
then
-- the associative fact type maps to an association
as.oclIsNew() and as.oclIsTypeOf(AssociationCh) and
(if self.name <> 'has' or self.name <> 'is part of' or
     self.name <> 'includes'
 then
     as.name = self.name
 else self.name ->isEmpty()
 endif) and
 as.isAbstract = false and self.factTypeRole
  -> forAll(ro| me.oclIsNew() and
     me.oclIsTypeOf(AssociationMemberEnd)
     and me.associationCh = as and me.name = ro.name and
     me.typeName = ro.nounConcept.name and
     me.isDerived = false and me.isDerivedUnion = false and
     (if self.name = 'is part of' and
       self.factTypeRole->last = ro
      then
       me.aggregation = AggregationKind::shared
      else me.aggregation = AggregationKind::none
      endif) and
     (if self.factTypeRole->first = ro
     then
       str2 -> isEmpty() implies
       (me.lowerValue = 0 and
        me.upperValue.oclIsTypeOf(UnlimitedNatural)) and
       str2 -> notEmpty() implies
          (me.lowerValue =
              str2.quantificationType().lowerValue() and
           me.upperValue =
              str2.quantificationType().upperValue())
     else
       str1 ->isEmpty() implies
          (me.lowerValue = 0 and
          me.upperValue.oclIsTypeOf(UnlimitedNatural)) and
       str2 -> notEmpty() implies
          (me.lowerValue =
              str1.quantificationType().lowerValue() and
           me.upperValue =
              str1.quantificationType().upperValue())
      endif)
```

```
else
  -- the associative fact type maps to an association class
  asc.oclIsNew() and asc.oclIsTypeOf(AssociationClassCh)
  and asc.name = Objectif.nounConcept.name and
  self.factTypeRole forAll(ro|
    me.oclIsNew() and
    me.oclIsTypeOf(AssociationClassMemberEnd) and
    me.associationClassCh = asc and me.name = ro.name and
    me.typeName = ro.nounConcept.name and
    me.isDerived = false and me.isDerivedUnion = false and
    (if self.name = 'is part of' and
      self.factTypeRole->last = ro
     then
      me.aggregation = AggregationKind::shared
     else me.aggregation = AggregationKind::none
     endif) and
    (if self.factTypeRole.first = ro
    then
      str2 -> isEmpty() implies
        (me.lowerValue = 0 and
        me.upperValue.oclIsTypeOf(UnlimitedNatural)) and
      str2 -> notEmpty() implies
        (me.lowerValue =
           str2.quantificationType().lowerValue() and
         me.upperValue =
           str2.quantificationType().upperValue())
    else
      str1 ->isEmpty() implies
        (me.lowerValue = 0 and
        me.upperValue.oclIsTypeOf(UnlimitedNatural)) and
      str2 -> notEmpty() implies
        (me.lowerValue =
           str1.quantificationType().lowerValue() and
         me.upperValue =
           str1.quantificationType().upperValue())
    endif)
endif
```

6.3.2.7 *includedInUml() of categorization fact type*

General description

A *CategorizationFactType* maps to a characterization object of *Generalization.* The *specificClassName* attribute of the *GeneralizationCh* has the same value as the name of the concept that the first role of the *CategorizationFactType* ranges over. The *generalClassName* attribute of the *GeneralizationCh* has the same value as the name of the concept that the second role of the *CategorizationFactType* ranges over.

Formal specification in OCL

```
context CategorizationFactType::includedInUml():GeneralizationCh
post GeneralizationChCreated:
    ge.oclIsNew() and ge.oclIsTypeOf(Generalization) and
    ge.generalClassName = self.factTypeRole ->
        last.nounConcept.name and
    ge.specificClassName = self.factTypeRole ->
```

```
            ->first().nounConcept.name)
```

6.3.2.8 *includedInUml() of categorization schema*

General description

A *CategorizationSchema* and a *Segmentation* map to a characterization object of *GeneralizationSet.* The attribute *name* of the *GeneralizationSetCh* has the same value as the *name* attribute of the *CategorizationSchema*. For each *category* of the *CategorizationSchema*, the *GeneralizationSetCh* has a *Participant* whose *generalClassName* attribute has the value of the *name* attribute of the general concept of the *CategorizationSchema* and whose *specificClassName* attribute has the same value as the name attribute of the category. The *isCovering* and *isDisjoint* attributes have the value *true* if there are structural rules whose meanings are the covering and disjointness constraints, as described in Section 6.1.2.

Formal specification in OCL

```
context CategorizationSchema::includedInUml():GeneralizationSetCh
  post GeneralizationSetChCreated:
       let strCoveringAndDisjoint:StructuralRule =
         StructuralRule.allInstances() -> any(st|
         st.isDisjointAndCovering and st.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         introducedVariable.rangedOverConcept.name =
         self.generalConcept.name and
         self.category.name -> includesAll(
         st.closedLogicalFormulation.
              oclAsType(ClosedUniversalQuantification).
              scopeFormulation.sequenceOfCategories())
       in
       let strCovering:Set(StructuralRule) =
         StructuralRule.allInstances() -> select(st|
         st.isCovering and st.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         introducedVariable.rangedOverConcept.name =
         self.generalConcept.name and
         self.category.name -> includesAll(
         st.closedLogicalFormulation.
              oclAsType(ClosedUniversalQuantification).
              scopeFormulation.sequenceOfCategories())
       in
       let strDisjoint:StructuralRule =
         StructuralRule.allInstances()-> select(st|
         st.isDisjoint and st.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         introducedVariable.rangedOverConcept.name =
         self.generalConcept.name and
         self.category.name -> includesAll(
         st.closedLogicalFormulation.
              oclAsType(ClosedUniversalQuantification).
              scopeFormulation.sequenceOfCategories())
       in
       ge.oclIsNew() and ge.oclIsTypeOf(GeneralizationSetCh) and
       ge.name = self.name and self.category->forAll(ca|
         pa.oclIsNew() and pa.oclIsTypeOf(Participant) and
         pa.generalClassName = self.generalConcept.name and
         pa.specificClassName = ca.name and
         pa.generalizationSetCh = ge) and

       self.oclIsTypeOf(Segmentation) implies
         (ge.isCovering = true and ge.isDisjoint = true) and

       (strCoveringAndDisjoint->notEmpty() implies
         (ge.isCovering = true and ge.isDisjoint = true) and

       (strCovering-> size() = self.category ->size() - 1) implies
         (ge.isCovering = true) and

       (strDisjoint->size() = self.category ->size() - 1) implies
         (ge.isDisjoint = true)
```

Given a logical formulation that structures a rule that means the covering and/or disjointness of a categorization scheme, the *sequenceOfCategories()* operation returns the sequence of names of the categories of the categorization scheme included in the rule.

```
context LogicalFormulation::sequenceOfCategories():
          Sequence(String)
  body:   if self.oclIsTypeOf(ExclusiveDisjunction)
          then
            self.oclAsType(ExclusiveDisjunction).logicalOperand1.
            oclAsType(AtomicFormulation).factType.factTypeRole ->
            first().nounConcept.name -> union(
            self.oclAsType(ExclusiveDisjunction).logicalOperand2.
              oclAsType(AtomicFormulation).factType.factTypeRole ->
              first().nounConcept.name)
          else
            if self.oclIsTypeOf(Disjunction) then
              self.oclAsType(Disjunction).logicalOperand1.
              oclAsType(AtomicFormulation).factType.factTypeRole
              ->first().nounConcept.name -> union(
              if self.oclAsType(Disjunction).logicalOperand2.
              oclIsTypeOf(AtomicFormulation)
              then
                self.oclAsType(Disjunction).logicalOperand2.
              oclAsType(AtomicFormulation).factType.factTypeRole
              -> first().nounConcept.name
              else
              self.oclAsType(Disjunction).logicalOperand2.
                sequenceOfCategories()
              endif
            else
              self.oclAsType(NorFormulation).logicalOperand1.
              oclAsType(AtomicFormulation).factType.factTypeRole
              -> first().nounConcept.name -> union(
              if self.oclAsType(NorFormulation).logicalOperand2.
                oclIsTypeOf(AtomicFormulation)
              then
                self.oclAsType(NorFormulation).logicalOperand2.
                oclAsType(AtomicFormulation).factType.factTypeRole
                -> first().nounConcept.name
```

```
                  else
                     self.oclAsType(NorFormulation).logicalOperand2.
                     sequenceOfCategories()
                  endif
               endif
            endif
```

6.3.2.9 *includedInUml() of reference scheme*

General description

A *ReferenceScheme* that is a schema unit maps to a characterization object of a constraint, *ConstraintCh*, as follows: (i) the value of the *namespace* attribute of the *ConstraintCh* is the same as the *name* attribute of the referenced concept of the reference scheme; (ii) the *symbolExpression* attribute is the concatenation of the referenced concept with "allInstances->isUnique(" and with the name of the simply used roles of the reference scheme; and (iii) for each simply used role of the reference scheme, there is an instance of *ConstrainedElement* whose *name* attribute has the same name as the role and whose type is "property."

Formal specification in OCL

```
context ReferenceScheme::includedInUml():ConstraintCh
  post ConstraintCh:
       ch.oclIsNew() and ch.oclIsTypeOf(ConstraintCh) and
       ch.namespace = self.referencedConcept.name -> any(true) and
       ch.bodyOpaqueExpression = self.referencedConcept.name ->
       any(true).concat('.allInstances-> isUnique('.concat(
       self.simplyUsedRole -> any(true).name.concat(')'))) and
       self.simplyUsedRole -> forAll(ro |
         co.oclIsNew() and co.oclIsTypeOf(ConstrainedElement) and
         co.constraintCh = ch and co.name = ro.name and
         co.type = TypeCons::property)
```

6.3.2.10 *includedInUml() of structural rule*

General description

Depending on how its closed universal quantification is structured, a *StructuralRule* that is a schema unit may map to different characterization objects of *ElementCh*:

- A structural rule whose meaning corresponds to a multiplicity constraint is mapped to a characterization object (*AttributeCh*, *AssociationCh* or *AssociationClassCh*) resulting from the mapping of the fact type that the atomic formulation is based on.
- A structural rule whose meaning corresponds to a covering and/or disjointness constraint is mapped to a characterization object of a generalization set, *GeneralizationSetCh.*
- A structural rule whose meaning corresponds to an XOR constraint is mapped to a characterization object of a constraint, *ConstraintCh*.

Formal specification in OCL

```
context StructuralRule::includedInUml():ElementCh
  post ElementChCreated:
       self.isMultiplicity() implies
         self.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         scopeFormulation.oclAsType(Quantification).
         scopeFormulation.oclAsType(AtomicFormulation).
         factType^includedInUml() and

       self.isDisjointAndCovering() implies
         self.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         scopeFormulation.oclAsType(ExclusiveDisjunction).
         logicalOperand1.oclAsType(AtomicFormulation).
         factType.oclAsType(CategorizationFactType).factTypeRole->
         first().categorizationScheme -> any(true)^includedInUml()
       and
       self.isCovering() implies
         self.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         scopeFormulation.oclAsType(Disjunction).logicalOperand1.
         oclAsType(AtomicFormulation).factType.
         oclAsType(CategorizationFactType).factTypeRole ->
         first().categorizationScheme -> any(true)^includedInUml()
       and
       self.isDisjoint() implies
         self.closedLogicalFormulation.
         oclAsType(ClosedUniversalQuantification).
         scopeFormulation.oclAsType(NorFormulation).
         logicalOperand1.oclAsType(AtomicFormulation).factType.
         oclAsType(CategorizationFactType).factTypeRole ->
         first().categorizationScheme -> any(true)^includedInUml()
       and
       self.isXOR() implies
         (co.oclIsNew() and co.oclIsTypeOf(ConstraintCh) and
       vs.symbolExpression = 'XOR' and
       co.namespace  = self.closedLogicalFormulation.
       oclAsType(ClosedUniversalQuantification).
       scopeFormulation.oclAsType(Disjunction).logicalOperand1.
       oclAsType(AtomicFormulation).factType.
       oclAsType(AssociativeFactType).factTypeRole ->
       first().nounConcept.name and
       st.closedLogicalFormulation.
       oclAsType(ClosedUniversalQuantification).
       scopeFormulation.restrictedFactTypes() ->
       forAll( rc| ce.oclIsNew() and
         ce.oclIsTypeOf(ConstrainedElement) and
         ce.constraintCh = co and ce.type = ConsType::Association
         and ce.name = rc.name and
         ce.membersName = rc.factTypeRole->collect(name) and
         ce.membersType = rc.factTypeRole ->
              collect(nounConcept.name))
```

Given a logical formulation that structures a rule that means, partially, an XOR constraint, the *restrictedFactTypes()* operation returns the sequence of fact types included in the rule.

```
context LogicalFormulation::restrictedFactTypes():
        Sequence(FactType)
  body: self.oclAsType(NorFormulation).logicalOperand1.
        oclAsType(AtomicFormulation).factType  -> union(
        if self.oclAsType(NorFormulation).logicalOperand2.
          oclIsTypeOf(AtomicFormulation)
        then
        self.oclAsType(NorFormulation).logicalOperand2.
          oclAsType(AtomicFormulation).factType
        else
        self.oclAsType(NorFormulation).logicalOperand2.
          restrictedFactTypes()
        endif
```

6.4 Translation mapping constraints

As described in Chapter 3, let $M = (MS_1, MS_1, \Sigma)$ be a mapping. M is a translation mapping when, for any S_1 and S_2 such that $\langle S_1, S_2 \rangle$ is an instance of M, then S_1 and S_2 are translations of each other. Therefore, in a translation mapping, the set of constraints Σ is satisfied when the two schemas are translations of each other. As stated in Section 3.3.5, the translation mapping constraints, Σ, consist of exactly two constraints, called *complete and consistent mapping to* S_2 and *complete and consistent mapping to* S_1. The intuitive meaning of the constraints, as described in Section 3.3.5, is that $\langle S_1, S_2 \rangle$ is an instance of the translation mapping M if each translatable schema unit of S_1 is consistently mapped to S_2 and each translatable schema unit of S_2 is consistently mapped to S_1.

Therefore, let $M = (MUml, MSbvr, \Sigma)$ be a translation mapping where $MUml$ and $MSbvr$ are instances of the UML metaschema and the SBVR metaschema, respectively. The translation mapping constraints Σ consist of exactly two constraints, called *complete and consistent mapping to Sbvr* and *complete and consistent mapping to Uml*.

```
context Element inv completeAndConsistentMappingToSbvr:
  isSchemaUnit() and
  (sbvrMappingKind() = MappingKind::HasEquivalents or
  sbvrMappingKind() = MappingKind::IsIncluded) implies
    mappedToSbvr()

context Meaning inv completeAndConsistentMappingToUml:
  isSchemaUnit() and
  (umlMappingKind() = MappingKind::HasEquivalents or
  umlMappingKind() = MappingKind::IsIncluded) implies
     mappedToUml()
```

The consistency condition is checked by two operations: *mappedToSBVR()* and *mappedToUml()*, which return a *True* value if the condition is satisfied and a *False* value otherwise. The following is the formal specification of the two operations, in OCL:

```
context Element::mappedToSbvr():Boolean
 body: if sbvrMappingKind() = MappingKind::HasEquivalents
       then
       self.sbvrEquivalents() -> forAll(th:MeaningCh|
         th.schemaUnit()->notEmpty() and
         th.schemaUnit().umlMappingKind() =
           MappingKind::IsIncluded and
         th.schemaUnit().includedInUml().schemaUnit() = self)
       else
         if sbvrMappingKind() = MappingKind::IsIncluded
         then
           self.includedInSbvr().schemaUnit()->notEmpty() and
           self.includedInSbvr().schemaUnit().umlMappingKind()
             = MappingKind::HasEquivalents and
           self.includedInSbvr().schemaUnit().umlEquivalents().
             schemaUnit()->includes(self)
         else
           False
         endif
       endif

context Meaning::mappedToUml():Boolean
 body: if umlMappingKind() = MappingKind::HasEquivalents
       then
       self.umlEquivalents() -> forAll(el:ElementCh|
         el.schemaUnit()->notEmpty() and
         el.schemaUnit().sbvrMappingKind() =
           MappingKind::IsIncluded
         and  el.schemaUnit().includedInSbvr().schemaUnit() = self)
       else
         if umlMappingKind() = MappingKind::IsIncluded
         then
           self.includedInUml().schemaUnit()->notEmpty() and
           self.includedInUml().schemaUnit().sbvrMappingKind()
             = MappingKind::HasEquivalents and
           self.includedInUml().schemaUnit().sbvrEquivalents().
             schemaUnit()->includes(self)
         else
           False
         endif
       endif
```

6.5 Translating UML and SBVR meanings schemas

Chapter 3 described how to use the operations defined in the previous sections in the translation of schemas. In general, let $M = (MS_1, MS_2, \Sigma)$ be a mapping and $S_1 = \{u_{1,1}, \ldots, u_{1,n}\}$ an instance of MS_1. The translation of S_1 into MS_2 is a schema $S_2 = \{u_{2,1}, \ldots, u_{2,m}\}$ such that $\langle S_1, S_2 \rangle$ is an instance of M. The translation of S_2 into MS_1 is defined similarly. The approach to the translation of a schema $S_1 = \{u_{1,1}, \ldots, u_{1,n}\}$ consists in translating each of its schema units $u_{i,j}$ following the order given by the operation *predecessors*, starting with the units that have no predecessors. The translation is done by applying an operation called *translateTo*S_j() to the schema units. An instance $u_{i,k}$ of S_i*Element* can be translated into S_j if it represents a schema unit whose mapping kind is

HasEquivalents or *IsIncluded*. The effect of the operation *translateToS$_j$*() must be that $u_{i,k}$ is mapped to S_j.

The translation of a UML schema to an SBVR meanings schema is done by applying the operation called *translateToSbvr()* to the UML schema units. The specification of the pre- and postconditions of the operation, in OCL, is as follows:

```
context Element:translateToSbvr()
 pre: isSchemaUnit() and
       (sbvrMappingKind() = MappingKind::HasEquivalents or
       sbvrMappingKind() = MappingKind::IsIncluded)
 post: mappedToSbvr()
```

Similarly, the translation of an SBVR meanings schema to a UML schema is done by applying the operation called *translateToUml()* to the SBVR schema units. The specification of the pre- and postconditions of the operation, in OCL, is as follows:

```
context Meaning:translateToUml()
 pre: isSchemaUnit() and
       (sbvrMappingKind() = MappingKind::HasEquivalents or
       sbvrMappingKind() = MappingKind::IsIncluded)
 post: mappedToUml()
```

There is no need to refine the specification of the two operations in the subtypes of *Element* or *Thing*. The specifications *mappedToSbvr* and *mappedToUml* are implemented in a fairly straightforward manner, as explained in Section 6.4, using the methods of the operations *createUnit* (Section 4.4.3 in UML and Section 5.4.3 in SBVR), *sbvrEquivalents()* (Section 6.2.1), *umlEquivalents* (Section 6.2.2), *includedInSbvr* (Section 6.3.1) and *includedInUml()* (Section 6.3.2).

The implementation of the methods of *translateToSbvr* and *translateToUml* are described in the appendices using the USE procedural language (Gogolla, Büttner & Richters 2007). Specifically, Appendix G describes the methods of the *sbvrEquivalents()* operations, Appendix H describes the methods of the *umlEquivalents* operations, Appendix I describes the methods of the *includedInSbvr* operations and Appendix J describes the methods of the *includedInUml()* operations.

The implementation is applied to translate the instances of the DBLP example from UML to SBVR and vice versa. In both cases, the time required to carry out the translation and check its completeness and consistency is less than four minutes, which seems quite acceptable when one is using research-oriented tools in a research environment.

7 SBVR Structured English representations

The *Semantics of Business Vocabulary and Business Rules (SBVR), v.1.0* document (Object Management Group 2008a), as described in Chapter 5, defines the metamodel for documenting the semantics of business vocabulary, business facts and business rules. SBVR is targeted to capture business concepts and business rules in a language close enough to ordinary language to facilitate business experts to read them. The SBVR specification proposes different notations to represent the instances of SBVR Meanings. Moreover, the specification in its metamodel includes different types of *Representation* to obtain, more easily, vocabularies and rules in any of these notations.

In particular, the specification defines an English vocabulary, called *SBVR Structured English*, as one of the possibly many notations that may be obtained from the SBVR representations. SBVR Structured English uses a small number of English structures and common words to elaborate vocabularies and rules. The SBVR specification also provides some predefined language patterns to map these SBVR Structured English notations to SBVR instances. Unfortunately, the SBVR specification does not provide a straightforward nor complete mapping from SBVR instances to such notations.

This chapter overviews the SBVR Structured English notation and describes the subset of the SBVR metamodel concerning representations of meanings. Note that some additional elements have been added to the SBVR Representations metaschema to have a straightforward SBVR Structured English notation.

The instances of the SBVR Structured English may derive from SBVR Meanings and this chapter provides the operations to derive the instances of SBVR Structured English from SBVR Meanings. These operations and the ones defined in the previous chapters may be used to automatically represent an SBVR schema in SBVR Structured English Notation. The representation is done by applying an operation called *vocabularyEntry(),* which is also described.

This chapter finishes by showing the result of applying said operation to the DBLP example introduced in Chapter 4 as an SBVR Structured English vocabulary.

As in the previous chapters, the SBVR Structured English metaschema, an example of instantiation and the specification and implementation of the operations have been specified in the USE tool. The detailed specifications are provided in the Appendices.

The rest of this chapter is structured as follows:

- Section 7.1 overviews SBVR Structured English as one of the possible notations of the SBVR representations.
- Section 7.2 shows the figures that form the abstract syntax of the subset of SBVR used to represent meanings in SBVR Structured English and briefly describes the concepts included in the abstract syntax.
- Section 7.3 defines the *newRepresentation()* operation on the schema units of SBVR to generate the instances of subtypes of SBVR *Representation.*
- Section 7.4 defines the *vocabularyEntry()* query operation that gives the representation of a schema unit in in SBVR Structured English notation.
- Section 7.5 shows the DBLP example as a SBVR Structured English Vocabulary resulting from the application of the operations described in the previous sections.

7.1 Overview of SBVR Structured English

SBVR Structured English is a proposed notation to express meanings. This section, reviews the main characteristics of the notation, to describe a vocabulary, which includes necessities of SBVR.

7.1.1 Expressions in SBVR Structured English

Any expression, in SBVR may be written in one of the four font styles:

term	The 'term' font is used for a designation of a type, one that is part of a vocabulary being used or defined (e.g., person, paper).
Name	The 'name' font is used for a designation of an individual concept (instances) — a name. Names tend to be proper nouns (e.g., Antoni).
verb	The 'verb' font is used for designations of fact types — usually a verb, preposition or combination thereof. Such a designation is defined in the context of a form of expression.
keyword	The 'keyword' font is used for linguistic symbols used to construct statements – the words that can be combined with other designations to form statements and definitions (e.g., "**each**" and "**it is obligatory that**")

The SBVR Structured English uses designations and forms of expressions exactly as they are defined in a vocabulary. Plural forms are not used to avoid linguistic difficulties. For example, a formal statement would say "each concept" rather than "all concepts." Both the active form and the passive form of a verb need to be defined in a vocabulary if both are used.

7.1.1.1 *Key words and phrases for logical formulations*

Key words and phrases are shown below for expressing each kind of logical formulation. The letters '*n*' and '*m*' represent use of a literal whole number. The letters '*p*' and '*q*' represent expressions of propositions.

Quantification

each	universal quantification
some	existential quantification
at least one	existential quantification
at least *n*	at-least-n quantification
at most one	at-most-one quantification
at most *n*	at-most-n quantification
exactly one	exactly-one quantification
exactly *n*	exactly-n quantification
at least *n* and at most *m*	numeric range quantification
more than one	at-least-n quantification with *n* = 2

Logical Operations

it is not the case that *p*	logical negation
p and *q*	conjunction
p or *q*	disjunction
p or *q* but not both	exclusive disjunction
if *p* then *q*	implication
q if *p*	implication
p if and only if *q*	equivalence
not both *p* and *q*	nand formulation
neither *p* nor *q*	nor formulation
p whether or not *q*	whether-or-not formulation

Where a subject is repeated when using "and" or "or," the repeated subject can be elided.

7.1.1.2 *Modal Operations*

A possible style of SBVR Structured English for modal operations is the Prefix Style that introduces rules by prefixing a statement with keywords that convey a modality. An structural rule uses the keyword: It is necessary that

7.1.1.3 *Other Keywords*

the	Used with a designation to make a pronominal reference to a previous use of the same designation. This is formally a binding to a variable of a quantification.
a, an	Universal or existential quantification, depending on context based on English rules.

another	(Used with a term that has been previously used in the same statement) existential quantification plus a condition that the referent thing is not the same thing as the referent of the previous use of the term.
a given	Universal quantification pushed outside of a demonstrative expression where "a given" is used such that it represents one thing at a time – this is used to avoid ambiguity where the "a" by itself could otherwise be interpreted as an existential quantification.

7.1.2 Describing a Vocabulary

In SBVR Structured English, a vocabulary is described in a document section having glossary-like entries for concepts having representations in the vocabulary.

7.1.2.1 *Vocabulary Entries*

Each entry is for a single concept, which is called the entry concept. It starts with a representation of the concept, either a designation or a form of expression.

Any of several kinds of captioned details can be listed under the representation. A skeleton of a vocabulary entry is shown below followed by an explanation of the use of each caption. Only those entries considered for the mapping between UML and SBVR are showed.

<primary representation>
Definition:
General Concept:
Concept type:
Necessity:
Reference Scheme:

Primary Representation: Designation or Form of Expression

The designation or form of expression, called the "primary representation" with respect to each entry, can be for any concept type. The primary representation for a fact type is a form of expression. Three examples are given below:

person

person ***has*** **name**

Catalunya

Definition

A definition is shown as an expression that can be logically substituted for the primary representation. A definition is fully formal if all of it is styled as described above.

General concept

The "General Concept" caption can be used to indicate a concept that generalizes the entry concept.

Concept Type

The "Concept Type" caption is used to specify a type of the entry concept. This is typically not used if the concept has no particular type other than what is obvious from the primary representation. A name is implicitly for an individual concept. Any term is implicitly for a noun concept. A form of expression is implicitly for a fact type.

Necessity

A "Necessity" caption is used to state something that is necessarily true. A necessity is an element of guidance expressed as a structural business rule statement. A guidance statement can be expressed formally or informally. A statement that is formal uses only formally styled text — all necessary vocabulary is available (by definition or adoption) so that no external concepts are required. Such a statement can be represented as a logical formulation. For example:

It is necessary that each authored publication ***has*** at least one author.

The above example includes three key words or phrases ("it is necessary that," "each" and "at least one"), two designations for types and one for a fact type (from a form of expression).

The key phrase "it is necessary that" can be omitted from a statement of a structural rule captioned "Necessity" because it is implied in the caption.

Reference Scheme

The "Reference Scheme" caption is used to state how things denoted by a term can be distinguished from each other based on one or more facts about the things. A reference scheme is expressed by referring to at least one role of a binary fact type.

7.2 SBVR Representations

Figure 7.1 shows the fragment of the SBVR metamodel (Object Management Group 2008a) that describes the representations in SBVR. Note that, in order to have a straightforward notation in SBVR Structured English, the *StructuredEnglishText* metaclass has been added. It has two attributes: *value* that constrain the expression of the representation; and the attribute *font* which represents any of the four font styles used in SBVR Structured English. In SBVR Structured English, a representation is composed by a set of ordered instances of *StructuredEnglishText*. Additionally, two additional abstract classes, *PrimaryRepresentation* and *Caption* have been added to distinguish the primary representation caption from the other captions and also the relationship among them. Moreover, there are three additional metaclasses: *GeneralConceptCaption*, *ConceptTypeCaption* and *ReferenceSchemeCaption*. The first one represents the designation of a general concept; the second one represents the type of concept; and the third one represents the reference scheme of a concept.

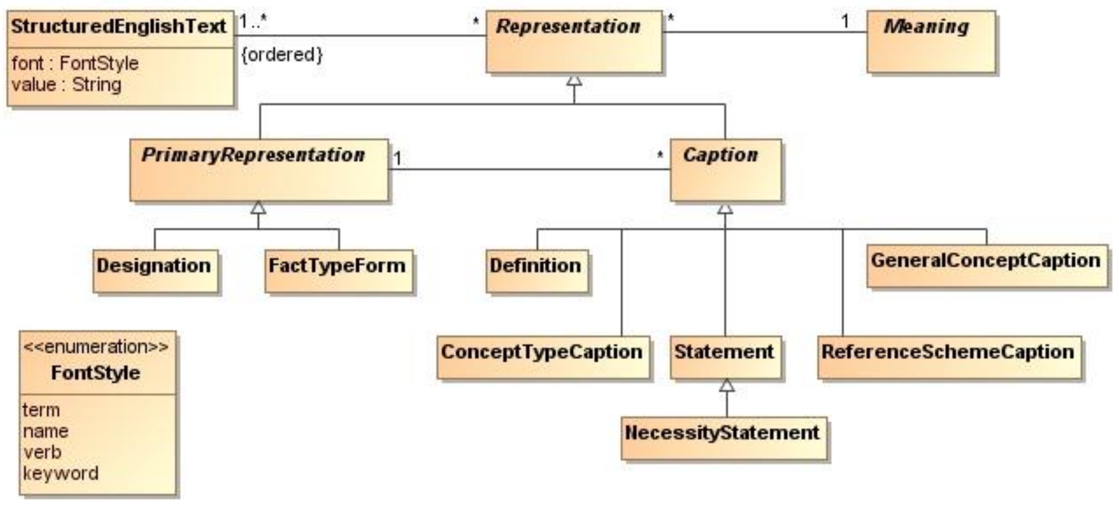


Figure 7.1 SBVR Representations

Appendix I describes the complete specification of the SBVR Structured English metamodel in the USE tool.

7.3 *newRepresentation()* operation

This section describes the *newRepresentation()* operation, in the context of *Meaning*. The evaluation of the *newRepresentation()* operation on a SBVR schema unit gives the set of *Representations* that are derived from said schema unit:

```
context Meaning::newRepresentation():Set(Representation)
  pre: isSchemaUnit()
  post definingTheRepresentation:
         result = Representation.allInstances() -
                        Representation.allInstances@pre()
```

The effect of the operation is defined in the subtypes of *Meaning* such that some or all of their instances represent schema units.

The *newText(name:String, font:FontStyle, order:Integer)* is an operation that gives the characteristics of a new instance of *StructuredEnglishText* associated, ordered, to an instance of a subtype of *Representation*:

```
context Representation::newText(name:String,
         font:FontStyle, order:Integer):StructuredEnglishText
  post StructuredEnglishTextCreated:
       se.oclIsNew() and se.oclIsTypeOf(StructuredEnglishText) and
       se.font = font and se.value = name and
       self.structuredEnglishText -> at(order) = se
```

Note that in the following sections the *hasSent* ('^') operator is used to invoke the *newText* operation within a postcondition (Object Management Group 2006b, p. 29). The use of this operator, as already used in the previous chapter, allows to better structure the postconditions. However, note that the USE tool does not allow the use of such operator. Therefore, the implementation, in USE, of a postcondition with an invocation to an operation includes, within the postcondition, the fragment corresponding to the invoked operation.

7.3.1 *newRepresentation()* of value type and object type

Each value type or object type schema unit is represented by an instance of *Designation* associated to it. The instance is also associated to an instance of *StructuredEnglishText* having the *font* and *value* attributes with values *FontStyle::term* and the name of the *ValueType* or *ObjectType*, respectively. For example, the value type named "gender" has an instance of *Designation* whose straightforward representation in Structured English notation is "gender".

Additionally, if the value type is associated to a closed projection which means that the value type is the enumeration of individual concepts, then the value type also includes a *Definition*. The definition is associated to a sequence of instances of *StructuredEnglishText* whose value and font attributes are the names of the individual concepts that define the value type, and *FontStyle::name*, respectively. For example, the previous value type is also represented by a *Definition* whose straithforward representation in Structured English notation is "**male or female**".

If the object type is associated to a closed projection which means that the instances of the object type are the union of instances of other concepts (like an abstsract UML class), then the objection type includes a *Definition*. The definition is associated to the sequence of instances of *StructuredEnglishText* that defines the object type as an a sequence of noun concepts joined by an "or" clause. For example, the object type named "authored publication" is represented by the *Designation*: **"authored publication**" and the *Definition*: **"authored book or book chapter or journal paper**".

Finally, if the object type is associated to a closed projection which means that the object type is an objectification of an associative fact type, then the object type also includes a *Definition*. The definition is associated to the sequence of instances of *StructuredEnglishText* that defines the object type as an "actuality" of an associative fact type. For example, the object type named "authorship" is represented by the *Designation*: "**authorship**" and the *Definition*: "**actuality that an author *has* an authored publication**".

```
context NounConcept::newRepresentation():Set(Representation)
  post RepresentationCreated:
       let asft:AssociativeFactType = self.closedProjection.
            logicalFormulation.oclAsType(Objectification).
            consideredLogicalFormulation.
            oclAsType(AtomicFormulation).
            factType.oclAsType(AssociativeFactType)in

       -- new designation
       d.oclIsNew() and d.oclIsTypeOf(Designation) and
       d.meaning = self and d^newText(self.name, FontSyle::term,1)
       and  if self.closedProjection -> notEmpty()
            then

            -- new definition
            def.oclIsNew() and def.oclIsTypeOf(Definition) and
            def.meaning.oclAsType(Concept) = self and
            def.primaryRepresentation = d and
```

```
        self.closedProjection.isEnumeration() implies
          self.closedProjection.logicalFormulation.
          sequenceOfLiterals() -> forAll(lit |
            def^newText(lit, FontStyle::term,
            self.closedProjection.logicalFormulation.
            sequenceOfLiterals() ->indexOf(lit)*2 -1) and
            if lit <> self.closedProjection.
               logicalFormulation.sequenceOfLiterals()->last()
            then def^newText('or', FontStyle::keyword,
                    self.closedProjection.logicalFormulation.
                    sequenceOfLiterals() ->indexOf(lit)*2)
            else true
            endif) and

        self.closedProjection.isAbstract() implies
          self.closedProjection.logicalFormulation.
          sequenceOfCategories() -> forAll(cat |
            def^newText(cat, FontStyle::term,
          self.closedProjection.logicalFormulation.
            sequenceOfCategories() ->indexOf(lit)*2 -1)  and
            if cat <> self.closedProjection.
               logicalFormulation.sequenceOfCategories()->
               last()
            then def^newText('or', FontStyle::keyword,
                    self.closedProjection.logicalFormulation.
                    sequenceOfCategories() ->indexOf(lit)*2)
            else true
            endif) and

        self.closedProjection.isObjectification() implies
          (def^newText('actuality', FontStyle::term, 1) and
          def^newText('that a', FontStyle::keyword, 2) and
          asft.factTypeRole -> forAll(rol |
            def^newText(rol, FontStyle::term,
            asft.factTypeRole -> indexOf(rol)*2 +1) and
            if asft.factTypeRole -> size()> 2 and
               def^newText(',', FontStyle::keyword,
               asft.factTypeRole -> indexOf(rol)*2 +2)
            else
               def^newText(asft.name, FontStyle::verb,
                    asft.factTypeRole -> indexOf(rol)*2 +2)
            endif))
      else
        true
      endif
```

7.3.2 *newRepresentation()* of individual concept

Each individual concept schema unit is represented by an instance of *Designation* associated to it. The instance is also associated to an instance of *StructuredEnglishText* with the *font* and *value* attributes with values *FontStyle::name* and the name of the individual concept. For example, the individual concept named "male" has an instance of *Designation:* "male."

```
context IndividualConcept::newRepresentation():Set(Representation)
  post RepresentationCreated:
       -- new designation
       d.oclIsNew() and d.oclIsTypeOf(Designation) and
       d.meaning = self and d^newText(self.name, FontSyle::name,1)
```

7.3.3 *newRepresentation()* of characteristic schema unit

Each *Characteristic* schema unit is represented by a *FactTypeForm* associated to a sequence of instances of *StructuredEnglishText.* The sequence has the following structure: (i) an instance the *value* attribute of which has the name of the fact type role (if the role does not have a name, then the name of the concept that the role scopes over) and the font style is "term;' and (ii) an instance the *value* attribute of which has the name of the fact type and the font sytle is "verb.' The *Characteristic* is represented also with *ConceptTypeCaption* which the associated *StructuredEnglishText* indicates that is a characteristic. For example the characteristic schema unit named "being conference paper" that has the role that ranges over "book chapter" is represented by a *FactTypeForm*: "book chapter *being conference paper*" and a *ConceptTypeCaption*: "characteristic."

```
context Characteristic::newRepresentation():
           Set(Representation)
  post RepresentationCreated :
       -- new fact type form
       f.oclIsNew() and f.oclIsTypeOf(FactTypeForm) and
       f.meaning.oclAsType(Characteristic) = self and
       f^newText(if self.factTypeRole -> first().name ->notEmpty()
                 then self.factTypeRole -> first.name
                 else self.factTypeRole -> first.nounConcept.name
                 endif, FontStyle::term, 1) and
       f^newText(self.name, FontStyle::verb, 2)

       -- new concept type form
       r.oclIsNew() and r.oclIsTypeOf(ConceptTypeCaption) and
       r.meaning.oclAsType(Characteristic) = self and
       r.primaryRepresentation = f and
       r^newText('characteristic', FontStyle::term, 1)
```

7.3.4 *newRepresentation()* of associative, is-property-of or partitive fact type schema unit

Each *AssociativeFactType* including (*PartitiveFactType* and *IsPropertyOfFactType*) is represented by a *FactTypeForm* associated to a sequence of instances of *StructuredEnglishText.* The sequence has the following structure: (i) an instance the *value* attribute of which has the name of the first fact type role (if the role does not have a name, then the name of the concept that the role scopes over) and the font style is 'term'; (ii) an instance the *value* attribute of which has the name of the fact type and the font sytle is 'verb'; and (iii) an instance the *value* attribute of which has the name of the last fact type role (if the role does not have a name, then the name of the concept that the role scopes over) and the font style is 'term'. The *AssociativeFactType* includes also a

ConceptTypeCaption the associated *StructuredEnglishText* of which indicates the type of associative fact type. For example, the associative fact type, named "is published in" between the concepts named "conference edition" and "edited book" is represented by the *FactTypeForm*: "conference edition ***is published in*** edited book."

Additionally, each fact type role of the *AssociativeFactType* that has a name is represented by an *Designation* and two *ConceptTypeCaption*. The *Designation* has a *StructuredEnglishText* with the *value* attribute as the name of the *Role* and the font style is 'term'. One of the *ConceptTypeCaption* has a *StructuredEnglishText* having the *value* and *font* attributes with values "role" and "term", respectively. The other *ConceptTypeCaption* has a *StructuredEnglishText* with the *value* the name of the concept that ranges over and the font style 'term'. For example, the associative fact type of above has also the *ConceptTypeCaption*: "associative fact type." And for example, the associative fact type between "editor" and "edited book" has the following representations: FactTypeForm: "editor has edited book", the ConceptType: "associative fact type," the Designation: "editor" and ConceptType: "role"

```
    context AssociativeFactType::newRepresentation():
                Set(Representation)
      post RepresentationCreated :
           -- new fact type form
           f.oclIsNew() and f.oclIsTypeOf(FactTypeForm) and
           f.meaning.oclAsType(AssociativeFactType) = self and
           self.factTypeRole -> forAll( ro|
             f^newText(if ro.name ->notEmpty() then ro.name
                        else ro.nounConcept.name  endif,
               FontStyle::term, self.factTypeRole ->indexOf(ro)*2-1)
               and
               if ro <> self.factTypeRole -> last()
               then if self.factTypeRole ->size() > 2
                    then f^newText(',', FontStyle::keyword,
                                   self.factTypeRole ->indexOf(ro)*2)
                    else f^newText(self.name, FontStyle::verb,
                                   self.factTypeRole ->indexOf(ro)*2)
                    endif
               else true endif) and

             -- new concept type caption
             r.oclIsNew() and r.oclIsTypeOf(ConceptTypeCaption) and
             r.meaning.oclAsType(AssociativeFactType) = self and
             r.primaryRepresentation = f and
             r^newText(if self.oclIsTypeOf(AssociativeFactType)
                        then 'associative fact type'
                        else if self.oclIsTypeOf(PartitiveFactType)
                                then 'partitive fact type'
                                else 'is-property-of fact type'
                                endif
                        endif, FontStyle::term,1) and
             self.factTypeRole->select(ft| ft.name -> notEmpty()) ->
               forAll(ro:FactTypeRole |
             -- new designation
             d.oclIsNew() and d.oclIsTypeOf(Designation) and
             d.meaning.oclAsType(AssociativeFactType) = self and
```

```
d^newText(ro.name, FontStyle::term,1) and
-- new concept type caption
c1.oclIsNew() and c1.oclIsTypeOf(ConceptTypeCaption) and
c1.meaning.oclAsType(AssociativeFactType) = self and
c1.primaryRepresentation = d and
c1^newText('role', FontStyle::term,1) and

-- new concept type caption
c2.oclIsNew()and c2.oclIsTypeOf(ConceptTypeCaption) and
c2.meaning.oclAsType(AssociativeFactType) = self and
c2.primaryRepresentation = d and
c2^newText(ro.nounConcept.name, FontStyle::term,1)
```

7.3.5 *newRepresentation()* of categorization fact type schema unit

Each categorization fact type schema unit is represented by a *GeneralConceptCaption*. The *GeneralConceptCaption* is associated to a *StructuredEnglishText* whose *value* attribute has the name of the general concept of the categorization fact type and its font style is 'term'. For example, the categorization fact type between "book chapter" and "authored publication" is represented by a *General concept*: "authored publication" of the primary representation "book chapter."

```
context CategorizationFactType::newRepresentation():
           Set(Representation)
  post RepresentationsCreated :
        f.oclIsNew() and f.oclIsTypeOf(GeneralConceptCaption) and
        f.meaning.oclAsType(CategorizationFactType) = self and
        f.primaryRepresentation = self.factTypeRole ->
           first().nounConcept.representation.oclAsType(Designation)
        and f^newText(self.factTypeRole -> last().nounConcept.name,
           FontStyle::term,1)
```

Note that, in SBVR Structured English, categorization fact types are represented by general concept captions of the general concepts.

7.3.6 *newRepresentation()* of categorization schema schema unit

Each categorization schema or segmentation schema unit is represented by several instances of Representation: (1) a *Designation* that has an *StructuredEnglishText* with the *value* attribute as the name of the *CategorizationScheme* and the font style is 'term'; (2) a *Definition* associated to a sequence of instances of *StructuredEnglishTex* which could be readed in Structured English as "categorization scheme that *is for* the generalConcept", where generalConcept refers to the name of the general concept of the categorization scheme; and (3) a *NecessityStatement* associated to a sequence of instances of *StructuredEnglishText* which could be readed in Structured English as "generalConcept *contains* the categories $category_1$, ...and $category_n$" where $category_i$ refers to the name of one of the categories of the categorization scheme and it follows the list of the names of categories separated by coma and "and." In the case of *Segmentation*, the definition changes the term of categorization scheme by segmentation. For example the categorization scheme of "type of authored publication" is represented as follows:

type of authored publication

Definition:	categorization scheme that *is for* authored publication
Necessity:	type of authored publication *contains* the categories journal paper, authored book and book chapter

```
 context CategorizationScheme::newRepresentation():
          Set(Representation)
   post RepresentationsCreated:
        -- new designation
        d.oclIsNew() and d.oclIsTypeOf(Designation) and
        d.meaning.oclAsType(CategorizationScheme) = self and
        d^newText(self.name, FontStyle::term,1) and

        -- new definition
        def.oclIsNew() and def.oclIsTypeOf(Definition) and
        def.meaning.oclAsType(CategorizationScheme) = self and
        def.primaryRepresentation = d and
        def^newText('categorization scheme', FontStyle::term, 1) and
        def^newText('that', FontStyle::keyword, 2) and
        def^newText('is for', FontStyle::verb, 3) and
        def^newText('the', FontStyle::keyword, 4) and
        def^newText('concept', FontStyle::term, 5) and
        def^newText(self.generalConcept.name, FontStyle::term, 6) and

        -- new necessity statement
        nes.oclIsNew() and nes.oclIsTypeOf(NecessityStatement) and
        nes.meaning.oclAsType(CategorizationScheme) = self and
        nes.primaryRepresentation = d and
        nes^newText(self.name, FontStyle::term, 1) and
        nes^newText('contains', FontStyle::verb, 2) and
        nes^newText('the', FontStyle::keyword, 3) and
        nes^newText('categories', FontStyle::term, 4) and
        nes^newText(self.category -> asSequence() -> forAll(ca|
           nes^newText(ca.name, FontStyle::term,
           self.category -> asSequence() -> indexOf(ca) *2 + 3 ))
           if self.category ->asSequence ->indexOf(ca) <
              self.category -> size()- 1
           then
              nes^newText(',', FontStyle::keyword,
              self.category -> asSequence() -> indexOf(ca) *2 + 4) and
           else if self.category ->asSequence ->indexOf(ca) =
                     self.category -> size()- 1
                then
                  nes^newText('and', FontStyle::keyword,
                     self.category -> asSequence() -> indexOf(ca)*2 + 4)
                else true
                endif
           endif)
```

7.3.7 *newRepresentation()* of reference scheme

Each instance of *ReferenceScheme* that is a schema unit is represented by a *ReferenceSchemeCaption*. The *ReferenceSchemeCaption* is associated to a sequence of instances of *StructuredEnglishText*. The sequence has the following structure: (i) for each fact type role that identifies the concept, there is an instance the value attribute of which has the name of said fact type role and the font style is 'term', and (ii) there is also an instance with the value attribute "and" and the font style 'keyword' between the previous instances. For example, the reference scheme schema unit meaning that "title" is the reference scheme of "conference edition," is represented by a *ReferenceSchemeCaption*: "title." The caption is associated to the primary representation of the object type named "conference edition."

```
context ReferenceScheme::newRepresentation():
         ReferenceSchemeCaption
  post RepresentationCreated:
       let roleNames:Sequence(String) =
          self.simplyUsedRole -> collect(ro|
             if ro.name -> notEmpty()
             then ro.name
             else ro.nounConcept.name
             endif)->asSequence
       in
       ref.oclIsNew() and ref.oclIsTypeOf(ReferenceSchemeCaption)
       and ref.meaning.oclAsType(ReferenceSchemeCaption) = self and
       ref.primaryRepresentation = self.referencedConcept.
          representation.oclAsType(Designation) and
       roleNames -> forAll(ron|
          ref^newText(ron,FontStyle::term,
             roleNames -> indexOf(ron)*2 -1) and
          if roleNames -> last() <> ron
          then ref^newText('and',FontStyle::keyword,
                   roleNames -> indexOf(ron)*2)
          else true
          endif)
```

Note that, in SBVR Structured English, reference scheme captions are incorporated as captions of the designation of the concept that incorporates this reference scheme.

7.3.8 *newRepresentation()* of structural rule schema unit

Each *StructuralRule* is represented by an instance of *NecessityStatement.*

Depending on the type of structural rule, the necessity may be attached to the designation of a concept or to a fact type form of a fact type. Three general cases have been considered:

- If the structural rule is structuring a multiplicity constraint, the necessity statement is attached to the fact type form representing the fact type that constrains;
- If the structural rule is structuring a covering or disjointness constraint, then the necessity statement is attached to the designation of the general concept;
- If the structural rule is structuring an xor-constraint, then necessity statement is attached to the designation of the concept that is constrained.

Two different examples are given below:

editor ***has*** edited book

Necessity: **each edited book *has* at least one editor** book

Necessity: each book *is* a edited book or *is* a authored book but not both

```
    context StructuralRule::newRepresentation():
             Set(Representation)
      post RepresentationsCreated:
           let form:ClosedLogicalFormulation =
                self.closedLogicalFormulation.
                oclAsType(ClosedUniversalQuantification)
           in
           let pos:Integer = form.introducedVariable ->size()*2+1
           in

           nes.oclIsNew() and nes.oclIsTypeOf(NecessityStatement) and
           nes.meaning.oclAsType(NecessityStatement) = self and
           nes^newText('each', FontStyle::keyword, 1) and
           form.introducedVariable ->forAll(va:Variable|
             nes^newText(va.rangedOverConcept.name, FontStyle::term,
             form.introducedVariable->indexOf(va)*2-1) and
             if va <> form.introducedVariable -> last()
                 then nes^newText('of a', FontStyle::keyword,
                   form.introducedVariable->indexOf(va)*2 )
                 else true endif) and

           self.isMultiplicity()
           implies
           (nes.primaryRepresentation = form.scopeFormulation.
             oclAsType(Quantification).scopeFormulation.
             oclAsType(AtomicFormulation).factType.representation.
             oclAsType(FactTypeForm) and
           nes^newText('has', FontStyle::verb, pos) and
           form.scopeFormulation^pharaphraseQuantification(nes,pos+1))
           and

           self.isDisjointAndCovering() or self.isCovering()
           implies
           (nes.primaryRepresentation = form.scopeFormulation.
             oclAsType(ExclusiveDisjunction).logicalOperand1.
             oclAsType(AtomicFormulation).factType.
             oclAsType(CategorizationFactType).factTypeRole
               -> last().nounConcept.representation.
               oclAsType(Designation) and
             form.scopeFormulation.sequenceOfCategories ->
              forAll(cat:String|
               nes^newText('is', FontStyle::verb,
                 form.scopeFormulation.sequenceOfCategories ->
                   indexOf(cat)* 4 + pos -3) and
               nes^newText('a', FontStyle::keyword,
                   form.scopeFormulation.sequenceOfCategories ->
                   indexOf(cat)* 4 + pos -2) and
               nes^newText(cat, FontStyle::term,
                   form.scopeFormulation.sequenceOfCategories ->
                   indexOf(cat)* 4 + pos -1 ) and

               if cat <> form.scopeFormulation.sequenceOfCategories ->
                 last()
               then
                 nes^newText('or', FontStyle::keyword,
                   form.scopeFormulation.sequenceOfCategories ->
                   indexOf(cat)* 4 + pos )
               else true
               endif) and
           if self.isDisjointAndCovering() then
             nes^newText('but not both', FontStyle::keyword,
             form.scopeFormulation.sequenceOfCategories -> size()* 4
               + pos + 1 )
           else
             true
           endif) and

           self.isDisjoint()
           implies
           (nes^newText('that', FontStyle::keyword, pos + 1) and
           nes^newText('is', FontStyle::verb, pos + 2) and
           nes^newText('a', FontStyle::keyword, pos + 3) and
           nes^newText(form.scopeFormulation.sequenceOfCategories() ->
           first(), FontStyle::term, pos + 5) and
           nes^newText('neither', FontStyle::keyword, pos + 6) and
           nes^newText('is', FontStyle::verb, pos + 7) and
           nes^newText('a', FontStyle::keyword, pos + 8) and
           form.scopeFormulation.sequenceOfCategories ->
             excludes(form.scopeFormulation.sequenceOfCategories() ->
             first()) -> forAll(cat:String |
               nes^newText(cat, FontStyle::term,
               form.scope.sequenceOfCategories() ->indexOf(cat)*2-7)
               and
               if cat <> form.scopeFormulation.sequenceOfCategories()
                 -> last()
               then
                 nes^newText('nor a', FontStyle::keyword,
                 form.scope.sequenceOfCategories() ->indexOf(cat)*2-7)
               else true
               endif)) and

           self.isXOR()
           implies
           (nes^newText('that', FontStyle::keyword, pos + 1) and
           nes^newText(form.introducedVariable.restrictingFormulation.
           oclAsType(AtomicFormulation).factType.name, FontStyle::verb,
           pos + 2)
           and nes^newText('a', FontStyle::keyword, pos + 3) and
           nes^newText(form.introducedVariable.restrictingFormulation.
           oclAsType(AtomicFormulation).factType.factTypeRole ->
           last().nounConcept.name, FontStyle::term, pos + 4) and
           nes^newText('does not', FontStyle::keyword, pos + 5) and
           nes^newText('have', FontStyle::verb, pos + 6) and
           nes^newText('either', FontStyle::keyword, pos + 7) and
           form.scopeFormulation.restrictedFactTypes ->
             forAll(ft:FactType |
               nes^newText(ft.factTypeRole->last().nounConcept.name,
                  FontStyle::term,
                  form.scopeFormulation.restrictedFactTypes ->
                  indexOf(ft) * 2 + 7) and
                if ft <> form.scopeFormulation.restrictedFactTypes ->
                  last()
                then
                  nes^newText('or', FontStyle::keyword,
                  form.scopeFormulation.restrictedFactTypes ->
                  indexOf(ft) * 2 + 8)
                else true
                endif))
```

This means that a closed universal quantification starts with the **each** keyword followed by the name of the concept that the variable, introduced by the quantification, ranges over. The rest of the statement is structured depending on the logical formulation associated to the universal quantification.

Note that the *sequenceOfCategories()* and *restrictedFactTypes()* operations were defined in Sections 6.3.2.8 and 6.3.2.9 of Chapter 6, respectively.

Additionally, to facilitate the pharaphrasing of the different subtypes of *Quantification*, the *pharaphraseQuantification(nes:Necessity, iniPos:Integer)* operation has been defined in *Quantification*. It constrains, depending on the subtype of *Quantification*, the instances of *StructuredEnglishText* included in the statement that represents a multiplicity rule. The specification of the operation is defined abstract and redefined in the subtypes of *Quantification* as follows:

```
context AtLeastNQuantification::pharaphraseQuantification(
         ne:NecessityStatement, iniPos:Integer):
         Set(StructuredEnglishText)
  post:  nes^newText('at least', FontStyle::keyword, iniPos) and
         nes^newText(self.minimumCardinality, FontStyle::term,
           iniPos + 1) and nes^newText(
           self.introducedVariable.rangedOverConcept.name,
             FontStyle::term, iniPos + 2)

context AtMostNQuantification::pharaphraseQuantification(
         ne:NecessityStatement, iniPos:Integer):
         Set(StructuredEnglishText)
  post:  nes^newText('at most', FontStyle::keyword, iniPos) and
         nes^newText(self.maximumCardinality, FontStyle::term,
           iniPos + 1) and nes^newText(
           self.introducedVariable.rangedOverConcept.name,
           FontStyle::term, iniPos + 2)

context ExactlyNQuantification::pharaphraseQuantification(
         ne:NecessityStatement, iniPos:Integer):
         Set(StructuredEnglishText)
  post:  nes^newText('exactly', FontStyle::keyword, iniPos) and
         nes^newText(self.cardinality, FontStyle::term, iniPos+1)
         and nes^newText(
             self.introducedVariable.rangedOverConcept.name,
             FontStyle::term, iniPos + 2)
```

```
context NumericRangeQuantification::pharaphraseQuantification(
        ne:NecessityStatement, iniPos:Integer):
        Set(StructuredEnglishText)
  post: nes^newText('at least', FontStyle::keyword, iniPos) and
        nes^newText(self.minimumCardinality, FontStyle::term,
          iniPos + 1) and nes^newText(
            self.introducedVariable.rangedOverConcept.name,
            FontStyle::term, iniPos + 2) and
        nes^newText('and at most', FontStyle::keyword, iniPos+3)
        and nes^newText(self.maximumCardinality, FontStyle::term,
            iniPos + 4) and nes^newText(
            self.introducedVariable.rangedOverConcept.name,
            FontStyle::term, iniPos + 5)
```

7.4 *vocabularyEntry()* operation

The *vocabularyEntry()* query operation applied to an instance of a subtype of *Meaning* gives the vocabulary entry of said meaning in the Structured English notation. The specification, in OCL, of the operation is the following:

```
context Meaning::vocabularyEntry():Set(Tuple(
        primaryRepresentation: Sequence(Tuple(
          text:String, font:FontStyle)),
        captions:Set(Tuple(
          captionType:CaptionType,
          captionValue:Sequence(Tuple(text:String,
              font:FontStyle))))))

  body: let primary:PrimaryRepresentation =
          if self.representation.oclAsType(PrimaryRrepesentation)
              -> notEmpty()
          then
            self.representation->any(pr|
              pr.oclIsKindOf(PrimaryRepresentation))
              oclAsType(PrimaryRepresentation)
          else
            self.representation->any(pr|
              pr.oclIsKindOf(Caption)).
              oclAsType(Caption).primaryRepresentation)
          endif in

        Tuple{
          primaryRepresentation: primary.representation()
          captions:
            primary.caption->any(c|c.oclIsTypeOf(Definition))
                 .oclAsType(Definition)->collect(
                 captionRepresentation())-> union(
            primary.caption->any(c|
                 c.oclIsTypeOf(GeneralConceptCaption))
                 .oclAsType(GeneralConceptCaption)
                 ->collect(captionRepresentation())->union (
            primary.caption->any(c|
                 c.oclIsTypeOf(ConceptTypeCaption))
                 .oclAsType(ConceptTypeCaption)->
                 collect(captionRepresentation())-> union(
            primary.caption->any(c|
                 c.oclIsTypeOf(NecessityStatement))
                      .oclAsType(NecessityStatement)) ->
                 collect(captionRepresentation())))))
```

This means that an SBVR Structured English vocabulary entry is composed by its primary entry followed by its set of captions as described in Section 7.2.2.1 *Vocabulary Entries.*

The operation *representation()* defined in the context of *PrimaryRepresentation* gives the sequence of Structured English text that represents such primary representation.

```
context PrimaryRepresentation::representation():Sequence(
         Tuple(text:String,font:FontStyle))
  body:  self.structuredEnglishText -> collect(st|
           Tuple{text:st.value,font:st.font})
```

The operation *captionRepresentation()* defined in the context of *Caption* gives the caption type and sequence of Structured English text that represents such primary representation.

```
context Caption::captionRepresentation():TupleType(
         captionType:CaptionType,
         captionValue:Sequence(TupleType(text:String,
                                         font:FontStyle)))
  body:  Tuple{captionType:captionType(),
           captionValue: self.structuredEnglishText -> collect(st|
           Tuple{text:st.value,font:st.font})}
```

The *captionType()* operation is defined abstract in *Caption* and redefined in its subtypes as follows:

```
context Definition::captionType():CaptionType
  body: CaptionType::Definition

context GeneralConceptCaption::captionType():CaptionType
  body: CaptionType::General_concept

context ConceptTypeCaption::captionType():CaptionType
  body: CaptionType::Concept_type

context NecessityStatement::captionType():CaptionType
  body: CaptionType::Necessity

context ReferenceSchemeCaption::captionType():CaptionType
  body: CaptionType::Reference_scheme
```

CaptionType is defined as an enumeration of: *Definition, General_concept, Concept_type, Necessity and Reference_scheme*.

Concerning its implementation, the specification *seVocabulary* has a quite straightforward implementation using the methods of the operations *newRepresentation* (Section 7.3).

The implementation of the methods of *newRepresentation* are described in Appendix J in the procedural language described in USE (Gogolla, Büttner and Richters 2007).

7.5 DBLP vocabulary in SBVR Structured English notation

The instances of the SBVR Representations created by the application of the newRepresentation method to the SBVR Meanings instances of the DBLP example are shown in Appendix K. The result of the query `Meaning.allInstances()->collect(me| me.vocabularyEntry())` after reformatting the font style:

Female

Male

Gender
Definition: Female or Male

Natural

String

Year

acronym
Concept type: String
Concept type: role

authored book
General concept: authored publication
General concept: book

authored publication
Definition: authored book or book chapter or journal paper
General concept: publication
Necessity: each book *is* a authored book or *is* a book chapter or *is* a

journal paper
Necessity: each authored publication that *is* a authored book *is* not either a book chapter or a journal paper
Necessity: each authored publication that *is* a book chapter *is* not either a authored book or a journal paper

authorship
Definition: actuality that an author *has* an authored publication

book
Definition: edited book or authored book
General concept: publication
Necessity: each book *is* a edited book or *is* a authored book but not both
Reference scheme: isbn

book chapter
General concept: authored publication

book section

book series
Reference scheme: id

book series issue	
General concept:	book
city	
Concept type:	String
Concept type:	role
conference edition	
Reference scheme:	title
conference series	
Reference scheme:	name
country	
Concept type:	String
Concept type:	role
edited book	
General concept:	book
General concept:	publication
edition	
Concept type:	String
Concept type:	role
editorship	
Definition:	actuality that an editor *has* an edited book
end page	
Concept type:	Natural
Concept type:	role
gender	
Concept type:	Gender
Concept type:	role
home page	
Concept type:	String
Concept type:	role
id	
Concept type:	String
Concept type:	role
ini page	
Concept type:	Natural
Concept type:	role
isbn	
Concept type:	String
Concept type:	role
issn	
Concept type:	String
Concept type:	role

journal

Reference scheme:	issn
Reference scheme:	title

journal issue

journal paper

General concept:	authored publication

journal section

journal volume

month

Concept type:	String
Concept type:	role

name

Concept type:	String
Concept type:	role

num pages

Concept type:	Natural
Concept type:	role

num publications

Concept type:	Natural
Concept type:	role

number

Concept type:	Natural
Concept type:	role

order

Concept type:	Natural
Concept type:	role

person

Reference scheme:	name

publication

Definition:	edited book or authored publication
Necessity:	each publication *is* a edited book or *is* a authored publication but not both

publication year

Concept type:	Year
Concept type:	role

publisher

Concept type:	String
Concept type:	role

title

Concept type:	String
Concept type:	role

type of authored publication

Definition:	categorization scheme that *is for* authored publication
Necessity:	type of authored publication *contains* the categories journal paper, authored book and book chapter

volume

Concept type:	Natural
Concept type:	role

year

Concept type:	Year
Concept type:	role

author *has* authored publication

Concept type:	associative fact type
Necessity:	each authored publication *has* at least one author

authorship *has* order

Concept type:	is-property-of fact type
Necessity:	each authorship *has* exactly one order

book *has* home page

Concept type:	is-property-of fact type
Necessity:	each book *has* at most one home page

book *has* isbn

Concept type:	is-property-of fact type
Necessity:	each book *has* exactly one isbn

book *has* num pages

Concept type:	is-property-of fact type
Necessity:	each book *has* exactly one num pages

book *has* publication year

Concept type:	is-property-of fact type
Necessity:	each book *has* exactly one publication year

book *has* publisher

Concept type:	is-property-of fact type
Necessity:	each book *has* exactly one publisher

book chapter *being conference paper*

Concept type:	characteristic

book chapter *has* end page

Concept type:	is-property-of fact type
Necessity:	each book chapter *has* exactly one end page

book chapter *has* ini page

Concept type:	is-property-of fact type
Necessity:	each book chapter *has* exactly one ini page

book chapter *is part of* book section

Concept type:	associative fact type
Necessity:	each book chapter *is part of* at most one book section
Necessity:	each book section *has* at least one book chapter

book chapter *is part of* book series issue

Concept type:	associative fact type
Necessity:	each book chapter *is part of* at most one book series issue
Necessity:	each book series issue *has* at least one book chapter

book chapter *is part of* edited book

Concept type:	associative fact type
Necessity:	each book chapter *is part of* at most one edited book
Necessity:	each edited book *has* at least one book chapter

book section *has* order

Concept type:	is-property-of fact type
Necessity:	each book section *has* exactly one order

book section *has* title

Concept type:	is-property-of fact type
Necessity:	each book section *has* exactly one title

book section *is part of* edited book

Concept type:	associative fact type
Necessity:	each book section *is part of* at most one edited book

book series *has* id

Concept type:	is-property-of fact type
Necessity:	each book series *has* exactly one id

book series *has* publisher

Concept type:	is-property-of fact type
Necessity:	each book series *has* exactly one publisher

book series *includes* book series issue

Concept type:	partitive fact type
Necessity:	each book series issue *has* exactly one book series

book series issue *has* number

Concept type:	is-property-of fact type
Necessity:	each book series issue *has* exactly one number

conference edition *has* city

Concept type:	is-property-of fact type
Necessity:	each conference edition *has* exactly one city

conference edition *has* country

Concept type:	is-property-of fact type
Necessity:	each conference edition *has* exactly one country

conference edition *has* home page

Concept type:	is-property-of fact type
Necessity:	each conference edition *has* at most one home page

conference edition *has* title

Concept type:	is-property-of fact type
Necessity:	each conference edition *has* exactly one title

conference edition *has* year

Concept type:	is-property-of fact type
Necessity:	each conference edition *has* exactly one year

conference edition *is published in* book series issue

Concept type:	associative fact type
Necessity:	each book series issue *has* at most one conference edition
Necessity:	each conference edition *is published in* at most one book series issue
Necessity:	each conference edition that *is published in* a book series issue *is* not *published in* a edited book nor *is published in* a journal issue

conference edition *is published in* edited book

Concept type:	associative fact type
Necessity:	each conference edition *is published in* at most one edited book
Necessity:	each edited book *has* at most one conference edition
Necessity:	each conference edition that *is published in* a edited book *is* not *published in* a book series issue nor *is published in* a journal issue

conference edition *is published in* journal issue

Concept type:	associative fact type
Necessity:	each conference edition *is published in* at most one journal issue
Necessity:	each journal issue *has* at most one conference edition
Necessity:	each conference edition that *is published in* a journal issue *is* not *published in* a edited book nor *is published in* a book series issue

conference series *has* acronym

Concept type:	is-property-of fact type
Necessity:	each conference series *has* exactly one acronym

conference series *has* name

Concept type:	is-property-of fact type
Necessity:	each conference series *has* exactly one name

conference series *includes* conference edition

Concept type:	partitive fact type
Necessity:	each conference edition *has* exactly one conference series

editor *has* edited book

Concept type:	associative fact type
Necessity:	each edited book *has* at least one editor

editorship *has* order

Concept type:	is-property-of fact type
Necessity:	each editorship *has* exactly one order

journal *has* issn

Concept type:	is-property-of fact type
Necessity:	each journal *has* exactly one issn

journal *has* title

Concept type:	is-property-of fact type
Necessity:	each journal *has* exactly one title

journal *includes* journal volume

Concept type:	partitive fact type
Necessity:	each journal volume *has* exactly one journal

journal issue *has* month
Concept type: is-property-of fact type
Necessity: each journal issue *has* at most one month

journal issue *has* num pages
Concept type: is-property-of fact type
Necessity: each journal issue *has* exactly one num pages

journal issue *has* number
Concept type: is-property-of fact type
Necessity: each journal issue *has* exactly one number

journal issue *has* year
Concept type: is-property-of fact type
Necessity: each journal issue *has* exactly one year

journal issue *includes* journal section
Concept type: partitive fact type
Necessity: each journal section *has* exactly one journal issue

journal paper *being conference paper*
Concept type: characteristic

journal paper *has* end page
Concept type: is-property-of fact type
Necessity: each journal paper *has* exactly one end page

journal paper *has* ini page
Concept type: is-property-of fact type
Necessity: each journal paper *has* exactly one ini page

journal paper *is part of* journal issue
Concept type: associative fact type
Necessity: each journal issue *has* at least one journal paper
Necessity: each journal paper *is part of* exactly one journal issue

journal paper *is part of* journal section
Concept type: associative fact type
Necessity: each journal paper *is part of* at most one journal section
Necessity: each journal section *has* at least one journal paper

journal section *has* order
Concept type: is-property-of fact type
Necessity: each journal section *has* exactly one order

journal section *has* title
Concept type: is-property-of fact type
Necessity: each journal section *has* exactly one title

journal volume *has* volume
Concept type: is-property-of fact type
Necessity: each journal volume *has* exactly one volume

journal volume *includes* journal issue
Concept type: partitive fact type
Necessity: each journal issue *has* exactly one journal volume

person ***has*** gender

Concept type:	is-property-of fact type
Necessity:	each person ***has*** exactly one gender

person ***has*** home page

Concept type:	is-property-of fact type
Necessity:	each person ***has*** at most one home page

person ***has*** name

Concept type:	is-property-of fact type
Necessity:	each person ***has*** exactly one name

person ***has*** num publications

Concept type:	is-property-of fact type
Necessity:	each person ***has*** exactly one num publications

person ***publishes*** publication

Concept type:	associative fact type
Necessity:	each person ***publishes*** at least one publication
Necessity:	each publication ***has*** at least one person

publication ***has*** edition

Concept type:	is-property-of fact type
Necessity:	each publication ***has*** exactly one edition

publication ***has*** title

Concept type:	is-property-of fact type
Necessity:	each publication ***has*** exactly one title

publication ***has*** year

Concept type:	is-property-of fact type
Necessity:	each publication ***has*** exactly one year

8 Contributions and future research

This chapter summarizes the main contributions of the research and approach presented and points out the areas of future research.

8.1 Contributions

8.1.1 A generic object-oriented approach to the translation between MOF metaschemas

This thesis presents a new generic approach to the translation between MOF metaschemas. Various proposals describe generic schema translation, as summarized in Chapter 2. The approach proposed in this thesis enriches the previous research in several aspects.

First of all, the generic translations between MOF schemas are defined, at conceptual level, by exclusively using object-oriented concepts, particularly the use of operations (and their refinements) and invariants, both formalized in OCL. The translations mappings can be used to check that one schema is a translation of another, and also to translate one into another one, in both directions. The translation mappings are defined declaratively by means of preconditions and postconditions and invariants and they can be implemented in any suitable language. The approach leverages the object-oriented constructs embedded in MOF metaschemas to achieve the goals of the object-oriented software development in the schema translation problem. This is one of the main advantages of the approach presented in this thesis.

The research is framed in the context of MOF, UML and OCL. The benefit is that there is a wide set of available tools to implement the approach. For demonstration purposes, this work uses one of these tools, USE (Gogolla, Büttner and Richters 2007), in both the declarative and the procedural parts of the mappings. Other tools might be appropriate for other projects.

The approach to translate schemas consists in two steps: the structuring of metaschemas in schema units and the establishment of relationship among schema units of different metaschemas.

Even though various authors proposed similar ideas, the approach to structuring metaschemas with schema units has not been previously explicitly formulated as in this thesis. Schema units correspond to semantic units of knowledge within a schema and each one consists of a set of structural schema elements. Their definition, precedence relationship among them and the characterization objects to create them, in each schema, is independent from their use in any schema management operation. This means that they are defined only once for each metaschema.

The difficulty of finding the relationship among different metaschemas is clearly reduced by using the defined schema units. On the one hand, the number of translation mappings to define between two metaschemas is at most the number of schema units created; which is much less than the number of structural elements of each metaschema. On the other hand, the translation mapping definition is split into two simpler parts: one between the schema units of one side and the characterization objects of the other side, and one between the characterization object of the second side and its schema units. Additionally, the precedence relationship among the schema units ensures the executability of the translation and the translation mapping postconditions defined ensure the consistency of the mappings.

8.1.2 The application to the translation between UML and SBVR

The generic approach has been applied to the particular case of translating UML schemas to SBVR and vice versa.

An important issue to take into account when defining translation mappings is the size and complexity of the metaschemas. In the particular case of the application of the approach presented to the translation between UML and SBVR, the most challenging work has been the definition of the schema units and the precedence relationship among them. Moreover, the equivalences among schema units of different metaschemas were easily defined once schema units were defined.

8.1.3 The transformation of SBVR to Structured English

There are two additional contributions, derived from the non-existence of a straightforward writing in SBVR Structured English notation from the instances of SBVR metamodel: (i) the definition of a very simple metamodel to support the SBVR Structured English notation, and (ii) the definition of operations to obtain the instances of such metamodel from the defined SBVR schema units.

The splitting of SBVR between meanings and representations proposed in this work has two benefits: (i) the exclusion of SBVR representations facilitates the translation mapping definition and (ii) it is easier to accommodate new natural language notations, as

Attempto Controled English (Wagner, Lukichev, Fuchs and Spreeuwenberg 2005) and RuleSpeak English (Object Management Group 2008) in the translation approach.

8.2 Future research

The work reported in this thesis may be further researched in various directions: (1) facilitating the definition of translation mappings; (2) defining a generic/super schema; (3) including the translation of instances; (4) defining other schema management operators; (5) translating OCL to SBVR; (6) translating behavioral schemas; and (7) representing UML and SBVR in other languages and notations. Each research line is briefly sketched in the following.

8.2.1 Facilitating the definition of translation mappings

Two operations, *target-equivalents* and *includedIn-target*, have been used to define translation mappings. This is, given two schemas S_1 and S_2, the S_2*equivalents* of a schema unit s_1 represented by an instance of S_1 are the set of schema units of S_2 whose *isIncludedIn*S_1 results in s_1. Moreover, given a schema unit s_1 represented by an instance of S_1 the *isIncludedIn*S_2 gives an s_2 schema unit, represented by an instance of S_2, whose S_1*equivalents* includes s1.

Both operations are complementaries. Therefore, an interesting research is automatically deriving the *includedIn-target* operations from the *target-equivalents* ones. If this is possible, only half of the postconditions definitions will have to be provided by designers.

8.2.2 Defining a generic/super schema

Approaches that provide a specification of the *schemaGen* (*modelGen*) operator as Papotti and Torlone (2005), Bernstein, Melnik and Mork (2005), Hainaut (2006), Boyd and McBrien (2005) and Atzeni, Cappellari, Torlone, Bernstein and Gianforme (2008), among others, rely on some kind of pivot model. The concept was introduced in the early approach (prior to the definition of the model management concept), MDM, by Atzeni and Torlone (1996). It is an elegant way to solve the combinatorial explosion in situations in which mappings must be developed from any M schemas to N schemas. Theoretically, instead of formalizing NxM distinct mappings, only M+N mappings are required.

Atzeni, Capellari and Bernstein (2005), Atzeni, Cappellari, Torlone, Bernstein and Gianforme (2008) and Hainaut (2006), among others, define the concept of *supermodel* or *generic model* as a model that has constructs corresponding to all the *metaconstructs* known to the system. They define a limited set of generic (i.e., model independent) *metaconstructs*: *lexical*, *abstract*, *aggregation*, *generalization* and *function*, in the case of Atzeni, Capellari and Bernstein (2005) and Atzeni, Cappellari, Torlone, Bernstein and Gianforme (2008) and *schema*, *entity type*, *simple domain*, *atomic attribute*, *primary identifier*, *secondary identifier*, *reference group* and *GER names*, in the case of Hainaut (2006). Therefore, any two models are translations of each other if there is any set of transformations in the supermodel that translate one model to another, where both models are described in terms of the supermodel constructs.

A possible research is to study the alternative of defining "generic" schema units and to establish the correspondence between the schema units defined in each metamodel and the "generic" ones. In this context, the translations are defined only among the "generic" schema units.

The main concern of this research, as suggested in the previous section, is the complexity for the "generic" metaschema to cover all type of metaconstructs.

8.2.3 Translation of instances

Atzeni, Capellari and Bernstein (2005) and Atzeni, Cappellari, Torlone, Bernstein and Gianforme (2008) include in their approaches the possibility of translating not only metaschemas but also instances of them. Their approaches include dictionaries that contain the description of each generic construct and the description of each element of a model in terms of the generic construct. Additionally, the dictionaries include the instances of the model, also as instances of the generic constructs. Each transformation in the supermodel is implemented in such a way that also generates the changes in the instances of the supermodel.

From the conceptual point of view, the generation of instances of a model is close to works on the area of validation of models that generate instances of models to prove their correctness (Gogolla, Bohling and Richters 2005 and Rull, Farré, Teniente and Urpí 2008).

8.2.4 Defining other schema management operators

Chapter 2 contains four descriptions of families of problems found in schema management: (i) schema transformation, (ii) schema integration, (iii) schema translation, and (iii) propagation of changes between schemas due to evolution. In order to solve such problems, schema management proposes the definition of basic schema management operators: *match*, *compose*, *merge*, *diff* and *modelGen* (*schemaGen*).

This thesis proposes a specification of the *schemaGen* operator at a conceptual level. The operator is defined in terms of schema units which have been defined independently from the translation mapping definitions.

Schema units and the translation mappings defined as postconditions may be used for the specification of other schema management operators. For example, given two schemas m_1 and m_2 and the mapping between both, the *diff* operator gives a third schema m_3 that is a subset of m_1 that do not participate in the mapping. Possibly, m_3 may be defined, in terms of the schema units defined, as the union of the instances of untranslatable schema units of m_1 to m_2 and all those instances of schema units of m_1 that may not consistenly be translated to m_2.

8.2.5 Translation of OCL to SBVR

This thesis considers a limited set of UML constraints to be translated to SBVR. However, an extension of the work presented in this thesis would be the study of translating UML including all the possible OCL expressions to SBVR. A first attempt in this direction has already been done by Pau and Cabot (2008), where the authors present the pharaprasing of OCL to SBVR.

By including the whole OCL metamodel in the translation approach, the constraint schema unit becomes very complex. A mechanism to structure in a different way the constraint schema unit should be provided. Then, translations from a part of an OCL expression to SBVR and vice versa should be defined.

8.2.6 Translation of behavioral schemas

This thesis has only included the structural part of conceptual schemas for the translation between UML and SBVR. It would be of interest to study translations between behavioral schemas. An alternative of representing, in UML, the behavioral schema is to represent domain and action request events as a special type of entities (Olivé 2007). On the other hand, SBVR also distinguishes between structural business rules and operative business rules. The study of the relationship between UML events and SBVR deserves further research.

8.2.7 Representing UML and SBVR in other languages and notations

This thesis is a first step towards a tighter integration of the business communities and software UML communities. As a future research, it would be interesting to implement the approach in a tool framework in order to evaluate the quality of the resulting SBVR Structured English expressions in industrial cases. As part of this goal SBVR Structured Catalan or Spanish (among others) notations should be developed.

Moreover, the operations to represent SBVR instances should be provided not only in SBVR Structured English notation but also to other notations such as the Business Rule Speak notation (Ross 2003).

Finally, the translation from UML to other business rules languages such as Controlled English Rule Language (Wagner, Lukichev, Fuchs and Spreeuwenberg 2005) should also be studied.

References

Adya, A., Blakeley, J., Melnik, S. & Muralidhar, S. 2007, "Anatomy of the ADO.NET entity framework," *ACM SIGMOD 2007*, pp. 877-888.

Altova 2008, *Altova*. http://www.altova.com.

Atzeni, P. 2007, "Schema and data translation: a personal perspective," *Advances in Databases and Information Systems,* LNCS 4690, pp. 14-27.

Atzeni, P., Cappellari, P. & Bernstein, P.A. 2005, "A multilevel dictionary for model management." *ER 2005*, LNCS 3716, pp. 160-175.

Atzeni, P., Cappellari, P. & Bernstein, P.A. 2006, "Model-independent schema and data translation," *EDBT 2006*, LNCS 3896, pp. 368-385.

Atzeni, P., Cappellari, P. & Gianforme, G. 2007, "MIDST: model independent schema and data translation," *ACM SIGMOD 2007*, pp. 1134-1136.

Atzeni, P., Cappellari, P. Torlone, R., Bernstein, P.A. & Gianforme, G. 2008, "Model-independent schema translation," *VLDB 2008*, vol. 17, pp. 1347-1370.

Atzeni, P. & Torlone, R. 1996, "Management of multiple models in an extensible database design tool," *EDBT 1996*, LNCS 1057, pp. 79-95.

Batini, C., Lenzerini, M. & Navathe, S.B. 1986, "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys,* vol. 18, no. 4, pp. 323-364.

Becket, D. 2004, "W3C RDF/XML Syntax Specification (Revised)." http://www.w3.org/TR/rdf-syntax-grammar/.Benedikt, M., Chan, C.Y., Fan, W. & Rastogi, R. 2003, "Capturing both types and constraints in data integration," *ACM SIGMOD International Conference on Management of Data 2003*, pp. 277-288.

Berler, M., Eastman, J., Jordan, D., Craig, R., Schadow, O., Stanienda, T. & Velez, F. 2000, Cattel, R.G.G. & Barry, D.K. (eds.), *The Object Data Standard: ODMG 3.0,* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

Bergamaschi, S., Castano, S.V. & Vincini, M. 1999, "Semantic integration of semistructured and structured data sources," *ACM SIGMOD Record,* vol. 28, no. 1, pp. 54-59.

Bernstein, P.A. 2003, "Applying model management to classical meta data problems," *CIDR 2003*, pp. 209-220.

Bernstein, P.A., Haas, L.M., Jarke, M., Rahm, E. & Wiederhold, G. 2000, "Panel: Is generic metadata management feasible?" *VLDB 2000*, pp. 660-662.

Bernstein, P.A., Halevy, A.Y. & Pottinger, R.A. 2000, "A vision for management of complex models," *ACM SIGMOD Record,* vol. 29, no. 4, pp. 55-63.

Bernstein, P.A. & Melnik, S. 2007, "Model management 2.0: manipulating richer mappings," *ACM SIGMOD 2007*, pp. 1-12.

Bernstein, P.A., Melnik, S. & Mork, P. 2005, "Interactive schema translation with instance-level mappings," *VLDB 2005*, pp. 1283-1286.

Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I. & Lindow, A. 2006, "Model transformations? Transformation models!" *MoDELS 2006*, LNCS 4199, pp. 440-453.

Bitpipe 2007, *Query Tools: Products*. http://www.bitpipe.com/plist/term/Query-Tool.html.

Bloor, M.S. & Owen, J. 1994, *Product Data Exchange,* UCL Press, Ltd.

Boronat, A., Carsí, J.A. & Ramos, I. 2006, "Algebraic specification of a model transformation engine," *FASE at the ETAPS 2006*, LNCS 3922, pp. 262-277.

Boronat, A., Carsí, J.A. & Ramos, I. 2005a, "Automatic reengineering in MDA using rewriting logic as transformation engine," *CSMR 2005*, pp. 228-231.

Boronat, A., Carsí, J.A. & Ramos, I. 2005b, "MOMENT: a formal Model management tool," *Summer School on Generative and Transformational Techniques in Software Engineering.* Braga, Portugal.

Bowers, S. & Delcambre, L. 2006, "Using the uni-level description (ULD) to support data-model interoperability." *Data & Knowledge Engineering*, vol. 59, pp. 511-533.

Boyd, M. & McBrien, P. 2005, "Comparing and transforming between data models via an intermediate hypergraph data model" *Journal on Data Semantics IV*, LNCS 3730, pp.69-109.

Buneman, P., Davidson, S.B. & Kosky, A. 1998, "Semantics of database transformations," *Selected Papers from a Workshop on Semantics in Databases*, LNCS 1358, pp. 55-91.

Business Rule Group 2003, *The Business Rule Manifesto. v.2.0*, Ronald G. Ross, http://www.businessrulesgroup.org/brmanifesto.htm.

Calvanese, D., De Giacomo, G. & Lenzerini, M. 2001. "Ontology of integration and integration of ontologies." *Description Logic Workshop (DL 2001)*, pp. 10-19.

Carey, M. 2006, "Data delivery in a service-oriented world: the BEA aquaLogic data services platform," *ACM SIGMOD 2006*, pp. 695-705.

Claypool, K.T. 2002, *Managing Schema Change in a Heterogeneous Environment. Ph. D. thesis*, Worcester Polytechnic Institute.

Chen, P. 2003, "The entity-relationship model: toward a unified view of data." *ACM TODS 2003*, vol. 1, no. 1, pp. 9-36.

Corcho, O., Fernández-López, M. & Gómez-Pérez, A. 2003, "Methodologies, tools and languages for building ontologies. Where is their meeting point?" *Data & Knowledge Engineering*, vol. 46, pp. 41-64.

Costal, D., Gómez, C., Queralt, A., Raventós, R. & Teniente, E. 2008 "Improving the definition of general constraints in UML," *Software and Systems Modeling*, vol. 7, no. 4, pp. 469-486.

Czarnecki, K. & Helsen, S. 2006, "Feature-based survey of model transformation approaches," *IBM Systems Journal,* vol. 45, no. 3, pp. 621-645.

Date, C.J. 2000, *WHAT Not HOW. The Business Rules Approach to Application Development.* Addison-Wesley. USA.

Davidson, S.B., Buneman, P., Harker, S., Overton, C. & Tannen, V. 1999, "Transforming and integrating biomedical data using Kleisli: a perspective," *SIGBIO Newsletter,* vol. 19, no. 2, pp. 8-13.

Eclipse 2008, *Eclipse Modeling Framework Project*, http://www.eclipse.org/emf/.

ETL 2007, *ETL Tool Survey 2006-2007*, http://www.etltool.com/.

Fagin, R., Kolaitis, P.G., Popa, L. & Tan, W.C. 2005, "Composing schema mappings: Second-order dependencies to the rescue." *ACM Trans. Database Syst.* vol. 30, no. 4, pp. 994-1055.

Rull, G., Farre, C., Teniente, E. & Urpí, T. 2008, "Validation of mappings between schemas." *Data & Knowledge Engineering*, vol. 66, no. 3, pp.414-437.

Fernandez, M.F., Florescu, D., Kang, J., Levy, A.Y. & Suciu, D. 1998, "Overview of Strudel: a web-site management system," *Networking and Information Systems,* vol. 1, no. 1, pp. 115-140.

Fuxman, A., Hernández, M.A., Ho, H., Miller, R.J., Papotti, P. & Popa, L. 2006, "Nested mappings: schema mapping reloaded," *VLDB 2006*, pp. 67-68.

Gangemi, A., Guarino, N., Masolo, C. & Oltramari, A. 2003, "Sweetening WORDNET with DOLCE," *AI Magazine,* vol. 24, no. 3, pp. 13-24.

Goedertier, S., Mues, C. & Vanthienen, J. 2007, "Specifying Process-Aware Access Control Rules in SBVR," RuleML 2007, LNCS 4824, pp. 39-52.

Gogolla, M. 2005, *Tales of ER and RE Syntax and Semantics*, Dagstuhl Seminar Proceedings 05161.

Gogolla, M., Bohling, J. & Richters, M. 2005, "Validating UML and OCL models in USE by automatic snapshot generation," *Software and System Modeling*, vol. 4, no. 4, pp. 386-398.

Gogolla, M., Büttner, F. & Richters, M. 2007, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, pp. 27-34.

Gogolla, M., Lindow, A., Richters, M. & Ziemann, P. 2002, "Metamodel transformation of data models. Position paper," *WISME at the UML 2002*.

Griethuysen, J.J. van, (ed.) 1982, *Concepts and terminology for the conceptual schema and the information base*. ISO TC97/SC5/WG3.

Grubb, P. & Takang, A.A. 2003, *Software Maintenance: Concepts and Practice*. World Scientific Publishing, Singapore.

Grunske, L., Geiger, L. & Lawley, M. 2005, "A graphical specification of model transformations with triple graph grammars," *Model Driven Architecture - Foundations and Applications*, LNCS 3748, pp.284-298.

Gruser, J.R., Raschid, L., Vidal, M.E. & Bright, L. 1998, "Wrapper generation for web accessible data sources," *COOPIS 1998*, pp. 14-23.

Hainaut, J-L. 1996, "Specification preservation in schema transformations - application to semantics and statistics," *Data & Knowledge Engineering*, vol. 19. pp. 99-134.

Hainaut, J-L. 2006, "The Transformational approach to database engineering." *GTTSE 2005*, LNCS 4143, pp. 95-143.

Halevy, A.Y. 2001, "Answering queries using views: A survey," *VLDB 2001*, LNCS 10, pp. 270-294.

Halevy, A.Y., Ashish, N., Bitton, D., Carey, M., Draper, D., Pollock, J., Rosenthal, A. & Sikka, V. 2005, "Enterprise information integration: successes, challenges and controversies," *ACM SIGMOD 2005*, pp. 778-787.

Halpin, T. 2001, *Information Modeling and Relational Databases*. Morgan Kaufman, San Francisco, CA, USA.

Halle, B.V. 1994, "Back to business rule basics," *Database Programming and Design*, pp. 15-18

Halle, B.V. 2001, "Building a business rule system," *Data Management Review*.

Harold, E.R. 2001, *The XML Bible*, Hungry Minds Inc. New York, USA.

Herbst, H. 1997, *Business Rule-Oriented Conceptual Modeling*, Physica-Verlag, Germany.

Hibernate 2007, *Hibernate Tools*. www.hibernate.org.

Hull, R. 1986, " Relative information capacity of simple relational database schemata," *SIAM J. Computing*, vol.15, no. 3, pp. 856-886.

Jarrar, M. 2007, "Towards automated reasoning on ORM schemes. Mapping ORM into the DLRidf Description Logic." *ER 2007*, LNCS 4801, pp. 181-197.

Kalfoglou, Y. & Schorlemmer, M. 2003, "Ontology mapping: the state of the art," *Knowledge Engineering Review,* vol. 18, no. 1, pp. 1-31.

Kementsietsidis, A., Arenas, M. & Miller, R.J. 2003, "Mapping data in peer-to-peer systems: semantics and algorithmic issues," *ACM SIGMOD International conference on Management of data 2003*, pp. 325-336.

Kimball, R. & Caserta, J. 2004, *The Data Warehouse ETL Tookit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data,* Wiley and Sons, Indianapolis, IN, USA.

Krammer, M. I. 1997, "Business rules: automating business policies and practices," *Distributed Computing Monitor.*

Lamb, D.A., Scott, M. & Heavey, T. 2005, *CASE Vendor List*. http://www.cs.queensu.ca/Software-Engineering/vendor.html.

Lenzerini, M. 2002, "Data integration: a theoretical perspective," *ACM PODS 2002*, pp. 233-246.

Li, C., Bohannon, P. & Narayan, P.P.S. 2003, "Composing XSL transformations with XML publishing views," *ACM SIGMOD International conference on Management of data 2003*, pp. 515-526.

Lien, Y.E. 1982, "On the equivalence of data models," *J. ACM*, vol. 29, pp. 333-362.

Louie, B., Mork, P., Martin-Sanchez, F., Halevy, A.Y. & Tarczy-Hornoch, P. 2007, "Methodological review: data integration and genomic medicine," *Journal of Biomedical Informatics,* vol. 40, no. 1, pp. 5-16.

Loucopoulos, P. & Wan Kadir, W.M.N 2008, "BROOD: Business rules-driven object oriented design," *Jornal of Database Management*, vol. 10, no. 1, pp. 41-73.

Madhavan, J., Bernstein, P.A., Domingos, P. & Halevy, A.Y. 2002, "Representing and reasoning about mappings between domain models," *AAAI'02*, pp. 80-86.

Madhavan, J. & Halevy, A.Y. 2003, "Composing mappings among data sources," *VLDB 2003*, vol. 29, pp. 572-583.

Mazón, J-N., Trujillo, J. & Lechtenbörger, J. 2007, "Reconciling requirements-driven data warehouses with data sources via multidimensional normal forms." *Data & Knowledge Engineering*, vol. 63, pp. 725-751.

McGuinness, D. L. & Harmelen, F. van (eds.) 2004, "W3C OWL web ontology language overview." http://www.w3.org/TR/owl-features/.

Mecca, G., Atzeni, P., Masci, A., Sindoni, G. & Merialdo, P. 1998, "The Araneus web-based management system," *ACM SIGMOD 1998*, pp. 544-546.

Melnik, S. 2004, *Generic Model management: Concepts and Algorithms,* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Melnik, S., Bernstein, P.A., Halevy, A.Y. & Rahm, E. 2005, "Supporting executable mappings in model management," *ACM SIGMOD 2005*, pp. 167-178.

Melnik, S., Rahm, E. & Bernstein, P.A. 2003, "Rondo: a programming platform for generic model management," *ACM SIGMOD 2003*, pp. 193-204.

Mens, T. & Van Gorp, P. 2006, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science,* vol. 152, pp. 125-142.

Meyer, B. 1997, *Object-oriented software construction*. Prentice Hall International Series in Computer Science.

Microsoft 2008, *Microsoft .NET Framework 3.5 Administrator Deployment Guide*. http://msdn.microsoft.com/en-us/library/cc160717.aspx

Microsoft 2007, *Microsoft Office InfoPath*. http://office.microsoft.com/en-us/infopath.

Microsoft 2006, *Microsoft BizTalk Server*. http://www.microsoft.com/biztalk.

Microsoft 2005, *Microsoft SQL Server Reporting Services*. http://www.microsoft.com/sql/technologies/reporting.

Miller, R.J., Ioannidis, Y.E. & Ramakrishnan, R. 1994, "Schema equivalence in heterogeneous systems: bridging theory and practice," *Information Systems*, vol. 19, no. 1, pp. 3-31.

Mitra, P., Wiederhold, G. & Kersten, M.L. 2000, "A graph-oriented model for articulation of ontology interdependencies," *EDBT 2000*, LNCS 1777, pp. 86-100.

Morgan, T. 2002, *Business Rules and Information Systems: Aligning IT with Business Goals.* Boston, MA, Addison-Wesley.

Moriarty, T. 1993, "The next paradigm," *Database Programming and Design.*

Niles, I. & Pease, A. 2001, "Towards a standard upper ontology," *FOIS 2001*, pp. 2-9.

Noy, N.F. 2004, "Semantic integration: a survey of ontology-based approaches," *ACM SIGMOD Record,* vol. 33, no. 4, pp. 65-70.

Object Management Group 2008a, *Semantics of Business Vocabulary and Business Rules (SBVR), v1.0. OMG Available specification. (formal/2008-01-02)*.

Object Management Group 2008b, *Business-Friendly Notation for Business Vocabulary and Rules. Request for Proposal. (bmi/2008-06-01)*.

Object Management Group 2007a, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, OMG Final Adopted Specification (ptc/2007-07-07)*.

Object Management Group 2007b, *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, OMG Available Specification without Change Bars (formal/2007-11-02)*.

Object Management Group 2006a, *Meta Object Facility (MOF) Core Specification, Version 2.0, OMG Available Specification (formal/2006-01-01).*

Object Management Group 2006b, *Object Constraint Language (OCL), Version 2.0, OMG Available Specification (formal/2006-05-01)*.

Object Management Group 2006c, *Unified Modeling Language: Superstructure, Version 2.1*.

Object Management Group 2004, Business Rule Team, Business Semantics of Business Rules (BSBR).

Object Management Group 2003, MDA Guide Version 1.0.1. (omg/2003-06-01).

Olivé, A. 2007, *Conceptual Modeling of Information Systems,* Springer-Verlag, Berlin Heidelberg.

Oracle 2007, *Oracle Toplink*. http://www.oracle.com/technology/products/ias/toplink/index.html

Papotti, P. & Torlone, R. 2005, "Heterogeneous data translation through XML conversion," *Journal of Web Engineering,* vol. 4, no. 3, pp. 189-204.

Papotti, P. & Torlone, R. 2007, "Automatic generation of model translations," *CAiSE 2007*, LNCS 4495, pp. 36-50.

Peterson, D., Sperberg-McQueen, C.M., Thompson, H.S., Gao, S. 2008, "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes: Working Draft in Last Call 20 June 2008."

Planas, E. & Olivé, A. 2006, *The DBLP Case Study*. http://guifre.lsi.upc.edu/DBLP.pdf.

Popa, L., Velegrakis, Y., Hernández, M.A., Miller, R.J. & Fagin, R. 2002, "Translating web data," *VLDB 2002*, pp. 598-609.

Pottinger, R.A. & Bernstein, P.A. 2003, "Merging models based on given correspondences," *VLDB 2003*, pp. 826-873.

Rahm, E. & Bernstein, P.A. 2001, "A survey of approaches to automatic schema matching," *VLDB 2001*, 10, pp. 334-350.

Raventós, R. 2008a, *Application examples of the object-oriented operation-based translation approach to fragments of the UML, ER and Relational metaschemas.* LSI-08-10-R. http://www.lsi.upc.edu/dept/techreps/llistat_detallat.php?id=1004.

Raventós, R. 2008b, *Implementation of the translation between UML and SBVR in USE*. http://www.lsi.upc.edu/~raventos/PhDthesis/IMPLEMENTATION.htm

Riehle, D. 2006, "Value object," *PLoP 2006*, http://hillside.net/plop/2006/Papers/Library/ValueObject-%20vo5.pdf.

Rosca, D., Greenspan, S., Feblowitz, M. & Wild, C. 1997, "A decision support methodology in support of the business rules lifecycle," *Proceedings of the Third IEEE International Symposium on Requirements Engineering, 1997*, pp. 236-246.

Ross, R. G. 2003, *Principles of the Business Rule Approach*. Boston, MA: Addison-Wesley.

Ross, R.G. 2005, *Business Rule Concepts: Getting to the Point of Knowledge (2nd ed.)*, Business Rule Solutions, LLC.

Rumbaugh, J., Jacobson, I. & Booch, G. 2004, *The Unified Modeling Language Reference Manual,* 2nd ed., Addison-Wesley Pub Co.

Shu, N.C., Housel, B.C., Taylor, R.W., Ghosh, S.P. & Lum, V.Y. 1977, "EXPRESS: A Data EXtraction, Processing, and REStructuring System," *ACM TODS,* vol. 2, no. 2, pp. 134-174.

Shvaiko, P. & Euzenat, J. 2005, "A survey of schema-based matching approaches." in *J. Data Semantics IV*, pp. 146-171.

Sperberg-McQueen, C.M., Gao, S. & Thompson, H.S. (eds.) 2008, "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures: Working Draft in Last Call 20 June 2008."Stylus Studio 2008, *Stylus Studio*. http://www.stylusstudio.com/.

Tommasi, M. De & Corallo, A. 2006, "SBVEAVER: A tool for modeling business vocabularies and business rules," *KES 2006,* Part III, LNAI vol. 4253, pp. 1083-1091.

Truex, D.P., Baskerville, R. & Klein, H. 1999, "Growing systems in emergent organisations," *Communications of ACM*, vol. 42, no.8, pp. 117-123.

University of Bremen Database Systems Group 2007, *A UML-based Specification Environment (USE)*. http://www.db.informatik.uni-bremen.de/projects/USE/.

Vara. J.M., Vela, B., Cavero, J.M. & Marcos, E. 2007, "Model transformation for object-relational database development," *SAC'07*, pp. 1012-1019.

Wagner, G., Lukichev, S., Fuchs, N. E., Spreeuwenberg, S. 2005, "First-Version Controlled English Rule Language." REWERSE project, *IST506779/Eindhoven/I1-D2/D/PU/b1*.

Wan-Kadir, W.M.N. & Loucopoulos, P. 2004, "Relating evolving business rules to software design," *Journal of Systems Architecture,* vol. 50, pp. 367-382.

Wiederhold, G. 1977, *Database Design,* McGraw-Hill Higher Education.

Zave, P. 1997, "Classification of research efforts in requirements engineering," *ACM Computing Surveys*, vol. 29, no. 4, pp. 315-321.

Appendix A (Chapter 4): UML metaschema in USE

The following is a complete specification, suitable for validation with the USE tool, of the UML metaschema presented in Chapter 4. The operations related to the translation to SBVR are not included. Note that all the associations that are ordered have been specified as an *order* attribute because the ordered keyword in the USE tool does not seem to matter when inserting association links. The keyword is only used to distinguish between *Set* and *Sequence* types when using navigational syntax in OCL expressions[8]. Note also that the type of the *value* attribute of the *LiteralUnlimitedNatura*l, *UnlimitedNatural*, has been defined as an enumeration since the USE tool does not support *UnlimitedNatural* data types. Finally, note also that some notation is slightly different from the standard OCL.

```
----------------------------------------
-- Fragment of UML Metaschema v.2.1.2
----------------------------------------

model UMLMetaschema

----------------------------------------
--- Enumeration
----------------------------------------

enum AggregationKind { none, shared, composite }
enum UnlimitedNatural { asterisk }

----------------------------------------
--- Classes
----------------------------------------

class Association < Relationship, Classifier
attributes
  isDerived : Boolean
operations
-- derived association
  endType():Set(Type) =
 self.memberEnd->collect(e|e.type)->asSet
end

class AssociationClass < Class, Association

class Class < Classifier
operations
-- derived association
  superClass():Set(Class)=
 self.general().oclAsType(Class)->asSet
end
```

[8] Information provided directly by Mark Richters, developer of USE tool.

```
abstract class  Classifier < RedefinableElement, Namespace, Type
attributes
  isAbstract : Boolean
operations
-- derived associations
  attribute():Set(Property)=
    self.oclAsType(Class).ownedAttribute->union(
  self.oclAsType(DataType).ownedAttribute)->asSet

  general():Set(Classifier)=
    self.parents()
-- additional operations

  parents():Set(Classifier)=
    generalization.general->asSet()

  allParents():Set(Classifier)=
    self.parents()->union( self.parents()->collect(p|
  p.allParents())->flatten()->asSet() )
end

class Constraint < NamedElement
end

class DataType < Classifier
end

abstract class DirectedRelationship < Relationship
end

abstract class Element
end

class Enumeration < DataType
end

class EnumerationLiteral < InstanceSpecification
   attributes
     order : Integer
end

class Expression < ValueSpecification
attributes
  symbol : String
end

abstract class Feature < RedefinableElement
end

class Generalization < DirectedRelationship
end

class GeneralizationSet < NamedElement
attributes
  isCovering : Boolean
  isDisjoint : Boolean
end

class InstanceSpecification < NamedElement
```

```
end

class LiteralBoolean < LiteralSpecification
attributes
  value : Boolean
end

class LiteralInteger < LiteralSpecification
attributes
  value : Integer
end

class LiteralNull < LiteralSpecification
end

abstract class LiteralSpecification < ValueSpecification
end

class LiteralString < LiteralSpecification
attributes
  value : String
end

class LiteralUnlimitedNatural < LiteralSpecification
attributes
  value : UnlimitedNatural
end

abstract class MultiplicityElement < Element
operations
-- derivated attributes
  lower():Integer=
 lowerBound()

  upper():UnlimitedNatural=
 upperBound()

-- additional operations
  lowerBound():Integer =
 if self.lowerValue->isEmpty then 1 else
  self.lowerValue.oclAsType(LiteralInteger).value
 endif

  upperBound():UnlimitedNatural =
 self.upperValue.oclAsType(LiteralUnlimitedNatural).value
end

abstract class NamedElement < Element
attributes
  name : String
end

abstract class Namespace < NamedElement
end

class OpaqueExpression < ValueSpecification
attributes
 body : String
 language : String
```

```
end

class PrimitiveType < DataType
end

class Property < StructuralFeature
attributes
  isDerived : Boolean
  isDerivedUnion : Boolean
  aggregation_ : AggregationKind
 order : Integer
operations
-- derived attributes
  isComposite():Boolean=
 self.aggregation_=#composite
end

abstract class RedefinableElement < NamedElement
end

abstract class Relationship < NamedElement
end

abstract class StructuralFeature < MultiplicityElement,
 TypedElement, Feature
end

abstract class Type < NamedElement
end

abstract class TypedElement < NamedElement
end

abstract class ValueSpecification
end

----------------------------------------
--- Associations
----------------------------------------

composition OwningUpper_UpperValue between
  MultiplicityElement[0..1] role owningUpper
  ValueSpecification[0..1] role upperValue
end

composition OwningLower_LowerValue between
  MultiplicityElement[0..1] role owningLower
  ValueSpecification[0..1] role lowerValue
end

association TypedElement_Type between
  TypedElement[*]
  Type[0..1]
end

composition Context_OwnedRule between
  Namespace[0..1] role context_
  Constraint[*] role ownedRule
end
```

```
association Constraint_ConstrainedElement between
  Constraint[*]
  Element[*] role constrainedElement ordered
end

composition OwningConstraint_Specification between
  Constraint[0..1] role owningConstraint
  ValueSpecification[1] role specification
end

composition OwningInstanceSpec_Specification between
 InstanceSpecification[0..1] role owningInstanceSpec
 ValueSpecification[0..1] role specification
end

association InstanceSpecification_Classifier between
 InstanceSpecification[*]
 Classifier[*] role classifier
end

association Classifier_RedefinedClassifier between
 Classifier[*]
 Classifier[*] role redefinedClassifier
end

association General_Generalization between
  Classifier[1] role general
  Generalization[*] role generalization_
end

composition Specific_Generalization between
  Classifier[1] role specific
  Generalization[*] role generalization
end

association Class_OwnedAttribute between
  Class[0..1] role class_
  Property[*] role ownedAttribute ordered
end

composition Class_NestedClassifier between
 Class[0..1] role class_
 Classifier[*] role nestedClassifier
end

association Association_MemberEnd between
  Association[0..1] role association_
  Property[2..*] role memberEnd ordered
attributes
 order : Integer
end

association OwningAssociation_OwnedEnd between
  Association[0..1] role owningAssociation
  Property[*] role ownedEnd ordered
end

association Property_SubsettedProperty between
```

```
  Property[*]
  Property[*] role subsettedProperty
end

association Property_RedefinedProperty between
  Property[*] role property_
  Property[*] role redefinedProperty
end

association DataType_OwnedAttribute between
  DataType[0..1] role dataType
  Property[*] role ownedAttribute ordered
end

association Enumeration_EnumerationLiteral between
  Enumeration[0..1] role enumeration
  EnumerationLiteral[1..*] role ownedLiteral ordered
end

composition AssociationEnd_Qualifier between
  Property[0..1] role associationEnd
  Property[*] role qualifier
end

association Powertype_PowertypeExtent between
  Classifier[0..1] role powertype
  GeneralizationSet[*] role powertypeExtent
end

association GeneralizationSet_Generalization between
  GeneralizationSet[*] role generalizationSet
  Generalization[1..*] role generalization
end
----------------------------------------
---   Constraints
----------------------------------------

constraints

context Association inv OnlyBinaryAssociationCanBeAggregations:
  self.memberEnd->exists(aggregation_<>#none) implies
    self.memberEnd->size=2

context Association inv
 AssociationsEndsWithMoreThan2MustBeOwnedByAssociation:
  if self.memberEnd->size>2 then ownedEnd->
 includesAll(memberEnd) else true endif

context Classifier inv GeneralizationsAreAcyclical:
  not self.allParents()->includes(self)

context Constraint inv AConstraintCannotBeAppliedToItself:
  not constrainedElement -> includes(self)

context GeneralizationSet inv
 EveryGeneralizationMustHaveTheSameGeneralClassifier:
  self.generalization->collect(g|g.general)->asSet->size<=1

context MultiplicityElement inv TheLowerBoundMustBeNonNegative:
  lowerBound()<>oclUndefined(Integer) implies lowerBound()>=0

context MultiplicityElement inv UpperBoundGreaterThanLowerBound:
 upperValue.oclAsType(LiteralInteger) <>
oclUndefined(LiteralInteger) implies
upperValue.oclAsType(LiteralInteger).value>=lowerBound()

context Property inv IsDeriveUnionImpliesIsDerived:
 self.isDerivedUnion implies self.isDerived

context Expression_Operand inv CorrectOrder:
 Expression_Operand.allInstances->sortedBy(order)->
 last.order=Expression_Operand.allInstances->size()

context Constraint_ConstrainedElement inv CorrectOrder:
 Constraint_ConstrainedElement.allInstances->
 sortedBy(order)->last.order=
 Constraint_ConstrainedElement.allInstances->size()

context Class_OwnedAttribute inv CorrectOrder:
 Class_OwnedAttribute.allInstances->sortedBy(order)->
 last.order=Class_OwnedAttribute.allInstances->size()

context Association_MemberEnd inv CorrectOrder:
 Association_MemberEnd.allInstances->sortedBy(order)->
 last.order=Association_MemberEnd.allInstances->size()

context OwningAssociation_OwnedEnd inv CorrectOrder:
 OwningAssociation_OwnedEnd.allInstances->sortedBy(order)->
 last.order=OwningAssociation_OwnedEnd.allInstances->size()

context DataType_OwnedAttribute inv CorrectOrder:
 DataType_OwnedAttribute.allInstances->sortedBy(order)->
 last.order=DataType_OwnedAttribute.allInstances->size()
```

Appendix B (Chapter 4): DBLP as an instance of UML metaschema

This Appendix shows a representative list (there is an example for each type of schema unit) of commands that have been used to create the structural schema of the DBLP example, introduced in Chapter 4, in the USE tool. The schema is created as instances of the UML Metaschema. The complete instantiation is available at (Raventós 2008b).

```
-- Primitive Types

!create PrimitiveType1 : PrimitiveType
!set PrimitiveType1.name := 'String'
!set PrimitiveType1.isAbstract := false

-- Enumeration

!create Enumeration1 : Enumeration
!set Enumeration1.isAbstract := false
!set Enumeration1.name := 'Gender'
!create EnumerationLiteral1 : EnumerationLiteral
!set EnumerationLiteral1.name := 'Male'
!insert (Enumeration1,EnumerationLiteral1) into
Enumeration_OwnedLiteral
!set EnumerationLiteral1.order := 1
!create EnumerationLiteral2 : EnumerationLiteral
!set EnumerationLiteral2.name := 'Female'
!insert (Enumeration1,EnumerationLiteral2) into
Enumeration_OwnedLiteral
!set EnumerationLiteral2.order := 2

-- Data Type

!create DataType1 : DataType
!set DataType1.name := 'Natural'
!set DataType1.isAbstract := false

-- Class

!create Class_1 : Class
!set Class_1.name := 'person'
!set Class_1.isAbstract := false

-- Generalization

!create Generalization1 : Generalization
!insert (Class_2,Generalization1) into General_Generalization
!insert (Class_4,Generalization1) into Specific_Generalization

-- Generalization Set
```

```
!create GeneralizationSet1 : GeneralizationSet
!insert (GeneralizationSet1,Generalization2) into
GeneralizationSet_Generalization
!insert (GeneralizationSet1,Generalization3) into
GeneralizationSet_Generalization
!set GeneralizationSet1.isCovering := true
!set GeneralizationSet1.isDisjoint := true

-- Attribute

!create Attribute1 : Property
!set Attribute1.name := 'gender'
!insert (Class_1,Attribute1) into Class_OwnedAttribute
!insert (Attribute1,Enumeration1) into TypedElement_Type
!set Attribute1.isDerived := false
!set Attribute1.isDerivedUnion := false
!set Attribute1.aggregation_ := #none
!create LiteralInteger1l : LiteralInteger
!set LiteralInteger1l.value := 1
!create LiteralInteger1u : LiteralInteger
!set LiteralInteger1u.value := 1
!insert (Attribute1,LiteralInteger1l) into OwningLower_LowerValue
!insert (Attribute1,LiteralInteger1u) into OwningUpper_UpperValue

-- Association

!create Association4 : Association
!set Association4.isDerived := false
!set Association4.isAbstract := false
!create Property4a : Property
!insert (Property4a,Class_7) into TypedElement_Type
!set Property4a.isDerived := false
!set Property4a.isDerivedUnion := false
!set Property4a.aggregation_ := #none
!create Property4b : Property
!insert (Property4b,Class_5) into TypedElement_Type
!set Property4b.isDerived := false
!set Property4b.isDerivedUnion := false
!set Property4b.aggregation_ := #shared
!set Property4a.order := 1
!set Property4b.order :=2
!insert (Association4,Property4a) into Association_MemberEnd
!insert (Association4,Property4a) into OwningAssociation_OwnedEnd
!insert (Association4,Property4b) into Association_MemberEnd
!insert (Association4,Property4b) into OwningAssociation_OwnedEnd
!create LiteralInteger4al : LiteralInteger
!set LiteralInteger4al.value := 1
!create LiteralUnlimitedNatural4au : LiteralUnlimitedNatural
!set LiteralInteger4au.value := #asterisk
!insert (Property4a,LiteralInteger4al) into OwningLower_LowerValue
!insert (Property4a,LiteralUnlimitedNatural4au) into
 OwningUpper_UpperValue
!create LiteralInteger4bl : LiteralInteger
!set LiteralInteger4bl.value := 0
```

```
!create LiteralInteger4bu : LiteralInteger
!set LiteralInteger4bu.value := 1
!insert (Property4b,LiteralInteger4bl) into OwningLower_LowerValue
!insert (Property4b,LiteralInteger4bu) into
 OwningUpper_UpperValue
!create LiteralUnlimitedNatural3bu:LiteralUnlimitedNatural
!set LiteralUnlimitedNatural3bu.value := 1
!insert (Property3b,LiteralInteger3bl) into OwningLower_LowerValue
!insert (Property3b,LiteralUnlimitedNatural3bu) into
OwningUpper_UpperValue

-- AssociationClasses

!create AssociationClass1 : AssociationClass
!set AssociationClass1.name := 'editorship'
!set AssociationClass1.isDerived := false
!set AssociationClass1.isAbstract := false
!create Property1a : Property
!insert (Property1a,Class_1) into TypedElement_Type
!set Property1a.name := 'editor'
!set Property1a.isDerived := false
!set Property1a.isDerivedUnion := false
!set Property1a.aggregation_ := #none
!create Property1b : Property
!insert (Property1b,Class_5) into TypedElement_Type
!set Property1b.isDerived := false
!set Property1b.isDerivedUnion := false
!set Property1b.aggregation_ := #none
!set Property1a.order := 1
!set Property1b.order := 2
!insert (AssociationClass1,Property1a) into Association_MemberEnd
!insert (AssociationClass1,Property1a) into
 OwningAssociation_OwnedEnd
!insert (AssociationClass1,Property1b) into Association_MemberEnd
!insert (AssociationClass1,Property1b) into
 OwningAssociation_OwnedEnd
!create LiteralInteger1al : LiteralInteger
!set LiteralInteger1al.value := 1
!create LiteralUnlimitedNatural1au : LiteralUnlimitedNatural
!set LiteralInteger1au.value := #asterisk
!insert (Property1a,LiteralInteger1al) into OwningLower_LowerValue
!insert (Property1a,LiteralUnlimitedNatural1au) into
 OwningUpper_UpperValue
!create LiteralInteger1bl : LiteralInteger
!set LiteralInteger1bl.value := 0
!create LiteralUnlimitedNatural1bu : LiteralUnlimitedNatural
!set LiteralInteger1bu.value := #asterisk
!insert (Property1b,LiteralInteger1bl) into OwningLower_LowerValue
!insert (Property1b,LiteralUnlimitedNatural1bu) into
 OwningUpper_UpperValue

-- Constraint

!create Constraint2 : Constraint
!set Constraint2.name := 'nameIsKeyOfPerson'
```

```
!create Constraint_ConstrainedElement2:
Constraint_ConstrainedElement between
(Constraint2,Attribute2)
!set Constraint_ConstrainedElement2.order := 1
!insert (Class_1, Constraint2) into Context_OwnedRule
!create Expression2 : Expression
!set Expression2.symbol := 'person.allInstances()->isUnique(name)'
!insert (Constraint2, Expression2) into
OwningConstraint_Specification
```

Appendix C (Chapter 4): methods for creating UML schema units

This Appendix describes the methods for creating instances of UML schema units from their characterization objects, as described in Chapter 4. For each chacterization objects there is a procedure in an .assl file. The description of all methods is available at (Raventós 2008b).

The method for creating the schema units of all instances of *ClassCh* is defined as follows:

```
procedure CreateUnitOfClassCh()
  var c:Class, el:ClassCh;
begin
 for nameCh:String in [ClassCh.allInstances -> collect(ch:ClassCh|
     ch.name)->asSet->asSequence]
  begin
    el := Any([ClassCh.allInstances -> select(ch|
      ch.name = nameCh)-> asSequence]);
    c:=Create(Class);
    [c].name := [el.name];
    [c].isAbstract := [el.isAbstract];
  end;
end;
```

The method for creating the schema units of all the instances of *DataTypeCh* is defined as follows:

```
procedure CreateUnitOfDataTypeCh()
  var d:DataType;
  begin
  for el:DataTypeCh in [DataTypeCh.allInstances->asSequence]
   begin
    if [el.isPrimitiveType] then
    begin
      p:=Create(PrimitiveType);
      [p].name := [el.name];
      [p].isAbstract := [false];
    end
    else
    begin
      d:=Create(DataType);
      [d].name := [el.name];
      [d].isAbstract := [false];
    end;
   end;
  end;
```

The method for creating the schema unit of all the instances of *EnumerationCh* is defined as follows:

```
procedure CreateUnitOfEnumerationCh()
  var e:Enumeration, eli:EnumerationLiteral, el:EnumerationCh;
  begin
```

```
for nameCh:String in [EnumerationCh.allInstances ->
  collect(ch:EnumerationCh| ch.name)->asSet->asSequence]
begin
  el := Any([EnumerationCh.allInstances->select(ch|
    ch.name = nameCh)->asSequence]);
  e := Create(Enumeration);
  [e].name := [el.name];
  [e].isAbstract := [false];
  for li:Literal in [el.literal]
    begin
      eli := Create (EnumerationLiteral);
      [eli].name := [li.name];
      [eli].order := [li.order];
      Insert(Enumeration_OwnedLiteral, [e],[eli]);
    end;
  end;
end;
```

The method for creating the schema units of all instances of *PropertyCh* is defined as follows:

```
procedure CreateUnitOfPropertyCh()
var p:Property, cl:Class, dt:DataType, li:LiteralInteger,
lu1:LiteralUnlimitedNatural, lu2:LiteralInteger, d:DataType,
el:PropertyCh, pr:PrimitiveType;
begin
  for pro:Tuple(cl:String, na:String) in
   [PropertyCh.allInstances->collect(ch:PropertyCh|
   Tuple{cl:ch.ownerClassName, na:ch.name})->asSet->asSequence]
  begin
   el := Any([PropertyCh.allInstances->select(p|
     p.ownerClassName=pro.cl and p.name=pro.na)->asSequence]);
   p := Create(Property);
   [p].name := [el.name];

   if [el.type<>'Boolean'] then
   begin
     d := Any([DataType.allInstances->select(e:DataType|
       (e.oclIsTypeOf(DataType) or e.oclIsTypeOf(Enumeration) or
       e.oclIsTypeOf(PrimitiveType)) and e.name=el.type)->
       asSequence]);
     Insert(TypedElement_Type, [p],[d]);
   end
   else
   begin
     pr:= Create(PrimitiveType);
     [pr].name := ['Boolean'];
     Insert(TypedElement_Type, [p],[pr]);
   end;

   if [el.ownerClassName<>oclUndefined(String)] then
   begin
     cl := Any([Class.allInstances->select(c:Class|
       c.name=el.ownerClassName)->asSequence]);
     Insert(Class_OwnedAttribute, [cl], [p]);
   end;
   if [el.ownerDataTypeName<>oclUndefined(String)] then
   begin
     dt := Any([DataType.allInstances->select(d:DataType|
       d.name=el.ownerDataTypeName)->asSequence]);
     Insert(DataType_OwnedAttribute, [dt], [p]);
   end;
   [p].isDerived := [el.isDerived];
   [p].isDerivedUnion := [el.isDerivedUnion];
   [p].aggregation_ := [el.aggregation_];
   li := Create(LiteralInteger);
   [li].value := [el.lowerValue];
   Insert(OwningLower_LowerValue, [p], [li]);
   if [el.upperValue=oclUndefined(Integer)] then
   begin
     lu1 := Create(LiteralUnlimitedNatural);
     [lu1].value := [#asterisk];
     Insert(OwningUpper_UpperValue, [p], [lu1]);
   end
   else
   begin
     lu2 := Create(LiteralInteger);
     [lu2].value := [el.upperValue];
     Insert(OwningUpper_UpperValue, [p], [lu2]);
     end;
   end;
 end;
```

The method for creating the schema units of all instances of *AssociationCh* is defined as follows:

```
procedure CreateUnitOfAssociationCh()
  var  a:Association, p:Property, cl:Class, li:LiteralInteger,
    lu2:LiteralInteger, lu1:LiteralUnlimitedNatural,
    el:AssociationCh;
  begin
   for pro:Tuple(cl1:String, cl2:String) in
    [AssociationCh.allInstances->collect(ch:AssociationCh|
      Tuple{cl1:ch.associationMemberEnd->sortedBy(order)->
      first.typeName, cl2:ch.associationMemberEnd->
      sortedBy(order) ->last.typeName})->asSet->asSequence]
   begin
    for el:AssociationCh in [AssociationCh.allInstances
        ->asSequence]
   begin
    a := Create(Association);
    if [el.name<>oclUndefined(String)] then
    begin
        [a].name := [el.name];
    end;
    [a].isAbstract := [el.isAbstract];
    for ame:AssociationMemberEnd in
          [el.associationMemberEnd]
    begin
      p := Create (Property);
      Insert(Association_MemberEnd, [a], [p]);
      Insert(OwningAssociation_OwnedEnd, [a], [p]);
      if [ame.name<>oclUndefined(String)] then
      begin
          [p].name := [p.name];
      end;
      cl := Any([Class.allInstances -> select(c:Class|
              c.name = ame.typeName)->asSequence]);
      Insert(TypedElement_Type, [p], [cl]);
```

```
       [p].isDerived := [ame.isDerived];
       [p].isDerivedUnion := [ame.isDerivedUnion];
       [p].aggregation_ := [ame.aggregation_];
       li := Create(LiteralInteger);
       [li].value := [ame.lowerValue];
       Insert(OwningLower_LowerValue, [p], [li]);
       if [ame.upperValue=oclUndefined(Integer)] then
       begin
         lu1 := Create(LiteralUnlimitedNatural);
         [lu1].value := [#asterisk];
         Insert(OwningUpper_UpperValue, [p], [lu1]);
       end
       else
       begin
         lu2 := Create(LiteralInteger);
         [lu2].value := [ame.upperValue];
         Insert(OwningUpper_UpperValue, [p], [lu2]);
         end;
       [p].order := [ame.order];
     end;
   end;
 end;
```

The method for creating the schema units of all instances of *AssociationClassCh* is defined as follows:

```
 procedure CreateUnitOfAssociationClassCh()
   var  ac:AssociationClass, p:Property, cl:Class,
     li:LiteralInteger, lu2:LiteralInteger,
     lu1:LiteralUnlimitedNatural, el:AssociationClassCh;
   begin
    begin
    for nameCh:String in [AssociationClassCh.allInstances->
      collect(ch:AssociationClassCh| ch.name)->asSet->asSequence]
    begin
      for el:AssociationClassCh in
            [AssociationClassCh.allInstances -> asSequence]
    begin
      ac := Create(AssociationClass);
      [ac].name := [el.name];
      for ame:AssociationClassMemberEnd in
            [el.associationClassMemberEnd]
      begin
        p := Create (Property);
        Insert(Association_MemberEnd, [ac], [p]);
        Insert(OwningAssociation_OwnedEnd, [ac], [p]);
        if [ame.name <> oclUndefined(String)] then
          begin
            [p].name := [p.name];
          end;
        cl := Any([Class.allInstances -> select(c:Class|
                  c.name = ame.typeName) -> asSequence]);
        Insert(TypedElement_Type, [p], [cl]);
        [p].isDerived := [ame.isDerived];
        [p].isDerivedUnion := [ame.isDerivedUnion];
        [p].aggregation_ := [ame.aggregation_];
        li := Create(LiteralInteger);
        [li].value := [ame.lowerValue];
        Insert(OwningLower_LowerValue, [p], [li]);
        lu := Create(LiteralUnlimitedNatural);
        if [ame.upperValue=oclUndefined(Integer)] then
        begin
          lu1 := Create(LiteralUnlimitedNatural);
          [lu1].value := [#asterisk];
          Insert(OwningUpper_UpperValue, [p], [lu1]);
        end
        else
        begin
          lu2 := Create(LiteralInteger);
          [lu2].value := [ame.upperValue];
          Insert(OwningUpper_UpperValue, [p], [lu2]);
          end;
          [p].order := [ame.order];
      end;
    end;
  end;
```

The method for creating the schema units of all instances of *GeneralizationCh* is defined as follows:

```
    procedure CreateUnitOfGeneralizationCh()
      var g:Generalization, gc:Class, sc:Class, el:GeneralizationCh;
    begin
      for pro:Tuple(cl1:String, cl2:String) in
        [GeneralizationCh.allInstances->collect(ch:GeneralizationCh|
          Tuple{cl1:ch.generalClassName, cl2:ch.specificClassName})->
            asSet->asSequence]
      begin
        el := Any([GeneralizationCh.allInstances->select(p|
          p.generalClassName = pro.cl1 and p.specificClassName =
          pro.cl2)->asSequence]);
        g := Create(Generalization);
        gc := Any([Class.allInstances->select(c:Class| c.name =
          el.generalClassName)->asSequence]);
        Insert(General_Generalization, [gc], [g]);
        sc := Any([Class.allInstances->select(c:Class|
          c.name = el.specificClassName)->asSequence]);
        Insert(Specific_Generalization, [sc], [g]);
      end;
    end;
```

The method for creating the schema units of all instances of *GeneralizationSetCh* is defined as follows:

```
 procedure CreateUnitOfGeneralizationSetCh()
   var gs:GeneralizationSet, g:Generalization, pw:Classifier,
   el:GeneralizationSetCh;
 begin

   for pro:String in [GeneralizationSetCh.allInstances->
    collect(ch:GeneralizationSetCh| ch.name)->asSet->asSequence]
   begin
    el := Any([GeneralizationSetCh.allInstances->select(p|
      p.name=pro)->asSequence]);
    gs := Create(GeneralizationSet);
    [gs].name := [el.name];
    [gs].isCovering := [el.isCovering];
    [gs].isDisjoint := [el.isDisjoint];
    for p:Participant in [el.participant->asSequence]
    begin
      g := Any([Generalization.allInstances->
         select(ge:Generalization| ge.general.name =
         p.generalClassName and ge.specific.name =
         p.specificClassName)->asSequence]);
      Insert(GeneralizationSet_Generalization, [gs], [g]);
    end;
   end;
 end;
```

The method for creating the schema units of all instances of *ConstraintCh* is defined as follows:

```
 procedure CreateUnitOfConstraintCh()
   var c:Constraint, na:Namespace, ex:Expression, ele:Element,
   el:ConstraintCh;
 begin
   for pro:Tuple(na:String,sp:String, ce:Set(ConstrainedElement_)) in
    [ConstraintCh.allInstances->select(ch| ch.name<>'XOR')->
    collect(ch:ConstraintCh| Tuple{na:ch.name, sp:ch.namespace,
      ce:ch.constrainedElement})->asSet->asSequence]
   begin
    el := Any([ConstraintCh.allInstances->select(p| p.name = pro.na
      and p.namespace=pro.sp and p.constrainedElement=pro.ce)->
      asSequence]);
    c := Create(Constraint);
    if [el.name<>oclUndefined(String)] then
    begin
      [c].name := [el.name];
    end;
    na := Any([Class.allInstances->select(n| n.name = el.namespace)
      ->asSequence]);
    Insert(Context_OwnedRule, [na], [c]);
    ex := Create(Expression);
    [ex].symbol := [el.symbolExpression];
    Insert(OwningConstraint_Specification,[c],[ex]);
    for coe:ConstrainedElement_ in [el.constrainedElement->
      asSequence]
    begin
      ele := Any([Element.allInstances->select(e:Element|
         if coe.type=#property then
           e.oclAsType(Property).class_.name=el.namespace and
           e.oclAsType(Property).name=coe.name
         else e.oclAsType(Association).memberEnd->collect(name)->
              asSet = coe.membersName->asSet and
              e.oclAsType(Association).memberEnd->collect(type.name)
              ->asSet = coe.membersType->asSet
         endif)->asSequence]);
      Insert(Constraint_ConstrainedElement, [c], [ele]);
    end;
   end;
   el := Any([ConstraintCh.allInstances->select(p| p.name = 'XOR')->
    asSequence]);
   c := Create(Constraint);
   if [el.name<>oclUndefined(String)] then
   begin
    [c].name := [el.name];
   end;
   na := Any([Class.allInstances->select(n| n.name = el.namespace)->
    asSequence]);
```

```
  Insert(Context_OwnedRule, [na], [c]);
  ex := Create(Expression);
  [ex].symbol := [el.symbolExpression];
  Insert(OwningConstraint_Specification,[c],[ex]);
  for coe:ConstrainedElement_ in [el.constrainedElement->asSequence]
  begin
   ele := Any([Element.allInstances->select(e:Element|
     if coe.type=#property then
        e.oclAsType(Property).class_.name=el.namespace and
        e.oclAsType(Property).name=coe.name
     else e.oclAsType(Association).memberEnd->collect(name)->asSet
        = coe.membersName->asSet and
        e.oclAsType(Association).memberEnd ->collect(type.name)->
        asSet = coe.membersType->asSet
     endif)->asSequence]);
   Insert(Constraint_ConstrainedElement, [c], [ele]);
  end;
end;
```

Appendix D (Chapter 5): SBVR meanings metaschema in USE

The following is a complete specification, suitable for validation with the USE tool, of the SBVR Meanings metaschema presented in Chapter 5. Note, as in the case of UML, that all the associations that are ordered have been specified as an *order* attribute because the ordered keyword in the USE tool does not seem to matter when inserting association links. The keyword is only used to distinguish between *Set* and *Sequence* types when using navigational syntax in OCL expressions[9].

```
---------------------------------------------
-- Fragment of SBVR Meanings Metaschema v.1.0
---------------------------------------------

   model SBVR Meanings

----------------------------------------
--- Enumerations
----------------------------------------

   enum FactTypeType { Associative, IsPropertyOf, Partitive,
            Categorization, Characteristic }
   enum BinaryOperationType { Conjunction, Disjunction, Equivalence,
            ExclusiveDisjunction, NandFormulation, NorFormulation,
            Implication, WhetherOrNotFormulation}
   enum QuantificationType { Universal, AtLeastN, Existential,
            AtMostN, AtMostOne, ExactlyN, ExactlyOne, NumericRange }

----------------------------------------
--- Classes
----------------------------------------

   class AssociativeFactType < FactType
   end

   class AtLeastNQuantification < Quantification
   end

   class AtMostNQuantification < Quantification
   end

   class AtMostOneQuantification < AtMostNQuantification
   end

   class AtomicFormulation < LogicalFormulation
   end
```

[9] Information provided by Mark Richters, developer of USE tool.

```
abstract class BinaryLogicalOperation < LogicalOperation
end

abstract class BindableTarget < Concept
end

class CategorizationScheme < ObjectType
end

class CategorizationFactType < FactType
end

class Characteristic < FactType
end

abstract class ClosedLogicalFormulation <
ClosedSemanticFormulation, LogicalFormulation
end

class ClosedProjection < Projection, ClosedSemanticFormulation
end

abstract class ClosedSemanticFormulation < SemanticFormulation
end

abstract class Concept < Meaning
attributes
  name: String
end

class Conjunction < BinaryLogicalOperation
end

class Disjunction < BinaryLogicalOperation
end

class Equivalence < BinaryLogicalOperation
end

class ExactlyNQuantification < Quantification
end

class ExactlyOneQuantification < ExactlyNQuantification
end

class ExclusiveDisjunction < BinaryLogicalOperation
end

class ExistentialQuantification < AtLeastNQuantification
end

abstract class FactType < Concept
end

class FactTypeRole < Role
```

```
attributes
  order : Integer
end

class Implication < BinaryLogicalOperation
end

class IndividualConcept < NounConcept, BindableTarget
end

class InstantiationFormulation < LogicalFormulation
end

class IsPropertyOfFactType < AssociativeFactType
end

abstract class LogicalFormulation < SemanticFormulation
end

class LogicalNegation < LogicalOperation
end

abstract class LogicalOperation < LogicalFormulation
end

abstract class Meaning < Thing
end

class NandFormulation < BinaryLogicalOperation
end

class NorFormulation < BinaryLogicalOperation
end

class NounConcept < Concept
end

class NonNegativeInteger < NounConcept
end

class NumericRangeQuantification < Quantification
end

class Objectification < LogicalFormulation
end

class ObjectType < NounConcept
end

class PartitiveFactType < AssociativeFactType
end

class Projection < SemanticFormulation
end
```

```
   class Proposition < Meaning
   attributes
     isTrue:Boolean
   end

   abstract class Quantification < LogicalFormulation
   end

   class ReferenceScheme < Concept
   end

   class Role < NounConcept
   end

   class RoleBinding < Concept
   end

   class Rule < Proposition
   end

   class Segmentation < CategorizationScheme
   end

   abstract class SemanticFormulation < Thing
   end

   class StructuralRule < Rule
   end

   class Text < NounConcept
   attributes
     value:String
   end

   abstract class Thing
   end

   class UniversalQuantification < Quantification
   end

   class Variable < BindableTarget
   end

   class WhetherOrNotFormulation < BinaryLogicalOperation
   end

   class ClosedQuantification < ClosedLogicalFormulation,
       Quantification
   end

   class ClosedUniversalQuantification < ClosedQuantification
   end

----------------------------------------
--- Associations
----------------------------------------
```

```
association NounConcept_Role between
  NounConcept[1] role nounConcept
  Role[*] role role_
end

association FactTypeRole_FactType between
  FactTypeRole[*] role factTypeRole ordered
  FactType[1] role factType
end

association ReferenceScheme_SimplyUsedRole between
  ReferenceScheme[*] role referenceScheme
  FactTypeRole[*] role simplyUsedRole
end

association ReferenceScheme_IdentifyingCharacteristic between
  ReferenceScheme[*] role referenceScheme
  Characteristic[*] role identifyingCharacteristic
end

association Concept_ReferenceScheme between
  Concept[1..*] role referencedConcept
  ReferenceScheme[*] role referenceSchemeOfConcept
end

association CategorizationScheme_Category between
  CategorizationScheme[*] role scheme
  Concept[1..*] role category
end

association CategorizationScheme_GeneralConcept between
  CategorizationScheme[*] role categorizationScheme
  NounConcept[1] role generalConcept
end

association LogicalFormulation_Projection between
  LogicalFormulation[0..1] role logicalFormulation
  Projection[*] role projection
end

association SemanticFormulation_FreeVariable between
  SemanticFormulation[*] role semanticFormulation
  Variable[*] role freeVariable
end

association RestrictingFormulation_Variable between
  LogicalFormulation[0..1] role restrictingFormulation
  Variable[*] role variable
end

association Variable_RangedOverConcept between
  Variable[*] role variable
  Concept[0..1] role rangedOverConcept
end
```

```
association Proposition_ClosedLogicalFormulation between
  Proposition[1] role proposition
  ClosedLogicalFormulation[0..1] role closedLogicalFormulation
end

association AtomicFormulation_FactType between
  AtomicFormulation[*] role atomicFormulation
  FactType[1] role factType
end

association AtomicFormulation_RoleBinding between
  AtomicFormulation[1] role atomicFormulation
  RoleBinding[*] role roleBinding
end

association FactTypeRole_RoleBinding between
  FactTypeRole[1] role factTypeRole
  RoleBinding[*] role roleBinding
end

association RoleBinding_BindableTarget between
  RoleBinding[*] role roleBinding
  BindableTarget[1] role bindableTarget
end

association InstantiationFormulation_BindableTarget between
  InstantiationFormulation[*] role
       boundedToInstantiationFormulation
  BindableTarget[1] role bindableTarget
end

association InstantiationFormulation_ConceptConsidered between
  InstantiationFormulation[*] role instantiationFormulation
  Concept[1] role conceptConsidered
end

association LogicalNegation_LogicalOperand between
  LogicalNegation[*] role LogicalNegation
  LogicalFormulation[1] role logicalOperand
end

association BinaryLogicalOperation_LogicalOperand1 between
  BinaryLogicalOperation[*] role binaryLogicalOperation1
  LogicalFormulation[1] role logicalOperand1
end

association BinaryLogicalOperation_LogicalOperand2 between
  BinaryLogicalOperation[*] role binaryLogicalOperation2
  LogicalFormulation[1] role logicalOperand2
end

association Implication_Antecedent between
  Implication[*] role implication1
  LogicalFormulation[1] role antecedent
```

```
end

association Implication_Consequent between
  Implication[*] role implication2
  LogicalFormulation[1] role consequent
end

association WhetherOrNotFormulation_Consequent between
  WhetherOrNotFormulation[*] role whetherOrNotFormulation1
  LogicalFormulation[1] role consequent
end

association WhetherOrNotFormulation_Inconsequent between
  WhetherOrNotFormulation[*] role whetherOrNotFormulation2
  LogicalFormulation[1] role inconsequent
end

association Quantification_ScopeFormulation between
  Quantification[*] role quantification
  LogicalFormulation[0..1] role scopeFormulation
end

association Quantification_IntroducedVariable between
  Quantification[0..1] role quantification
  Variable[1] role introducedVariable
end

association AtLeastNQuantification_MinimumCardinality between
  AtLeastNQuantification[*] role atLeastNQuantification
  NonNegativeInteger[1] role minimumCardinality
end

association AtMostNQuantification_MaximumCardinality between
  AtMostNQuantification[*] role atMostNQuantification
  NonNegativeInteger[1] role maximumCardinality
end

association NumericRangeQuantification_MinimumCardinality between
  NumericRangeQuantification[*] role numericRangeQuantification1
  NonNegativeInteger[1] role minimumCardinality
end

association NumericRangeQuantification_MaximumCardinality between
  NumericRangeQuantification[*] role numericRangeQuantification2
  NonNegativeInteger[1] role maximumCardinality
end

association ExactlyNQuantification_Cardinality between
  ExactlyNQuantification[*] role exactlyNQuantification
  NonNegativeInteger[1] role cardinality
end

association BindableTarget_Objectification between
  BindableTarget[1] role bindableTarget
  Objectification[*] role objectification
```

```
end

association Objectification_ConsideredLogicalFormulation between
  Objectification[*] role objectification
  LogicalFormulation[1] role consideredLogicalFormulation
end

association ProjectionVariable_IsInProjection between
  Variable[1..*] role projectionVariable
  Projection[*] role isInProjection
end

association Variable_FactTypeRole between
  Variable[*] role roleVariable
  FactTypeRole[0..1] role factTypeRole
end

association ClosedProjection_NounConcept between
  ClosedProjection[0..1] role closedProjection
  NounConcept[0..1] role nounConcept
end

association ClosedProjection_FactType between
  ClosedProjection[0..1] role closedProjection
  FactType[0..1] role factType
end
```

Appendix E (Chapter 5): DBLP as an instance of SBVR meanings metaschema

This Appendix lists the commands that have been used to create part of the structural schema of the DBLP example in the USE tool. The schema is created as instances of the SBVR Meanings Metaschema. The whole instantiation is available at (Raventós 2008b).

```
-- ObjectType

!create ObjectType1 : ObjectType
!set ObjectType1.name := 'person'

-- IndividualConcept

!create IndividualConcept1 : IndividualConcept
!set IndividualConcept1.name := 'Male'
!create IndividualConcept2 : IndividualConcept
!set IndividualConcept2.name := 'Female'

-- ValueType

!create Gender1 : ObjectType
!set Gender1.name := 'gender'

!create ClosedProjection4 : ClosedProjection
!insert (ClosedProjection4,Gender1) into
    ClosedProjection_NounConcept
!create VariableP4 : Variable
!insert (VariableP4,ClosedProjection4) into
    ProjectionVariable_IsInProjection
!insert (VariableP4,Gender1) into Variable_RangedOverConcept
!create DisjunctionP4 : Disjunction
!insert (DisjunctionP4,ClosedProjection4) into
    LogicalFormulation_Projection
!create InstantiationFormulationP41: InstantiationFormulation
!insert (DisjunctionP4, InstantiationFormulationP41) into
    BinaryLogicalOperation_LogicalOperand1
!insert (InstantiationFormulationP41,IndividualConcept1) into
    InstantiationFormulation_BindableTarget
!insert (InstantiationFormulationP41,VariableP4) into
    InstantiationFormulation_ConceptConsidered
!create InstantiationFormulationP42: InstantiationFormulation
!insert (DisjunctionP4, InstantiationFormulationP42) into
    BinaryLogicalOperation_LogicalOperand2
!insert (InstantiationFormulationP42,IndividualConcept2) into
    InstantiationFormulation_BindableTarget
!insert (InstantiationFormulationP42,VariableP4) into
    InstantiationFormulation_ConceptConsidered

-- CategorizationFactType
```

```
!create CategorizationFactType1 : CategorizationFactType
!create Role_1g2 : FactTypeRole
!insert (ObjectType2,Role_1g2) into NounConcept_Role
!create Role_1s4 : FactTypeRole
!insert (ObjectType4,Role_1s4) into NounConcept_Role
!set CategorizationFactType1.name := 'is a category of'
!insert (Role_1g2,CategorizationFactType1) into
FactTypeRole_FactType
!insert (Role_1s4,CategorizationFactType1) into
FactTypeRole_FactType
!set Role_1s4.order := 1
!set Role_1g2.order := 2

-- IsPropertyOfFactType

!create IsPropertyOfFactType1 : IsPropertyOfFactType
!set IsPropertyOfFactType1.name := 'has'
!create Role_155 : FactTypeRole
!insert (ObjectType1,Role_155) into NounConcept_Role
!create Role_26 : FactTypeRole
!set Role_26.name := 'name'
!insert (Text1,Role_26) into NounConcept_Role
!insert (Role_26,IsPropertyOfFactType1) into FactTypeRole_FactType
!insert (Role_155,IsPropertyOfFactType1) into FactTypeRole_FactType
!set Role_155.order := 1
!set Role_26.order := 2

-- AssociativeFactType

!create AssociativeFactType3 : AssociativeFactType
!set AssociativeFactType3.name := 'has'
!create Role_90 : FactTypeRole
!insert (ObjectType8,Role_90) into NounConcept_Role
!insert (Role_90,AssociativeFactType3) into FactTypeRole_FactType
!set AssociativeFactType3.name := 'is part of'
!create Role_91 : FactTypeRole
!insert (ObjectType5,Role_91) into NounConcept_Role
!insert (Role_91,AssociativeFactType3) into FactTypeRole_FactType
!set Role_90.order := 1
!set Role_91.order := 2

-- PartitiveFactType

!create PartitiveFactType1 : PartitiveFactType
!create Role_106 : FactTypeRole
!insert (ObjectType10,Role_106) into NounConcept_Role
!insert (Role_106,PartitiveFactType1) into FactTypeRole_FactType
!set PartitiveFactType1.name := 'includes'
!create Role_107 : FactTypeRole
!insert (ObjectType9,Role_107) into NounConcept_Role
!insert (Role_107,PartitiveFactType1) into FactTypeRole_FactType
!set Role_106.order := 1
!set Role_107.order := 2

-- Characteristic
```

```
!create Characteristic1: Characteristic
!create Role_51 : FactTypeRole
!insert (ObjectType7,Role_51) into NounConcept_Role
!insert (Role_51,Characteristic1) into FactTypeRole_FactType
!set Characteristic1.name := 'being conferencePaper'
!set Role_51.order := 1

-- ReferenceScheme

!create ReferenceScheme1 : ReferenceScheme
!insert (ObjectType1, ReferenceScheme1) into
Concept_ReferenceScheme
!insert (ReferenceScheme1, Role_26) into
ReferenceScheme_SimplyUsedRole

-- StructuralRule

!create StructuralRule1 : StructuralRule
!create UniversalQuantification1 : ClosedUniversalQuantification
!insert (StructuralRule1,UniversalQuantification1) into
Proposition_ClosedLogicalFormulation
!create Variable1X : Variable
!insert (Variable1X,ObjectType1) into Variable_RangedOverConcept
!insert (UniversalQuantification1,Variable1X) into
Quantification_IntroducedVariable
!create ExactlyOneQuantification1 : ExactlyOneQuantification
!create NumberOne1 : NonNegativeInteger
!set NumberOne1.value := 1
!insert (ExactlyOneQuantification1, NumberOne1) into
ExactlyNQuantification_Cardinality
!insert (UniversalQuantification1,ExactlyOneQuantification1) into
Quantification_ScopeFormulation
!create Variable1Y : Variable
!insert (Variable1Y,Role_26) into Variable_RangedOverConcept
!insert (ExactlyOneQuantification1,Variable1Y) into
Quantification_IntroducedVariable
!create AtomicFormulation1: AtomicFormulation
!insert (ExactlyOneQuantification1,AtomicFormulation1) into
Quantification_ScopeFormulation
!insert (AtomicFormulation1,IsPropertyOfFactType1) into
AtomicFormulation_FactType
!create RoleBinding1X : RoleBinding
!insert (AtomicFormulation1,RoleBinding1X) into
AtomicFormulation_RoleBinding
!insert (Role_155, RoleBinding1X) into FactTypeRole_RoleBinding
!insert (RoleBinding1X, Variable1X) into RoleBinding_BindableTarget
!create RoleBinding1Y : RoleBinding
!insert (AtomicFormulation1,RoleBinding1Y) into
AtomicFormulation_RoleBinding
!insert (Role_26, RoleBinding1Y) into FactTypeRole_RoleBinding
!insert (RoleBinding1Y, Variable1Y) into RoleBinding_BindableTarget
```

Appendix F (Chapter 5): methods for creating SBVR meanings schema units

This Appendix describes some of the methods for creating instances of SBVR Meanings schema units from their characterization objects, as described in Chapter 5. For each chacterization objects there is a procedure in an .assl file.

Note that the USE tool does neither allow that a method class a second method nor to define recursive processes within a method. Additionally, the only type of loop allowed is the "for .. in." With these limitations on the executable language it is not possible to fully automatize the generation of schema units of all characterization objects as defined, declaratively, in Chapter 6. In particular, the methods that create structural rules or closed projections only covers the cases found in the DBLP example. The description of all methods is available at (Raventós 2008b).

The method for creating the schema units of all instances of *IndividualConceptCh* is defined as follows:

```
procedure CreateUnitOfIndividualConceptCh()
  var i:IndividualConcept;
  begin
  for el:IndividualConceptCh in
        [IndividualConceptCh.allInstances->asSequence]
   begin
     i := Create(IndividualConcept);
     [i].name := [el.name];
   end;
  end;
```

The method for creating the schema units of all instances of *FactTypeCh* is defined as follows:

```
procedure CreateUnitOfFactTypeCh()
  var  a:AssociativeFactType, i:IsPropertyOfFactType,
     c:CategorizationFactType, p:PartitiveFactType,
     ch:Characteristic, fr:FactTypeRole, n:NounConcept;
  begin
  for el:FactTypeCh in [FactTypeCh.allInstances->asSequence]
   begin
     if [el.type = #Associative] then
     begin
        a := Create(AssociativeFactType);
        [a].name := [el.name];
        for ro:RoleOfFactType in [el.roleOfFactType ->
                   asSequence]
        begin
           fr := Create(FactTypeRole);
           Insert(FactTypeRole_FactType, [fr], [a]);
           [fr].order := [ro.order];
           if [ro.name<>oclUndefined(String)] then
              begin
                 [fr].name := [ro.name];
              end;
```

```
      n := Any([NounConcept.allInstances ->
            select(n:NounConcept| n.name =
                  ro.rangesOverConcept)->Sequence]);
      Insert(NounConcept_Role,  [n], [fr]);
    end;
  end;

  if [el.type = #IsPropertyOf] then
  begin
    i := Create(IsPropertyOfFactType);
    [i].name := [el.name];
    for ro:RoleOfFactType in [el.roleOfFactType ->
            asSequence]
    begin
      fr := Create(FactTypeRole);
      Insert(FactTypeRole_FactType, [fr], [i]);
      [fr].order := [ro.order];
      if [ro.name<>oclUndefined(String)] then
        begin
          [fr].name := [ro.name];
        end;
      n := Any([NounConcept.allInstances ->
            select(n:NounConcept| n.name =
                ro.rangesOverConcept)->asSequence]);
      Insert(NounConcept_Role,  [n], [fr]);
    end;
  end;

  if [el.type = #Partitive] then
  begin
    p := Create(PartitiveFactType);
    [p].name := [el.name];
    for ro:RoleOfFactType in [el.roleOfFactType ->
            asSequence]
    begin
      fr := Create(FactTypeRole);
      Insert(FactTypeRole_FactType, [fr], [p]);
      [fr].order := [ro.order];
      if [ro.name<>oclUndefined(String)] then
        begin
          [fr].name := [ro.name];
        end;
      n := Any([NounConcept.allInstances ->
            select(n:NounConcept| n.name =
                ro.rangesOverConcept)->asSequence]);
      Insert(NounConcept_Role,  [n], [fr]);
    end;
  end;

  if [el.type = #Categorization] then
  begin
    c := Create(CategorizationFactType);
    [c].name := [el.name];
    for ro:RoleOfFactType in [el.roleOfFactType ->
            asSequence]
    begin
      fr := Create(FactTypeRole);
      Insert(FactTypeRole_FactType, [fr], [c]);
      [fr].order := [ro.order];
```

```
        if [ro.name<>oclUndefined(String)] then
          begin
            [fr].name := [ro.name];
          end;
        n := Any([NounConcept.allInstances ->
              select(n:NounConcept| n.name =
                  ro.rangesOverConcept)->asSequence]);
        Insert(NounConcept_Role,  [n], [fr]);
      end;
    end;

    if [el.type = #Characteristic] then
    begin
      ch := Create(Characteristic);
      [ch].name := [el.name];
      for ro:RoleOfFactType in [el.roleOfFactType ->
              asSequence]
      begin
        fr := Create(FactTypeRole);
        Insert(FactTypeRole_FactType, [fr], [ch]);
        [fr].order := [ro.order];
        if [ro.name<>oclUndefined(String)] then
          begin
            [fr].name := [ro.name];
          end;
        n := Any([NounConcept.allInstances ->
              select(n:NounConcept| n.name =
                  ro.rangesOverConcept)->asSequence]);
        Insert(NounConcept_Role,  [n], [fr]);
      end;
    end;
  end;
 end;
```

The method for creating the schema units of all instances of *CategorizationSchemeCh* is defined as follows:

```
     procedure CreateUnitOfCategorizationSchemeCh()
   var  c:CategorizationScheme, se:Segmentation, co:Concept,
        ob:ObjectType;
   begin
   for el:CategorizationSchemeCh in
     [CategorizationSchemeCh.allInstances->asSequence]
     begin
       if [el.isSegmentation] then
       begin
         se := Create(Segmentation);
         [se].name := [el.name];
         ob := Any([ObjectType.allInstances ->
                select(o:Concept|o.name = el.generalConcept)->
                asSequence]);
         Insert(CategorizationScheme_GeneralConcept, [se],[ob]);
         for st:String in [el.category -> asSequence]
         begin
           co := Any([Concept.allInstances ->select(c:Concept|
                         c.name = st) -> asSequence]);
           Insert(CategorizationScheme_Category,[se], [co]);
         end;
```

```
      end
      else
      begin
        c := Create(CategorizationScheme);
        [c].name := [el.name];
        ob := Any([ObjectType.allInstances ->
                select(o:Concept|o.name = el.generalConcept)->
                asSequence]);
        Insert(CategorizationScheme_GeneralConcept,[c], [ob]);
        for st:String in [el.category -> asSequence]
        begin
            co := Any([Concept.allInstances ->select(c:Concept|
                        c.name = st) ->asSequence]);
            Insert(CategorizationScheme_Category,[c], [co]);
        end;
      end;
    end;
  end;
```

Appendix G (Chapter 6): methods to materialize *sbvrEquivalents()* operations

This Appendix describes some of the methods to materialize the *sbvrEquivalents()* operations described in Chapter 6. In particular it describes the methods to materialize the *Class::sbvrEquivalents()* and *Association::sbvrEquivalents()*. The description of all methods is available at (Raventós 2008b).

```
procedure sbvrEquivalentsOfClass()
var ob:NounConceptCh,genSet:GeneralizationSet,bio1:BinaryOperation,
 bio2:BinaryOperation, bio3:BinaryOperation, v1:Variable2,
 v2:Variable2, at1:Atomic, at2:Atomic, at3:Atomic, at4:Atomic,
 at5:Atomic, bin1:Binding, bin2:Binding, bin3:Binding,
 bin4:Binding,bin5:Binding,       bin6:Binding,       bin7:Binding,
bin8:Binding,
 bin9:Binding, bin10:Binding;

begin
for e:Class in [Class.allInstances->select(c| c.isSchemaUnit() and
            not c.oclIsTypeOf(AssociationClass))-> asSequence]
  begin
  ob := Create( NounConceptCh );
  [ob].name := [e.name];
  Insert( SbvrEquivalents, [e], [ob] );
  if [e.generalization_->exists(ge| ge.generalizationSet->
  notEmpty)] then
  begin
    genSet := Any([e.generalization_->select(ge|
      ge.generalizationSet->notEmpty)->any(true).
      generalizationSet->asSequence]);
   end;

  if [e.isAbstract and genSet<>oclUndefined(GeneralizationSet) and
      genSet.generalization->size()=2] then
  begin

    bio1 := Create(BinaryOperation);
    Insert(NounConceptCh_Formulation, [ob],[bio1]);

    v1 := Create(Variable2);
    Insert(NounConceptCh_ProjectionVariable, [ob],[v1]);
    [v1].rangedOverConcept := [e.name];

    [bio1].type := [#Disjunction];

    at1 := Create(Atomic);
    Insert (First_BinaryOperation, [at1],[bio1]);
    [at1].factTypeName := ['is a category of'];
    [at1].type := [#Categorization];

    bin1 := Create(Binding);
    [bin1].order := [2];
    Insert(Atomic_Binding, [at1], [bin1]);
    Insert(Binding_Variable, [bin1], [v1]);
    [bin1].rangesOverConcept := [e.name];

    bin2 := Create(Binding);
    [bin2].order := [1];
    Insert(Atomic_Binding, [at1], [bin2]);
    Insert(Binding_Variable, [bin2], [v1]);
    [bin2].rangesOverConcept := [genSet.generalization->
      asSequence->first.specific.name];

     at2 := Create(Atomic);
    Insert (Second_BinaryOperation, [at2],[bio1]);
    [at2].factTypeName := ['is a category of'];
    [at2].type := [#Categorization];

    bin3 := Create(Binding);
    [bin3].order := [2];
    Insert(Atomic_Binding, [at2], [bin3]);
    Insert(Binding_Variable, [bin3], [v1]);
    [bin3].rangesOverConcept := [e.name];

    bin4 := Create(Binding);
    [bin4].order := [1];
    Insert(Atomic_Binding, [at2], [bin4]);
    Insert(Binding_Variable, [bin4], [v1]);
    [bin4].rangesOverConcept := [genSet.generalization->
      asSequence->last.specific.name];

end;

if [e.isAbstract and genSet<>oclUndefined(GeneralizationSet) and
    genSet.generalization->size()=3] then
begin
    bio2 := Create(BinaryOperation);

    v2 := Create(Variable2);
    Insert(NounConceptCh_ProjectionVariable, [ob],[v2]);
    [v2].rangedOverConcept := [e.name];

    [bio2].type := [#Disjunction];
    Insert(NounConceptCh_Formulation, [ob],[bio2]);

    at3 := Create(Atomic);
    Insert (First_BinaryOperation, [at3],[bio2]);
    [at3].factTypeName := ['is a category of'];
    [at3].type := [#Categorization];

    bin5 := Create(Binding);
    [bin5].order := [2];
    Insert(Atomic_Binding, [at3], [bin5]);
    Insert(Binding_Variable, [bin5], [v2]);
    [bin5].rangesOverConcept := [e.name];
    bin6 := Create(Binding);
    [bin6].order := [1];
    Insert(Atomic_Binding, [at3], [bin6]);
    Insert(Binding_Variable, [bin6], [v2]);
    [bin6].rangesOverConcept := [genSet.generalization->
      asSequence->first.specific.name];

    bio3 := Create(BinaryOperation);
    [bio3].type := [#Disjunction];
    Insert (Second_BinaryOperation, [bio3],[bio2]);

    at4 := Create(Atomic);
    Insert (First_BinaryOperation, [at4],[bio3]);
    [at4].factTypeName := ['is a category of'];
    [at4].type := [#Categorization];

    bin7 := Create(Binding);
    [bin7].order := [2];
    Insert(Atomic_Binding, [at4], [bin7]);
    Insert(Binding_Variable, [bin7], [v2]);
    [bin7].rangesOverConcept := [e.name];

    bin8 := Create(Binding);
    [bin8].order := [1];
    Insert(Atomic_Binding, [at4], [bin8]);
    Insert(Binding_Variable, [bin8], [v2]);
    [bin8].rangesOverConcept := [genSet.generalization->
      asSequence->at(2).specific.name];

    at5 := Create(Atomic);
    Insert (Second_BinaryOperation, [at5],[bio3]);
    [at5].factTypeName := ['is a category of'];
    [at5].type := [#Categorization];

    bin9 := Create(Binding);
    [bin9].order := [2];
    Insert(Atomic_Binding, [at5], [bin9]);
    Insert(Binding_Variable, [bin9], [v2]);
    [bin9].rangesOverConcept := [e.name];

    bin10 := Create(Binding);
    [bin10].order := [1];
    Insert(Atomic_Binding, [at5], [bin10]);
    Insert(Binding_Variable, [bin10], [v2]);
    [bin10].rangesOverConcept := [genSet.generalization->
      asSequence->last.specific.name];
  end;

 end;
end;

procedure sbvrEquivalentsOfAssociation()
var ro:RoleCh, fa:FactTypeCh, r1:RoleOfFactType, r2:RoleOfFactType,
 asstype:AggregationKind, st:StructuralRuleCh,
 qf1:QuantificationForm, v1:Variable2, v2:Variable2,
qf2:QuantificationForm, at:Atomic, bi1:Binding, bi2:Binding;

 begin
 for e:Association in [Association.allInstances->select(a|
   a.isSchemaUnit())->asSequence]
 begin
 asstype := [if e.memberEnd->exists(pr|
      pr.aggregation_=#composite) then #composite
      else if e.memberEnd->exists(pr|pr.aggregation_=#shared)
           then #shared else #none endif endif];

   fa := Create(FactTypeCh);
   if [asstype = #composite] then
   begin
     [fa].name := [if e.name=oclUndefined(String) then 'includes'
          else e.name endif];
     [fa].type := [#Partitive];
   end
   else
     begin
        [fa].type := [#Associative];
        [fa].name := [if asstype = #shared then 'is part of' else
                   if (e.name<>oclUndefined(String) and not
                         e.oclIsTypeOf(AssociationClass))
               then e.name else 'has' endif endif];
     end;

   Insert( SbvrEquivalents, [e], [fa] );

   r1 := Create(RoleOfFactType);
   Insert( FactTypeCh_RoleOfFactType, [fa], [r1] );
   [r1].name := [e.memberEnd->sortedBy(order)->first.name];
   [r1].rangesOverConcept := [e.memberEnd->sortedBy(order)->
        first.type.name];
   [r1].order := [1];

   r2 := Create(RoleOfFactType);
   Insert( FactTypeCh_RoleOfFactType, [fa], [r2] );
    [r2].name := [e.memberEnd->sortedBy(order)->last.name];
    [r2].rangesOverConcept := [e.memberEnd->sortedBy(order)->
        last.type.name];
    [r2].order := [2];

    for me:Property in [e.memberEnd]
     begin
        if [me.lower()>0 or
           me.upperValue.oclIsTypeOf(LiteralInteger)] then
       begin
          st := Create(StructuralRuleCh);
          Insert( SbvrEquivalents, [e], [st] );

          qf1 := Create(QuantificationForm);
          Insert( StructuralRuleCh_Formulation, [st],[qf1]);
          [qf1].type := [#ClosedUniversal];
```

```
v1 := Create(Variable2);
Insert(QuantificationForm_IntroducedVar, [qf1],[v1]);
[v1].rangedOverConcept := [if e.memberEnd->
  select(other| other<>me)->
  any(true).name<>oclUndefined(String) then
  e.memberEnd->select(other|other<>me)->any(true).name
  else
  e.memberEnd->select(other|other<>me)->
  any(true).type.name endif ];

qf2 := Create(QuantificationForm);
Insert( Formulation_QuantificationForm, [qf2],[qf1]);

if [me.lower() =
   me.upperValue.oclAsType(LiteralInteger).value and
   me.lower() = 1] then
begin
 [qf2].type := [#ExactlyOne];
 [qf2].card := [1];
end;
if [me.lower() =
  me.upperValue.oclAsType(LiteralInteger).value and
  me.lower() <> 1] then
begin
 [qf2].type := [#ExactlyN];
 [qf2].card := [me.lower()];
end;
if [me.lower() = 1 and
   me.upperValue.oclIsTypeOf(LiteralUnlimitedNatural)]
then
begin
 [qf2].type := [#Existential];
 [qf2].minimCard := [1];
end;
if [me.lower()>1 and
  me.upperValue.oclIsTypeOf(LiteralUnlimitedNatural)]
then
begin
 [qf2].type := [#AtLeastN];
 [qf2].minimCard := [me.lower()];
end;
if [me.lower()=0 and
me.upperValue.oclIsTypeOf(LiteralInteger)] then
begin
 [qf2].type := [#AtMostN];
 [qf2].maxCard :=
  [me.upperValue.oclAsType(LiteralInteger).value];
end;
if [me.lower()=0 and
  me.upperValue.oclAsType(LiteralInteger).value=1] then
begin
 [qf2].type := [#AtMostOne];
 [qf2].maxCard := [1];
end;
if [me.lower()>1 and
```

```
 me.upperValue.oclIsTypeOf(LiteralInteger)] then
begin
 [qf2].type := [#NumericRange];
 [qf2].minimCard := [me.lower()];
 [qf2].maxCard :=
   [me.upperValue.oclAsType(LiteralInteger).value];
end;
v2 := Create(Variable2);
Insert(QuantificationForm_IntroducedVar, [qf2],[v2]);
[v2].rangedOverConcept := [if me.name <>
        oclUndefined(String) then me.name
        else me.type.name endif];

at := Create(Atomic);
Insert( Formulation_QuantificationForm, [at],[qf2]);
[at].type := [fa.type];

if [me.order = 1] then
begin
   [at].factTypeName := [fa.name];
end
else
begin
   [at].factTypeName :=
        [if asstype = #composite then
              if e.name<>oclUndefined(String)
              then 'is included in' else e.name endif
        else
              if asstype = #shared then 'is part of'
              else
        if (e.name<>oclUndefined(String) and not
                    e.oclIsTypeOf(AssociationClass))
              then e.name else 'has' endif
              endif
        endif];
end;

bi1 := Create(Binding);
Insert (Atomic_Binding, [at],[bi1]);
[bi1].rangesOverConcept :=
        [if e.memberEnd->select(other| other<>me)->
              any(true).name<>oclUndefined(String)
        then e.memberEnd->select(other|other<>me)->
              any(true).name
        else e.memberEnd->select(other|other<>me)->
              any(true).type.name
        endif];
Insert (Binding_Variable,[bi1],[v1]);
[bi1].order := [if me.order=1 then 2 else 1 endif];

bi2 := Create(Binding);
Insert (Atomic_Binding, [at],[bi2]);
[bi2].rangesOverConcept :=
        [if me.name<>oclUndefined(String) then me.name
        else me.type.name endif];
```

```
          Insert (Binding_Variable,[bi2],[v2]);
          [bi2].order := [me.order];

        end;
      end;
  end;
end;
```

Appendix H (Chapter 6): methods to materialize *includedInUml()* operations

This Appendix describes the method to materialize the *ObjectType::includedInUml()* operation described in Chapter 6. The description of all methods is available at (Raventós 2008b).

```
procedure includedInUmlOfObjectType()
  var cl:ClassCh, acl:AssociationClassCh, as:AssociativeFactType,
str:Sequence(StructuralRule),str1:StructuralRule,
str2:StructuralRule, quan1:Quantification, quan2:Quantification,
me:AssociationClassMemberEnd;

begin
for e:ObjectType in [ObjectType.allInstances->reject(ob|
  ob.oclIsTypeOf(CategorizationScheme) or
  ob.oclIsTypeOf(Segmentation))->asSequence]
  begin
     if [e.closedProjection->isEmpty() or
        (e.closedProjection->notEmpty() and
        ((e.closedProjection.logicalFormulation.
        oclAsType(Disjunction).logicalOperand1.
        oclAsType(AtomicFormulation) <>
        oclUndefined(AtomicFormulation)  and
        e.closedProjection.logicalFormulation.
        oclAsType(Disjunction).logicalOperand1.
        oclAsType(AtomicFormulation).factType.
        oclIsTypeOf(CategorizationFactType) and
        e.closedProjection.logicalFormulation.
        oclAsType(Disjunction).logicalOperand2.
        oclAsType(AtomicFormulation) <>
        oclUndefined(AtomicFormulation) and
        e.closedProjection.logicalFormulation.
        oclAsType(Disjunction).logicalOperand2.
        oclAsType(AtomicFormulation).factType.
        oclIsTypeOf(CategorizationFactType)) or
        (e.closedProjection.logicalFormulation.
        oclAsType(Disjunction).logicalOperand1.
        oclAsType(AtomicFormulation) <>
        oclUndefined(AtomicFormulation) and
        e.closedProjection.logicalFormulation.
        oclAsType(Disjunction).logicalOperand1.
        oclAsType(AtomicFormulation).factType.
        oclIsTypeOf(CategorizationFactType) and
        e.closedProjection.logicalFormulation.
        oclAsType(Disjunction).logicalOperand2.
        oclAsType(Disjunction).logicalOperand1.
        oclAsType(AtomicFormulation) <>
        oclUndefined(AtomicFormulation) and
        e.closedProjection.logicalFormulation.
        oclAsType(Disjunction).logicalOperand2.
```

```
    oclAsType(Disjunction).logicalOperand1.
    oclAsType(AtomicFormulation).factType.
    oclIsTypeOf(CategorizationFactType) and
    e.closedProjection.logicalFormulation.
    oclAsType(Disjunction).logicalOperand2.
    oclAsType(Disjunction).logicalOperand2.
    oclAsType(AtomicFormulation) <>
    oclUndefined(AtomicFormulation) and
    e.closedProjection.logicalFormulation.
    oclAsType(Disjunction).logicalOperand2.
    oclAsType(Disjunction).logicalOperand2.
    oclAsType(AtomicFormulation).factType.
    oclIsTypeOf(CategorizationFactType))))]
  then
  begin
    cl := Create( ClassCh );
    [cl].name := [e.name];
    Insert( IncludedInUml, [e], [cl] );

    if [e.closedProjection->notEmpty() and
      ((e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand1.
      oclAsType(AtomicFormulation) <>
      oclUndefined(AtomicFormulation)  and
      e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand1.
      oclAsType(AtomicFormulation).factType.
      oclIsTypeOf(CategorizationFactType) and
      e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand2.
      oclAsType(AtomicFormulation) <>
      oclUndefined(AtomicFormulation) and
      e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand2.
      oclAsType(AtomicFormulation).factType.
      oclIsTypeOf(CategorizationFactType)) or
      (e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand1.
      oclAsType(AtomicFormulation) <>
      oclUndefined(AtomicFormulation) and
      e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand1.
      oclAsType(AtomicFormulation).factType.
      oclIsTypeOf(CategorizationFactType) and
      e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand2.
      oclAsType(Disjunction).logicalOperand1.
      oclAsType(AtomicFormulation) <>
      oclUndefined(AtomicFormulation) and
      e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand2.
      oclAsType(Disjunction).logicalOperand1.
      oclAsType(AtomicFormulation).factType.
      oclIsTypeOf(CategorizationFactType) and
      e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand2.
      oclAsType(Disjunction).logicalOperand2.
      oclAsType(AtomicFormulation) <>
      oclUndefined(AtomicFormulation) and
      e.closedProjection.logicalFormulation.
      oclAsType(Disjunction).logicalOperand2.
        oclAsType(Disjunction).logicalOperand2.
      oclAsType(AtomicFormulation).factType.
      oclIsTypeOf(CategorizationFactType)))]
    then
    begin
      [cl].isAbstract := [true];
    end
    else
    begin
      [cl].isAbstract := [false];
    end;
  end;
  if [e.closedProjection->notEmpty() and
  e.closedProjection.logicalFormulation.
      oclIsTypeOf(Objectification)]
  then
  begin
    acl := Create( AssociationClassCh );
    [acl].name := [e.name];
    Insert( IncludedInUml, [e], [acl] );

    as := [e.closedProjection.logicalFormulation.
      oclAsType(Objectification).
      consideredLogicalFormulation.
      oclAsType(AtomicFormulation).factType.
      oclAsType(AssociativeFactType)];
    str1 := [oclUndefined(StructuralRule)];
    str2 := [oclUndefined(StructuralRule)];
    quan1 := [oclUndefined(Quantification)];
    quan2 := [oclUndefined(Quantification)];

    if [StructuralRule.allInstances->select(st|
      st.closedLogicalFormulation.
      oclAsType(ClosedUniversalQuantification).
      scopeFormulation.oclAsType(Quantification).
      scopeFormulation.oclAsType(AtomicFormulation).
        factType = as and
      st.closedLogicalFormulation.
      oclAsType(ClosedUniversalQuantification).
      scopeFormulation.oclAsType(Quantification).
      scopeFormulation.oclAsType(AtomicFormulation).
      factType.factTypeRole->sortedBy(order) = as.factTypeRole
        ->sortedBy(order))->notEmpty]
    then
    begin
      str := [StructuralRule.allInstances->select(st|
        st.closedLogicalFormulation.
        oclAsType(ClosedUniversalQuantification).
          scopeFormulation.oclAsType(Quantification).
          scopeFormulation.oclAsType(AtomicFormulation).
          factType = as and
        st.closedLogicalFormulation.
        oclAsType(ClosedUniversalQuantification).
        scopeFormulation.oclAsType(Quantification).
        scopeFormulation.oclAsType(AtomicFormulation).factType.
        factTypeRole->sortedBy(order) = as.factTypeRole->
          sortedBy(order))->asSequence];

      if [str->select(st| st.closedLogicalFormulation.
        oclAsType(ClosedUniversalQuantification).
        introducedVariable.rangedOverConcept =
        as.factTypeRole->sortedBy(order)->last.nounConcept)->
        notEmpty]
      then
      begin
        str2 := Any([str->select(st|
          st.closedLogicalFormulation.
            oclAsType(ClosedUniversalQuantification).
            introducedVariable.rangedOverConcept =
          as.factTypeRole->sortedBy(order)->last.nounConcept
            and st.closedLogicalFormulation.
            oclAsType(ClosedUniversalQuantification).
            scopeFormulation.oclAsType(Quantification).
            scopeFormulation.oclAsType(AtomicFormulation).
            factType = as) ]);

        quan2 := [str2.closedLogicalFormulation.
          oclAsType(ClosedUniversalQuantification).
            scopeFormulation.oclAsType(Quantification)];
      end;

      if [str->select(st| st.closedLogicalFormulation.
        oclAsType(ClosedUniversalQuantification).
          introducedVariable.rangedOverConcept =
        as.factTypeRole->sortedBy(order)->first.nounConcept)->
        notEmpty()]
      then
      begin
        str1 := Any([str->select(st|
          st.closedLogicalFormulation.
          oclAsType(ClosedUniversalQuantification).
          introducedVariable.rangedOverConcept =
          as.factTypeRole->sortedBy(order)->first.nounConcept
          and st.closedLogicalFormulation.
          oclAsType(ClosedUniversalQuantification).
          scopeFormulation.oclAsType(Quantification).
          scopeFormulation.oclAsType(AtomicFormulation).
          factType = as )]);

        quan1 := [str1.closedLogicalFormulation.
          oclAsType(ClosedUniversalQuantification).
          scopeFormulation.oclAsType(Quantification)];
      end;
    end;
```

```
for ro:FactTypeRole in [as.factTypeRole->sortedBy(order)]
begin
  me := Create(AssociationClassMemberEnd);
  Insert(AssociationClassCh_AssociationClassMemberEnd,
    [acl], [me]);
  [me].name := [ro.name];
  [me].typeName := [ro.nounConcept.name];
  [me].isDerived := [false];
  [me].isDerivedUnion := [false];
  [me].aggregation_ := [if e.name = 'includes' and
    ro.order = 1 then #composite else
    if e.name = 'is part of' and ro.order = 2 then #shared
    else #none endif endif];
  [me].order := [ro.order];

  if [ro.order = 1] then
  begin
    if [str2 = oclUndefined(StructuralRule)] then
    begin
        [me].lowerValue := [0];
    end;
    if [str2 <> oclUndefined(StructuralRule)] then
    begin

      if [quan2.oclIsTypeOf(AtLeastNQuantification) or
        quan2.oclIsTypeOf(ExistentialQuantification)]
      then
      begin
      [me].lowerValue :=
      [quan2.oclAsType(AtLeastNQuantification).
           minimumCardinality.value];
      end;
      if [quan2.oclIsTypeOf(AtMostNQuantification) or
        quan2.oclIsTypeOf(AtMostOneQuantification)] then
      begin
        [me].lowerValue := [0];
        [me].upperValue :=
           [quan2.oclAsType(AtMostNQuantification).
                 maximumCardinality.value];
      end;
      if [quan2.oclIsTypeOf(ExactlyNQuantification) or
        quan2.oclIsTypeOf(ExactlyOneQuantification)] then
      begin
        [me].lowerValue :=
           [quan2.oclAsType(ExactlyNQuantification).
           cardinality.value];
        [me].upperValue :=
           [quan2.oclAsType(ExactlyNQuantification).
           cardinality.value];
      end;
      if [quan2.oclIsTypeOf(NumericRangeQuantification)]
      then
      begin
        [me].lowerValue :=
               [quan2.oclAsType(NumericRangeQuantification).
               minimumCardinality.value];
            [me].upperValue :=
               [quan2.oclAsType(NumericRangeQuantification).
                     maximumCardinality.value];
         end;
       end;
     end
     else
     begin
       if [str1 = oclUndefined(StructuralRule)] then
       begin
            [me].lowerValue := [0];
       end;
       if [str1 <> oclUndefined(StructuralRule)] then
       begin

         if [quan1.oclIsTypeOf(AtLeastNQuantification) or
           quan1.oclIsTypeOf(ExistentialQuantification)] then
         begin
            [me].lowerValue :=
               [quan1.oclAsType(AtLeastNQuantification).
               minimumCardinality.value];
         end;
         if [quan1.oclIsTypeOf(AtMostNQuantification) or
           quan1.oclIsTypeOf(AtMostOneQuantification)] then
         begin
            [me].lowerValue := [0];
            [me].upperValue :=
               [quan1.oclAsType(AtMostNQuantification).
               maximumCardinality.value];
         end;
         if [quan1.oclIsTypeOf(ExactlyNQuantification) or
           quan1.oclIsTypeOf(ExactlyOneQuantification)] then
         begin
            [me].lowerValue :=
               [quan1.oclAsType(ExactlyNQuantification).
               cardinality.value];
            [me].upperValue :=
               [quan1.oclAsType(ExactlyNQuantification).
               cardinality.value];
         end;
         if [quan1.oclIsTypeOf(NumericRangeQuantification)]
         then
         begin
            [me].lowerValue :=
               [quan1.oclAsType(NumericRangeQuantification).
               minimumCardinality.value];
            [me].upperValue :=
               [quan1.oclAsType(NumericRangeQuantification).
               maximumCardinality.value];
         end; end;
       end; end;
   end; end;
end;
```

Appendix I (Chapter 7): SBVR Structured English metaschema in USE

The following is a complete specification, suitable for validation with the USE tool, of the SBVR Structured English metaschema presented in Chapter 7. Note that all the associations that are ordered have been specified as an *order* attribute because the ordered keyword in the USE tool does not seem to matter when inserting association links. The keyword is only used to distinguish between *Set* and *Sequence* types when using navigational syntax in OCL expressions[10].

```
-- SBVR Representations

enum FontStyle { term, name, verb, keyword }
enum CaptionType {General_concept, Concept_type, Definition,
                Necessity, Reference_scheme}

abstract class Representation
end

abstract class PrimaryRepresentation < Representation
end

abstract class Caption < Representation
end

class Definition < Caption
end

class Designation < PrimaryRepresentation
end

class FactTypeForm < PrimaryRepresentation
end

class GeneralConceptCaption < Caption
end

class ConceptTypeCaption < Caption
end

class ReferenceSchemeCaption < Caption
end

class Statement < Caption
end

class NecessityStatement < Statement
end

class StructuredEnglishText
```

[10] Information provided by Mark Richters, developer of USE tool.

```
attributes
 value : String
 font : FontStyle
 order : Integer
end

association StructuredEnglishText_Representation between
 StructuredEnglishText[1..*] role structuredEnglishText ordered
 Representation[1] role representation
end

association Meaning_Representation between
 Meaning[1] role meaning
 Representation[*] role representation
end

association PrimaryRepresentation_Caption between
 PrimaryRepresentation[1] role primaryRepresentation
 Caption[*] role caption
end
```

Appendix J (Chapter 7): methods to materialize *newRepresentation()* operations

This Appendix describes some of the methods to materialize the *newRepresentation()* operations described in Chapter 7. The description of all the methods is available at (Raventós 2008b).

```
    procedure CreateNewRepresentationOfNounConcept()
    var d:Designation, st:StructuredEnglishText,def:Definition,
         dis1:Disjunction, dis2:Disjunction, atom1:AtomicFormulation,
         atom2:AtomicFormulation, atom3:AtomicFormulation,
         cp:ClosedProjection,st1:StructuredEnglishText,
         st2:StructuredEnglishText, st3:StructuredEnglishText,
         st4:StructuredEnglishText, st5:StructuredEnglishText,
         st6:StructuredEnglishText, st7:StructuredEnglishText,
         st8:StructuredEnglishText, st9:StructuredEnglishText,
         st10:StructuredEnglishText,st11:StructuredEnglishText,
  st12:StructuredEnglishText, st13:StructuredEnglishText,
         cat1:CategorizationFactType, cat2:CategorizationFactType,
         cat3:CategorizationFactType, con:Concept,
         ins1:InstantiationFormulation, ins2:InstantiationFormulation;

    begin
    for el:NounConcept in [NounConcept.allInstances->asSequence]
      begin
         d := Create(Designation);
         Insert(Meaning_Representation, [el], [d]);
         st := Create(StructuredEnglishText);
         Insert(StructuredEnglishText_Representation,[st],[d]);
         [st].value := [el.name];
         [st].order := [1];
         [st].font := [#term];

         if [el.closedProjection->notEmpty] then
         begin
         if [el.closedProjection.logicalFormulation->notEmpty and
              el.closedProjection.logicalFormulation.
              oclIsTypeOf(Disjunction) and el.closedProjection.
              logicalFormulation.oclAsType(Disjunction).
              logicalOperand1.oclIsTypeOf(AtomicFormulation) and
              el.closedProjection.logicalFormulation.
              oclAsType(Disjunction).logicalOperand2.
              oclIsTypeOf(AtomicFormulation)] then
            begin
            cp := [el.closedProjection];
            dis1 := [cp.logicalFormulation.oclAsType(Disjunction)];
            atom1 :=
                   [dis1.logicalOperand1.oclAsType(AtomicFormulation)];

            atom2 :=
            [dis1.logicalOperand2.oclAsType(AtomicFormulation)];

   cat1 := [atom1.factType.oclAsType(CategorizationFactType)];
   cat2 := [atom2.factType.oclAsType(CategorizationFactType)];
   con := [cat1.factTypeRole->sortedBy(order)->
            last.nounConcept];

   def := Create(Definition);
   Insert(Meaning_Representation, [con], [def]);
   Insert(PrimaryRepresentation_Caption, [d],[def]);

   st1 := Create(StructuredEnglishText);
   Insert(StructuredEnglishText_Representation,[st1],[def]);
   [st1].value := [cat1.factTypeRole->sortedBy(order)->
     first.nounConcept.name];
   [st1].order := [1];
   [st1].font := [#term];

   st2 := Create(StructuredEnglishText);
   Insert(StructuredEnglishText_Representation,[st2],[def]);
   [st2].value := ['or'];
   [st2].order := [2];
   [st2].font := [#keyword];

   st3 := Create(StructuredEnglishText);
   Insert(StructuredEnglishText_Representation,[st3],[def]);
   [st3].value := [cat2.factTypeRole->sortedBy(order)->
     first.nounConcept.name];
   [st3].order := [3];
   [st3].font := [#term];
end
else
begin

if [el.closedProjection.logicalFormulation->notEmpty and
     el.closedProjection.logicalFormulation.
     oclIsTypeOf(Disjunction) and el.closedProjection.
     logicalFormulation.oclAsType(Disjunction).
     logicalOperand1.oclIsTypeOf(AtomicFormulation) and
     el.closedProjection.logicalFormulation.
     oclAsType(Disjunction).logicalOperand2.
     oclIsTypeOf(Disjunction) and
     el.closedProjection.logicalFormulation.
     oclAsType(Disjunction).logicalOperand2.
     oclAsType(Disjunction).logicalOperand1.
     oclIsTypeOf(AtomicFormulation) and
     el.closedProjection.logicalFormulation.
     oclAsType(Disjunction).logicalOperand2.
     oclAsType(Disjunction).logicalOperand2.
     oclIsTypeOf(AtomicFormulation)] then
begin
   cp := [el.closedProjection];

   dis1 := [cp.logicalFormulation.oclAsType(Disjunction)];
   atom1 :=
     [dis1.logicalOperand1.oclAsType(AtomicFormulation)];

dis2 := [dis1.logicalOperand2.oclAsType(Disjunction)];

atom2 :=
     [dis2.logicalOperand1.oclAsType(AtomicFormulation)];

atom3 :=
     [dis2.logicalOperand2.oclAsType(AtomicFormulation)];

cat1 := [atom1.factType.oclAsType(CategorizationFactType)];
cat2 := [atom2.factType.oclAsType(CategorizationFactType)];
cat3 := [atom3.factType.oclAsType(CategorizationFactType)];

def := Create(Definition);
Insert(Meaning_Representation, [el], [def]);
Insert(PrimaryRepresentation_Caption, [d],[def]);

st1 := Create(StructuredEnglishText);
Insert(StructuredEnglishText_Representation,[st1],[def]);
[st1].value := [cat1.factTypeRole->sortedBy(order)->
                   first.nounConcept.name];
[st1].order := [1];
[st1].font := [#term];

st2 := Create(StructuredEnglishText);
Insert(StructuredEnglishText_Representation,[st2],[def]);
[st2].value := ['or'];
[st2].order := [2];
[st2].font := [#keyword];

st3 := Create(StructuredEnglishText);
Insert(StructuredEnglishText_Representation,[st3],[def]);
[st3].value := [cat2.factTypeRole->sortedBy(order)->
                   first.nounConcept.name];
[st3].order := [3];
[st3].font := [#term];

st4 := Create(StructuredEnglishText);
Insert(StructuredEnglishText_Representation,[st4],[def]);
[st4].value := ['or'];
[st4].order := [4];
[st4].font := [#keyword];

st5 := Create(StructuredEnglishText);
Insert(StructuredEnglishText_Representation,[st5],[def]);
[st5].value := [cat3.factTypeRole->sortedBy(order)->
                   first.nounConcept.name];
[st5].order := [5];
[st5].font := [#term];

end
else
begin
if [el.closedProjection.logicalFormulation->notEmpty and
```

```
            el.closedProjection.logicalFormulation.
            oclIsTypeOf(Disjunction) and el.closedProjection.
            logicalFormulation.oclAsType(Disjunction).
            logicalOperand1.oclIsTypeOf(InstantiationFormulation)
            and el.closedProjection.logicalFormulation.
            oclAsType(Disjunction).logicalOperand2.
            oclIsTypeOf(InstantiationFormulation)]
       then
       begin

         cp := [el.closedProjection];
         dis1 := [cp.logicalFormulation.oclAsType(Disjunction)];
         ins1 := [dis1.logicalOperand1.
                    oclAsType(InstantiationFormulation)];

         ins2 := [dis1.logicalOperand2.
                    oclAsType(InstantiationFormulation)];

       def := Create(Definition);
       Insert(Meaning_Representation, [el], [def]);
       Insert(PrimaryRepresentation_Caption, [d], [def]);

       st1 := Create(StructuredEnglishText);
       Insert(StructuredEnglishText_Representation,[st1],[def]);
       [st1].value := [ins1.bindableTarget.
                          oclAsType(IndividualConcept).name];
       [st1].order := [1];
       [st1].font := [#name];

       st2 := Create(StructuredEnglishText);
       Insert(StructuredEnglishText_Representation,[st2],[def]);
       [st2].value := ['or'];
       [st2].order := [2];
       [st2].font := [#keyword];

       st3 := Create(StructuredEnglishText);
       Insert(StructuredEnglishText_Representation,[st3],[def]);
       [st3].value := [ins2.bindableTarget.
                          oclAsType(IndividualConcept).name];
       [st3].order := [3];
       [st3].font := [#name];
       end;
     end;
   end;
end;
  end;
end;

procedure CreateNewRepresentationOfIndividualConcept()
var d:Designation, st:StructuredEnglishText;
begin
for el:IndividualConcept in [IndividualConcept.allInstances->
                    asSequence]
begin
    d := Create(Designation);
    Insert(Meaning_Representation, [el], [d]);
    st := Create(StructuredEnglishText);
    Insert(StructuredEnglishText_Representation,[st],[d]);
    [st].value := [el.name];
    [st].order := [1];
    [st].font := [#name];

    end;
  end;
```

Appendix K (Chapter 7): DBLP as an instance of SBVR Structured English metaschema

This Appendix lists the commands that have been used to create a fragment of the structural schema of the DBLP example in the USE tool. The schema is created as instances of the SBVR Structured English Metaschema. The whole instantiation is available at (Raventós 2008b).

```
-- Designation

!create Designation2 : Designation
!insert (ObjectType1,Designation2) into Meaning_Representation
!create StructuredEnglishText2 : StructuredEnglishText
!insert (StructuredEnglishText2,Designation2) into
  StructuredEnglishText_Representation
!set StructuredEnglishText2.value := 'person'
!set StructuredEnglishText2.order := 1
!set StructuredEnglishText2.font := #term

-- FactTypeForm

!create FactTypeForm1 : FactTypeForm
!insert (AssociativeFactType1,FactTypeForm1) into
    Meaning_Representation
!create StructuredEnglishText100 : StructuredEnglishText
!insert (StructuredEnglishText100,FactTypeForm1) into
  StructuredEnglishText_Representation
!set @StructuredEnglishText100.value := 'editor'
!set @StructuredEnglishText100.order := 1
!set @StructuredEnglishText100.font := #term
!create StructuredEnglishText101 : StructuredEnglishText
!insert (StructuredEnglishText101,FactTypeForm1) into
  StructuredEnglishText_Representation
!set @StructuredEnglishText101.value := 'is editor of'
!set @StructuredEnglishText101.order := 2
!set @StructuredEnglishText101.font := #verb
!create StructuredEnglishText102 : StructuredEnglishText
!insert (StructuredEnglishText102,FactTypeForm1) into
   StructuredEnglishText_Representation
!set @StructuredEnglishText102.value := 'editedBook'
!set @StructuredEnglishText102.order := 3
!set @StructuredEnglishText102.font := #term

-- Concept Type caption

!create ConceptTypeCaption45 : ConceptTypeCaption
!insert (AssociativeFactType1,ConceptTypeCaption45) into
  Meaning_Representation
!insert (FactTypeForm1,ConceptTypeCaption45) into
  PrimaryRepresentation_Caption
!create StructuredEnglishText103 : StructuredEnglishText
!insert (StructuredEnglishText103,ConceptTypeCaption45) into
  StructuredEnglishText_Representation
```

```
!set @StructuredEnglishText103.value := 'associative fact type'
!set @StructuredEnglishText103.order := 1
!set @StructuredEnglishText103.font := #term

-- Definition

!create Definition1 : Definition
!insert (ObjectType2,Definition1) into Meaning_Representation
!insert (Designation2,Definition1) into
 PrimaryRepresentation_Caption
!create StructuredEnglishText856 : StructuredEnglishText
!insert (StructuredEnglishText856,Definition1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText856.value := 'editedBook'
!set @StructuredEnglishText856.order := 1
!set @StructuredEnglishText856.font := #term
!create StructuredEnglishText857 : StructuredEnglishText
!insert (StructuredEnglishText857,Definition1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText857.value := 'or'
!set @StructuredEnglishText857.order := 2
!set @StructuredEnglishText857.font := #keyword
!create StructuredEnglishText858 : StructuredEnglishText
!insert (StructuredEnglishText858,Definition1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText858.value := 'authoredPublication'
!set @StructuredEnglishText858.order := 3
!set @StructuredEnglishText858.font := #term

-- GeneralConceptCaption

!create GeneralConceptCaption1 : GeneralConceptCaption
!insert (Designation4,GeneralConceptCaption1) into
 Meaning_Representation
!insert (Designation1,GeneralConceptCaption1) into
 PrimaryRepresentation_Caption
!create StructuredEnglishText324 : StructuredEnglishText
!insert (StructuredEnglishText324,GeneralConceptCaption1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText324.value := 'publication'
!set @StructuredEnglishText324.order := 1
!set @StructuredEnglishText324.font := #term

-- ConceptTypeCaption

!create ConceptTypeCaption1 : ConceptTypeCaption
!insert (Role_10,ConceptTypeCaption1) into Meaning_Representation
!insert (Designation1,ConceptTypeCaption1) into
 Meaning_Representation
!create StructuredEnglishText24 : StructuredEnglishText
!insert (StructuredEnglishText24,ConceptTypeCaption1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText24.value := 'role'
!set @StructuredEnglishText24.order := 1
!set @StructuredEnglishText24.font := #term

-- NecessityStatement
!create NecessityStatement1 : NecessityStatement
!insert (StructuralRule1,NecessityStatement1) into
```

```
 Meaning_Representation
!insert (FactTypeForm3,NecessityStatement1) into
 PrimaryRepresentation_Caption
!create StructuredEnglishText333 : StructuredEnglishText
!insert (StructuredEnglishText333,NecessityStatement1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText333.value := 'each'
!set @StructuredEnglishText333.order := 1
!set @StructuredEnglishText333.font := #keyword
!create StructuredEnglishText334 : StructuredEnglishText
!insert (StructuredEnglishText334,NecessityStatement1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText334.value := 'person'
!set @StructuredEnglishText334.order := 2
!set @StructuredEnglishText334.font := #term
!create StructuredEnglishText335 : StructuredEnglishText
!insert (StructuredEnglishText335,NecessityStatement1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText335.value := 'has'
!set @StructuredEnglishText335.order := 3
!set @StructuredEnglishText335.font := #verb
!create StructuredEnglishText336 : StructuredEnglishText
!insert (StructuredEnglishText336,NecessityStatement1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText336.value := 'exactly'
!set @StructuredEnglishText336.order := 4
!set @StructuredEnglishText336.font := #keyword
!create StructuredEnglishText337 : StructuredEnglishText
!insert (StructuredEnglishText337,NecessityStatement1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText337.value := 'one'
!set @StructuredEnglishText337.order := 5
!set @StructuredEnglishText337.font := #term
!create StructuredEnglishText338 : StructuredEnglishText
!insert (StructuredEnglishText338,NecessityStatement1) into
 StructuredEnglishText_Representation
!set @StructuredEnglishText338.value := 'name'

-- ReferenceSchemeCaption

!create ReferenceSchemeCaption1 : ReferenceSchemeCaption
!insert (ObjectType1,ReferenceSchemeCaption1) into
 Meaning_Representation
!create StructuredEnglishText93 : StructuredEnglishText
!insert     (StructuredEnglishText93,ReferenceSchemeCaption1)     into
  StructuredEnglishText_Representation
!set @StructuredEnglishText93.value := 'name'
!set @StructuredEnglishText93.order := 1
!set @StructuredEnglishText93.font := #term
```