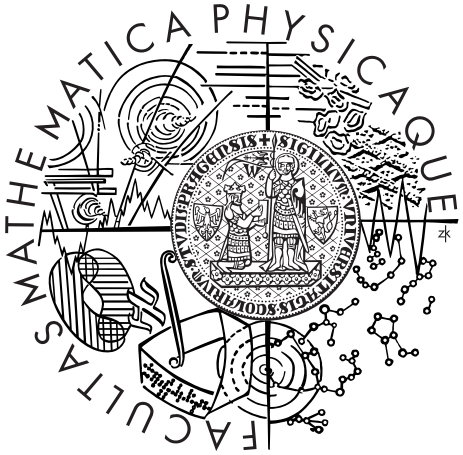
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Vladimír Kudelas

Adapting Service Interfaces when Business Processes Evolve

Department of Software Engineering

Supervisor of the master thesis: Mgr. Martin Nečaský, Ph.D.

Study programme: Informatics

Specialization: Software systems

Prague 2012

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on July 31, 2012 Vladimír Kudelas

Název práce: *Adaptácia Rozhraní Služieb pri Evolúcii Business Procesov*
Autor: *Vladimír Kudelas*
Katedra: *Katedra Softwarového Inženýrství*
Vedoucí diplomové práce: *Mgr. Martin Nečaský, Ph.D.*
E-mail vedoucího: *necasky@ksi.mff.cuni.cz*

Abstrakt: *V prezentovanej práci študujeme odvodenie optimálnych komunikačných XML schém pre daný konceptuálny model business procesu, doplnený o konceptuálny model vymieňaných dát a vplyv zmeny užívateľských požiadaviek na odvodenú XML schému. Práca obsahuje prehľad dôležitých problémov spojených s touto témou. Taktiež popisuje už existujúce prístupy k týmto problémom. Práca detailnejšie rozoberá konkrétne odvodenie optimálnej XML schémy. Prístup predstavený v tejto práci ukazuje možnosť modelovania vymieňaných dát spolu s konceptuálnym modelom business procesu. Práca ďalej popisuje, ako automaticky odvodiť konkrétnu XML schému z daného konceptuálneho modelu vymieňaných dát. V neposlednom rade, práca obsahuje prototypovú implementáciu uvedeného riešenia.*

Klíčová slova: *BPMN, dátový artefakt, konceptuálny model, PIM, business pravidlo*

Title: *Adapting Service Interfaces when Business Processes Evolve*
Author: *Vladimír Kudelas*
Department: *Department of Software Engineering*
Supervisor: *Mgr. Martin Nečaský, Ph.D.*
Supervisor's e-mail address: *necasky@ksi.mff.cuni.cz*

Abstract: *In the presented work, we study a derivation of an optimal communication XML schemas for a given conceptual model of a business process, complemented with a conceptual model of exchanged data and influence of a change in user requirements on derived XML schema. The work contains a review of important problems related to the topic. It also describes existing approaches of these problems. The work analyses a derivation of an optimal XML schema in detail. The approach presented in this thesis shows an ability to model exchanged data with a conceptual model of a business process. The thesis describes a solution, how to derive a concrete XML schema automaticly from a given conceptual model of exchanged data. Finally, the work contains a prototype implementation of the presented solution.*

Keywords: *BPMN, data artefact, conceptual model, PIM, business rule*

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays, there are very popular different notations of business process modelling, e.g. BPMN [13], for a software analysis. The modelling of business processes at the conceptual level allows domain experts to cooperate in the analysis and to design the software. There is a common language for domain experts, software architects and analysts.

Business processes are modelled like units of business processes and communications between them. Units are usually modelled as simple tasks or subprocesses. They are composed of other units and communications. Communications are usually modeled as sequence flows. One sequence flow represents one communication (data object) from one task to another. This data object represents data transferred between two tasks. Business process modelling notations are usually extended with some conceptual data modelling language, e.g. Unified Markup Language (UML) [20]. It is used to support the modelling of these data objects. It is also common that different business rules are connected to tasks and sequence flows. Mostly, there are conditions and restrictions on exchanged data. More exactly they are conditions and restrictions on data objects.

A whole business process model can be later translated to web services and to executable BPEL [14] scripts, which orchestrates all parts together. Besides this automatization, it is necessary to define structure of each data object in the business model. Web services usually communicate by exchanging XML documents [15]. Therefore, a software architect has to define an XML schema of XML documents by some XML schema language, e.g. XML Schema [16]. There are common requirements on structure of the XML schema used in web services. Structure should be *readable* and complex from the view of business rules. It should not contain redundant data.

Since most of current applications are dynamic, sooner or later business processes and created web services need to be changed. Related issues like XML schemas of data objects also have to be changed. We speak about the evolution and adaptability of applications.

The adaptation of business processes covers many related issues:

- The evolution of a software
  Business processes evolve very often. It is necessary to adapt the whole

software to this evolution very quickly. We can understand the evolution as a change of conceptual data models of some data objects or a change of constraints of data objects.

- Integration with other softwares
  As a company is getting bigger or the scope of the business is adapting to new requirements, it is necessary to integrate different software tools and functionalities. It is very common that a new task or a sub-process has to be integrated into an existing business process model. It includes adding or removing sequence flows, tasks and business rules.

All of these changes can cause several problems:

- Change of a conceptual data model of a data object
  It can cause a malfunction of the whole system. It is necessary to change an XML schema of the data object.

- Change of business rules of a data object
  It can cause an inadequate behaviour and response of the system. A previous XML schema of the data object was designed for previous business rules.

- Adding or removing sequence flows and tasks
  It can cause a malfunction of the whole system. It is necessary to change an XML schema of the data object reconnected to new or different tasks. The XML schema was defined for a sequence flow, which leads to a different task before change. Therefore, it can represent an XML schema of totally different data.

All these presented possibilities lead to an incorrect or inadequate functionality of the system. Therefore, XML schemas of data objects have to be hand-designed by a software architect. Assume that for one sequence flow *Order*, which transfers data about user's order, a data object was designed. This data object uses a conceptual data model of the whole system. The business of a company was extended and it is necessary to have more information about the user. Therefore, the conceptual model of the system was extended with two more information about the user, e.g. *phone* and *email*. This information has to be transferred everywhere with the user's information. That is why the sequence flow *Order* and all others, which contain user's information, have to be updated.
The first problem is to identify all XML schemas working with the user's information. The second problem is to define a change for each found XML schema from the first phase.

1.2 Aim of the Thesis

The aim of this thesis is a research on possibilities and limitations of deriving and adapting optimal communication XML schemas for a given conceptual schema of a business process, complemented with a conceptual schema of exchanged data. The thesis analyses related issues in general and discusses their key problems and solutions. The core of the work is a proposal and implementation of own

approach dealing with selected disadvantages and open issues. The purpose of the thesis is:

- to propose an algorithm to derive an optimal communication XML schema for a given conceptual schema of a business process, complemented with a conceptual schema of exchanged data
- to analyse an influence of user's changes on the derived optimal communication XML schema
- to propose algorithms for updating of the derived XML schema with the user's cooperation according to user's changes

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 briefly describes the Business Process Model and Notation [13], selected parts of the Unified Markup Language [20] and selected parts of the Object Constraint Language [22] related to the aim of this thesis. Chapter 3 presents a short introduction to the Model Driven Architecture.

In Chapter 4, there are discussed related works dealing with a generation of an optimal XML schema and related topics to this thesis. Chapter 5 describes the architecture of the work presented in this thesis. Chapter 6 describes theoretical models used in this work.

Chapter 7 contains a main part of this thesis. It proposes an algorithm for derivation of an optimal communication XML schema. In Chapter 8, there is analysed an evolution of the data object. It contains proposed solutions. Chapter 9 briefly presents an experimental implementation of the presented solution and describes experiments with real-word examples. Finally, Chapter 10 concludes and provides future research directions of this approach.

Chapter 2

Business Process Model and Notation

This chapter describes the most popular business process modelling notation - Business Process Model and Notation, [13]. Then, it describes selected parts of the UML language and the Object Constraint Language (OCL) [22] related to the aim of this thesis.

2.1 Business Process Model and Notation

The *Business Process Model and Notation* (BPMN) is a public standard maintained by Object Management Group (OMG). It describes a graphical notation that business process analysts and users can use to model and to define business processes. It is also used as a support for process interactions and a documentation of a system.

A business process diagram is a simple diagram consisted of five primary elements:

- Flow Objects
  They represent core elements of the diagram. The BPMN specification defines three flow object groups: *activities*, *events* and *gateways*.

- Connection Objects
  They are used to connect flow objects. The BPMN specification defines two connection objects: *sequence flow* and *message flow*.

- Swimlanes
  These are elements used to organize processes and responsibilities in the diagram. The BPMN specification defines two swimlanes: *lane* and *pool*.

- Artifacts
  These are mechanisms used to extend a basic BPMN. The BPMN version 2.0 [13] defines three artifacts: *Associations*, *Groups* and *Text Annotations*.

- Data
  In the previous version 1.2, *Data Object* was also a part of artifacts. In this version 2.0, it is defined as a part of Data primary elements. Except

Data Object, it also contains *Data Input*, *Data Output*, *Data Store*, etc. For simplicity, we write about *Data Object* as about an artifact.

2.1.1 Flow Objects

Activities

An activity is any work that is done in a process. This work can be done by a computer or by a human interaction. The specification defines three activities: *Task*, *Sub-Process* and *Call Activity*. For the purpose of this thesis, it is important only how these activities can be connected with connection objects.

All types of activities can be a source or a target of connection objects. If there are more outgoing connection objects, then it means that a parallel path is created for each connection object.

Events

Event is something that occurs during the running of a process. These events affect a flow of a whole process. Events can be divided into three types: *start events*, *end events* and *intermediate events*. Start events indicate the beginning of the process and end events indicate the end of the process. Intermediate events indicate something between the start and the end of this process. Beside this categorization, some events can be categorized as separate elements or they can be on the boundary of the activities.

Events can have different types, e.g. message, timer, conditional, etc. All of them have some special functionality used in the designing of the whole process. According to their type, they can *catch* the flow or they can *throw* the flow. To *catch* the flow means, that it requires some input. It can stop the flow for some time or until something is happened, e.g. arrival of a message. To *throw* the flow means, that it produces some output. It can stop the flow for some time or until something is happened, e.g. send a message or a signal.

Events placed on boundaries of activities can be only *catch* events. All of the events only *take a look* at the flow of the process. They can change a direction of the process. However, they do not change data, which are a part of the flow.

Start and end events are not important for the purpose of this thesis. A connection of connection objects and intermediate events is important. Event must be a target and a source of sequence flows. Events can have multiple incoming and outgoing sequence flows. There is only one exception from this connection rule. Event of type *Link Intermediate Event* must not be both a target and a source of a sequence flow.

Message flow can be used only with the event of the type *Message Intermediate Event*. The *Message Intermediate Event* may have incoming or outgoing message flows, but not both.

To simplify the rest of this thesis, we do not work with *Link Intermediate Event* and with events placed on boundaries of activities.

Gateways

Gateways are used to control the flow of an execution semantics of a given business process. They are capable to consum and to generate and additional control

tokens. Single gateway can have multiple input and output connection objects. Gateways do not represent a work done in a process.

The specification defines five types of gateways: *Exclusive*, *Event-Based*, *Inclusive*, *Parallel* adn *Complex*. Event-Based gateway controls the flow of an execution. This control is based on events that occur at that point of the process. A parallel gateway is used to create parallel paths performed at the same time. It can be used to synchronize these paths too. All other types of gateways use some conditions to control and navigate the flow.

2.1.2 Connection Objects

A sequence flow is used to show the flow of an execution of a business process. It has only one source and only one target. It can contain a condition. The flow can use this sequence flow, only if it satisfies this condition.

A message flow is used to show the flow of messages between two flow objects. It must connect separate pools. It can be connected to the pool's boundary or to flow objects in the pool.

To simplify the solution of this thesis, we do not work with message flows connected to the boundary of the pool.

2.1.3 Swimlanes

A pool is a graphical representation of a Participant (Actor) in a process. It is used to separate participant's activities. A lane is a sub-partition within a process, sometimes within a pool. It is used to organise activities.

2.1.4 Artifacts and Data

The Group is a graphical element, which separates some acitivities in a diagram visually. It does not have any influence on the flow of an execution.

The Text Annotation is only a mechanism for designers to write additional information or notes.

The Annotation is used to connect artifacts with other BPMN graphical objects.

This Data primary elements do not solve the topic of this thesis. It is more complicated than it was in the previous version 1.2 of the specification. These elements are designed to describe data structure. They are used in other elemenents to describe data inputs and outputs. They are not general enough for the purpose of this thesis. Therefore, we have created a new artifact *Data Artefact* (see 9.2.1).

2.2 Unified Markup Language

The *Unified Markup Language* (UML) specification contains a lot of different types of diagrams. A class diagram is the most important for the purpose of this thesis. An explanation of the class diagram is out of the scope of this thesis. We mention only basics. The full explanation can be found in [20].

We use this class diagram to create a conceptual model of data objects transferred between activities in a business process model.

2.2.1 Class Diagram

A class diagram describes static structure of the system. It divides the system into classes with different connections and relationships. This type of a diagram can be used on different levels of an abstraction. For our purpose, it is needed a conceptual class diagram, that represents concepts of the problem domain.

A Class is a specification of one object structure and its behaviour. It contains attributes. They represent object's structure and operations, which show the behaviour. The class can be in relations to other classes. We mention only relation types, which are important for this thesis:

- Association
  It represents a simple association between two classes. It has some properties, which you can find in a detail explanation in [20]. A general class diagram supports n-ary associations. In this thesis, we work only with binary relations.

- Aggregation
  It is a special type of the association. It is more specific and it defines a part-whole or part-of relationship.

- Composition
  It is a stronger variant of the aggregation and also a special type of the association. It has a strong lifecycle dependency between instances of a container class and a contained class. If the container class is destroyed, then all contained classes are destroyed too.

- Generalization
  It is a special kind of relation, which relates a general and a specific class. Each instance of the specific class is also an instance of the general class and it contains all features (attributes, operations, relations) of its general class.

An Association class is a special type of the class. It is connected to an association (aggregation or composition). It defines an additional information of the connected association. In a general class diagram, it has all features of a normal class. But for simplicity of this thesis, we do not allow any relations with association classes. In all other ways, it acts as a normal class.

2.2.2 PIM and PIM-View Class Diagrams

We use two types of diagrams that are founded on principles of the class diagram. One of them, Platform Independent Model (PIM), uses almost all capabilities of the class diagram. All functionalities are described in the documentation of a project DaemonX [23].

The second one, Platform Independent Model View (PIM-View), is based on PIM. It uses only a part of PIM capabilities:

- Class element
- Attributes of the class element
- Association element
- Generalization element
- Aggregation element
- Composition element
- Association class element
- Attributes of an association class element

The Class element and the Association class element are extended with one property *Count*. It defines a maximal count of instances of the class, where $Count \in (\mathbb{N} \cup \{*\})$.

2.3 Object Constraint Language

The *Object Constraint Language* (OCL) is a formal language used to describe expressions on UML models, especially on class diagrams. These expressions usually define invariants and conditions over objects described in the model. They do not have any side effects. It is a very strong tool to define invariants of the system in a modelling phase or to create queries over the model, or to define operations and actions. The whole explanation of OCL can be found in [22]. In the next paragraphs, there are briefly explained only basics needed to show the usage of OCL in this thesis.

We use OCL to create business rules. These rules are connected to data objects or different parts of the business process model.

Usually, each OCL expression has a context. It can be defined directly in expressions. If expression has a graphical representation, it can be also connected to its context. The context is an instance of a class from the class diagram. This context is a beginning of each path in the expression.

Expressions consist of paths and functions from a standard OCL library. Paths are in the meaning of paths through the class diagram.

The OCL expression can work with operations. It can create pre-conditions and post-conditions. This will not be needed, because our diagrams do not contain operations. The next difference from the classic OCL is in a navigation. OCL uses association end descriptions for the navigation. In our diagrams, we do not have any association end descriptions. Therefore, we use class names as association end descriptions. In these diagrams, it is not allowed to have duplicate names of classes. Therefore, there are no possibilities to construct an expression with an ambiguous path.

Chapter 3

Model Driven Architecture

In this chapter, we briefly explain *model driven architecture*. It is an approach to a system development.

3.1 Model Driven Architecture

The *Model driven architecture* (MDA) [21], is a software design approach to a system development. One of the main aims is to separate a design from an architecture. It shifts a development process from a code-centric to model-centric approach. MDA separates the business logic from platform specifics.

Instead of writing lines of a code, architects model systems according to business processes. They generate a *Platform Independent Model* (PIM). Then, using tools, they can generate a *Platform Specific Model* (PSM), that is targeted to a specific platform, e.g. C++, .NET, etc.

MDA uses more levels of an abstraction. It uses three levels of models and viewpoints. A viewpoint on a system is an abstraction. It uses a selected set of architectural concepts and structuring rules, in order to focus on specific concerns within that system. The abstraction is used as an action of suppressing specific details to establish a simpler model.

Using models and some additional information, like mapping and marking, it is possible to make a transformation from PIM to PSM. Nowadays, this architecture is used in several places. You can see it in automatic generation of a platform specific code from UML models.

3.1.1 Computation Independent Model

A *computation independent model* (CIM) focuses on the environment of a system and on requirements. It is a model of business rules of the system. It has no detail specification of computer implementation. It is transformed to PIM, but not to PSM.

3.1.2 Platform Independent Model

A *platform independent model* (PIM) focuses on an operation of the system while hiding details necessary for a particular platform. It is a maximal part of the system specification that does not change from one platform to another.

Models at this level of an abstraction should be closer to the client's point of view. Clients and other domain experts should be able to recognize objects in the model.

Nowadays, the most common general modelling language for PIM is UML. But it is also possible to use another language, which is more specific for the area, where the system will be used.

3.1.3 Platform Specific Model

A *platform specific model* (PSM) focuses on details of the use of a specific platform by the system. It takes PIM and add all platform specific details.

3.1.4 Model Transformation

A *model transformation* is the most important part of this architecture. With an additional information, like mappings from PIM to PSM for specific platforms, it is possible to generate PSM models or concrete PSM instances of PSM models. This is the main advantage of MDA. It makes models portable and reusable.

Chapter 4

Related Works

In this chapter, papers related to a generation of an XML schema from other model and commercial tools, are discussed.

The chapter contains three parts. The first one is focused on the generation of the XML schema from another model. The second one is focused on commercial tools for UML and BPMN modelling. The last part of this chapter gives a summarizing comparison of all discussed works and briefly introduces the main issue of this work.

4.1 UML and XML Schema

Paper [1] describes an approach for mapping between the UML class diagram and the XML Schema using a three level design approach. The main aims of the paper are:

- To define the three level design approach.
- To introduce a logical level of the design.
- To propose a general algorithm for a transformation from a conceptual level to a logical level.

4.1.1 Three Level Design Approach

The paper introduces the three level design approach. It is depicted in Figure 4.1.

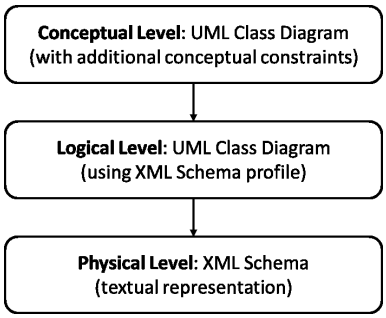


Figure 4.1: Three level design approach

The conceptual level is a UML class diagram with some non-standard annotations, such as a primary identification of a class. The logical level uses a defined UML profile for an XML Schema. It uses mainly stereotypes of classes to define XML Schema concepts in the UML profile. A detail description of this UML profile can be found in section 3 of paper [1]. The physical level defines data structures by using the implementation language - in this case the XML Schema. It is a concrete XML Schema generated from the logical level model. There is a direct bidirectional mapping from the logical to the physical level and vice versa.

4.1.2 Conceptual to Logical Level Mapping

An XML schema is hierarchical structure from the nature. The generating of a logical level model from a conceptual one requires to choose one or more conceptual classes to begin the tree-hierarchy. An approach in this paper aims to minimize redundancy and to maximize a connectivity of XML data structures. The approach used in this paper is directly based on [2]. There are differences in the algorithm because of differences between an *Object Role Modeling* (ORM) [3] and UML. One of the main differences is that the concept of *Anchors* used in [2] to identify the direction of the nesting, was not used in this paper. Instead of this, they use the navigation and cardinalities on a relation.

The mapping consists of four steps:

- Generate Type Definitions
  It creates type definitions for each attribute and class in the conceptual diagram. It means, the creation of complex types and primitive types with restrictions.

- Determine Class Groupings
  It determines, how to group and to nest conceptual classes based on relations between them. It uses simple rules based on the navigation of relations and their cardinalities. These rules try to reduce redundancy of the created XML schema and to maximize the connectivity.

  Example of the rule:
  If exactly one association end has a cardinality of $\langle 1, 1 \rangle$, then we nest the class at the another relation end towards it. Both nesting possibilities are depicted in Figure 4.2.

  If we do not use the rule above, we create a redundancy mapping. It is depicted in the part b) in Figure 4.2. Because *Employee* can be a head lecturer of more *Subject*s than one. Therefore, the information about *Employee* will be repeated for all *Subject*s, where it is a head lecturer.

- Build the Complex Type Nestings
  After identifying of nesting directions from the previous step, it is necessary to perform the complex type nesting. This is done in this step. It creates hierarchical structure.

- Create a Root Element
  At the end, it is necessary to define the root element of the created XML schema.

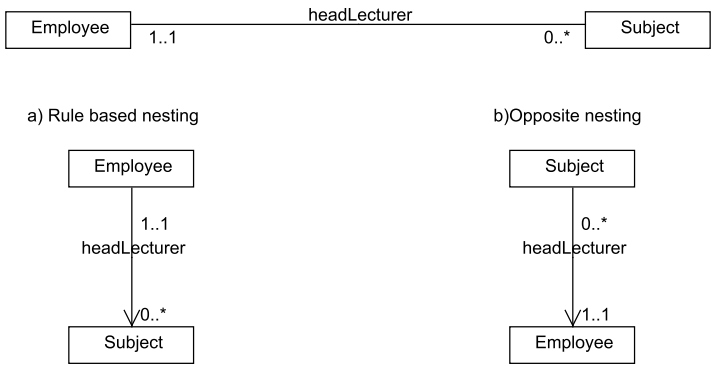


Figure 4.2: Rule example

4.1.3 Discussion

The paper presents an interesting approach to the automatic generation of the XML schema from the UML model. It uses the three level design, which is similar to MDA. From the user's point of view, this method is useful in reducing redundancy and in maximizing the connectivity in the created XML schema.

Beside advantages of this paper, it has some limitations also mentioned in the paper. It needs some additional constructs in UML. OCL cannot express all constraints, which can be expressed in XML schema regular expressions. In the solution presented in this paper, it is also not possible to define a mixed content in the XML schema.

As a big advantage of the paper can be found an introducing the generation method, which minimizes redundancy and maximizes connectivity of the created XML schema.

4.2 XSEM - A Conceptual Model for XML

Paper [4] describes an approach for a conceptual modelling of XML data. It presents a two layer design. The first layer, XSEM-ER represents an overall conceptual layer. It uses an extension of the Entity-relationship (ER) model [6]. The second layer, XSEM-H represents hierarchical organization of structures from the first layer. It introduces the transformation from XSEM-ER to XSEM-H by *Operators*. It is possible to create more than one XSEM-H model from one XSEM-ER model. It is also possible to use only a part of XSEM-ER in the transformed XSEM-H model.

During the transformation of XSEM-ER to XSEM-H, an interconnection between two models is created. Thanks to this, it is possible to support evolution operations, which were described in [5]. The evolution is based on a propagation of changes from the XSEM-ER model to the connected XSEM-H model. Changes on XSEM-ER are made by atomic operations, which have their counterparts on XSEM-H.

A visualization of the XSEM-H model is the most important part of this paper for the purpose of this thesis. This model and the visualization with a

small extension are used for the modelling of XML schema results. Therefore, this part of the paper is described bellow.

4.2.1 XSEM-H

It is an extension of the UML class diagram. It takes part of class diagram components. It adds some new components and a profile named XSEM. There is a bidirectional mapping between XSEM-H and the XML schema. This functionality of the mapping was implemented in the XCase tool [12].

The visualization of XSEM-H was implemented in [23] as a modelling plugin. This software was used for an implementation part of this thesis.

Components of XSEM-H:

- Class
  It represents a class from the XSEM-ER model. Each class has a name, a label and it contains attributes. It can be connected with other classes by associations.

- Attribute
  It represents an attribute from the XSEM-ER model. It can exists only as a part of the class. It can represent zero or one attribute from the XSEM-ER model. Each attribute has a name, a data type, a cardinality and a position.

- Association
  It represents a connection between two XSEM-H classes in the relation parent-child. It can represent a whole set of XSEM-ER associations.

- Content Choice
  It represents *choice element* of the XML schema.

- Content Sequence
  It represents *sequence element* of the XML schema.

- Content Set
  It represents *all element* of the XML schema.

4.2.2 Discussion

The paper presents a detailed approach for conceptual modelling. Authors continue with this issue. They present the approach for the evolution between two conceptual layers.

There is introduced a complex conceptual model for XML data. In related paper [5], the evolution in this conceptual model is described .

The paper is presented as a transformation from an extended ER model. It can be also used as a transformation from the restricted UML class diagram, as it was presented in [12] and [23].

4.3 An Extension of Business Process Model for XML Schema Modeling

Paper [7] describes an approach for deriving optimal communication XML formats for a given conceptual schema of a business process with a conceptual schema of exchanged data. Authors use the approach of MDA. They introduce PIM and PSM models for a conceptual model of exchanged data. They extend it with OCL constraints over the PIM model.

4.3.1 Metrics

An optimal XML format is defined by two metrics computed on the PSM model. The paper shows a detail description of two metrics and the explanation of their using. To choose the optimal PSM model, it is necessary to create all possible PSM models from the given PIM model. This approach has an exponential complexity in all cases.

This paper is important, because of presented metrics:

- Redundancy Metric
  It defines functions, which compute a positive number for each class of the PSM model. The value of the metric is a sum of computed class values.

- Business Rules Metric
  For each OCL expression defined over a PIM model, it creates paths of classes from the PIM model. For each path, it finds a subpath in a PSM model. Classes from the PSM model are connected to classes from the PIM model. This connection was created during the derivation of all possible PSM models from the PIM model. For each path, there is computed a positive number based on subpaths in the PSM model. The value of the metric is a sum of computed path values.

For each PSM model, there are computed these two metrics. According to user's preferences, which metric is more important, the final value for one PSM model is computed. The optimal PSM model has the lowest value.

4.3.2 Discussion

The paper presents the novel approach to the generation of the conceptual model of exchanged data in the business process modelling. It defines two metrics used to choose the optimal XML schema for exchanged data. These metrics try to choose a conceptual model with minimal redundancy and a maximal connectivity.

It uses a restricted UML class diagram as a PIM model. Because of that, it is not usable in larger systems. In larger system, there are required more complicated concepts of the UML class diagram to create a conceptual model of exchanged data.

4.4 Related Commercial Software Solutions

There is a lot of commercial tools for UML and BPMN modelling. We choose two of them and we discuss their functionalities in a connection to the topic of this thesis.

4.4.1 Altova UModel

The commercial tool Altova UModel [11] supports all fourteen UML 2.3 diagram types. It supports also the XML Schema modelling by a special UML profile. There are two options how to create the XML Schema model. The first one is to create a diagram with a UML profile for XML Schema manually. The second one is to use a model transformation, which is one of functionalities of this software.

It uses an approach of MDA to support model transformations. A definition of the transformation requires the mapping between types of models. After definition, a new model transformation is created. But you cannot update the result easily. The transformation is stored as a separate package.

UModel also supports BPMN. It is posible to create a business process model and to define data objects. However, you cannot connect data objects with other diagrams defining the conceptual model of data.

This tool supports different diagram types and levels of modelling. It is a useful tool to create models of the system. Though, it does not support connections between different models (diagrams).

4.4.2 Enterprise Architect

The commercial tool Enterprise Architect [10] supports all fourteen UML 2.4.1 diagram types. It also supports the modelling of an XML schema by a special UML profile. Again, you can manually create the XML Schema model or you can use a generation. In compare to the first tool, it is possible to generate the XML Schema model directly from a UML class diagram and there is no transformation stored. It stores only a new model. The new model does not have hierarchical structure. It is a normal UML class diagram with a special UML profile. User has to create the tree design of the model manually.

It is also possible to generate an XSD file from the XML Schema model directly and vice versa. The generation process of the XSD file (from the XML Schema model) and the XML Schema (from the UML class diagram) does not care about redundancy or the connectivity. It creates a valid XSD file, but it is up to the user to define or to repair the XML Schema model.

The tool also supports BPMN and BPEL. It is possible to create a business process model and define a data object. But you cannot connect data objects with other diagrams directly. There is a possibility to connect this data object with one type from other models.

This tool supports a lot of diagram types and levels of modelling. It is useful tool to create models and a specification for large systems.

4.5 Comparison of the Related Works

All presented works study problems related to a conceptual modelling of an XML schema. Each of them focuses on the problem from a different point of view.

Paper [1] describes an approach to the conceptual modelling and to the generation of a logical level by using UML and the UML profile. It uses a set of rules to define nesting and hierarchical structure.

Paper [4] describes an approach to the conceptual modelling and to the generation of the PSM model. It defines an extended ER model for conceptual modelling. It uses own PSM model XSEM-H. It introduces a transformation, which allows more PSM models for one PIM model. Authors continue with the work and introduce methods for a propagation of evolution operations.

Paper [7] describes an approach to the conceptual modelling and to the generation of the PSM model by defining own PIM and PSM model. It uses metrics to choose an optimal PSM model from the set of all possible PSM models.

Commercial tools [11] and [10] support general functionalities of UML diagram types and BPMN. These tools are useful for modelling. They are determined for a general purpose. They are not focused on redundancy or the connectivity of the generated XML schema and they are not focused on the evolution in business process models.

In this thesis, we focus on the problem of the derivation an optimal XML schema for business process models and on the influence of changes in the business process model on generated XML schemas. We focus especially on following problems:

- to identify metrics for choosing the optimal XML schema for the business process.
- to propose an algorithm for creating PSM models from the given PIM model.
- to analyse changes in the business process model and their influence on the generated XML schema.
- to propose algorithms to update the derived XML format, with user's cooperation, according to changes made in the business process model.

Chapter 5

Architecture

In this chapter, we introduce the architecture of the work presented in this thesis and extensions made in the software DaemonX [23]. The work uses an approach of a Model Driven Architecture. Especially, it uses an idea of a Platform Independent Model and a Platform Specific Model. The software tool DaemonX is used for the purpose of this thesis. Therefore, it also uses the architecture presented in this software.

5.1 Architecture

The purpose of this thesis is to propose a method to derive an optimal communication XML format for a given conceptual schema of a business process, complemented with a conceptual schema of exchanged data. Therefore, it is necessary to use BPMN model and some conceptual model for exchanged data. The idea behind this thesis is, that there are conceptual models of the whole problem domain of the system. Exchanged data are always associated only with a small part of this problem domain. We introduce two conceptual models for exchanged data. The first one models the whole problem domain and the second one serves as a view on a part of the first model. This approach of the one conceptual model for the problem domain and other related models was also mentioned in [8].

We use *PIM* as the first model for the whole domain. It was implemented in DaemonX. It is a restricted UML Class Diagram. We use a new model *PIM-View*, as the second model, for the view on the part of the *PIM*. This new conceptual model is more explained in section 5.2.

The resulting optimal XML schema (format) is stored as a conceptual model, which was briefly described in subsection 4.2.1. This model was also implemented in DaemonX and its name is *PSM XML*. For the purpose of this thesis, it was necessary to extend this model (see 9.2.1).

For the BPMN model, we choose a model, which was implemented in DaemonX. Its name is a *PIM process* model. We extended this model with one new artifact named *Data Artefact* (see 9.2.1).

As it was mentioned in section 2.3, we use OCL to express business rules

over a conceptual model of exchanged data. We use the master thesis [9], which implements *Universal Constraint Language* (UCL) and OCL in the DaemonX tool. We created a simple UCL model for the *PIM-View* model based on the implementation of OCL over the UML class diagram in [9].

Figure 5.1 depicts the architecture with connections between models. The figure shows the Schema and Physical level, which are not implemented in this thesis. It is depicted to show the idea behind the introduced solution.

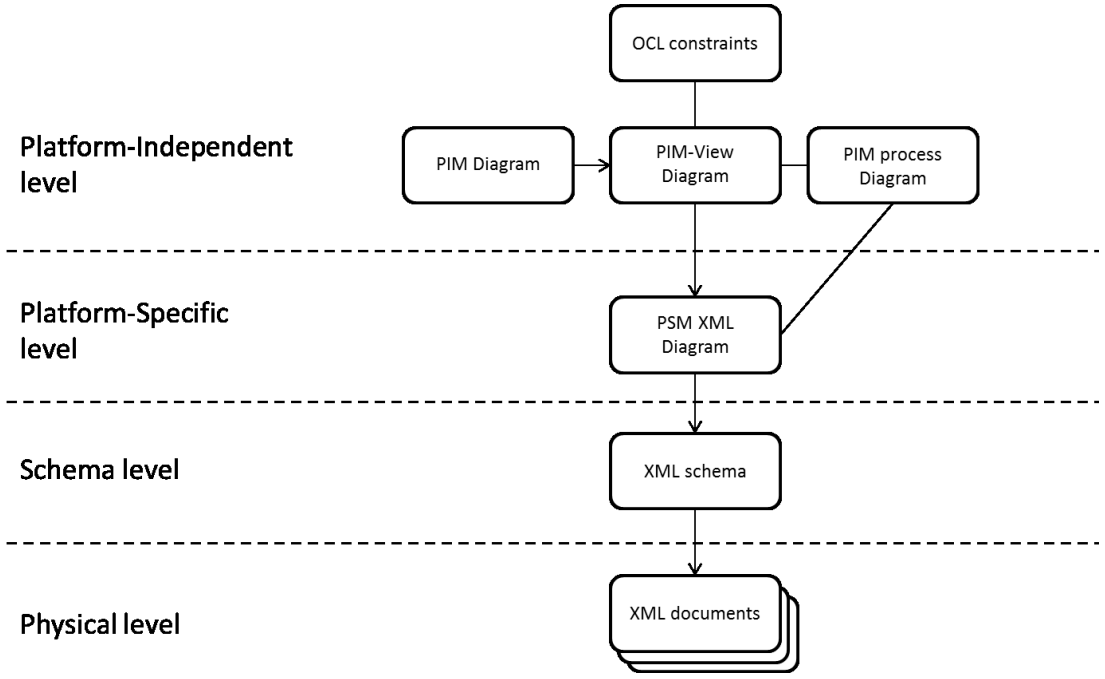


Figure 5.1: Application architecture

It is possible to generate a concrete XML Schema or Schematron [17] from the created conceptual model of the XML schema. According to these XML schemas we can have XML documents.

5.1.1 Architecture Example

In this subsection, we present a complex example of all used models and connections between them. It is depicted in Figure 5.2.

The *PIM* diagram contains the conceptual model of the whole problem domain. The *PIM-View* diagram is based on the *PIM* diagram. Connections between elements of diagrams are created by *evolution references* (see [23]). These connections are depicted by blue lines. Both of diagrams have to be created by a domain expert (user).

OCL constraints have to be created over the *PIM-View* diagram. They have to be connected to elements from the *PIM process* (BPMN) diagram. This is done by choosing correct values in drop-down lists. Connections between OCL constraints, the source *PIM-View* diagram and *PIM process* (BPMN) elements are depicted by yellow lines.

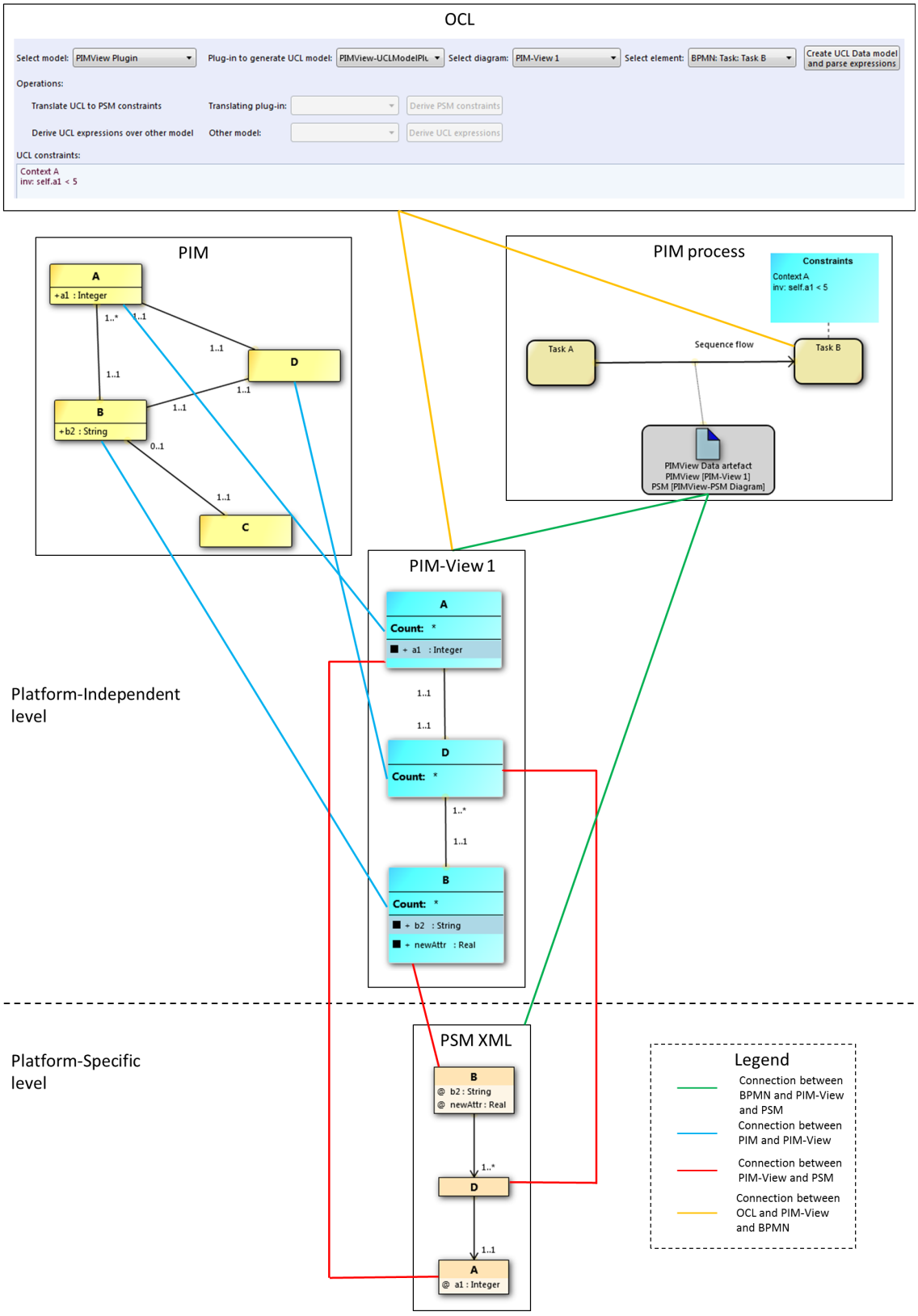


Figure 5.2: Architecture example

The *PIM process* (BPMN) diagram has to be created by the domain expert too. The *Data Artefact* has to be connected to the created *PIM-View* diagram. This connection is depicted in figure by a green line.

One of purposes of this thesis is to propose an algorithm to derive an optimal XML schema for exchanged data in the business process model. Therefore, a Platform Specific Model represented by a *PSM XML* diagram is created by the proposed algorithm. Connections between elements of the *PIM-View* diagram and the *PSM XML* diagram are created by *evolution references*. They are depicted by red lines. After the creation of the *PSM XML* diagram, the connection between *Data Artefact* and the created diagram is made. This connection is depicted in figure by a green line.

5.2 PIM-View Model

A *PIM-View* model is a Platform Independent Model that shows only a part of *PIM* models. The main sense of this model is described in the section above. It is a restricted UML Class Diagram. One diagram of the *PIM-View* model can reference elements from different diagrams of the *PIM* model.

There is one feature, which is applied to one diagram of the *PIM-View* model. The diagram cannot contain two classes with the same name. This restriction is added to simplify the creation of OCL expressions.

5.2.1 Elements

Class

It represents a class from a *PIM* model. The name of the class is taken from the referenced class in the *PIM* model. It can be connected with relations and it can contain attributes.

Association class

It represents an association class from a *PIM* model. The name of the class is taken from the referenced class in the *PIM* model. It can be connected only with one association (aggregation or composition). This connection is visualized as a dashed line. It can contain attributes. This association class is restricted in compare to the general UML Class Diagram. It is restricted in the similar way as in the *PIM* model. The association class in the general class diagram can be connected with other relations. It means, that it can be in a relation with other classes. This restriction is made to simplify the problem studied in this thesis.

Attribute

It represents one property of the class. There are two types of attributes in this model:

- Reference attributes
  They represent a *PIM* attribute from the *PIM* class referenced by the *PIM-View* class. This *PIM-View* class contains reference attributes. The name and the type of the attribute is taken from the referenced attribute in the *PIM* model.

- Normal attributes
  They are normal attributes like in the *PIM* model. Sometimes, there is a computed or an aggregated value in exchanged data. Therefore, there are these normal attributes.

Both of attributes have one additional feature. They can be marked as optional. Attributes in the *PIM* model do not have a cardinality, so it is not possible to define their occurence.

Relations

A *PIM-View* model supports four kind of relations like a *PIM* model. It is *association*, *aggregation*, *composition* and *generalization*. All of them only reference equivalent relations in the *PIM* model. In the time of the creation, they take cardinalities from referenced relations. There is also a possibility to create own cardinalities. Ends of the new relation and the direction have to be the same as in the referenced relation from the *PIM* model.

Chapter 6

Models of the System

In this chapter, we describe theoretical models used in this thesis. These models are used in next chapters to present the solution of this work. Most of the definitions also have an example to explain the definition properly.

6.1 *PIM* Model

Definition 6.1. (*PIM* schema). A platform independent model schema is a 15-tuple $\mathcal{S} = (\mathcal{C}, \mathcal{C}_\mathcal{A}, \mathcal{A}, \mathcal{R}, \mathcal{R}_\mathcal{G}, \mathcal{R}_\mathcal{A}, \mathcal{T}_\mathcal{R}, name, class, ends, aends, type, rtype, orient, rcard)$.

- $\mathcal{C}$, $\mathcal{C}_\mathcal{A}$, $\mathcal{A}$ and $\mathcal{R}$ denote the set of classes, association classes, attributes and relations in $\mathcal{S}$, respectively.
- $\mathcal{R}_\mathcal{G}$ denote the set of generalizations. It is a special type of a relation.
- $\mathcal{R}_\mathcal{A}$ denote the set of association-class relations.
- $\mathcal{T}_\mathcal{R}$ denote the set of relation types $\{association, aggregation, composition\}$.
- Function $name : \mathcal{C} \cup \mathcal{C}_\mathcal{A} \cup \mathcal{A} \rightarrow \mathcal{L}$ assigns a name to each class, association class or attribute. $\mathcal{L}$ denote the set of all names.
- Function $class : \mathcal{A} \rightarrow \mathcal{C} \cup \mathcal{C}_\mathcal{A}$ assigns a class or an association class to each attribute.
- Function $ends : \mathcal{R} \cup \mathcal{R}_\mathcal{G} \rightarrow \mathcal{C} \times \mathcal{C}$ assigns a pair of classes to each relation or generalization. Then, for a given relation $R \in \mathcal{R}$, where $ends(R) = (C_1, C_2)$, we say that C_1 and C_2 participate in R. The ordering of both classes in $ends(R)$ is not important, i.e. $ends(R) = (C_1, C_2) \Leftrightarrow ends(R) = (C_2, C_1)$. Similarly for the generalization.
- Function $aends : \mathcal{R}_\mathcal{A} \rightarrow \mathcal{C}_\mathcal{A} \times \mathcal{R}$ assigns a pair of an association class and a relation to each association-class relation. Then, for a given association-class-relation $R_A \in \mathcal{R}_\mathcal{A}$, where $aends(R_A) = (C_{A_1}, R_1)$, we say that C_{A_1} and R_1 participate in R_A. The ordering of an association class and a relation in $aends(R_A)$ is not important, i.e. $aends(R_A) = (C_{A_1}, R_1) \Leftrightarrow aends(R_A) = (R_1, C_{A_1})$.

- Function $type : \mathcal{A} \to \gamma$ assigns a data type to each attribute. γ denote the set of all types.
- Function $rtype : \mathcal{R} \to \mathcal{T}_{\mathcal{R}}$ assigns a relation type $\mathcal{T}_{\mathcal{R}}$ to each relation.
- Function $orient : \mathcal{R} \to \{\emptyset\} \cup \mathcal{C}; \mathcal{R}_{\mathcal{G}} \to \mathcal{C}$, where $\mathcal{C} \in rng(ends)$. It defines an orientation of a relation or generalization. This function returns the class, which the relation or generalization is oriented to. This class has to be one of ends of the relation or generalization. The generalization is always oriented to some class. The relation does not have to be oriented to any class, in this case the function returns $\emptyset$. Then, for a given relation $R \in \mathcal{R}$, where $ends(R) = (C_1, C_2)$ and $orient(R) = C_1$, we say that R is oriented to the class C_1. Similarly for the generalization. The relation or generalization can be oriented only to one class from $rng(ends)$, i.e. $orient(R) = C_1 \wedge orient(R) = C_2$ is not possible. For each relation or generalization $orient$ returns only one class from $rng(ends)$ or for the relation it can return $\emptyset$.
- Function $rcard : \mathcal{C} \times \mathcal{R} \to \langle \mathbb{N}_0 \times (\mathbb{N} \cup \{*\}) \rangle$ assigns a cardinality to each pair of a class and relation, s.t. the class participates the relation. A cardinality is a pair $\langle min, max \rangle$, s.t. $min \geq 0 \wedge max > 0 \wedge min < max$ or $max = *$.

We also use auxiliary functions:

- Function $assoc_{rel} : \mathcal{C}_{\mathcal{A}} \to \mathcal{R}_{\mathcal{A}}$ returns an association-class relation connected to a defined association class by the function $aends$.
- Function $aends_{rel} : \mathcal{R}_{\mathcal{A}} \to \mathcal{R}$ returns a relation connected to adefined association-class relation by the function $aends$.
- Function $first_{rel} : \mathcal{R} \to \mathcal{C}$ returns the first class from a pair returned by the function $ends$ for a defined relation.
- Function $second_{rel} : \mathcal{R} \to \mathcal{C}$ returns the second class from a pair returned by the function $ends$ for a defined relation.

Example

The definition of the *PIM* schema represents one diagram of the *PIM* model mentioned in section 5.1. As it was mentioned, it is a restricted UML class diagram. Figure 6.1 depicts a visualization of a *PIM* model diagram. Because not all *PIM* schema elements have a name, the diagram is extended with auxiliary names used only for this explanation. These names are depicted by a red colour.

- The set of classes $\mathcal{C} = \{$Address, Person, Professor, Student, Faculty$\}$
- The set of association classes $\mathcal{C}_{\mathcal{A}} = \{$DurationOfAddress$\}$
- The set of attributes $\mathcal{A} = \{$city, street, psc, name, phone, salary, averageMark, from, to$\}$
- The set of relations $\mathcal{R} = \{$Rel1, Rel2$\}$
- The set of generalizations $\mathcal{R}_{\mathcal{G}} = \{$Gen1, Gen2$\}$

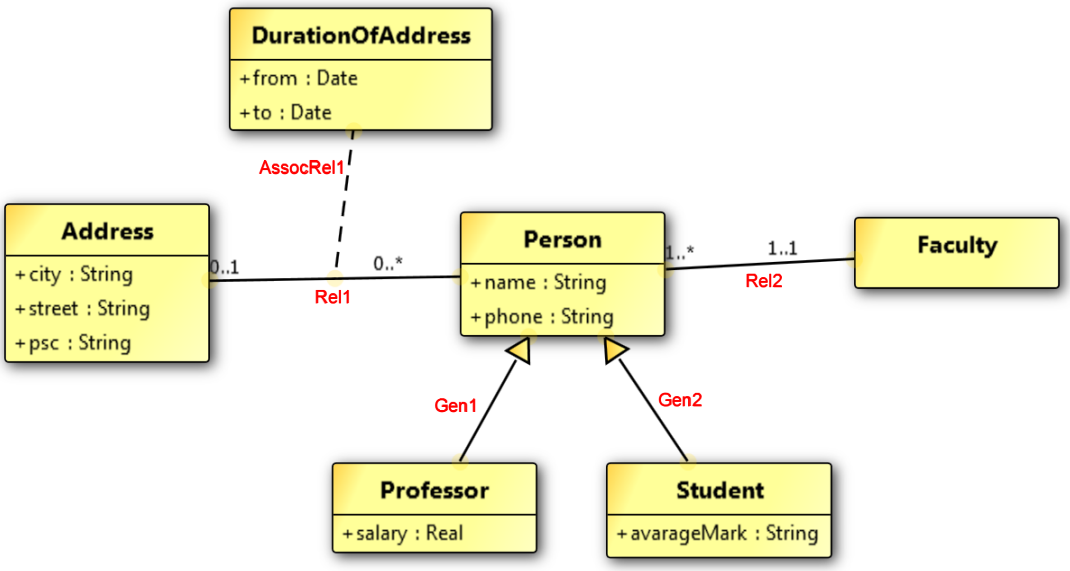


Figure 6.1: Platform independent model schema visualization example

- The set of association-class relations $\mathcal{R}_{\mathcal{A}} = \{\text{AssocRel1}\}$
- Some examples of the function *class*: $class(\text{from}) = \text{DurationOfAddress}$, $class(\text{phone}) = \text{Person}$
- Some examples of the function *ends*: $ends(\text{Rel1}) = (\text{Address, Person})$, $ends(\text{Gen1}) = (\text{Person, Professor})$
- An example of the function *aends*: $aends(\text{AssocRel1}) = (\text{DurationOfAddress, Rel1})$
- An example of the function *type*: $type(\text{from}) = \text{date}$
- An example of the function *rtype*: $rtype(\text{Rel1}) = association$
- Some examples of the function *orient*: $orient(\text{Rel1}) = \emptyset$, $orient(\text{Gen1}) = \text{Person}$
- An example of the function *rcard*: $rcard(\text{Address, Rel1}) = \langle 0, * \rangle$

6.2 *PIM-View* Model

Definition 6.2. (*PIM-View* schema). A platform independent model view schema on a *PIM* $\mathcal{S} = (\mathcal{C}, \mathcal{C}_{\mathcal{A}}, \mathcal{A}, \mathcal{R}, \mathcal{R}_{\mathcal{G}}, \mathcal{R}_{\mathcal{A}}, \mathcal{T}_{\mathcal{R}}, name, class, ends, aends, type, rtype, orient, rcard)$ is a 16-tuple $\mathcal{S}_v = (\mathcal{C}_v, \mathcal{C}_{\mathcal{A}v}, \mathcal{A}_v, \mathcal{R}_v, \mathcal{R}_{\mathcal{G}_v}, \mathcal{R}_{\mathcal{A}v}, \mathcal{T}_{\mathcal{R}}, name_v, class_v, ends_v, aends_v, type_v, rtype_v, orient_v, rcard_v, count)$ such that $\mathcal{C}_v \subseteq \mathcal{C}$, $\mathcal{C}_{\mathcal{A}v} \subseteq \mathcal{C}_{\mathcal{A}}, \mathcal{R}_v \subseteq \mathcal{R}, \mathcal{R}_{\mathcal{G}_v} \subseteq \mathcal{R}_{\mathcal{G}}, \mathcal{R}_{\mathcal{A}v} \subseteq \mathcal{R}_{\mathcal{A}}$. Functions $ends_v$, $aends_v$, $rtype_v$, $orient_v$, $rcard_v$ are restrictions of their counter parts in $\mathcal{S}$ to $\mathcal{S}_v$. The function $rcard_v$ may change cardinalities assigned by the function *rcard*.

- Set $\mathcal{A}_v$: a part of the set is a subset of $\mathcal{A}$ and a part contains new attributes.

- Function $name_v$: for classes and association classes from $\mathcal{C}$ and $\mathcal{C}_\mathcal{A}$, it is a restriction of the function counter part in $\mathcal{S}$ to $\mathcal{S}_v$. For attributes from $\mathcal{A}$, it is a restriction of the function counter part in $\mathcal{S}$ to $\mathcal{S}_v$. For new attributes, it assigns a new name.

- Function $class_v$: for attributes from $\mathcal{A}$, it is a restriction of the function counter part in $\mathcal{S}$ to $\mathcal{S}_v$. For new attributes, it assigns a class or association class to each new attribute.

- Function $type_v$: for attributes from $\mathcal{A}$, it is a restriction of the function counter part in $\mathcal{S}$ to $\mathcal{S}_v$. For new attributes, it assigns a data type to each new attribute.

- Function $count : \mathcal{C}_v \cup \mathcal{C}_{\mathcal{A}_v} \rightarrow (\mathbb{N} \cup \{*\})$ assigns a number to each class or association class in the *PIM-View* schema. This number is a maximal number of instances of $C_v \in \mathcal{C}_v$ or $C_{A_v} \in \mathcal{C}_{\mathcal{A}_v}$ in the *PIM-View* schema. It is assigned by a domain expert.

In this thesis, we also use an *extended PIM-View* schema $\mathcal{S}_{v_e}$, which contains one new function $nest_{v_e}$: $\mathcal{R}_v \rightarrow \mathcal{C}_v$. It defines a nesting class for the relation.

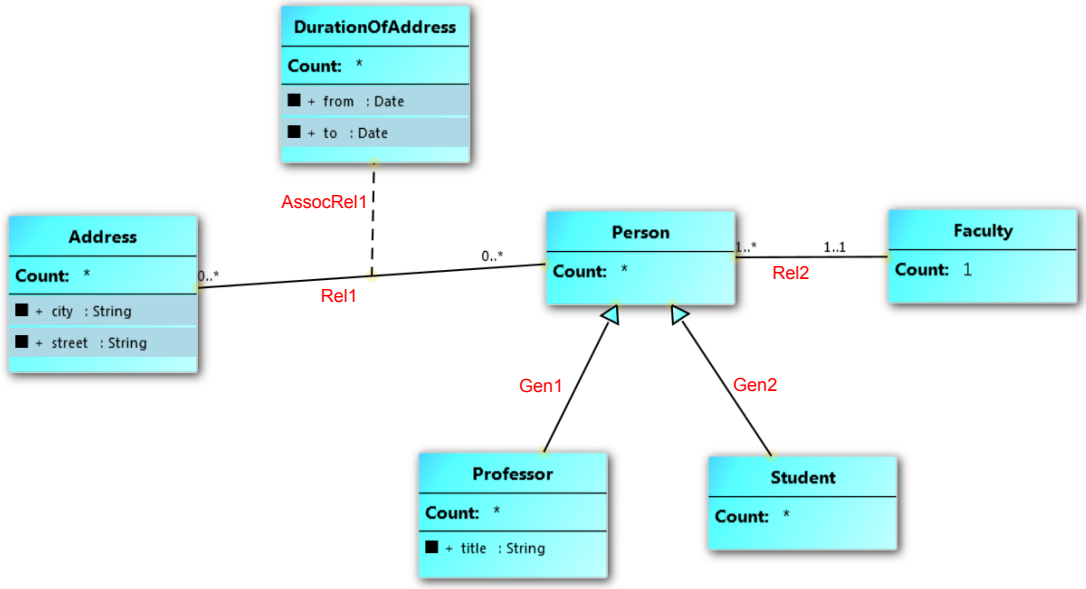


Figure 6.2: Platform independent model view schema visualization example

Example

A platform independent model schema models a whole problem domain related to a business processe(s). One data object of the business process model is modelled by the platform independent model view schema. Figure 6.2 depicts a visualization of the *PIM-View* schema as a UML class diagram. Because not all *PIM-View* schema elements have a name, the diagram is extended with auxiliary names used only for this explanation. These names are depicted by a red colour.

We describe only some parts of the diagram, which are different in compare to an example of the *PIM* schema.

- The set of attributes $\mathcal{A}_v = \{\text{city, street, from, to}\} \cup \{\text{title}\}$
- Some examples of the function $class_v$: $class_v(\text{from}) = \text{DurationOfAddress}$, $class_v(\text{title}) = \text{Professor}$
- An example of the function $type_v$: $type_v(\text{from}) = \text{date}$, $type_v(\text{title}) = \text{string}$
- An example of the function $count$: $count(\text{Address}) = *$, $count(\text{Faculty}) = 1$
- An example of a changed value of the cardinality: $rcard(\text{Person, Rel1}) = \langle 0, * \rangle$

Definition 6.3. (*PIM* path). A *PIM* path P is a sequence $(R_1, \dots R_n)$ of relations or generalizations from $\mathcal{R} \cup \mathcal{R}_\mathcal{G}$, where $(\forall i \in \{1, n\})(ends(R_i) = (C_{i-1}, C_i))$. C_0 and C_n are called the start and the end of P. Functions *start* and *end* return for P the start and end of P, respectively. $\mathcal{P}$ denotes the set of all *PIM* paths in $\mathcal{S}$. This definition can be applied to the *PIM-View* schema similarly with restricted sets and functions.

A *PIM* path represents a path through classes of a concrete *PIM* schema. It is used with business rules expressed as OCL constraints.

6.3 *PSM* Model

Definition 6.4. (*PSM* schema). A platform specific model schema is a 14-tuple $\mathcal{S}' = (\mathcal{C}', \mathcal{K}', \mathcal{A}', \mathcal{R}', \mathcal{R}'_\mathcal{K}, \mathcal{R}'_\mathcal{S}, \mathcal{C}'_\mathcal{S}, name', type', class', ends', rcard', kends', class'_{key})$.

- $\mathcal{C}'$, $\mathcal{K}'$, $\mathcal{A}'$, $\mathcal{R}'$ denote the set of classes, keys, attributes and relations in $\mathcal{S}'$, respectively.
- $\mathcal{R}'_\mathcal{K}$ denote the set of key-relations in $\mathcal{S}'$.
- $\mathcal{R}'_\mathcal{S}$ denote the set of specializations in $\mathcal{S}'$. It is a special type of the relation.
- $\mathcal{C}'_\mathcal{S} \in \mathcal{C}'$ is a class called *schema class* of $\mathcal{S}'$.
- Function $name' : \mathcal{C}' \cup \mathcal{K}' \cup \mathcal{A}' \to \mathcal{L}$ assigns a name to each class, key and attribute. $\mathcal{L}$ denote the set of all names.
- Function $type' : \mathcal{A}' \to \gamma$ assigns a data type to each attribute. γ denote the set of all types.
- Function $class' : \mathcal{A}' \to \mathcal{C}'$ assigns a class to each attribute.
- Function $ends' : \mathcal{R}' \cup \mathcal{R}'_\mathcal{S} \to \mathcal{C}' \times \mathcal{C}'$ assigns a pair of classes to each relation or specialization. Then, for a given relation $R' \in \mathcal{R}'$, where $ends(R') = (C'_1, C'_2)$, we call C'_1 and C'_2 a parent and a child of R', respectively. Therefore, the ordering in $ends(R')$ is important in contrast to PIM relations. We also say that C'_1 is a parent of C'_2 and that C'_2 is a child of C'_1. Similarly for the specialization.

- Function $rcard' : (\mathcal{C}' \cup \mathcal{K}') \times (\mathcal{R}' \cup \mathcal{R}'_{\mathcal{K}}) \rightarrow \langle \mathbb{N}_0 \times (\mathbb{N} \cup \{*\}) \rangle$ assigns a cardinality to each pair of a relation and class or to each pair of a key-relation and class, or to each pair of a key-relation and key, s.t. the class or key participates in the relation or key-relation.

- Function $kends' : \mathcal{R}'_{\mathcal{K}} \rightarrow \mathcal{C}' \times \mathcal{K}'$ assigns a pair of a class and key to each key-relation. The ordering in $kends'(R'_K)$ is important. Similarly as for the function $ends'$. For a given key-relation $R'_k \in \mathcal{R}'_{\mathcal{K}}$, where $kends(R'_k) = (C', K')$, we call C' and K' a parent and a child of R'_K, respectively.

- Function $class'_{key} : \mathcal{K}' \rightarrow \mathcal{C}'$ assigns a class to each key.

The graph $(\mathcal{C}' \cup \mathcal{K}', \mathcal{R}' \cup \mathcal{R}'_{\mathcal{K}} \cup \mathcal{R}'_{\mathcal{S}})$ has to be a directed tree rooted in the *schema class* $\mathcal{C}'_{\mathcal{S}}$. We call each class $\mathcal{C}'$, which is a child of $\mathcal{C}'_{\mathcal{S}}$, a root class.

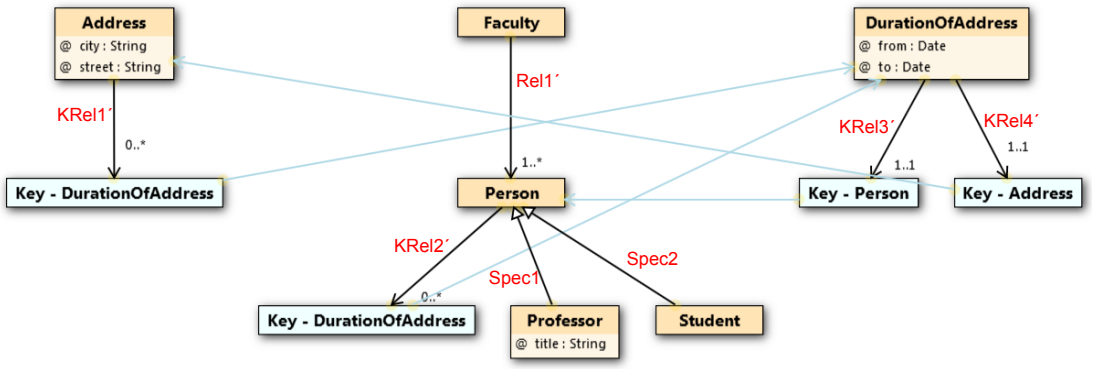


Figure 6.3: Platform specific model schema visualization example

Keys represent a key-ref construct from an XML Schema. In our model, it is a leaf of a directed tree. We use key-relations to represent it. Each key is referencing some *PSM* schema class defined by the function $class'_{key}$. It does not have any attributes, because it is only a reference to some *PSM* schema class.

In our derivation process bellow, we create keys from *PIM-View* schema classes. We create them only in specific cases, which are described in the next chapter.

Example

A platform specific model schema models one data object of a business process model for a concrete platform. In this case, it is Extensible markup language (XML). Figure 6.3 depicts the *PSM* schema in a graphical representation similar to a UML class diagram. Because not all *PSM* schema elements have a name, the diagram is extended with auxiliary names used only for this explanation. These names are depicted by a red colour.

- The set of classes $\mathcal{C}'$ = {Address, Faculty, Person, Professor, Student, DurationOfAddress}

- The set of keys $\mathcal{K}'$ = {Key-DurationOfAddress, Key-DurationOfAddress, Key-Person, Key-Address}

- The set of attributes $\mathcal{A}' = \{$city, street, from, to, title$\}$
- The set of relations $\mathcal{R}' = \{$Rel1'$\}$
- The set of key-relations $\mathcal{R}'_{\mathcal{K}} = \{$KRel1', KRel2', KRel3', KRel4'$\}$
- The set of specializations $\mathcal{R}'_{\mathcal{S}} = \{$Spec1, Spec2$\}$
- The schema class $\mathcal{C}'_{\mathcal{S}}$ is not depicted in the figure. It would be a parent class of Address, Faculty and DurationOfAddress.
- An example of the function *type'*: *type'*(title) = string
- Some examples of the function *class'*: *class'*(title) = Professor, *class'*(from) = DurationOfAddress
- Some examples of the function *ends'*: *ends'*(Rel1') = (Faculty, Person), *ends'*(Spec1) = (Person, Professor)
- Some examples of the function *rcard'*: *rcard'*(Rel1', Faculty) = $\langle 1, * \rangle$, *rcard'*(KRel1', Address) = $\langle 0, * \rangle$
- An example of the function *kends'*: *kends'*(KRel1') = (Address, Key-DurationOfAddress)
- An example of the function $class'_{key}$: $class'_{key}$(Key-Person) = (Person)

Definition 6.5. (*PSM* schema *Forest*). A platform specific model schema forest is a pair $\mathcal{F}' = (\mathcal{S}', \mathcal{R}'_{\mathcal{R}})$.

- $\mathcal{S}'$ denote the *PSM* schema.
- $\mathcal{R}'_{\mathcal{R}}$ denote the list of root classes from $\mathcal{S}'$

A conversion between the *PSM* schema and the *PSM* schema *Forest* is straightforward. We need only the *PSM* schema to get the list of root classes and a given *PSM* schema *Forest* contains the *PSM* schema.

Definition 6.6. (*PSM* root-path $\mathcal{P}_r^{PSM}$) A *PSM* root-path $\mathcal{P}_r^{PSM}$ for *PSM* class $C' \in \mathcal{C}'$ is a sequence $(R'_1, ..., R'_n)$ of relations, key-relations and specializations from $\mathcal{R}'$, $\mathcal{R}'_{\mathcal{K}}$ and $\mathcal{R}'_{\mathcal{S}}$, where $(\forall i \in \{1..n\})(parent'(R'_i) = C_{i-1} \wedge child'(R'_i) = C_i)$ and C_0 is a root class, and $C_n = C'$.

A *PSM* root-path represents a path from a class to a root class by using relations, key-relations and specializations.

Example

We use the *PSM* schema depicted in Figure 6.3 in this example.

$\mathcal{P}_r^{PSM}(Key - DurationOfAddress) = \{$KRel'2, Rel'1$\}$

6.4 Interpretation

Definition 6.7. (Interpretation I_{pim_v}) An interpretation I_{pim_v} of the *PSM* schema $\mathcal{S}'$ against the PIM-View schema $\mathcal{S}_v$ is a partial function. I_{pim_v} maps:

- $\mathcal{C}' \cup \mathcal{K}' \rightarrow \mathcal{C}_v \cup \{c_{a_1}, c_{a_2}, ..., c_{a_k}\}$:

$$(\forall i \in \{1..k\}) c_{a_i} \in \mathcal{C}_{\mathcal{A}v}, k \geq 1$$
$$\wedge$$
$$i \neq j \Rightarrow c_{a_i} \neq c_{a_j}$$
$$\wedge$$
$$(\exists! R' \in \mathcal{R}')((\forall i \in \{1..k\}) aends_{rel}(assoc_{rel}(c_{a_i})) = R')$$

- $\mathcal{A}' \rightarrow \mathcal{A}_v$
- $\mathcal{R}' \cup \mathcal{R}'_{\mathcal{K}} \rightarrow \mathcal{R}_v$
- $\mathcal{R}'_{\mathcal{S}} \rightarrow \mathcal{R}_{\mathcal{G}_v}$

from $\mathcal{S}'$ to $\mathcal{S}_v$. Only the relation $R' \in \mathcal{R}'$ such that parent$(R') \neq \mathcal{C}'_{\mathcal{S}}$, may have a defined interpretation. Moreover, following conditions must be satisfied:

- $(\forall A' \in \mathcal{A}')(class_v(I_{pim_v}(A')) = I_{pim_v}(class'(A')))$
- $(\forall R'_S \in \mathcal{R}'_{\mathcal{S}})(ends_v(I_{pim_v}(R'_S)) = \{I_{pim_v}(parent'(R'_S)), I_{pim_v}(child'(R'_S))\})$
- $(\forall R' \in \mathcal{R}' \wedge \forall R' \in \mathcal{R}'_{\mathcal{K}})$:
  - if $I_{pim_v}(parent'(R')) \in \mathcal{C}_v \wedge I_{pim_v}(child'(R')) \in \mathcal{C}_v$ then holds: $ends_v(I_{pim_v}(R')) = \{I_{pim_v}(parent'(R')), I_{pim_v}(child'(R'))\}$
  - if $I_{pim_v}(parent'(R')) = \{c_{a_1}, c_{a_2}, ..., c_{a_k}\}$ then

$$first_{rel}(ends_v(I_{pim_v}(R'))) = I_{pim_v}(child'(R'))$$
$$\vee$$
$$second_{rel}(ends_v(I_{pim_v}(R'))) = I_{pim_v}(child'(R'))$$

  - if $I_{pim_v}(child'(R')) = \{c_{a_1}, c_{a_2}, ..., c_{a_k}\}$ then

$$first_{rel}(ends_v(I_{pim_v}(R'))) = I_{pim_v}(parent'(R'))$$
$$\vee$$
$$second_{rel}(ends_v(I_{pim_v}(R'))) = I_{pim_v}(parent'(R'))$$

Interpretation I_{pim_v} represents a mapping from a *PSM* schema to a *PIM-View* schema. Conditions, which must be satisfied, represent rules of this mapping. The first two conditions are simple. The last condition is more complex, because of *PIM-View* schema association classes. The first part of the last condition says, that if a parent and a child of the *PSM* schema relation are mapped on *PIM-View* schema classes, then values of functions $ends_v$ and $ends'$ have to be mapped on each other respectively. The next two parts of the last condition say, that if a parent or a child is mapped on association classes, then the first or the second value of functions $ends_v$ and $ends'$ has to be mapped on each other.

In this thesis, we also use an inverse interpretation $I^{-1}_{pim_v}$, which is defined like an inversion of a partial function I_{pim_v}.

Example

Figure 6.3 depicts the *PSM* schema, which has an interpretation against the *PIM-View* schema depicted in Figure 6.2. The interpretation I_{pim_v} is following:

- classes, association classes and keys:
  I_{pim_v}(Address) = Address
  I_{pim_v}(Person-Key)=Person
  I_{pim_v}(DurationOfAddress) = DurationOfAddress
  ...
- attributes:
  I_{pim_v}(name) = name
  I_{pim_v}(salary) = salary
  ...
- relations and key-relations:
  I_{pim_v}(KRel1') = Rel1
  I_{pim_v}(KRel4') = Rel1
  I_{pim_v}(Rel1') = Rel2
  ...
- specializations:
  I_{pim_v}(Spec1') = Gen1
  I_{pim_v}(Spec2') = Gen2

6.5 Bussiness Rules (OCL)

Definition 6.8. (Function $gen_{parents}$) The function $gen_{parents}$: $\mathcal{C}_v \rightarrow \{\mathcal{C}_v\}$ returns for a given *PIM-View* schema class all parents of this class from generalizations until root classes of generalization trees.

Example

We use an example depicted in Figure 6.2. Let *LazyStudent* be one more *PIM-View* schema class connected by the generalization *Gen3* to the class *Student*. This new class *LazyStudent* is a child of this generalization *Gen3*. The function $gen_{parents}$ returns for the *PIM-View* schema class *LazyStudent* a set of *PIM-View* schema classes: {Student, Person}.

Definition 6.9. (Simple business rule) A simple business rule is a sequence of *PIM-View* classes $(C_{v_1}, C_{v_2}, \ldots, C_{v_n})$, where $n > 1 \wedge (\forall i \in \{1, \ldots, n\}) C_{v_i} \in \mathcal{C}_v$, s.t. $(\forall j \in \{2, \ldots, n\})$ holds (a) $\vee$ (b):

(a) $(\exists R_k \in \mathcal{R}_v) ends_v(R_k) = (C_{v_{j-1}}, C_{v_j})$

(b) $gen_{parents}(C_{v_{j-1}}) \neq \emptyset \wedge \exists C_g \in gen_{parents}(C_{v_{j-1}})$, where
$(\exists R_k \in \mathcal{R}_v) ends_v(R_k) = (C_g, C_{v_j})$

Definition 6.10. (Business rule) A business rule is a pair $\mathcal{BR} = (\mathcal{CO}, \mathcal{RU})$, where

- $\mathcal{CO} \in \mathcal{C}_v$ is a context class. It is a beginning of each simple business rule from $\mathcal{RU}$
- $\mathcal{RU}$ denote the set of simple business rules. The first class in each simple business rule is a context class of this business rule.

The second condition represents a semantics of generalizations from a UML class diagram. Child classes of generalizations can be replaced by parent classes in OCL expressions.

OCL expressions can contain different functions, e.g. *sum*. These functions make a definition of OCL expressions complicated. Therefore, we use this simple definition. In example bellow, we explain a conversion from an OCL epxression to this definition of a business rule. It is a straightforward conversion. Therefore, we use only an example to describe it.

Example

We use the *PIM-View* schema depicted in Figure 6.2 for this example. OCL expressions:

```
Context Address
inv:  self.city <> "Praha"

Context Person
inv:  self->Address:collect(a|a.city = "Praha"):size() < 10

Context Faculty
inv:  self->Person:collect(p | p->Address:collect( a |
a.city = "Praha"):size() > 10):size() > 0
```

The first OCL expression cannot be converted to our business rule. It does not contain a sequence of *PIM-View* schema classes longer than one. It uses attributes of the context class directly.

The second OCL expression can be converted to our business rule. It contains two collection functions: *collect* and *size*. The condition in functions are converted to our business rule as new simple business rules. This OCL expression is converted as:

- $\mathcal{CO}$ = Person
- $\mathcal{RU}$ = {(Person, Address)}

It has only one simple business rule, because the sequence of *PIM-View* schema classes in the condition has only one *PIM-View* schema class.

The third OCL expression can be also converted to our business rule.

- $\mathcal{CO}$ = Faculty
- $\mathcal{RU}$ = {(Faculty, Person), (Person, Address)}

It has two simple business rules, because the sequence in the condtion has two *PIM-View* schema classes.

6.6 *PIM* Process Model

Definition 6.11. (BPM schema) A business process model schema is a 7-tuple $\mathcal{B} = (\mathcal{T}, \mathcal{O}, \mathcal{F}, name, ends_b, model, rules)$ where

- $\mathcal{T}, \mathcal{F}, \mathcal{O}$ denote the set of tasks, flows (sequence and message flows), events and gateways in $\mathcal{B}$, respectively.
- Function $name : \mathcal{T} \cup \mathcal{O} \to \mathcal{L}$ assigns a name to each task, event and gateway. $\mathcal{L}$ denote the set of all names.
- Function $ends_b : \mathcal{F} \to (\mathcal{T} \cup \mathcal{O}) \times (\mathcal{T} \cup \mathcal{O})$ assigns ends to a flow. The ordering of ends is important. It defines the beginning and end of the flow respectively. Then, for the given flow $F \in \mathcal{F}$, where $ends_b(F) = (T_1, T_2)$, we call T_1 and T_2 the beginning and end respectively. We also use auxiliary functions $begin_b(F) = T_1$ and $end_b(F) = T_2$.
- Function *model* assigns a *PIM-View* diagram to a flow in $\mathcal{F}$. *PIM-View* schemas assigned to all flows in $\mathcal{F}$ must be over the common *PIM* schema. Only the flow $F \in \mathcal{F}$, which has $begin_b(F) = T$, where $T \in \mathcal{T}$, can have an assigned *PIM-View* schema.
- Function *rules* assigns a set of business rules to each task or event, or gateway in $\mathcal{T}$ or $\mathcal{O}$. The set may be empty.

This definition of a business process is more general than the specification of BPMN. In this thesis, we do not update a model of the business process. This definition is only used for an analysis of changes in the business process model.

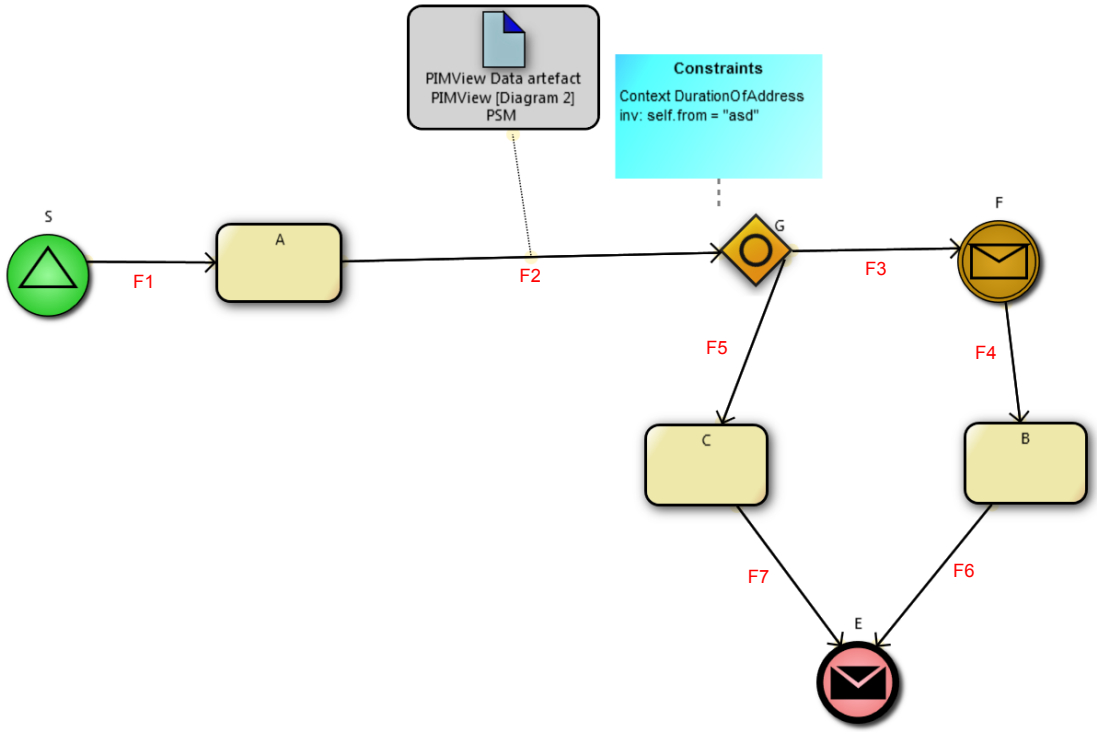


Figure 6.4: Business process model schema visualization example

We allow to assign business rules to events and gateways, because they work with exchanged data. They evaluate some conditions over this data. These conditions can be expressed by OCL expressions.

Example

A business process model schema models one concrete business process. Figure 6.4 depicts a visualization of the BPM schema as a BPMN diagram. Because not all BPM schema elements have a name, the diagram is extended with auxiliary names used only for this explanation. These names are depicted by a red colour.

- The set of tasks $\mathcal{T} = \{\text{A, B, C}\}$
- The set of events and gateways $\mathcal{O} = \{\text{S, G, F, E}\}$
- The set of flows $\mathcal{F} = \{\text{F1, F2, F3, F4, F5, F6, F7}\}$
- Some examples of the function $ends_b$: $ends_b(\text{F1}) = \{\text{S, A}\}$, $ends_b(\text{F3}) = \{\text{G, F}\}$
- An example of the function $model$: $model(\text{F2}) = \text{Diagram2}$ (name of the *PIM-View* schema)
- An example of the function $rules$:
  $rules(\text{G}) = \{$Context DurationOfAddress
  inv: self.from = "asd"$\}$

Chapter 7

Derivation of the Optimal Communication XML Schema

We use the approach of paper [7] to define features of an optimal communication XML schema for a given conceptual schema of a business process, complemented with a conceptual schema of exchanged data. Especially, we use metrics and we add one new metric to find the optimal communication XML schema.

Paper [7] solves the problem of the XML schema's generation from a given conceptual model of exchanged data by generating all possible XML schemas. It does not work with cardinalities of relations. It can lead to a lose of some information for cardinalities with lower value equal to zero. This thesis proposes another method for generating XML schemas. We use the idea presented in paper [1]. We use a navigation and cardinalities of relations in the conceptual model of exchanged data.

As first, we generate XML schemas, then we apply given metrics and choose the optimal XML schema.

7.1 Limitations

The method proposed in this thesis does not solve all problems related to the derivation of the optimal communication XML schema. There are some limitations of the proposed solution:

- An acyclic conceptual model of exchanged data
  The conceptual model of exchanged data can contain cyclic relations. This is a difficult problem to solve in a derivation of hierarchical structure. For simplicity, we work with the acyclic conceptual model.

  It is up to a domain expert to define the conceptual model of exchanged data, which does not contain cyclic relations between classes. This functionality is supported by a creation of another Platform Independent Model called *PIM-View* model. Therefore, the domain expert does not have to change the conceptual model of the problem domain. He or she has to change the conceptual model of the view on the part of the problem domain.

- An association class without relations
  In a general UML class diagram, an association class can contain relations

to another classes of the diagram. This functionality is not used very often in conceptual models of applications. It brings a few problems in deriving of hierarchical structure. We do not solve this concrete problem in the presented solution.

The permission of these relations often makes cycles between classes. The second problem ism how it should be derived to hierarchical structure. An association class contains some additional information about the association (aggregation or composition), which it is connected to. This information has to be derived to hierarchical structure in a close connection to the connected association (aggregation or composition). This can be a problem, if the association class is connected with another relations to another classes.

- A derivation of attributes
  Attributes in a general UML class diagram and in a used *PIM* model and *PIM-View* model can have different data types. XML schemas, as [16], have a strong support for data types. Therefore, it would be very beneficial, if there was some mechanism, which supports deriving data types from the conceptual model to the derived XML schema.

  The work presented in this thesis does not support any special mechanism for deriving data types of attributes and their constraints. But if the domain expert uses only basic data types for attributes and more complex data types are modelled by classes and relations, then the presented solution derives the optimal XML schema with correct data types.

  The derivation or transformation of constraints from the conceptual model to the XML schema is mentioned in papers [1] and [9].

- An ordering of children in generated XML schema
  A XML schema has hierarchical structure. An ordering of children is important in this hierachy. It can be used to store some information. Position can be imporant in some XML documents.

  As we derives an XML schema from the conceptual model, which does not have hierarchical structure, it can have some position information stored only in normal elements, e.g. a class or an attribute. Therefore, positions of children are not important in our proposed algorithm and we do not work with it.

- Multiple inheritance
  UML supports multiple inheritance. The most common XML schema language the XML Schema does not support multiple inheritance. Therefore, we do not solve this problem in a derivation of an optimal XML schema.

  The conceptual model of exchanged data cannot contain classes with more than one supertype.

7.2 Derivation of the First *PSM* Schemas

The first step in the deriving of an optimal XML schema is to derive a few *PSM* schemas by given rules.

7.2.1 Nesting of Classes

As it was mentioned above, we use the approach described in paper [1] to define nesting classes in generated hierarchical structure of a *PSM* schema. In this subsection, we analyse a possible nesting according to relation types and relation cardinalities.

Relation types are more important, because of their semantic information. Therefore, we discuss relation types as first.

Relation Types

Relation types of a *PIM-View* model are equivalent in semantics to relation types of a UML class diagram. According to this semantics, we discuss possibilities to create general rules for nesting, which make a compact and less data redundant *PSM* schema.

- *association* - is a general relationship between two classes.

  Because of it, we cannot use this information to reduce data redundancy by specifying the nesting of classes.

- *aggregation* - is a specialization of the association. It specifies a whole-part relationship between two classes. In basic aggregation relationships, the lifecycle of a part class is independent from the whole class's lifecycle.

  According to this semantics, we cannot use this type of the relation to specify the nesting of classes. A part class can exist without a whole class. We cannot create a general rule for nesting to reduce data redundancy.

- *composition* - is a stronger form of the aggregation, where the whole and parts have coincident lifetimes.

  We can use this type of a relation to specify the nesting of classes. Because one class of this relation is always a part of the other class of the relation. If we do not nest the part class into the whole class, we have to use keys to refer to this part class.

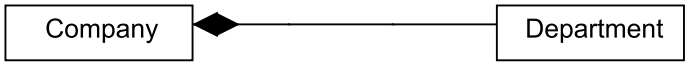


Figure 7.1: Composition example

Figure 7.1 depicts an example of the composition. The class Department, which is a part class, will be always a part of the class Company. It cannot exist without the Company.

Because of this semantics, we always nest classes according to the direction of the composition. In this concrete example, we nest the class Department into the class Company, which is depicted in Figure 7.2.

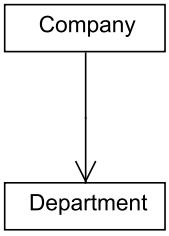


Figure 7.2: Nesting of composition example

Relation Cardinalities

Cardinalities are one of the main indicators for the nesting of classes. They define a number of objects, which participate in the relation.

A domain specialist can change cardinalities in a *PIM-View* schema. It does not have to be the same, as in a *PIM* schema.

For this work only few types of cardinalities are importnat: $\langle 0, 1\rangle$, $\langle 0, m\rangle$, $\langle 0, *\rangle$, $\langle 1, 1\rangle$, $\langle 1, m\rangle$, $\langle 1, *\rangle$, $\langle m, n\rangle$, $\langle m, *\rangle$, where $m, n \in \mathbb{N}_0$ and $m, n > 1$ and $m, n \neq *$.

We divide cardinality pairs in a few groups. They are discussed bellow.

(a) Figure 7.3 depicts cardinality pairs. Each of them has the cardinality $\langle 1, 1\rangle$ on one side of the relation.

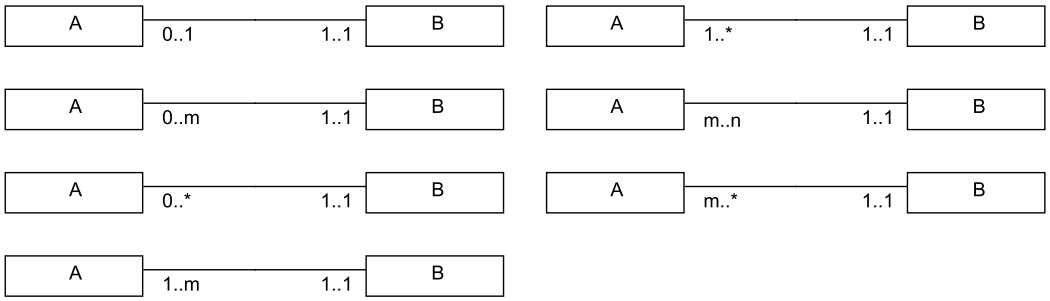


Figure 7.3: Relations with $\langle 1, 1\rangle$ cardinality on one side

In all seven cases, the class A participate in the relation with the cardinality $\langle 1, 1\rangle$. It means, that the class A is related to exactly one class B. In the first three cases, the lower value of the cardinality of the class B is zero. If we nest the class B into the class A, it leads to data redundancy and to acreation of a separate root class for the class B in the *PSM* schema.

Figure 7.4 depicts an example of this nesting. There are several problems with this nesting. The first one is data redundancy created by repeating *Employee* details with each *Subject* occurrence. This happens, because *Employee* can be a head lecturer of more than one *Subject*. The next issue is, that not all *Employee*s are assigned as a head lecturer of some *Subject*. Therefore, a separate root class has to be created for the class *Employee*.

In the next four cases, the nesting of the class B into the class A leads to data redundancy. An example is similar as above.

According to our discussion, we nest the class A into the class B.

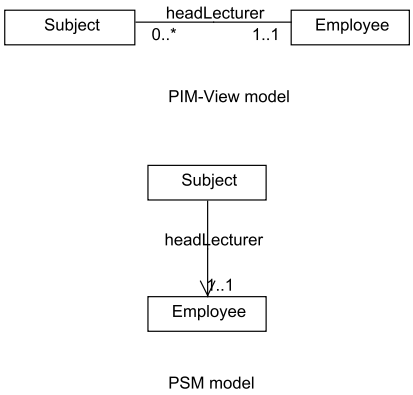


Figure 7.4: Example of nesting $\langle 0, x \rangle$ cardinality into $\langle 1, 1 \rangle$ cardinality

(b) Figure 7.5 depicts cardinality pairs. Each of them has a cardinality $\langle 0, x \rangle$, where $x \in \{1, m, *\}$. On the other side of the relation, there are cardinalities with a lower value ≥ 1.

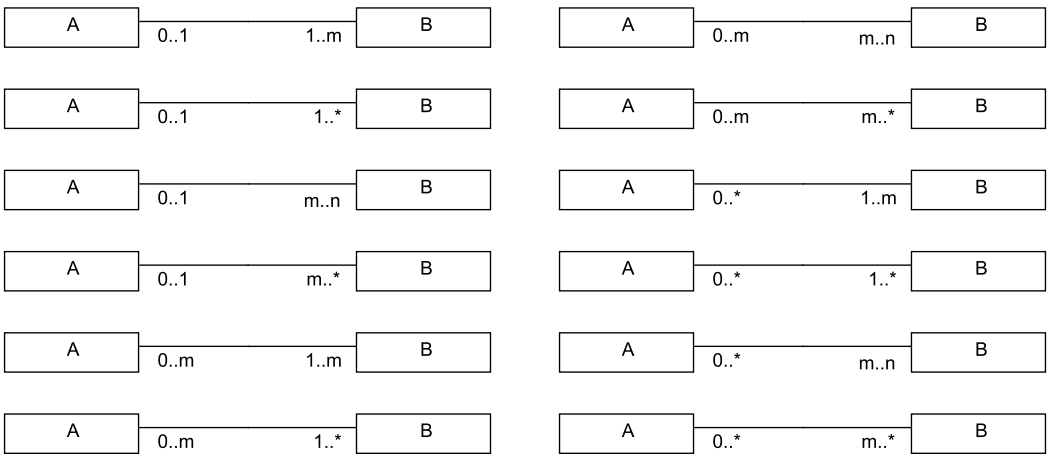


Figure 7.5: Relations with one $\langle 0, x \rangle$ cardinality

If we nest the class B into the class A, we get the same problem as in the previous set of cardinality pairs. If we nest the class A into the class B, we create data redundancy, but there is no need to create a separate global class. We try to generate a compact and less data redundant *PSM* schema. One *PSM* schema class is more compact than one class and one key of the *PSM* schema. We do not try to generate a totally redundancy-free *PSM* schema.

Therefore, we nest the class A into the class B.

(c) Figure 7.6 depicts cardinality pairs. Each of them has one cardinality $\langle x, * \rangle$, where $x \geq 1$ and the second cardinality $\langle x, y \rangle$, where $x \geq 1$ and $y \neq *$.

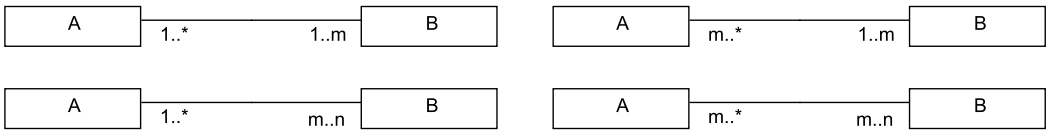


Figure 7.6: Relations with lower values of cardinalities ≥ 1 and one upper value $= *$

If we nest the class B into the class A, we get data redundancy. In the worst case, it is the upper value of the cardinality of the class B. In all cases, it is the biggest possible number $*$. If we nest the class A into the class B, the size of data redundancy is the upper value of the cardinality of the class A, what is m ($m \neq *$).

According to this discussion we nest the class A into the class B.

(d) Figure 7.7 depicts cardinality pairs. Both of them have an upper value of the cardinality > 1 and $\neq *$ and a lower value of the cardinality ≥ 1.

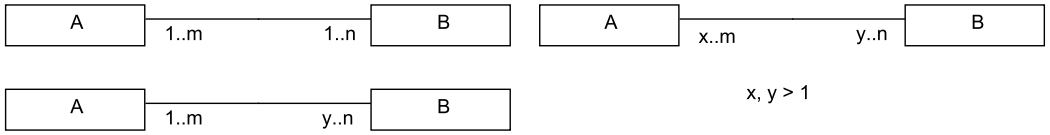


Figure 7.7: Relations with lower values of cardinalities ≥ 1 and upper values $\neq *$

In this case, we try to reduce data redundancy by nesting a class with smaller upper cardinality into a class with bigger upper cardinality. If $m < n$, we nest the class B into the class A. If $m \geq n$, we nest the class A into the class B.

(e) Figure 7.8 depicts cardinality pairs. Each of the cardinality has an upper value $= *$.

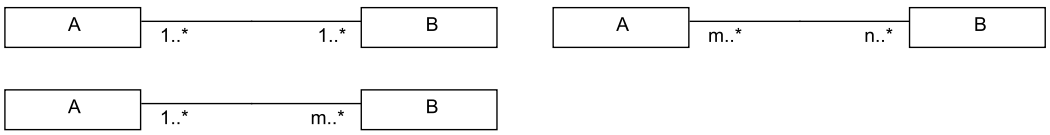


Figure 7.8: Relations with both upper values of cardinalities $= *$

In this cases, we cannot reduce data redundancy by using a cardinality. Here, we can use only an orientation, if it is defined. If the orientation is defined, the

function $orient_v$ returns one of the classes, which participate in the relation. We nest the second class (not returned by the function $orient_v$) returned by the function $ends_v$ into the class returned by the function $orient_v$, e.g. $ends(R) = (c_1, c_2)$ and $orient(R) = c_2$, then we nest the class c_1 into the class c_2.

If the orientation is not defined, we do not nest these classes. We mark the relation as processed and we create two keys. One will be the child of the class A and it will point on the class B by the function $class'_{key}$, the other one vice versa.

(f) Figure 7.9 depicts cardinality pairs. Each of the cardinality has a lower value $= 0$.

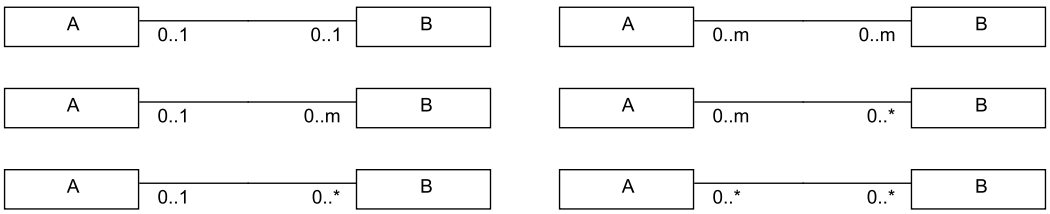


Figure 7.9: Relations with a lower value of cardinality $= 0$ on both sides

In this cases, we cannot nest classes. If we nest classes in any direction, we have to create a global class to be able to model whole data information from the *PIM-View* schema. Therefore, we create two keys. One will be the child of the class A and it will point on the class B by the function $orient_v$, the other one vice versa.

(g) Figure 7.10 depicts the relation, which has both cardinalities equal to $\langle 1, 1 \rangle$.

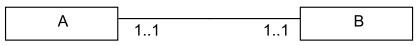


Figure 7.10: Relation with $\langle 1, 1 \rangle$ cardinality on both sides

In this case, we do not use cardinalities or directions to define nesting. If all other relations have defined a nesting class, we process these relations by Algorithm 7.4.

7.2.2 Transformation of Association Classes

In our *PIM-View* schema, we allow to have more association classes connected to one relation. We allow this possibility to support user's comfort in the modelling of a conceptual model. But in a *PSM* schema, which represents exchanged data, it is not important to have more association classes for one relation. Therefore, we merge this association classes into one association class for one relation.

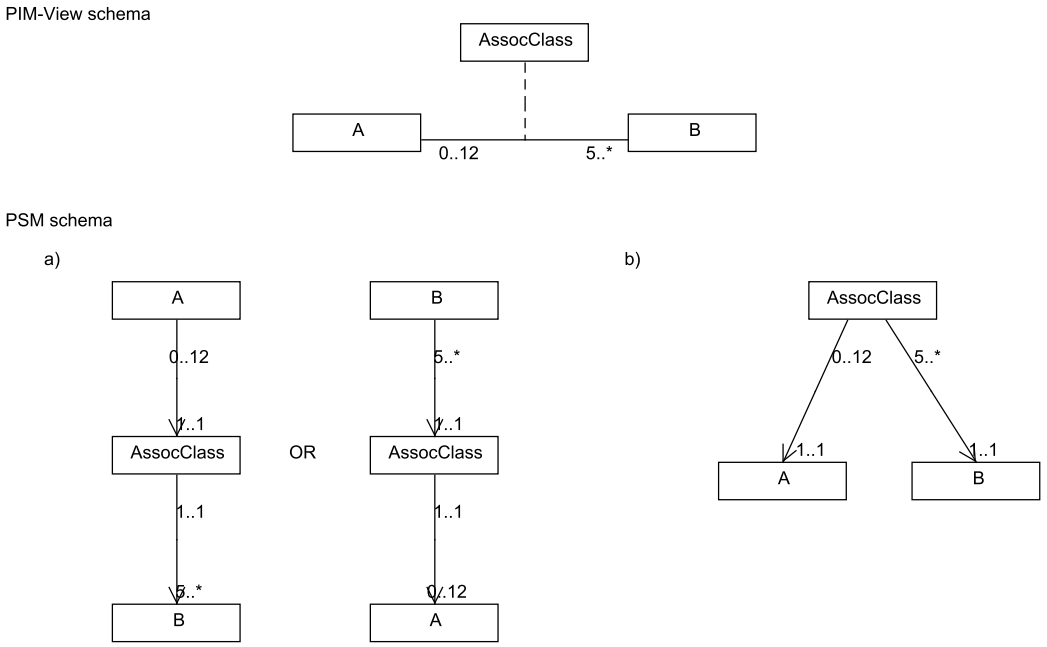


Figure 7.11: Example of an association class transformation

The next problem with association classes is, how we include them in hierarchical structure of a *PSM* schema. We try to maximize connectivity in the created *PSM* schema. Therefore, this association class has to be placed near classes, which participate in the connected relation. According to this discussion, we have two possibilities where to place the association class. Both are depicted in Figure 7.11.

The last thing, which has to be done, is to modify cardinalities in created hierarchical structure according to cardinalities in the *PIM-View* schema. Figure 7.11 depicts how cardinalities are modified.

7.2.3 Algorithms

In this subsection, we describe algorithms used in the process of a derivation of the first *PSM* schemas from a given conceptual model of exchanged data. This conceptual model is modelled by a *PIM-View* schema.

Check Circles

As it was mentioned in section 7.1, the proposed algorithm works only with the acyclic *PIM-View* schema. Therefore, the first Algorithm 7.1 searches for a cycles. If there is some cycle, it informs a user and it will not continue.

This algorithm uses a graph created from a *PIM-View* schema. Vertices are created from classes and association classes, edges are created from relations, generalizations and association-class relations.

Algorithm 7.1 CheckCircles

Input: *PIM-View* schema graph $G_{\mathcal{S}_v}$
Output: true - found circle, false otherwise

```
1: create queue Q_v of vertices
2: create list L_e of edges
3: enqueue random not marked vertex v as visited from G_{S_v} into Q_v
4: mark v as visited
5: while Q_v is not empty do
6:    t ← Q_v.dequeue
7:    for all edges e which contains vertex t in G_{S_v} and are not in L_e do
8:       o ← G_{S_v}.opposite(e, t)
9:       L_e.add(e)
10:      if o is marked as visited then
11:         return true
12:      else
13:         mark o as visited
14:         Q_v.enqueue(o)
15:      end if
16:   end for
17: end while
18: u ← any not marked vertex as visited in G_{S_v}
19: if u is not empty then
20:   result ← CheckCircles(G_{S_v})
21: end if
22: return false
```

Hide Generalizations

Algorithm 7.2 hide generalizations from the *PIM-View* schema. It creates one vertex from vertices connected by generalizations. All relations connected to these classes are reconnected to the new created vertex. Attributes are also moved to the new created vertex. The whole process of hiding is depicted in Figure 7.12.

Algorithm 7.2 HideGeneralizations

Input: PIM-View model graph $G_{\mathcal{S}_v}$
Output: PIM-View schema with hidden generalizations $G_{\mathcal{S}_v H_G}$.

```
1: copy G_{S_v} to G
2: for all generalizations g in G do
3:    create new class c_g
4:    add new class to G
5:    (c_1, c_2) ← ends(g)
6:    remove g from G
7:    reconnect all relations and generalizations from c_1 and c_2 to c_g
8:    reconnect all attributes from c_1 and c_2 to c_g
9:    store information about this hiding of c_1 and c_2 into c_g
10:   remove c_1 and c_2 from G
11: end for
12: return G
```

This operation is necessary to simplify the creation of hierarchical structure. When we hide generalizations, we can work with the *PIM-View* schema as with a graph. Therefore, during the finding of nesting classes, we work with the *PIM-View* schema, which has hidden generalizations.

An additional information about connections between hidden classes, and relations has to be stored in this new vertex. We need it to recreate generalizations.

Some of the next algorithms work with this modified *PIM-View* schema.

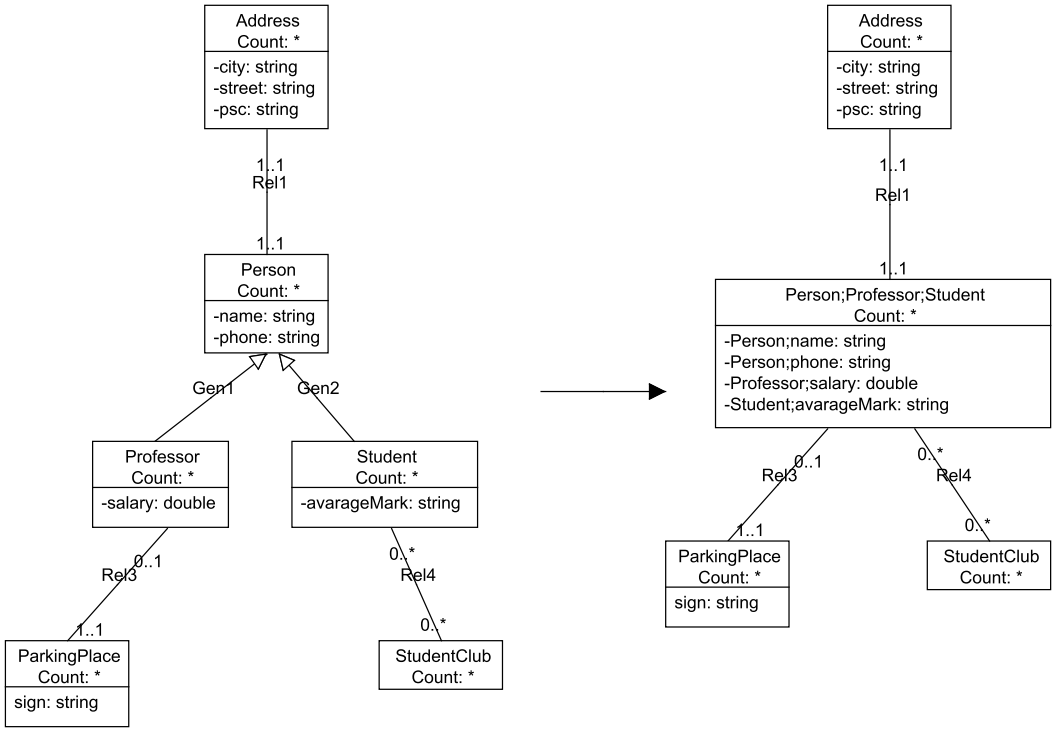


Figure 7.12: Hiding of generalizations in a *PIM-View* schema

Nesting of Classes

These algorithms define nesting classes for most of the relations. They use conclusions from subsection 7.2.1 about relation types and cardinalities. We use the *extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations.

First Algorithm 7.3 defines nesting classes according to the discussion about relation types and cardinalities from a) to e).

Next two Algorithms 7.4 and 7.5 define nesting classes for the relation with cardinalities of type g). This kind of the relation represents *equivalence*. If both or none of the involved classes is a nesting class in some other relation, we look on created subtrees of both involved classes. This step is done as the last one. Therefore, all nesting classes are already defined or the relation is splitted to keys. We can measure the depth of subtrees of both involved classes. According to this depth, we can define the nesting class. If only one of involved classes is a nesting class in some other relation, we can nest the other class.

Last Algorithm 7.6 uses all three algorithms together in the right order. At the beginning, it calls Algorithm 7.3 to find nesting classes. After that, it goes

through all relations of type g) and it calls Algorithm 7.4. At this state, all discussed cases of relations have the nesting class defined.

Algorithm 7.3 FindNestingClasses

Input: *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations
Output: *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations

1: **for all** relation $r \in \mathcal{R}_v, \mathcal{R}_v \in \mathcal{S}_{v_e}$ **do**
2: $(c_{v_1}, c_{v_2}) \leftarrow ends_v(r)$
3: **if** $rtype_v(r) = composition$ **then**
4: **if** $orient_v(r) = c_{v_1}$ **then**
5: $nest_{v_e}(r) \leftarrow c_{v_2}$
6: **else**
7: $nest_{v_e}(r) \leftarrow c_{v_1}$
8: **end if**
9: **else if** r has cardinalities pair of type a) from discussion **then**
10: **if** $rcard_v(c_{v_1}, r) = \langle 1, 1 \rangle$ **then**
11: $nest_{v_e}(r) \leftarrow c_{v_1}$
12: **else**
13: $nest_{v_e}(r) \leftarrow c_{v_2}$
14: **end if**
15: **else if** r has cardinalities pair of type b) from discussion **then**
16: **if** $rcard_v(c_{v_1}, r) = \langle 0, x \rangle$, where $x \in \{1, m, *\}$ and $m > 1 \wedge m \neq *$ **then**
17: $nest_{v_e}(r) \leftarrow c_{v_2}$
18: **else**
19: $nest_{v_e}(r) \leftarrow c_{v_1}$
20: **end if**
21: **else if** r has cardinalities pair of type c) from discussion **then**
22: **if** $rcard_v(c_{v_1}, r) = \langle x, m \rangle$, where $m \neq * \wedge m > 1$ and $x \geq 1$ **then**
23: $nest_{v_e}(r) \leftarrow c_{v_1}$
24: **else**
25: $nest_{v_e}(r) \leftarrow c_{v_2}$
26: **end if**
27: **else if** r has cardinalities pair of type d) from discussion **then**
28: $\langle x, m \rangle \leftarrow rcard_v(c_{v_1}, r)$
29: $\langle y, n \rangle \leftarrow rcard_v(c_{v_2}, r)$
30: **if** $m < n$ **then**
31: $nest_{v_e}(r) \leftarrow c_{v_1}$
32: **else**
33: $nest_{v_e}(r) \leftarrow c_{v_2}$
34: **end if**
35: **else if** r has cardinalities pair of type e) from discussion **then**
36: **if** $orient_v(r) = c_{v_1}$ **then**
37: $nest_{v_e}(r) \leftarrow c_{v_2}$
38: **else if** $orient_v(r) = c_{v_2}$ **then**
39: $nest_{v_e}(r) \leftarrow c_{v_1}$
40: **end if**
41: **end if**
42: **end for**

Algorithm 7.4 GetNestingClassFor1..1Relation

Input: Relation r with both cardinalties $\langle 1, 1 \rangle$, *extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations, list of processed relations P_L

Output: Nesting class c

1: $(c_1, c_2) \leftarrow ends_v(r)$
2: **if** ($c_1 \wedge c_2$ are nesting classes in some other relation) $\vee$ ($c_1 \wedge c_2$ are not nesting classes in any other relation) **then**
3: P_L.add(r)
4: $d \leftarrow GetDepthOfSubtreeInExtendedPIM-View(c_1, P_L)$
5: $e \leftarrow GetDepthOfSubtreeInExtendedPIM-View(c_2, P_L)$
6: P_L.remove(r)
7: **if** $d < e$ **then**
8: **return** c_1
9: **else**
10: **return** c_2
11: **end if**
12: **else**
13: **if** c_1 is nesting class in some other relation **then**
14: **return** c_2
15: **else**
16: **return** c_1
17: **end if**
18: **end if**

Algorithm 7.5 GetDepthOfSubtreeInExtendedPIM-View

Input: Class c, *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations, list of processed relations P_L

Output: Depth of subtree rooted in class c

1: $maxDepth \leftarrow 0$
2: **for all** relations r, where $c \in ends_v(r)$ **do**
3: **if** $nest_{v_e} = null \wedge rtype_v(r) \neq composition \wedge r$ is of type e) $\wedge r \notin P_L$ **then**
4: P_L.add(r)
5: $nest_{v_e}(r) \leftarrow GetNestingClassFor1..1Relation(r, P_L)$
6: P_L.remove(r)
7: **end if**
8: **if** $nest_{v_e} \neq null \wedge nest_{v_e} \neq c$ **then**
9: $(c_1, c_2) \leftarrow ends_v(r)$
10: **if** $c = c_1$ **then**
11: $c_n = c_2$
12: **else**
13: $c_n = c_1$
14: **end if**
15: $m \leftarrow GetDepthOfSubtreeInExtendedPIM-View(c_n, \mathcal{S}_{v_e}, P_L)$
16: $maxDepth \leftarrow max(maxDepth, m)$
17: **end if**
18: **end for**
19: **return** $maxDepth + 1$

Algorithm 7.6 SetNestingClasses

Input: *PIM-View* schema S_v with hidden generalizations
Output: *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations

1: $\mathcal{S}_{v_e} \leftarrow$ create *Extended PIM-View* schema from S_v
2: $\mathcal{S}1_{v_e} \leftarrow FindNestingClasses(\mathcal{S}_{v_e})$
3: **for all** relation $r \in \mathcal{R}_v, \mathcal{R}_v \in \mathcal{S}1_{v_e}$ **do**
4: **if** $nest_{v_e}(r) = null \wedge r$ is of type e) $\wedge\ rtype_v \neq composition$ **then**
5: create list of relations P_L
6: $nest_{v_e}(r) \leftarrow GetNestingClassFor1..1Relation(r, P_L)$
7: **end if**
8: **end for**
9: **return** $\mathcal{S}1_{v_e}$

Derivation of a First Partial *PSM* Schema

These algorithms create a first partial *PSM* schema. They do not solve generalizations and association classes.

The process of the derivation includes a few auxiliary functions. These functions are not described in detail and they are divided into two groups. Functions in the first group have three common parameters. They use a *PIM-View* schema, a *PSM* schema, an I_{pim_v} interpretation and an $I^{-1}_{pim_v}$ inverse interpretation. These parameters are used to store mapping (interpretation) from a *PIM-View* schema to a *PSM* schema and vice versa. They are also used to store new created elements of the *PSM* schema. Because they are common parameters, they are not listed in the description.

- $CreatePSMClass$(c - *PIM-View* schema class)
  At first, it tries to find an interpretation of c in $I^{-1}_{pim_v}$. If the interpretation exists, it returns the *PSM* schema class from this interpretation. If it does not exist, the function creates a new *PSM* schema class from c and its attributes. Mapping is stored into interpretations. The new *PSM* schema class with all its features is stored into the *PSM* schema. All features, it means all values of functions in the *PSM* schema.

- $CreatePSMClass$(L_ac - list of *PIM-View* schema association classes)
  It creates a new *PSM* schema class from L_ac. As it was mentioned in 7.2.2, association classes can be merged into one association class in the *PSM* schema. Mappings are stored into interpretations. The new *PSM* schema class with all its features is stored into the *PSM* schema.

- $CreatePSMKey$(c - *PIM-View* schema class, c' - *PSM* schema class)
  It creates a new *PSM* schema key from c and the value of the function $class'_{key}$ is set to c'. Mapping is stored into interpretations. The new *PSM* schema key with all its features is stored into the *PSM* schema.

- $CreatePSMRelation$(r - *PIM-View* schema relation, c'_p - *PSM* schema parent class, c'_c - *PSM* schema child class, $card_p$ - parent cardinality, $card_c$ - child cardinality)
  It creates a new *PSM* schema relation with a defined parent c'_p and child c'_c.

It also sets values of the function $rcard'$ to $card_p$ and $card_c$, respectively. Mapping is stored into interpretations. The new *PSM* schema relation with all its features is stored into the *PSM* schema.

- $CreatePSMKeyRelation$(*r - PIM-View* schema relation, c'_p - *PSM* schema parent class, k'_c - *PSM* schema child key, $card_p$ - parent cardinality, $card_c$ - child cardinality)
  It creates a new *PSM* schema key-relation with a defined *PSM* schema class c'_p and *PSM* schema key k'_c. It also sets values of the function $rcard'$ to $card_p$ and $card_c$, respectively. Mapping is stored into interpretations. The new *PSM* schema key-relation with all its features is stored into the *PSM* schema.

- $ReplaceWithKeys$(c' *PSM* schema class)
  It is used to replace the defined *PSM* schema class c' by a key. It finds all *PSM* schema relations, where c' is a child. For each of these relations, it creates a *PSM* schema key pointed on c' and another key pointed on a parent of this relation. Then, it adds two key-relations and the previous relation is removed. One of the key-relation has the child key pointed on c' and has the same parent as the relation. The second key-relation has the child key pointed on the parent of the relation and its parent is c'.

A function in the second group is used to get some information from a *PIM-View* schema. It is so simple, that we only give a short description of this function.

- $IsNestingClass$(*c - Extended PIM-View* schema class, *PIM-View* schema)
  It is used to check, if a given class c is used in the function $nest_{v_e}$.

Algorithm 7.7 CreatePSMSchema

Input: *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations
Output: *PSM* schema $\mathcal{S}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

1: create empty PSM model $\mathcal{S}'$
2: create empty interpretation I_{pim_v}
3: create empty inverse interpretation $I^{-1}_{pim_v}$
4: create list of used *PSM* schema classes U_C
5: create list of duplicate *PSM* schema classes D_C
6: create list of visited *PIM-View* schema relations V_R
7: $GetRootClasses(\mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
8: create list of *PSM* schema classes L_C
9: $L_C \leftarrow \mathcal{C}'$
10: **for all** root classes $c_r \in L_C$ **do**
11: $\quad CreateSubTree(c_r, U_C, D_C, V_R, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
12: **end for**
13: $ProcessDuplicitUse(D_C, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
14: $ProcessLastRelations(\mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
15: **return** $\mathcal{S}'$, I_{pim_v}, $I^{-1}_{pim_v}$

As first, we describe main Algorithm 7.7 and the process of this part. We find classes, which can be root classes of the *PSM* schema. Then, we create trees for

these root classes according to found nesting classes. During the creation of trees, we store information about the duplicate use of *PSM* schema classes. The *PSM* schema class can be used in more trees. After that, we process these duplicate classes by creating *PSM* schema keys. At the end, we process associations, which do not have a nesting class in a similar way as duplicate classes, by creating *PSM* schema keys.

Algorithm 7.8 is used to find *PIM-View* schema classes, which can be used as root classes of the *PSM* schema. We apply algorithms for finding nesting classes before this part. Therefore, we can take *PIM-View* schema classes, which are not used as a nesting class in any relation.

Algorithm 7.8 GetRootClasses

Input: *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations, *PSM* schema $\mathcal{S}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

1: **for all** classes $c \in \mathcal{C}_v$ **do**
2: **if** $IsNestingClass(c) = false$ **then**
3: $CreatePSMClass(c, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
4: **end if**
5: **end for**

Algorithm 7.9 CreateSubTree

Input: *PSM* schema class c', list of used *PIM-View* schema classes U_C, list of duplicit *PIM-View* schema classes D_C, list of visited relations V_R, *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations, *PSM* schema $\mathcal{S}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

1: **if** U_C.contains(c') **then**
2: **if** Not D_C.contains(c') **then**
3: D_C.add(c')
4: **end if**
5: **else**
6: U_C.add(c')
7: **end if**
8: $c_{pv} \leftarrow$ mapping for c' from I_{pim_v}
9: **for all** relations r where $c_{pv} \in ends_v(r)$ **do**
10: **if** $nest_{v_e}(r) \neq null \wedge nest_{v_e}(r) \neq c_{pv} \wedge$ Not V_R.contains(r) **then**
11: V_R.add(r)
12: $c'_c \leftarrow CreatePSMClass(nest_{v_e}(r), \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
13: $CreateSubTree(c'_c, U_C, D_C, V_R, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
14: $c_{cpv} \leftarrow$ mapping for c'_c from I_{pim_v}
15: $card_p \leftarrow rcard_v(c_{pv}, r)$
16: $card_c \leftarrow rcard_v(c_{cpv}, r)$
17: $CreatePSMRelation(r, c', c'_c, card_p, card_c, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
18: **end if**
19: **end for**

Algorithm 7.9 is used to create a *PSM* schema according to defined nesting classes. It is also storing information about the duplicate use of *PSM* schema

classes. It also maintains information about visited *PIM-View* schema relations, to avoid of processing some relation twice.

Algorithm 7.10 processes *PSM* schema classes, which are used more than once in Algorithm 7.9. It is a simple action, which replaces these classes with *PSM* schema keys.

It works with an inverse interpretation. In general, this interpretation can return more *PSM* schema classes for one *PIM-View* schema class. But at this point of processing, it is not possible. The function $CreatePSMClass$ tries to find an existing interpretation of the given *PIM-View* schema class, at first. Therefore, there cannot be two interpretations of any *PIM-View* schema class at this point of processing.

Algorithm 7.10 ProcessDuplicitUse

Input: List of duplicate *PIM-View* schema classes D_C, *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations, *PSM* schema $\mathcal{S}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

1: **for all** classes c in D_C **do**
2: $c' \leftarrow$ mapping for c from $I^{-1}_{pim_v}$
3: $ReplaceWithKeys(c', \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
4: **end for**

Algorithm 7.11 ProcessLastRelations

Input: *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations, *PSM* schema $\mathcal{S}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

1: **for all** relations $r \in \mathcal{R}_v$, where $nest_{v_e}(r) = null$ **do**
2: $(c_1, c_2) \leftarrow ends_v(r)$
3: **if** Not exists mapping for c_1 in $I^{-1}_{pim_v}$ **then**
4: $CreatePSMClass(c_1)$
5: **end if**
6: **if** Not exists mapping for c_2 in $I^{-1}_{pim_v}$ **then**
7: $CreatePSMClass(c_2)$
8: **end if**
9: $c'_1 \leftarrow$ mapping for c_1 from $I^{-1}_{pim_v}$
10: $c'_2 \leftarrow$ mapping for c_2 from $I^{-1}_{pim_v}$
11: $ReplaceWithKeys(c'_1, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
12: $ReplaceWithKeys(c'_2, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
13:
14: $card_{c1} \leftarrow rcard_v(c_1, r)$
15: $card_{c2} \leftarrow rcard_v(c_2, r)$
16:
17: $k'_1 \leftarrow CreatePSMKey(c_1, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
18: $CreatePSMKeyRelation(r, c'_2, k'_1, card_{c2}, card_{c1}, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
19: $k'_2 \leftarrow CreatePSMKey(c_2, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
20: $CreatePSMKeyRelation(r, c'_1, k'_2, card_{c1}, card_{c2}, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
21: **end for**

Algorithm 7.11 processes relations of the *PIM-View* schema, which do not have nesting class defined. The discussion, mentioned in subsection 7.2.1, shows cases, where *PSM* schema keys are necessary. This algorithm solves these concrete cases. It uses the function $ReplaceWithKeys$ to repair already done work. Then, it creates *PSM* schema keys, which are pointing on each other.

Unhide Generalizations

As it was mentioned in subsection 7.2.3, we hide generalizations to work with a simplified *PIM-View* schema. In this part we unhide generalizations.

This algorithm contains a few actions, which have to be done. Because, it is mostly a technical work with a *PIM-View* schema and *PSM* schema, we describe in detail only two parts of the algorithm.

At the beginning, we have to find all *PSM* schema classes, which represent hidden generalizations. Each of these classes has to be processed by Algorithm 7.12.

The function $CreatePSMClassesForHiddenPIMViewClasses$ creates a new *PSM* schema class for each hidden *PIM-View* schema class by the function $CreatePSMClass$.

The function $CreatePSMSpecsForHiddenGeneralizations$ creates a new *PSM* schema specialization for each hidden *PIM-View* schema generalization.

The function $ReconnectPSMRelationsAndFixPSMKeys$ reconnects the existing *PSM* schema relations from class c' to new created classes. It also changes *PSM* schema keys, which points on the class c' to new classes. This process is depicted in Figure 7.13.

The function $RemoveBadPSMRelations$ is described in detail by Algorithm 7.13. In the first partial *PSM* schema, which was created by algorithms from the previous part, can be the situation, where a child class of a new specialization can also be a child class of some other relation. This situation is depicted in Figure 7.13. Therefore, it is necessary to replace these relations with *PSM* schema keys.

The function $RemovePSMClass$ removes the PSM schema class and all its features from the *PSM* schema. It also removes interpretations from I_{pim_v} and $I^{-1}_{pim_v}$ for the defined *PSM* schema class.

The function $RemovePSMRelation$ removes the PSM schema relation and all its features from the *PSM* schema. It also removes interpretations from I_{pim_v} and $I^{-1}_{pim_v}$ for the defined *PSM* schema relation.

Algorithm 7.12 UnhideGeneralizations

Input: *PSM* schema class c', *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations, *PSM* schema $\mathcal{S}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

1: $c_h \leftarrow$ mapping for c' from I_{pim_v}
2: $CreatePSMClassesForHiddenPIMViewClasses(c_h, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
3: $CreatePSMSpecsForHiddenGeneralizations(c_h, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
4: $ReconnectPSMRelationsAndFixPSMKeys(c', c_h, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
5: $RemoveBadPSMRelations(c_h, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
6: $RemovePSMClass(c', \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$

The function $GetPSMClassesWithBadRelations$ finds *PSM* schema classes, which are a child in some specialization and also in some *PSM* schema relation.

The function $GetPSMRelationsWhereIsChild$ finds *PSM* schema relations, which have a *PSM* schema class defined as a child.

Algorithm 7.13 RemoveBadPSMRelations

Input: *PIM-View* schema class c_h, *Extended PIM-View* schema $\mathcal{S}_{v_e}$ with hidden generalizations, *PSM* schema $\mathcal{S}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

1: $L_R \leftarrow GetPSMClassesWithBadRelations(c_h, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
2: **for all** *PSM* schema class $c' \in L_R$ **do**
3: $\quad L_{CR} \leftarrow GetPSMRelationsWhereIsChild(c', \mathcal{S}')$
4: $\quad$ **for all** *PSM* schema relation $r' \in L_{CR}$ **do**
5: $\quad\quad c'_p \leftarrow parent'(r')$
6: $\quad\quad c_p \leftarrow$ mapping for c'_p from I_{pim_v}
7: $\quad\quad c'_c \leftarrow child'(r')$
8: $\quad\quad c_c \leftarrow$ mapping for c'_c from I_{pim_v}
9: $\quad\quad r \leftarrow$ mapping for r' from I_{pim_v}
10: $\quad\quad RemovePSMRelation(r', \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
11:
12: $\quad\quad card_1 \leftarrow rcard_v(c_c)$
13: $\quad\quad card_2 \leftarrow rcard_v(c_p)$
14: $\quad\quad k'_1 \leftarrow CreatePSMKey(c_p, c'_p, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
15: $\quad\quad CreatePSMKeyRelation(r, c'_c, k'_1, card_1, card_2, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
16: $\quad\quad k'_2 \leftarrow CreatePSMKey(c_c, c'_c, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
17: $\quad\quad CreatePSMKeyRelation(r, c'_p, k'_2, card_2, card_1, \mathcal{S}_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
18: $\quad$ **end for**
19: **end for**

Processing of Association Classes

In subsection 7.2.2, we discuss possibilities of deriving association classes to hierarchical structure of a *PSM* schema. In this part, we describe, how these possibilities are incorporated into the whole derivation process.

There are two possibilities, how a *PIM-View* schema relation can be transformed into a *PSM* schema in this derivation process. They are both depicted in Figure 7.14. The first one is a transformation to a normal *PSM* schema relation, the second one is a transformation to two key-relations and two keys.

If the *PIM-View* schema relation, connected to an association class, is tranformed to the normal *PSM* schema relation, we can apply both possibilities from the discussion about association classes. If it is transformed to the key-relation, we cannot apply the second possibility from the discussion. That is, beacuse key-relations are processed by a different transformation, which is similar to this second possibility.

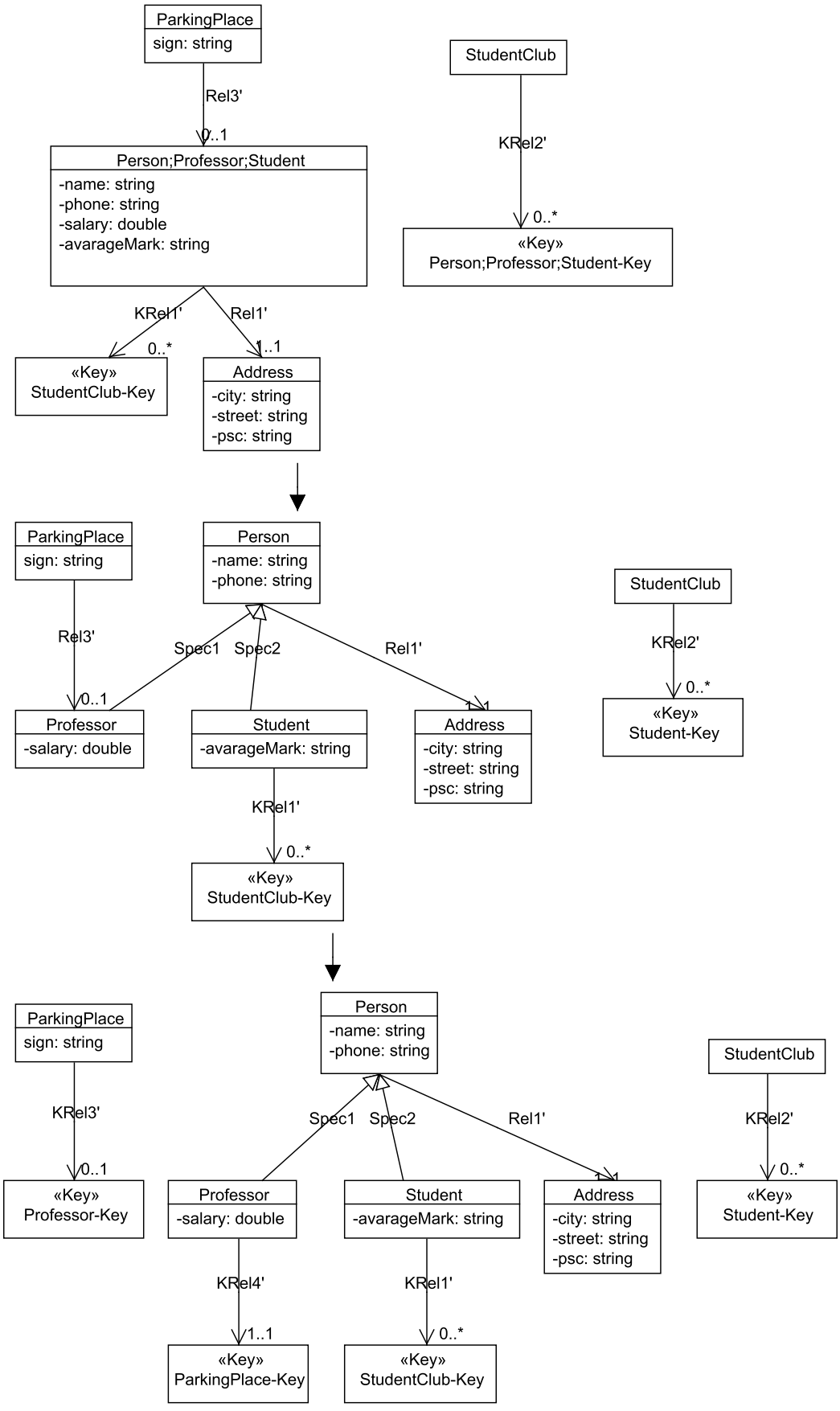


Figure 7.13: Unhiding of generalizations in a *PSM* schema

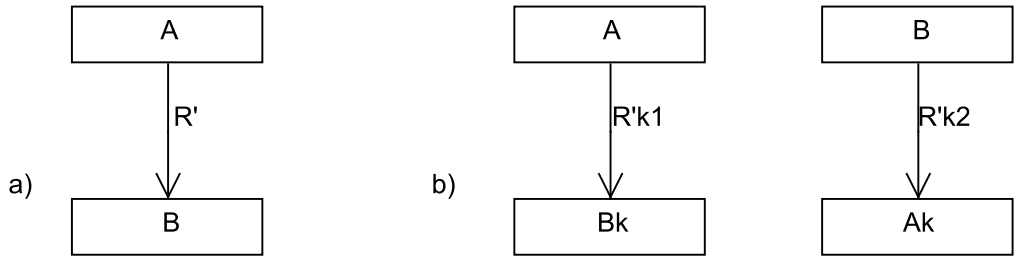


Figure 7.14: Transformations of a *PIM-View* schema relation into a *PSM* schema

There is also one special case associated with this second possibility of the transformation. If a parent class of the *PSM* schema relation is a child in some specialization, we cannot transform this relation by the second possibility of the transformation. The information about a specialization is more important than the transformation of association classes. If we transformed the relation by the second possibility, we would change the information about the generalization from the *PIM-View* schema. This problem is depicted in Figure 7.16.

All association class transformations used in the derivation process are depicted in Figure 7.15.

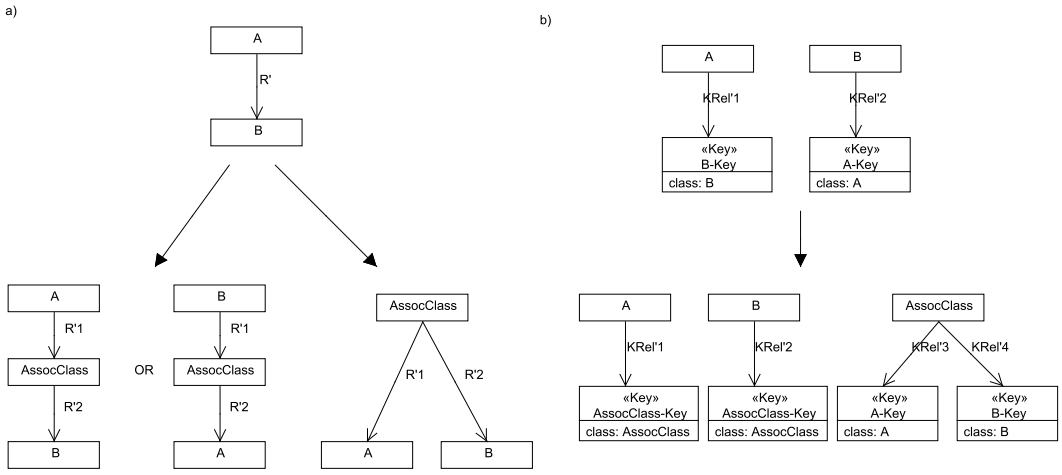


Figure 7.15: Example of an association class transformation in a derivation process

According to the discussion above, there are two possibilities of the association class's transformation in some cases. Therefore, an algorithm in this part takes one *PSM* schema and according to association classes in a *PIM-View* schema, it can create more instances of the *PSM* schema.

Algorithm 7.14 takes all *PIM-View* schema relations with connected association class(es). For each of this relation, it takes already created *PSM* schemas and calls Algorithm 7.15. This second algorithm applies discussed transformations. If it is necessary, it creates a copy of the *PSM* schema, which is updated.

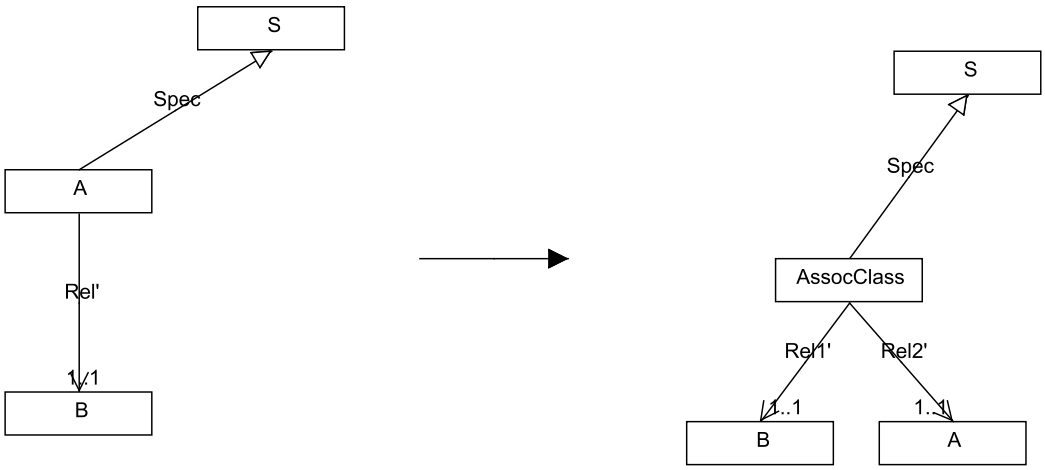


Figure 7.16: Problem with specializations during a transformation of association classes

Algorithm 7.14 ProcessAssociationClasses

Input: *Extended PIM-View* schema $\mathcal{S}_{v_e}$, *PSM* schema $\mathcal{S}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

Output: list of *PSM* schemas

1: create empty list of *PSM* schemas $L_{S'_{sch}}$
2: $L_{S'_{sch}}$.add($\mathcal{S}'$)
3: $L_R \leftarrow GetPIMViewRelationsWithAssocClass(\mathcal{S}_{v_e})$
4:
5: **for all** *PIM-View* schema relation $r \in L_R$ **do**
6: create list of *PSM* schemas $L_{S'_p}(L_{S'_{sch}})$
7: $L_{S'_{sch}}$.clear()
8: **for all** *PSM* schema $S1' \in L_{S'_p}$ **do**
9: $L_{S'1} \leftarrow ProcessOneAssociationClass(r, S1', \mathcal{S}_{v_e}, I_{pim_v}, I^{-1}_{pim_v})$
10: $L_{S'_{sch}}$.addRange($L_{S'1}$)
11: **end for**
12: **end for**
13: **return** $L_{S'_{sch}}$

Algorithm 7.15 ProcessOneAssociationClass

Input: *PIM-View* schema relation r, *PSM* schema $\mathcal{S}'$, *Extended PIM-View* schema $\mathcal{S}_{v_e}$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$
Output: list of *PSM* schemas

1: create empty list of *PSM* schemas $L_{S'_{sch}}$
2: $S'1 \leftarrow ProcessAssociationClassA(r, \mathcal{S}', \mathcal{S}_{v_e}, I_{pim_v}, I^{-1}_{pim_v})$
3: $L_{S'_{sch}}$.add($S'1$)
4: $S'2 \leftarrow ProcessAssociationClassB(r, \mathcal{S}', \mathcal{S}_{v_e}, I_{pim_v}, I^{-1}_{pim_v})$
5: **if** $S'2 \neq null$ **then**
6: $\quad L_{S'_{sch}}$.add($S'2$)
7: **end if**
8: **return** $L_{S'_{sch}}$

Main Algorithm

This main Algorithm 7.16 uses all previous parts and it calls them in the correct order. At the beginning, it creates a copy and a graph from a given *PIM-View* schema. It checks circles and if there are some, it informs user. If there are not circles, it hides generalizations. After that, it sets nesting classes and it creates the first partial *PSM* schema. Then, it calls the algorithm $UnhideGeneralizationsAll$. This algorithm represents the whole process described in the part Unhide generalizations. It uses Algorithm 7.12. At the end, it processes association classes.

Algorithm 7.16 DerivatePSMSchemas

Input: *Extended PIM-View* schema $\mathcal{S}_{v_e}$
Output: list of *PSM* schemas, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

1: create $\mathcal{S}1_{v_e}$ - copy of $\mathcal{S}_{v_e}$
2: create *PIM-View* schema graph $G_{\mathcal{S}_v}$ from $\mathcal{S}1_{v_e}$
3: **if** $CheckCircles(G_{\mathcal{S}_v}) =$ true **then**
4: $\quad$ inform user
5: $\quad$ **return** null
6: **end if**
7:
8: $G_{\mathcal{S}_v H_G} \leftarrow HideGeneralizations(G_{\mathcal{S}_v})$
9: create $\mathcal{S}2_{v_e}$ - *PIM-View* schema with hidden generalizations from $G_{\mathcal{S}_v H_G}$
10: $\mathcal{S}3_{v_e} \leftarrow SetNestingClasses(\mathcal{S}2_{v_e})$
11: $\{\mathcal{S}', I_{pim_v}, I^{-1}_{pim_v}\} \leftarrow CreatePSMSchema(\mathcal{S}3_{v_e})$
12: $UnhideGeneralizationsAll(\mathcal{S}3_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
13: $L_{\mathcal{S}'} \leftarrow ProcessAssociationClasses(\mathcal{S}1_{v_e}, \mathcal{S}', I_{pim_v}, I^{-1}_{pim_v})$
14: **return** $L_{\mathcal{S}'}, I_{pim_v}, I^{-1}_{pim_v}$

7.3 Derivation of Other *PSM* Schemas

In the previous section, we propose an algorithm, which derives first *PSM* . These schemas contain keys and key-relations, which can be replaced by normal relations. This replacement leads to a more redundant *PSM* schema, but it is

more compact and for some metrics it can have better results. Figure 7.17 depicts examples of reducing keys and key-relations.

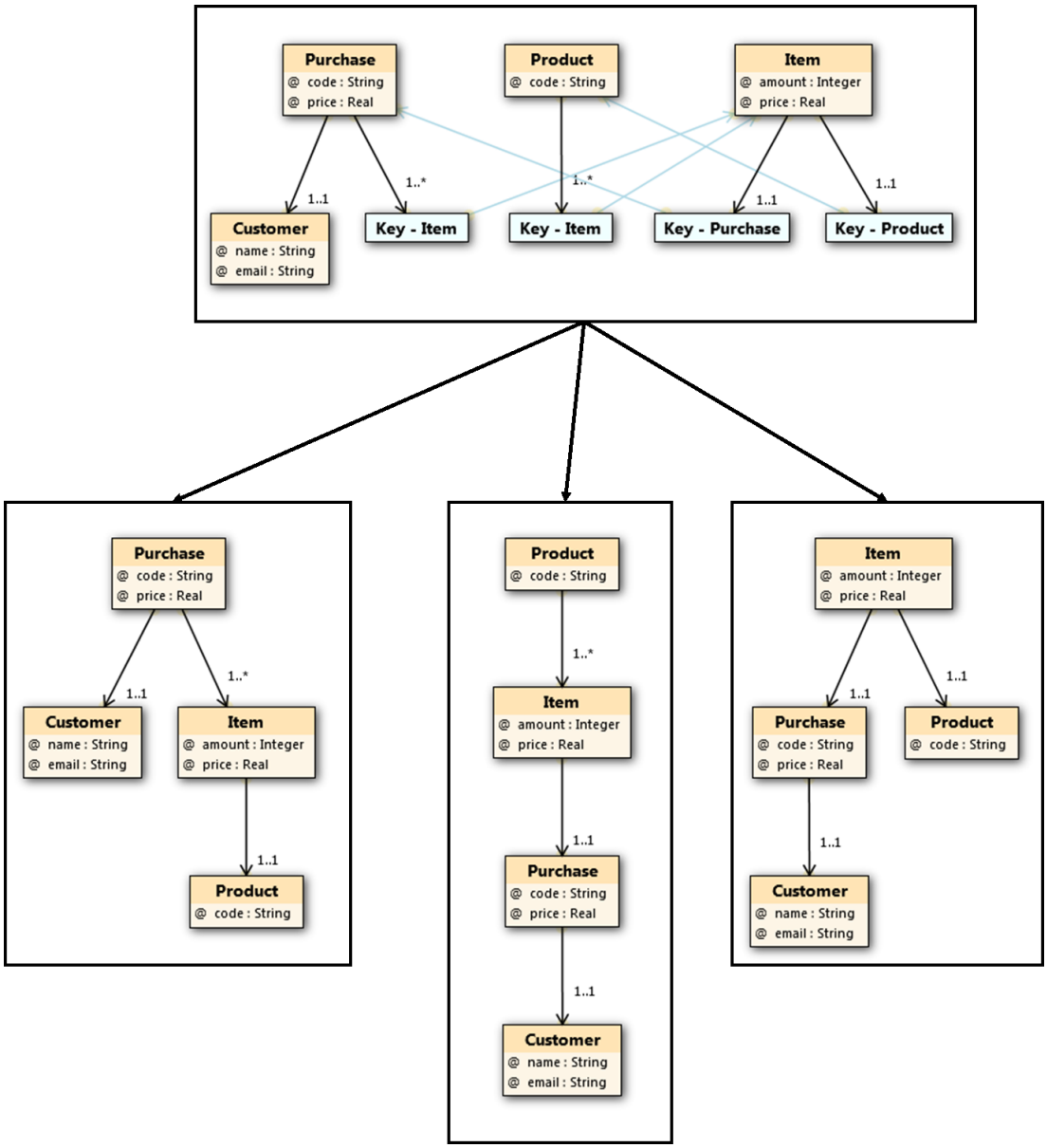


Figure 7.17: Examples of reducing keys in a *PSM* schema

7.3.1 Conditions

To be able to replace these keys and key-relations, we have to find conditions, under which it is possible. In the process of replacing, it is necessary to switch a parent, a child and cardinalities of some relations. Only cardinalities of relations and specializations can have a problem with this replacement.

- Specializations
  We cannot switch a parent and a child of a specialization. This replacement changes a stored information derived from *PIM-View* schema generalizations.

- Cardinalities
  There is a problem only with cardinalities, which have the lower value equal to zero. Replacement of these cardinalities do not solve the problem of keys and key-relations. Because we do not want to lose any information, we need to create new keys and key-relations to preserve this information.

We use the operation *ReplaceKeyWithForest* to replace a key and key-relation by a referenced *PSM* schema class. This operation is depicted in Figure 7.18. We do not describe this operation in detail, because it is mainly a technical work with the *PSM* schema. The main part of the operation is to reverse some relations in the tree, s.t. one concrete class is a new root class of the tree.

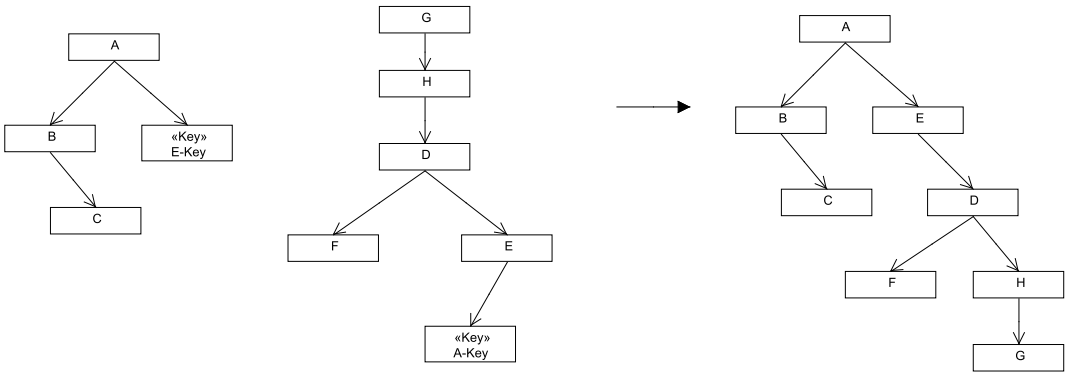


Figure 7.18: Examples of *ReplaceKeyWithForest* operation

As it was mentioned above, this operation can be done only on keys, which satisfy conditions bellow.

Condition 7.1. (Key-relation) Let R'_k be a key-relation, where $kends'(R'_k) = (c', k')$ for a defined key k'. For R'_k has to be satisfied:

- R'_k is mapped to a relation R in the *PIM-View* schema. Let c_1 be a *PIM-View* schema class, which c' was created from and let c_2 be a *PIM-View* schema class, which k' was created from. Then, for R there has to be satisfied (I) $\vee$ (II):

  (I) both cardinalities of R has lower values greater than zero

  (II) $rcard_v(c_1, R) = \langle 0, x\rangle \wedge rcard_v(c_2, R) = \langle z, x\rangle$, where $x \in (\mathbb{N} \cup \{*\})$, $z \geq 1$

Condition 7.2. (Reverse $\mathcal{P}_r^{PSM}$) Let c'_r be a class referenced by a defined key k' ($class'_{key}(k') = c'_r$). Then we have the *PSM* root-path $\mathcal{P}_r^{PSM} = (R'_1, ..., R'_n)$, where $C'_n = c'_r$ and C'_0 is a root class. For each R'_i, where $i \in \{1..n\}$ there has to be satisfied (a) $\wedge$ (b):

(a) R'_i is not a specialization

(b) R'_i is mapped to a *PIM-View* schema relation R_{pv} and both cardinalities of R_{pv} have lower values greater than zero

Let $CanBeKeyReplaced$ be a function returns the true, if both conditions 7.1 and 7.2 are satisfied for the specified key and the false otherwise.

First condition 7.1 checks cardinalities of the *PIM-View* schema relation, which the key-relation was derived from. Second condition 7.2 checks relations, key-relations and specializations in the whole *PSM* root-path. Both conditions checks cardinalities, but with different strictness. The first condition checks the key-relation, which is not going to be reversed. Therefore, it allows zero for the lower value of the parent cardinality. The second condition checks relations, key-relations and specializations, which are going to be reversed. Therefore, it does not allow any zero for lower values of both cardinalities and it also does not allow any specializations.

7.3.2 Algorithms

In this subsection, we describe algorithms used in the process of a derivation of other *PSM* schemas. As it was shown above, there is a possibility to make more compact *PSM* schemas.

In these algorithms, we work with trees (a *PSM* schema *Forest*) created from a *PSM* schema. We need to identify these trees by some unique identifier. The creation of these trees is simple. It is necessary to find root classes. We use the function $GetPSMForest$ to create the *PSM* schema *Forest*. A tree represents the part of the *PSM* schema. We assume, that they are already identified by some unique identifier.

As first, we describe main Algorithm 7.17. For each *PSM* schema, it creates a forest of trees, an empty forest and a list of used trees. After that, it calls Algorithm 7.18. It returns all created *PSM* schemas.

Algorithm 7.17 DeriveAllPSMSchemas

Input: *Extended PIM-View* schema $\mathcal{S}_{v_e}$, list of *PSM* schemas $\mathcal{L}'$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$
Output: list of *PSM* schemas

1: create empty list of *PSM* schemas $L_{S'_{sch}}$
2: **for all** *PSM* schema $S' \in \mathcal{L}'$ **do**
3: create empty *PSM* schema S'_e
4: $F'_e \leftarrow GetPSMForest(S'_e)$
5: create empty list of tree identifiers L_{T_i}
6: $F' \leftarrow GetPSMForest(S')$
7: $L \leftarrow ProcessForest(F', F'_e, L_{T_i}, \mathcal{S}_{v_e}, I_{pim_v}, I^{-1}_{pim_v})$
8: $L_{S'_{sch}}$.addRange(L)
9: **end for**
10: **return** $L_{S'_{sch}}$

Algorithm 7.18 is the most important part of this process. It uses a recursion to derivate *PSM* schemas. As input parameters, it takes the totally first *PSM* schema *Forest* F', the *PSM* schema *Forest*, which is now in progress F'_n, the list of used trees L_{T_i} from F' and other known parameters. It finds all trees, which have not been processed yet. For each of these trees, it creates a copy of F'_n

and L_{T_i} and it calls Algorithm 7.19 to process one tree. This processed tree is stored into the copy of F'_n and information about the processing is stored into the copy of L_{T_i}. If there is an unprocessed tree in this copy, it calls the recursion. If all trees are processed in this copy, it adds copied forest into the result of this algorithm.

The main idea is, that the order of processing of each *PSM* schema key is very important. For example, we have three keys k'_1, k'_2 and k'_3, each of them is referencing to some tree. The order of processing k'_1, k'_2, k'_3 can lead to a different *PSM* schema than the order of processing k'_2, k'_3, k'_1. This feature is also depicted in Figure 7.17.

It uses one auxiliary function $GetNotUsedTrees$. It compares identifiers of trees in two *PSM* schema *Forest*s. It returns trees from the first *PSM* schema *Forest*, which are not in the second *PSM* schema *Forest*.

It also uses *PSM* schema *Forest*'s auxiliary function $addTree$. This function adds a root class into the $\mathcal{R}'_{\mathcal{R}}$ of $\mathcal{F}'$ and it adds each element of the tree into the *PSM* schema $\mathcal{S}'$, the interpretation I_{pim_v} and the inverse interpretation $I^{-1}_{pim_v}$.

Algorithm 7.18 ProcessForest

Input: *PSM* schema *Forest* F', new *PSM* schema *Forest* F'_n, list of used trees L_{T_i}, *Extended PIM-View* schema $\mathcal{S}_{v_e}$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

Output: list of $Forest$s

1: $L_{nu} \leftarrow GetNotUsedTrees(F', L_{T_i})$
2: create empty list of $Forest$s L_{rf}
3: **for all** tree $t_{nu} \in L_{nu}$ **do**
4: create L_{CT_i} copy of L_{T_i}
5: create F'_{cn} copy of F'_n
6:
7: $t \leftarrow ProcessTree(F', F'_{cn}, t_{nu}, L_{CT_i}, null, \mathcal{S}_{v_e}, I_{pim_v}, I^{-1}_{pim_v})$
8: F'_{cn}.addTree(t, I_{pim_v}, $I^{-1}_{pim_v}$)
9: **if** Not L_{CT_i}.contains(t) **then**
10: L_{CT_i}.add(t)
11: **end if**
12:
13: $L_l \leftarrow GetNotUsedTrees(F', L_{CT_i})$
14: **if** L_l is empty **then**
15: L_{rf}.add(F'_{cn})
16: **else**
17: $L_{of} \leftarrow ProcessForest(F', F'_{cn}, L_{CT_i}, \mathcal{S}_{v_e}, I_{pim_v}, I^{-1}_{pim_v})$
18: L_{rf}.addRange(L_{of})
19: **end if**
20: **end for**
21: **return** L_{rf}

Algorithm 7.19 checks conditions discussed in subsection 7.3.1. It applies the operation $ReplaceKeyWithForest$. It uses the recursion to process keys in the tree.

It uses some auxiliary functions:

- $GetKeys(F'_n, t_c, c')$
  It finds all *PSM* schema keys in the given tree t_c except key k', which satisfies the condition $class'_{key}(k') = c'$.

- $IsInForest(F'_n, c')$
  It checks, if the given *PSM* schema class c' is in the given *PSM* schema *Forest*.

- $GetTree(F'_n, c')$
  It finds and returns a tree in the *PSM* schema *Forest* F'_n, which contains the *PSM* schema class c'.

- $Forest.remove(t)$
  It removes the given tree t from $\mathcal{R}'_{\mathcal{R}}$ and all its elements from $\mathcal{S}'$.

Algorithm 7.19 ProcessTree

Input: *PSM* schema *Forest* F', new *PSM* schema *Forest* F'_n, tree t, list of used trees L_{T_i}, *PSM* schema class c', *Extended PIM-View* schema $\mathcal{S}_{v_e}$, interpretation I_{pim_v}, inverse interpretation $I^{-1}_{pim_v}$

Output: tree

```
1: create t_c copy of t
2: L_k ← GetKeys(F'_n, t_c, c')
3: for all PSM schema key k' ∈ L_k do
4:   if CanBeKeyReplaced(k', t_c, F', F'_n) = true then
5:     if IsInForest(F'_n, class'_key(k')) = true then
6:       t_p ← GetTree(F'_n, class'_key(k'))
7:       F'_n.remove(t_p)
8:       ReplaceKeyWithForest(t_c, F'_n, k', t_p, S_ve, I_pimv, I^-1_pimv)
9:     else
10:      t_p ← GetTree(F', class'_key(k'))
11:      r' ← key-relation, where child'(kends'(r')) = k'
12:      c'_p ← parent'(r')
13:      t_n ← ProcessTree(F', F'_n, t_p, L_Ti, c'_p, S_ve, I_pimv, I^-1_pimv)
14:      ReplaceKeyWithForest(t_c, F'_n, k', t_n, S_ve, I_pimv, I^-1_pimv)
15:      if Not L_Ti.contains(t_n) then
16:        L_Ti.add(t_n)
17:      end if
18:    end if
19:  end if
20: end for
21: return t_c
```

At the beginning, the algorithm creates a copy of the tree and finds all keys in the copied tree. The whole algorithm updates only the copy of the tree. It tries to replace all found keys. It uses the function $CanBeKeyReplaced$ to check conditions. After that, it is important, if the given tree is already added into the

processing of the *PSM* schema *Forest*. If it is added, we have to remove it from the *PSM* schema *Forest* and to replace the given key with this tree. If it is not added, we have to process this tree by the recursion and to replace the given key with the processed tree.

7.4 Metrics

In the previous section, we describe algorithms, which create several *PSM* . We use metrics to choose the optimal one for exchanged data. We use metrics, which correspond to wanted features of an XML schema.

In this work, we work with two features of the XML schema. The first one is data redundancy and the second one is a connectivity or compactness. Data redundancy means an occurrence of the same data more than once. The compactness means the storing of related data together in hierarchical structure of the XML schema. We use business rules expressed by OCL for definition of preconditions, post-conditions and invariants. Exchanged data (XML schemas) are validated against these business rules. This validation is done by some XML query language like XQuery [18] or XSLT [19]. Therefore, we try to put data, which are used in business rules, closer to the top of hierarchical structure. This action simplify an access to data and their validation.

A character $*$, which is used in the function *count* and in cardinalities, represents ∞ in metrics.

Example

We explain each metric on some example. We use the *PIM-View* schema depicted in Figure 7.19 for these examples. For the given *PIM-View* schema, we use two *PSM* schemas, which are depicted in Figure 7.20. In this figure, child cardinalities are depicted by a red colour.

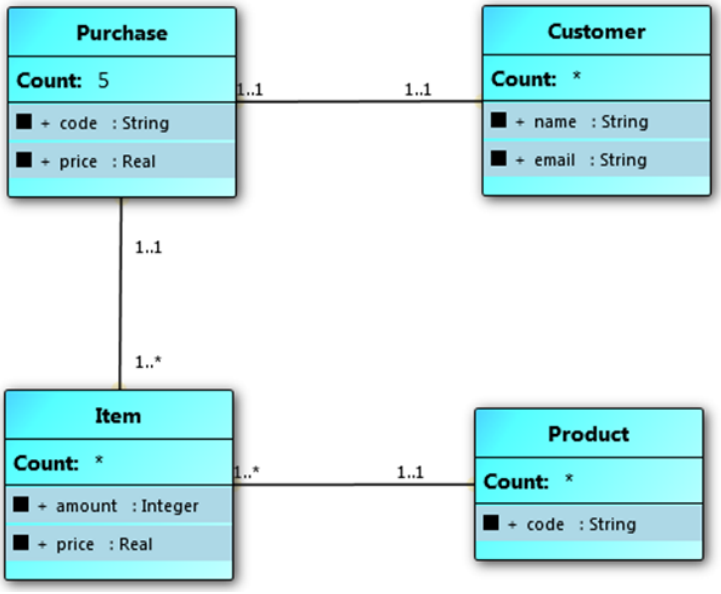


Figure 7.19: Example of a *PIM-View* schema for metrics

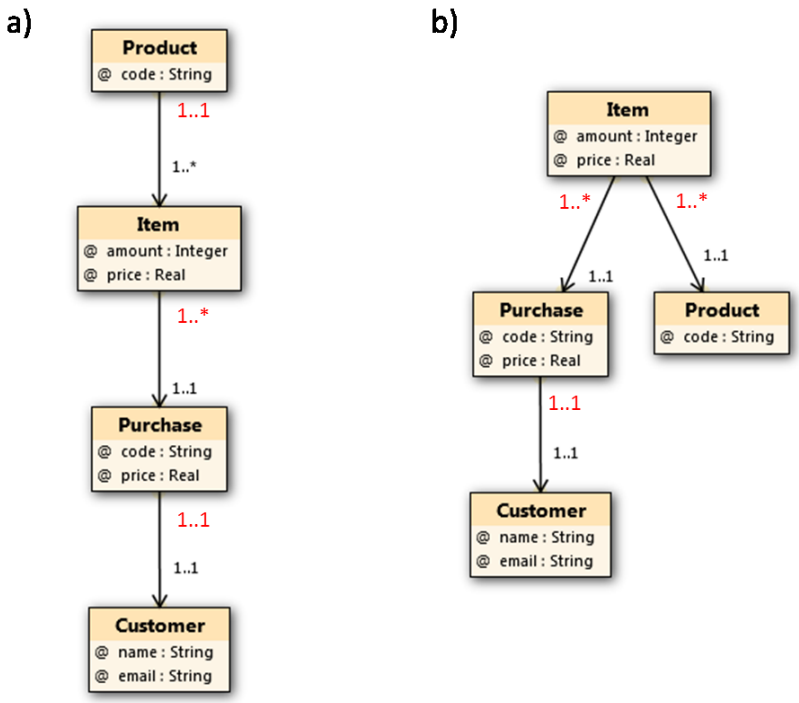


Figure 7.20: Examples of a *PSM* schemas for metrics

Two of these metrics work with business rules. Therefore, we need some OCL expressions. Expressions for our example are written bellow.

```
Context Item
pre:  self.price > 0.2

Context Purchase
inv:  self.price = self->Item:collect(a|a.price * a.amount):sum()
```

7.4.1 Redundancy Metric

The redundancy metric checks measure of redundancy in a given *PSM* schema. At first, it assigns a positive number to each *PSM* schema class and key. Then, it counts a number, which reflects redundancy for one *PSM* schema. This metric is based on paper [7].

Definition 7.1. (*PSM* Class Redundancy) A *PSM* Class Redundancy red_{psm} is a total function, which assigns a positive integer (including ∞) to each *PSM* schema class C' and key K' in $\mathcal{S}'$ except $\mathcal{C}'_{\mathcal{S}}$. For a given $D' \in \mathcal{C}' \cup \mathcal{K}'$, it is defined as follows:

- if D' is a root class and $I_{pim_v}(D') = C$, where $C \in \mathcal{C}_v$, then $red_{psm}(D') = count(I_{pim_v}(D'))$
- if D' is a root class and $I_{pim_v}(D') = \{c_{a_1}, c_{a_2}, ..., c_{a_k}\}$, where $(\forall i \in \{1..k\})$ $c_{a_i} \in \mathcal{C}_{\mathcal{A}v}, k \geq 1$, then $red_{psm}(D') = Max(I_{pim_v}(D'))$. The function Max returns maximum from *count* values of given association classes

- if D' is not a root class, then let's have R', that $R' \in \mathcal{R}' \cup \mathcal{R}'_{\mathcal{K}} \cup \mathcal{R}'_{\mathcal{S}}$, s.t. $child'(R') = D'$
  - if $R' \in \mathcal{R}' \cup \mathcal{R}'_{\mathcal{K}}$, then $red_{psm}(D') = red_{psm}(parent'(R')) * u$, where u is the upper value of $rcard'(R', parent'(R'))$
  - if $R' \in \mathcal{R}'_{\mathcal{S}}$, then $red_{psm}(D') = red_{psm}(parent'(R'))$

The *PSM* Class Redundancy reflects redundancy of one *PSM* schema class. If it is a root class, there are x instances of the class in the XML schema, where x is a value of the function *count* applied on the class. If it is not a root class, then a number of instances depends on a *PSM* schema relation, where the class is a child. The parent's red_{psm} means, that there may be more instances of the parent of this relation. The parent cardinality (participation in the relation) means, that the instance of the child of this relation may be repeated for different instances of the parent.

The number of child's instances directly depends on the number of instances of the parent and on the participation of the parent class in the relation (the cardinality).

We work with specializations in this definition. The specialization is not translated into an XML document directly. It just specifies semantics between types of elements in an XML schema. Therefore, a value of the *PSM* Class Redundancy of a child in a specialization is taken from a parent.

Definition 7.2. *Let $D' \in \mathcal{C}' \cup \mathcal{K}'$ and $R' \in \mathcal{R}' \cup \mathcal{R}'_{\mathcal{K}} \cup \mathcal{R}'_{\mathcal{S}}$, s.t. $child'(R') = D'$. We say that a PSM schema class or key inflicts redundancy, if $red_{psm}(D') > 1$ and $rcard'(R', D') > 1$.*

As it was mentioned, the *PSM* Class Redundancy reflects redundancy of one *PSM* schema class. Therefore, if a value of the red_{psm} is greater than one, it means that the class can represent redundant data in some XML document. This is not enough, because the parent can be redundant, but its subtree does not have to contain any redundant data. Therefore, we also check child's participation in the relation.

Definition 7.3. (Class Redundancy Metric) A Class Redundancy Metric ω_{ck} is a total function assigning a positive number to each *PSM* schema class $C' \in \mathcal{C}'$ and key $K' \in \mathcal{K}'$ as follows:

$$\omega_{ck}(C') = \{ \begin{array}{ll} 0 & \text{if } C' \text{ does not inflict redundancy} \\ size(C') & \text{if } C' \text{ inflicts redundancy} \end{array}$$

$size(C')$ denote the number of *PSM* schema classes and keys in the subtree of C' including C'. Similarly for the key K'.

The Class Redundancy Metric tries to represent, how badly redundant can be one *PSM* schema class. We count the size of the subtree to represent it.

Definition 7.4. (Redundancy PSM Metric) A Redundancy PSM Metric Ω is a function, which assigns a positive number or zero to $\mathcal{S}'$ as follows:

$$\Omega(\mathcal{S}') = \sum_{C' \in (\mathcal{K}' \cup (\mathcal{C}' \setminus \{\mathcal{C}'_S\}))} \omega_{ck}(C')$$

The Redundancy PSM Metric represents measure of data redundancy in the given *PSM* schema. It counts the Class Redundancy Metric for each class separately and sum values.

The Redundancy PSM Metric can be computed by the depth-first search traversal of $\mathcal{S}'$. It is a straightforward approach. Therefore, we omit its explanation.

Example

We use *PSM* schemas depicted in Figure 7.20 to compute redundancy metric. These *PSM* schemas are derived from the *PIM-View* schema depicted in Figure 7.19.

We start with the *PSM* schema *a)*. Only the class *Purchase* can inflict redundancy. All other child classes participate in the relation with the upper value of the cardinality equal to 1. The Class Redundancy Metric of its parent *Item* is equal to ∞, because the upper value of the *Product*'s cardinality is equal to ∞. The size of the subtree is two. Therefore, the value of the Redundancy PSM Metric for the *PSM* schema *a)* is two.

In the *PSM* schema *b)*, only classes *Purchase* and *Product* can inflict redundancy. It is because of the same reason as above. The Class Redundancy Metric of their parent *Item* is equal to ∞. Sizes of the subtrees are two and one. Therefore, the value of the Redundancy PSM Metric for the *PSM* schema *b)* is three.

7.4.2 Context Metric

The context metric checks a position of context classes from business rules. These context classes are important, because they are the beginning of all expressions used in business rules. Therefore, we try to put these classes close to the top of hierarchical structure of an XML schema.

The function $class_{context}$ is an auxiliary function used to get a context class from a business rule. It is a function: $business\ rule \rightarrow \mathcal{C}_v$. It gets context a *PIM-View* schema class from a business rule.

Definition 7.5. (Function $search_{to}$) Let $search_{to}$ be a function: $F \rightarrow \{\mathcal{T} \cup \mathcal{O}\}$, where $F \in \mathcal{F}$. It returns all $T_i \in \mathcal{T}$ and $O_i \in \mathcal{O}$, which satisfied a condition, that there exists a flow path $(F, F_1, \ldots, F_n)$, where $(\forall j \in \{0, \ldots, n\})\ F_j \in \mathcal{F}$. For F holds $ends_b(F) = (T, X)$, where $T \in \mathcal{T}$. For X and n holds

- $n = 0 \rightarrow X \in \mathcal{T}$
- $n = 1 \rightarrow X \in \mathcal{O}$ and $ends_b(F_1) = (O, T_i)$, where $O \in \mathcal{O}$ and $T_i \in \mathcal{T}$
- $n > 1 \rightarrow X \in \mathcal{O}$. For $j < n - 1$ holds $ends_b(F_j) = (O_1, O_2)$, where $O_1 \in \mathcal{O}$ and $O_2 \in \mathcal{O}$. For $j = n$ holds $ends_b(F_n) = (O, T_i)$, where $O \in \mathcal{O}$ and $T_i \in \mathcal{T}$

We allow to connect business rules to tasks, events and gateways. Therefore, we need this function to find all elements from BPM schema, which can have

influence on the given *PIM-View* schema. The *PIM-View* schema can be connected only to a flow F, where $begin_b(F) \in \mathcal{T}$. This function finds all flow paths from this flow to any other task. Each event, gateway and task, which is a part of these paths, can be connected to business rules with influence on the given *PIM-View* schema. Until reaching of a next task, all elements of the path work with same exchanged data.

Definition 7.6. (Context Metric) A Context Metric Φ is a total function, which assigns a positive number to F ($F \in \mathcal{F}$) as follows:

$$\Phi(F) = \sum_{E \in search_{to}(F)} (\sum_{r \in rules(E)} depth(class_{context}(r)))$$

depth denote the number of *PSM* schema relations in the *PSM* root path $\mathcal{P}_r^{PSM}$ for C'.

The Context Metric finds all elements of the BPM schema, which can have influence on the *PIM-View* schema connected to the given sequence flow F. It takes all business rules for found elements and for each it gets a depth of a context class. The Context Metric is a sum of these depths.

The Context Metric can be computed by the depth-first search traversal of $\mathcal{S}'$. We search only for context classes and their depths. We know a sequence flow used to compute the Context Metric. Therefore, the function $search_{to}$ can be computed by the breadth-first search traversal of $\mathcal{B}$. It is a straightforward approach. Therefore, we omit its explanation.

Example

We use *PSM* schemas depicted in Figure 7.20 and OCL expressions written above in section 7.4. These expressions contain two context classes *Item* and *Purchase*.

In the *PSM* schema *a)*, context classes are in depths two and one. Therefore, the Context Metric of this schema is three.

In the *PSM* schema *b)*, context classes are in depths zero and one. Therefore, the Context Metric of this schema is one.

7.4.3 Path Metric

This metric checks paths in business rules. Business rules contain different paths from a *PIM-View* schema. This metric tries to measure a placement of these paths in a *PSM* schema. It is similar as in the context metric, we try to place paths from business rules closer to root classes.

Definition 7.7. (*PSM* subpath) Let $P = (R_1, \ldots, R_k)$ be a *PIM* path in $\mathcal{S}$. A *PSM* subpath of P in $\mathcal{S}'$ is each *PSM* path $P' = (R'_1, \ldots, R'_n)$, where

- $R'_i \in \mathcal{R}' \cup \mathcal{R}'_{\mathcal{K}} \cup \mathcal{R}'_{\mathcal{S}}$
- $1 \leq k \leq n$
- $(1 \leq i \leq n-1)(child'(R'_i) = parent'(R'_{i+1}) \vee parent'(R'_i) = child'(R'_{i+1}))$

- $(\forall i \in \{1 \dots n\})(I_{pim_v}(R'_i) \in \{R_1, \dots, R_k\})$
- $start(P) = I_{pim_v}(start'(P')) \wedge end(P) = I_{pim_v}(end'(P'))$
- P' is maximal, i.e. adding any PSM relation to P' violates previous conditions

Functions $start'$ and end' are equivalents to functions $start$ and end from the definition of *PIM* path, respectively. We use $psmsubpaths(P)$ to denote the set of all *PSM* subpaths of P in $\mathcal{S}'$. Further, we use $length(P)$ and $length'(P')$ to denote the number of *PIM* schema relations and generalization and *PSM* schema relations, key-relations and specializations in P and P', respectively. We count key-relations twice to reflect referencing by using keys. We also use $depth_{path}(P')$ to denote minimal depth of the first or the last *PSM* schema class of the given path P'.

A *PSM* subpath of a *PIM* path represents a path in a *PSM* schema, which is semantically equivalent, when using an interpretation I_{pim_v} between a *PIM-View* schema and a *PSM* schema.

A *PSM* subpath can be longer than a *PIM* path, because of our derivation method of an association classes and because of using key-relations in a *PSM* schema.

The function *paths* is an auxiliary function used to get all *PIM* paths from business rules $rules(E)$, where $E \in \mathcal{T} \cup \mathcal{O}$. These expressions can contain more *PIM* paths for one expression, because they can contain functions working with collections. This example is shown bellow.

Definition 7.8. (Single Path Metric) Let E be a task from $\mathcal{T}$ or event, or gateway from $\mathcal{O}$. A Single Path Metric is a total function ψ_{path}, which assigns a positive number to a *PIM* path $P \in paths(rules(E))$ as follows:

$$\psi_{path}(P) = \sum_{P' \in psmsubpaths(P)} \frac{length'(P')}{length(P)} * depth_{path}(P')$$

The Single Path Metric computes a positive number for one *PIM* path. This number represents a position of a given *PIM* path in a *PSM* schema. Since, there can be more *PSM* subpaths for one *PIM* path, we have to compute a value for each *PSM* subpath. This value works with a length of a *PSM* path. If a *PSM* path is longer than a *PIM* path, it computes worse number.

Definition 7.9. (Path Metric) A Path Metric Ψ is a total function, which assigns a positive number to F ($F \in \mathcal{F}$) as follows:

$$\Psi(F) = \sum_{E \in search_{to}(F)} \left(\sum_{P \in paths(rules(E))} \psi_{path}(P) \right)$$

The whole process starts with a sequence flow. A *PIM-View* schema is connected to this sequence flow. From this connected *PIM-View* schema, we derive a *PSM* schema. From the sequence flow, we can find business rules connected to exchanged data. The Path Metric is computed like a sum of Single Path Metrics

for each PIM path contained in business rules.

We have a start and an end of each *PIM* path. We can use an interpretation I_{pim_v} to find the start and the end of this path in the given *PSM* schema $\mathcal{S}'$. We can use modified breadth-first search (BFS) to find *PSM* subpaths. BFS has to be modified, because of specializations and our derivation method of association classes. It is a straightforward approach. Therefore, we omit its explanation.

Example

For this example, we need to find *PIM* paths in OCL expressions written above in section 7.4. These expressions are defined over the *PIM-View* schema depicted in Figure 7.19.

For the first expression, there is no *PIM* path. The expression uses attributes of the context class. For the second expression, there is only one *PIM* path $(Purchase, Item)$. Other parts of expressions work directly in a place, where they start. Lengths of a *PIM* path and a *PSM* path are the same, both equal to one. Therefore, we only need to work with the depth of the beginning or the end of the path.

For the *PSM* schema *a)*, the end of the path, the class $Item$ is in a lower depth than the beginning. It is a number one.

For the *PSM* schema *b)*, the end of the path, the class $Item$ is in a lower depth than the beginning. It is a number zero.

7.4.4 Final Metric Formula

We have three metric values from a given *PSM* schema. To use these values correctly, we need three user-defined positive numbers. Each number is a weight of one metric. Each communication between tasks can have different requirements according to a nature of a business process. Therefore, we need a project analyst to define the weight of each metric.

Definition 7.10. (Final metric) A Final metric consists of metric functions and three user-defined positive numbers (a, b, c). It is a total function Δ, which assigns a positive number to $\mathcal{S}'$ as follows:

$$\Delta(\mathcal{S}', F, a, b, c) = a * \Omega(\mathcal{S}') + b * \Phi(F) + c * \Psi(F)$$

The Flow F is a flow with an assigned *PIM-View* schema by the function *model*. A *PSM* schema $\mathcal{S}'$ is created from this *PIM-View* schema.

For computation of the value of the Final metric, we need the sequence flow F, which is connected to the *PIM-View* schema of exchanged data. We also need the *PSM* schema derived from the given *PIM-View* schema and weights of metrics.

Example

We use results of examples discussed for each metric. For this example, we use these weights:

- a - redundancy metric weight = 0.33
- b - context metric weight = 0.34
- c - path metric weight = 0.33

For the *PSM* schema *a)*, we get equation:

$$0.33 * 2 + 0.34 * 3 + 0.33 * 1 = 2$$

For the *PSM* schema *b)*, we get equation:

$$0.33 * 3 + 0.34 * 1 + 0.33 * 0 = 1.35$$

Therefore, the *PSM* schema *b)* is more optimal for the given exchanged data and business rules.

7.5 The Optimal XML Schema

We already described all necessary algorithms to derive an optimal XML schema for a given conceptual schema of a business process, complemented with a conceptual schema of exchanged data. This last Algorithm 7.20 puts all things together.

At the beginning, it derives all *PSM* schemas by given rules. Then, it only computes a value of a Final metric for each *PSM* schema and it chooses the *PSM* schema with the lowest value.

Algorithm 7.20 GetOptimalPSMSchema

Input: *PIM-View* schema $\mathcal{S}_v$, (a, b, c) user-defined weights, F sequence flow
Output: *PSM* schema

1: create *Extended PIM-View* schema $\mathcal{S}_{v_e}$ from $\mathcal{S}_v$
2: $(L, I_{pim_v}, I^{-1}_{pim_v}) \leftarrow DerivatePSMSchemas(\mathcal{S}_{v_e})$
3: **if** $L = null$ **then**
4: **return** null
5: **end if**
6: $L_{psm} \leftarrow DeriveAllPSMSchemas(\mathcal{S}_{v_e}, L, I_{pim_v}, I^(-1)_{pim_v})$
7:
8: $resPSM \leftarrow null$
9: $bestMetric \leftarrow \infty$
10: **for all** $S' \in L_{psm}$ **do**
11: $u \leftarrow \Delta(S', F, a, b, c)$
12: **if** $u < bestMetric$ **then**
13: $bestMetric \leftarrow u$
14: $resPSM \leftarrow S'$
15: **end if**
16: **end for**
17: **return** $resPSM$

Chapter 8

Evolution of a Data Artefact

In the previous chapter, we describe algorithms for deriving optimal communication XML schema. In this chapter, we anaylse an influence of user's changes on a derived optimal communication XML schema. We use a data artefact to connect a conceptual model of exchanged data with a business model and business rules. Therefore, we focus on all user's changes with influence on our data artefact and on all parts used to derive the XML schema (*PSM* schema).

8.1 Analyse of User's Changes

A described derivation process uses a few inputs. Exactly, it uses a conceptual model of exchanged data and business rules connected to this conceptual model. Therefore, only changes made on these two inputs can have some influence on a derived *PSM* schema.

- A conceptual model of exchanged data
  Any change made in a conceptual model of exchanged data has a direct influence on a derived *PSM* schema. For example, a change of a name of some class has to be propagated into the derived *PSM* schema. This example is quite simple, but these changes can be more complex, e.g. adding a new relation or class. This change can lead to changes of hierarchical structure of the derived *PSM* schema.

  During derivation, we create connections between a derived *PSM* schema and a *PIM-View* schema. It is a propagation of changes from Platform Independent Model to Platform Specific Model. And in the same way, propagation of changes from PIM to a conceptual model of an XML schema. This problem is already discussed in papers [4] and [5]. The work in these papers already solve most of the problems of the related topic. Therefore, for these changes, we can use approach described in these papers.

- Business rules
  Any change made in business rules used in a derivation process has also a direct influence on a derived *PSM* schema. For example, adding of new business rules can rapidly change metrics used in the derivation process. If user adds three business rules with the same context class, then some other hierarchical structure can be more optimal than the one, which is already used.

During the derivation, we use business rules only in a metric part. We cannot make a partial propagation of changes made in business rules. There is no direct connection between the derived *PSM* schema and business rules. A user can also make some changes in the derived *PSM* schema.

8.2 Changes of Business Rules

According to previous discussion about changes of business rules, we propose an approach described in this section for updating of the derived *PSM* schema.

We apply a derivation process described in the previous chapter on a data artefact and we compute metrics for the *PSM* schema connected to the data artefact. We display some statistic information about a new *PSM* schema and the connected *PSM* schema to a user. After that, the user has to choose one of these *PSM* schemas. The chosen *PSM* schema will be reconnected to the data artefact.

8.2.1 Statistic Information

We choose this statistic information to help a domain expert in the selection of more optimal communication *PSM* schema.

- A value of the context metric
  To compare influence of context classes from business rules on *PSM* schemas.
- A value of the redundancy metric
  To compare measure of redundancy in both *PSM* schemas.
- A value of the path metric
  To compare influence of paths from business rules on *PSM* schemas.
- A value of the Final metric
  To compare all metrics together with user's weights for both *PSM* schemas.
- A count of trees
  To compare a number of trees in both *PSM* schemas. In most cases, a less number of trees signalise better compactness of hierarchical structure.
- A tree statistics
  It contains three values for each tree. A name of a root class, a depth of the tree and a count of nodes in a subtree defined by the root class. Tree's information shows main hierarchical structure of both *PSM* schemas.
- A count of *PSM* schema keys
  It signalises, how compact is a *PSM* schema. If one *PSM* schema contains more keys than another, then it is probably less compact and it also needs more time to process.

8.2.2 Example

In our example, we use the *PIM-View* schema depicted in Figure 7.19 and business rules (OCL expressions) written in an example in section 7.4. The derived optimal *PSM* schema is depicted in Figure 8.1. Other *PSM* schemas, for the given *PIM-View* schema, are depicted in Figure 7.20

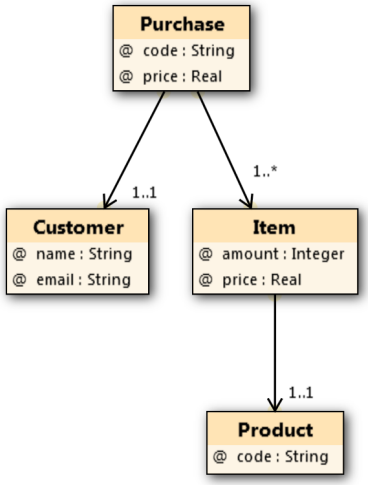


Figure 8.1: Derived optimal *PSM* schema

We add two more business rules for the data artefact:

```
Context Item
inv:  self.amount < 100

Context Item
inv:  self->Purchase.price >= self.price
```

A statistic information about the derived optimal *PSM* schema and the new optimal *PSM* schema is depicted in Figure 8.2.

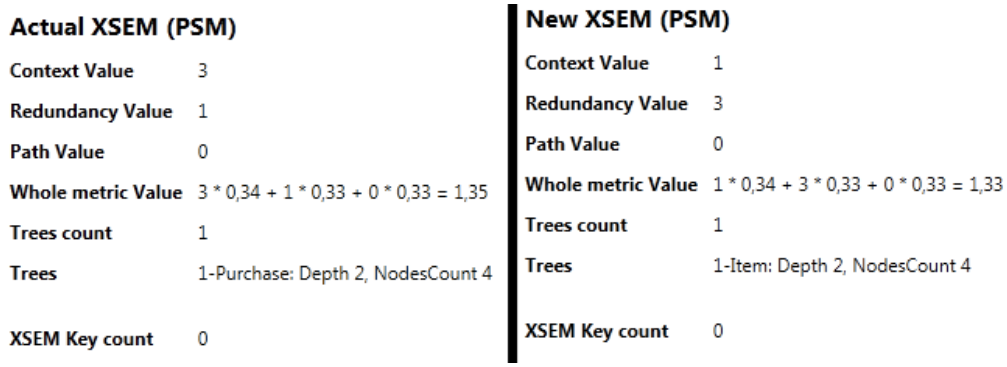


Figure 8.2: *PSM* schemas statistic

The new optimal *PSM* schema has worse value of the redundancy metric, but it has better value of the context metric. The value of the context metric changed, because the context class *Item* has more occurrences in business rules. Because of that, the value of the final metric is better for the new optimal *PSM* schema. This new optimal *PSM* schema is depicted in Figure 7.20 in the part b).

Chapter 9

Implementation and Experiments

In this chapter, we describe a prototype implementation of algorithms described in Chapters 7 and 8. The prototype is implemented as an extension of DaemonX, which is a framework for modelling and evolution processing [23].

9.1 DaemonX

DaemonX is an existing pluginable CASE tool for data and/or process modelling. It was developed as a software project on the Faculty of Mathematics and Physics, of the Charles University in Prague. The aim of the framework is to provide functionality for processing evolution between various models. Models are implemented as modelling plug-ins and evolution is provided by evolution plug-ins. The functionality of constraint languages and their evolution was added in the work [9]. All plug-ins use *application programmable interface* (API) of the framework. A developer of a new plug-in has to make a definition of a new model and define rules for a propagation of atomic operations from a source model to a target model. A detailed description and a documentation of the framework can be found on websites of the DaemonX project [23].

9.2 Implementation

A prototype implementation uses existing PIM, PSM XML and PIM process modelling plug-ins, which were developed as a part of the first release of the DaemonX project. It also uses an extension of DaemonX, which was created in [9]. There were needed some extensions of mentioned parts. They are discussed in subsection 9.2.1.

The implementation adds four new plug-ins. The first plug-in is a modelling plug-in for the *PIM-View* schema, it is described in detail in section 5.2. The second plug-in enables creation of mapping between the PIM model and the PIM-View model. The third plug-in enables creation of mapping between the PIM-View model and the PSM XML model. The fourth plug-in is used to express OCL expressions over the PIM-View model. It is an UCL modelling plug-in.

9.2.1 Extensions

As it was mentioned above, for the purpose of this thesis, we use the software tool DaemonX. It was necessary to make some changes of the software tool and its extension created in [9] to support all features of this thesis.

Extension of PIM process model

We added a new artifact similar to Data Object named *Data Artefact*. This new artifact can be connected only with sequence flows, source of which is some activity. It represents data exchanged in business process model. An example of the new element is depicted in Figure 9.1

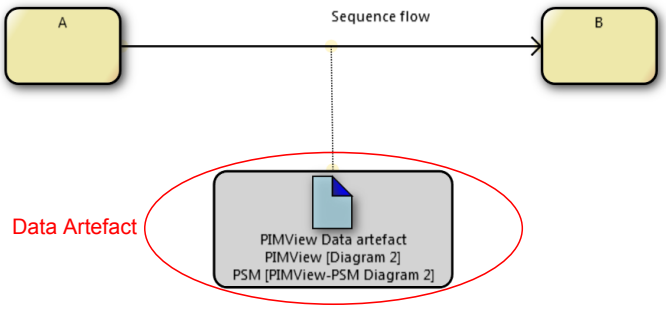


Figure 9.1: Data Artefact example

During the creation of this element, it is necessary to define a *PIM-View* diagram, which represents a conceptual model of this data.

After applying of a proposed algorithm for creation of an optimal XML schema, it stores a connection on the created *PSM XML* diagram.

Most of the operations and algorithms presented in this thesis are placed in a context menu of this element. Their implementation is stored in this plug-in.

Extension of PSM XML

We added two new concepts. The first one is *Specialization* to express inheritance in an XML schema. It is visualized like a generalization in the *PIM* model. An example of the *Specialization* is depicted in Figure 9.2

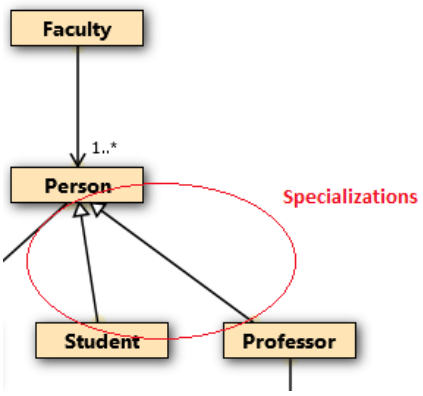


Figure 9.2: Specialization example

The second one is *Key* to express a simple *key-ref* in an XML schema. It is a simple *key-ref*, because it can refer only one other class from the same diagram. This *Key* concept does not solve all modelling problems of *key* and *key-ref* of an XML schema, e.g. it does not solve the problem, how *key* of the referenced class is specified. *Key* is visualized as a blue rectangle and it has a blue line, source of which is *Key*, and the target is the referenced class. The line has an arrow on the side of the target. An example of the element *Key* is depicted in Figure 9.3

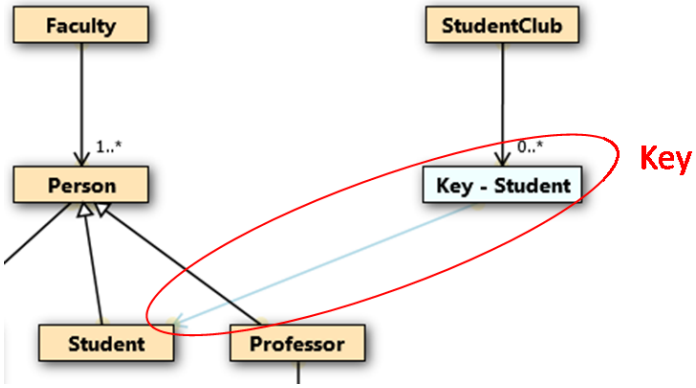


Figure 9.3: Key example

Extension of UCL

The work presented in [9] assumed, that UCL expressions are defined over all diagrams of one model (modelling plug-in) of DaemonX. For the purpose of this thesis, it is necessary to have a possibility to define UCL expressions for one diagram of a model.

It is also necessary to have a possibility to connect UCL expressions to elements of different diagrams. Different diagrams mean not only different diagrams of the same model, but also different diagrams of different models.

The last extension is associated with expressions themselves. In this thesis, we use paths created by UCL expressions. It was necessary to add functionality, which returns parsed paths from expressions.

Figure 9.4 depicts mentioned extension of an UCL part of DaemonX.

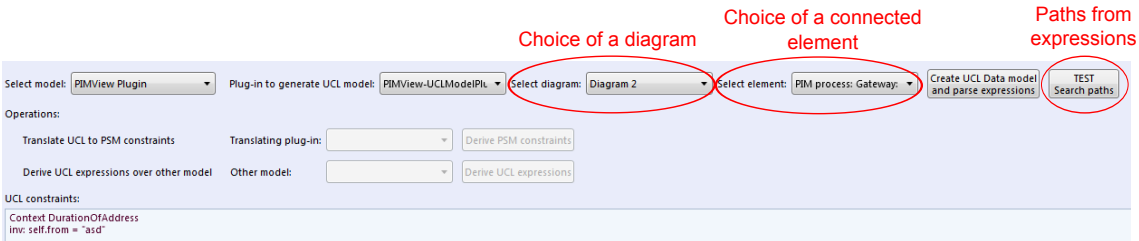


Figure 9.4: Extensions of a UCL part of DaemonX

9.3 Experiments

Simple experiments are described in Chapters 7 and 8 as examples. Because, there is no existing real-world project that provides similar abilities, it is not possible to compare our solution and results with others. Therefore, we created own complex example to test abilities of the solution.

Functionalities of important derivation parts have examples above in Chapters 7 and 8. Therefore, this example shows a more complex business process model. *PIM-View* schemas and derived *PSM* schemas are simple. Therefore, we show only one *PIM-View* schema and one derived *PSM* schema. OCL expressions (business rules) are also simple. They are not described in this thesis, but this example is part of a CD content (see A).

The conceptual model of a problem domain is depicted in Figure 9.5. It is not a whole model of the problem domain. It contains only a part, which was necessary for this example.

The business process model is depicted in Figure 9.6. It models booking of rooms in a hotel and paying by a credit card.

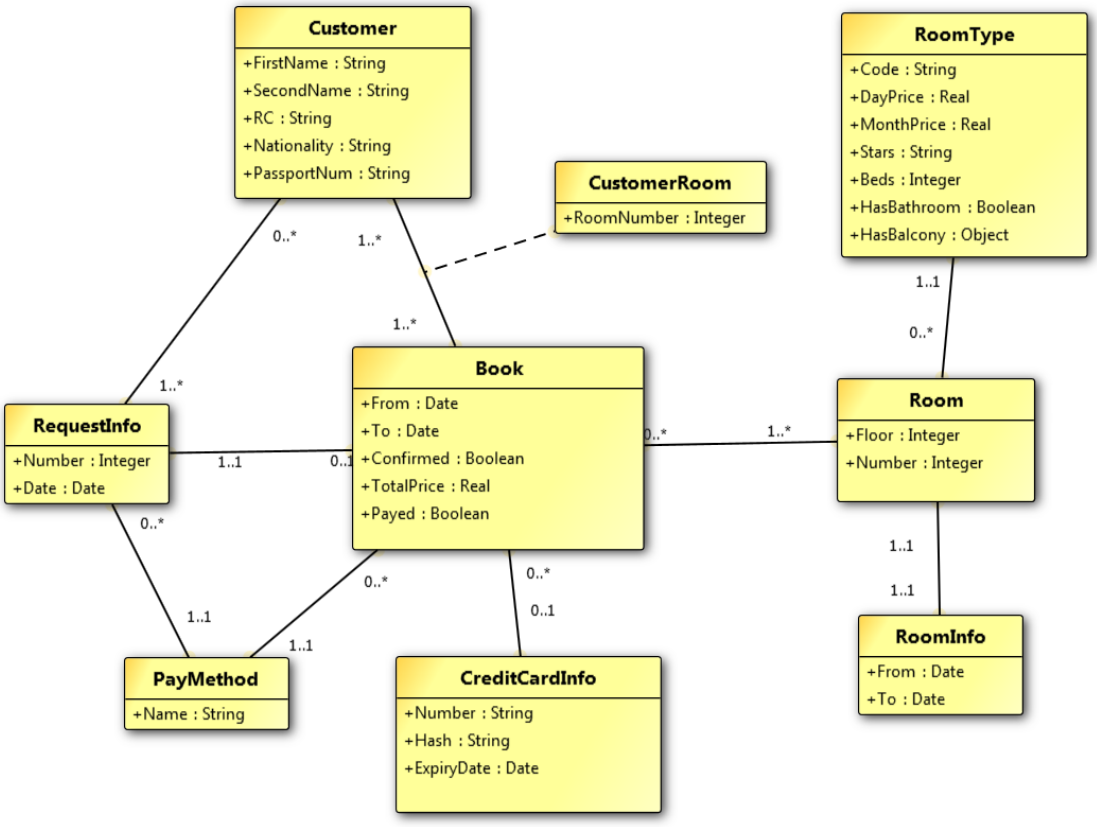


Figure 9.5: *PIM* schema of an experiment

In this thesis, we show the *PIM-View* schema named *PIMView CustomersInfo*. It is connected to the sequence flow, which starts in the task *Get Customers details*. This *PIM-View* schema is depicted in Figure 9.7.

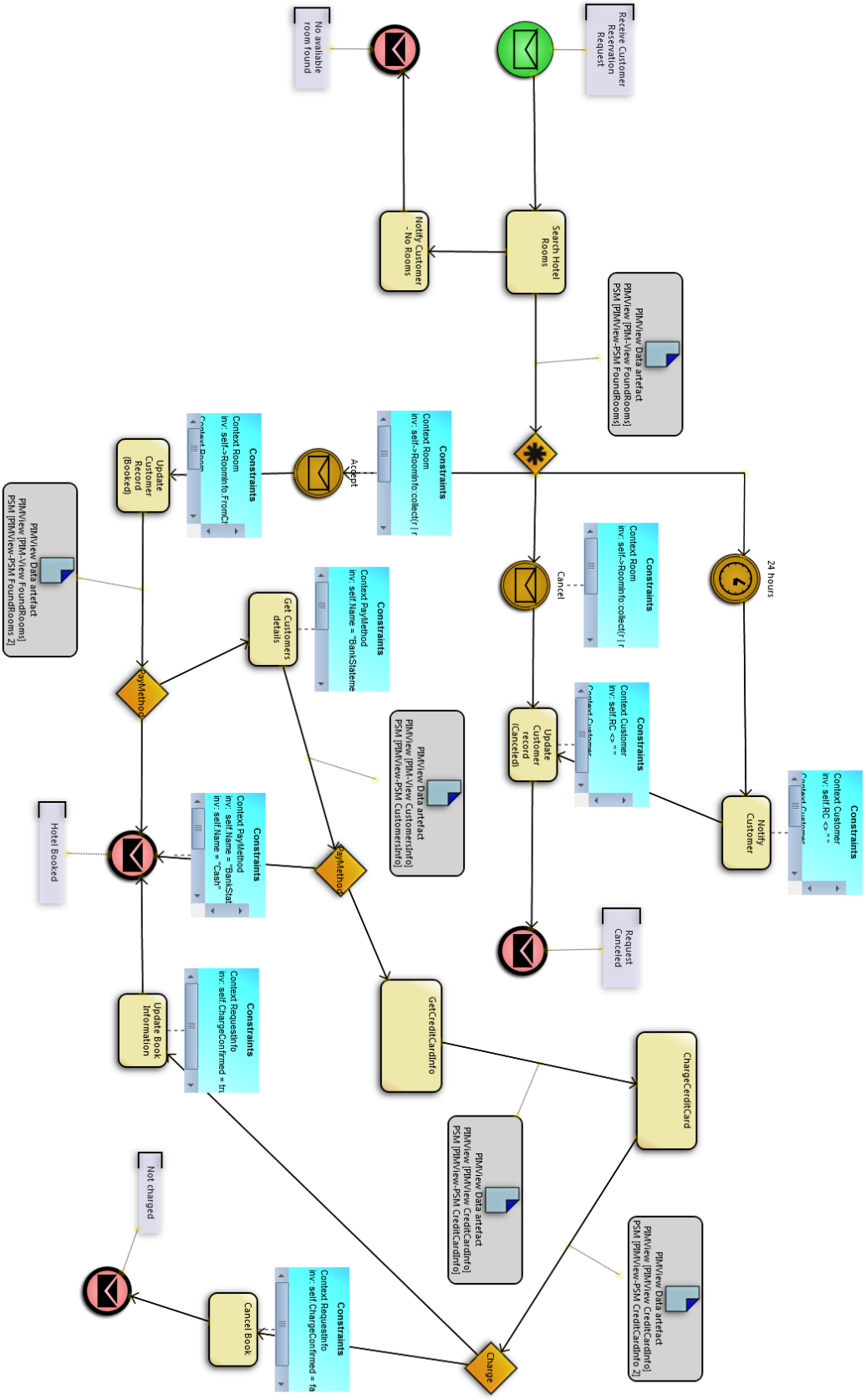


Figure 9.6: Business process model of an experiment

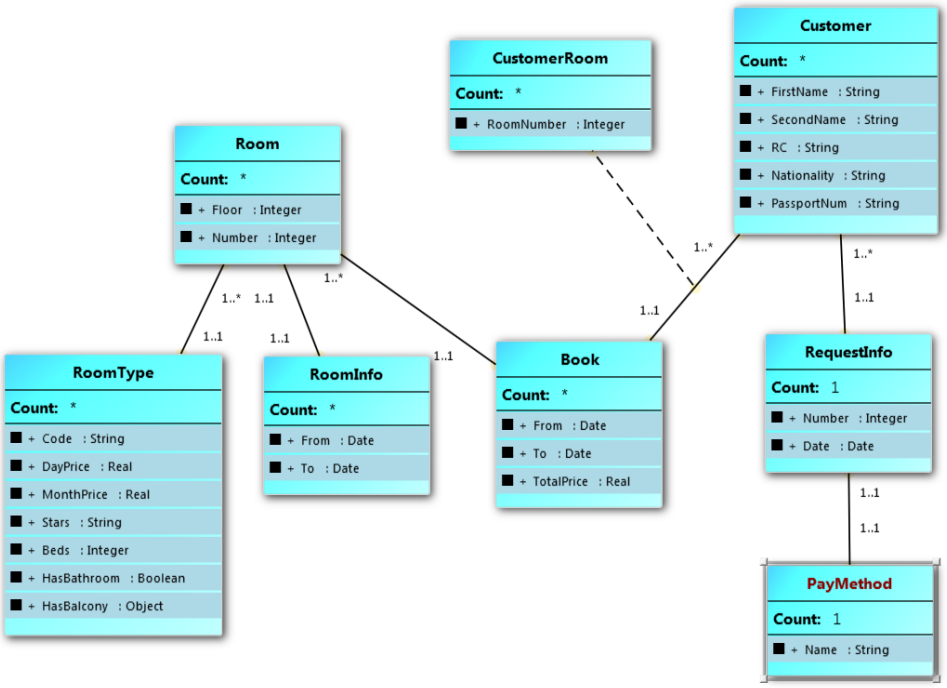


Figure 9.7: *PIM-View* schema *PIMView CustomersInfo*

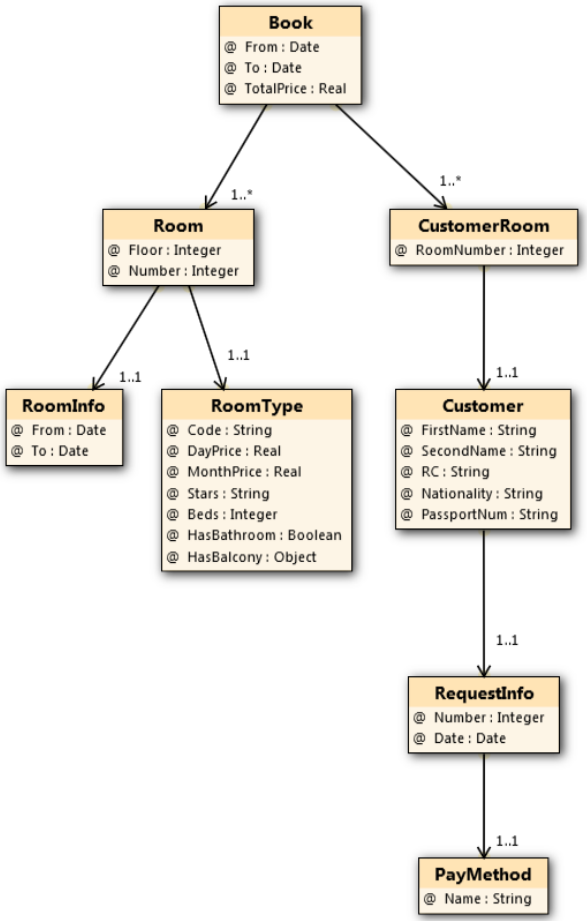


Figure 9.8: *PSM* schema derived from *PIMView CustomersInfo*

The *PSM* schema derived from *PIMView CustomersInfo* is depicted in Figure 9.8. It is a single tree, because we can take the class *Book* as a root class. In this case, there is no relation, which violates conditions from subsection 7.3.1. Therefore, it is possible to create a *PSM* schema with one tree.

This experiment shows a more complex business process model with a simple conceptual model of a problem domain. The derivation of an optimal XML schema is straightforward, because there are not complicated conceptual models of exchanged data.

Chapter 10

Conclusion

In this thesis, we presented an approach to derivation of optimal communication XML schemas for a given conceptual schema of a business process, complemented with a conceptual schema of exchanged data. We proposed algorithms for derivation of an optimal communication XML schema. We analysed impact of user's changes on a derived optimal communication XML schema. As well, we proposed an algorithm to update the derived XML schema according to user's changes of business rules.

We began with introducing of adaptation of business processes problem in general. We introduced the problem of generation of an optimal communication XML schema and the related problem of adaptation of this XML schema to user's changes (see Chapter 1).

In Chapter 2, we introduced theoretical background of this thesis. Chapter 3 introduced a base concept of a model driven architecture, which we used to define roles of proposed models.

Chapter 4 contains analysis of related works dealing with a generation of an optimal XML schema and of related topics. Chapter 5 describes the architecture of the work presented in this thesis. Chapter 6 describes theoretical models used in this work.

In Chapter 7, the main contribution of this thesis was described. In this chapter, we discussed problems related to the derivation of an optimal communication XML schema. This chapter contains algorithms used for derivation of several XML schemas from a given conceptual model. We proposed metrics, which are used to find an optimal XML schema. The end of this chapter describes an algorithm for derivation of an optimal communication XML schema.

In Chapter 8, we discussed an impact of user's changes on a derived XML schema. We identified a problem, which was not solved yet and we proposed a solution of this problem.

The implementation of the proposed solution is described in Chapter 9, together with test experiments. A prototype implementation of the proposed model and algorithms was implemented as an extension of the DaemonX framework. It is available on the attached CD.

10.1 Main Contribution

The main contribution of our approach is the ability to a create conceptual model of exchanged data by a view on a part of a conceptual model of a whole problem domain, automatic derivation of an optimal communication XML schema from a given conceptual model of exchanged data, complemented with business rules working with this exchange data. It provides an algorithm for updating already derived communication XML schema according to user's changes of business rules. It enables derivation of an XML schema, which has good features in use with business processes. This derivation requires minimal cooperation with a user.

10.2 Open Problem

10.2.1 Changes in Business Process Model

In our work, we focus on derivation of an optimal communication XML schema and on changes of inputs of the derivation process. The data artefact is connected to a sequence flow in a given business process model and business rules are connected to elements of this business process model. Any change in the business process model can have influence on the derived XML schema:

- reconnection of the sequence flow can change business rules, which are used in the derivation process
- adding of a new gateway or event can change business rules, which are used in the derivation process
- removing of a task can lead to a change in a conceptual model of exchanged data, which are used in the derivation process

10.3 Future Work

10.3.1 Richer Derivation Process

In section 7.1, we described some limitations of a derivation process. These limitations are not very binding, but for many cases, it is not enough. The main limitation, which is most binding for a domain expert, is an acyclic conceptual model of exchanged data. The domain expert has to create an acyclic conceptual model, but this can be in some cases really difficult.

10.3.2 Optimalization of Derivation Process

A derivation process presented in this thesis has better time complexity in some cases, than a derivation process presented in paper [7]. It still can have an exponential time complexity in cases, where there are relations with both lower values of cardinalities equal to $*$. To improve the time complexity, we need some additional information, which helps us with derivation. This additional information will probably contain some semantic information about a conceptual model of exchanged data.

Appendix A

CD Contents

The attached CD contains:

- PDF version of the thesis - *thesis.pdf*.
- The whole implemented application with sample projects in the directory *implementation*.
- The source codes of the whole DaemonX framework with implemented components for the purposes of this thesis in the directory *src*.
- The documentation of the whole DaemonX framework in the directory *doc*.
- The user documentation of the PIM-View modelling plug-in *doc/PIMView_plugin_user.pdf*

The sample projects are in the directory *implementation/Save*. There are files:

- *example_1.dx*
  Models of this project are shown in Chapter 6.
- *example_2.dx*, *example_2_1.dx*
  Models of this projects are shown in Chapter 7. The first project does not contain the derived optimal *PSM* schema. The second project contains the derived optimal *PSM* schema. The second project also have more OCL expressions.
- *example_3.dx*, *example_3_1.dx*
  Models of this projects are shown in Chapter 9. The first project does not contain derived optimal *PSM* schemas. The second project contains derived optimal *PSM* schemas.

Bibliography

[1] Routledge N., Bird L. and Goodchild A. *UML and XML schema* in *ADC '02: Proceedings of the 13th Australasian database conference* ACS, Melbourne, Australia, 2002

[2] Bird L., Goodchild A. and Halpin T. *Object Role Modeling and XML Schema* in *Proc. International Conceptual Modeling Conference* Salt Lake City, USA, 2000, p. 309-322, Springer.

[3] Halpin T. *Object-Role Modeling (ORM/NIAM), Handbook on Architectures of Information Systems*. Heidelberg, 1998, Chapter 4, Springer

[4] Nečaský M. *XSEM - A Conceptual Model for XML* in *Proc. of the fourth Asia-Pacific conference on Conceptual Modelling* ACS, Inc. Darlinghurst, Australia, 2007, p. 37-48.

[5] Nečaský M., Mlýnková I., Klímek J. *Model-Driven Approach to XML Schema Evolution* in *OTM'11 Proceedings of the 2011th Confederated international conference on On the move to meaningful internet systems* Heidelberg, Berlin, 2011, p. 514-523

[6] Codd E. F. *The Relational Model for Database Management: Version 2.* Reading, Mass, Addison-Wesley. 1990.

[7] Macek O., Nečaský M. *An Extension of Business Process Model for XML Schema Modeling* in *6th World Congress on Services* Miami, Florida, USA, July 2010, p. 383-390, ACM

[8] Nečaský M., Mlýnková I. *Five-Level Multi-Application Schema Evolution* in *DATESO* 2009, p. 90–104

[9] Piják P. *Universal Constraint Language*. Master Thesis Charles University in Prague, Prague, 2011

[10] Sparx Systems. *Enterprise Architect*. 2012 `http://www.sparxsystems.com.au`

[11] Altova. *UModel*. 2012 `http://www.altova.com/umodel.html`

[12] XCase team. *Xcase - tool for xml data modeling.* `http://xcase.codeplex.com/`

[13] Object Management Group. *Business Process Model And Notation (BPMN) Version 2.0*. Object Management Group, January 2011, `http://www.omg.org/spec/BPMN/2.0/PDF`.

[14] Jordan D. *Web Services Business Process Execution Language Version 2.0 Primer* 2007, http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm.

[15] Bray T., Paoli J., Sperberg-McQueen C. M., Maler E. and Yergeau F. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* W3C, November 2008, http://www.w3.org/TR/2008/REC-xml-20081126/

[16] Thompson H. S., Beech D., Maloney M. and Mendelshon N. *XML Schema Part 1: Structures* World Wide Web Consortium, 2004, http://www.w3.org/TR/xmlschema-1/.

[17] International Organization for Standardization. *ISO/IEC 19757-3:2006 Information technology - Document Schema Definition Language (DSDL) - Part 3: Rule-based validation – Schematron*, http://www.schematron.com/.

[18] Boag S., Chamberlin D., Fernández M. F., Florescu D., Robie J. and Siméon J. *XQuery 1.0: An XML Query Language (Second Edition)* World Wide Web Consortium, January 2011, http://www.w3.org/TR/xquery/.

[19] Clark J. *XSL Transformations (XSLT) Version 1.0* World Wide Web Consortium, November 1999, http://www.w3.org/TR/xslt.

[20] Object Management Group. *Unified Modeling Language^{TM} (OMG UML), Superstructure, V2.4.1* Object Management Group, August 2011, http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

[21] Miller J., Mukerji J. *MDA Guide Version 1.0.1.* Object Management Group, 2003. http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf.

[22] Object Management Group. *Object Constraint Language (OCL), Version 2.3.1* Object Management Group, January 2012, http://www.omg.org/spec/OCL/2.3.1/PDF.

[23] DaemonX Team. *DaemonX.* June 2011, http://daemonx.codeplex.com/.

List of figures