Charles University in Prague

Faculty of Mathematics and Physics

DOCTORAL THESIS

Jakub Klímek

XML Formats Evolution and Integration

Department of Software Engineering

Supervisor of the doctoral thesis:	Mgr. Martin Nečaský, Ph.D.
Study programme:	Computer Science
Specialization:	Software Systems

Prague 2013

Let me thank Eliška, my parents, friends and colleagues from the XML and Web Engineering research group for supporting me throughout my lengthy studies.

Let me also thank my supervisor Mgr. Martin Nečaský, Ph.D. for his exemplary leadership and direction and all the anonymous reviewers of my papers for helpful suggestions and insightful opinions.

Finally, let me thank the Czech Science Foundation (GAČR) and Grant Agency of Charles University (GAUK) for supporting me through their grants GAČR P202/10/0573, GAČR P201/09/0990, GAUK 3110, GAUK 572212, GAUK SVV-2010-261312, GAUK SVV-2011-263312, GAUK SVV-2012-265312 and GAUK SVV-2013-267312, respectively.

Název práce: Evoluce a integrace XML formátů

Autor: Jakub Klímek

Katedra: Katedra softwarového inženýrství

Vedoucí disertační práce: Mgr. Martin Nečaský, Ph.D.

Abstrakt: V poslední dekádě se XML stalo široce rozšířeným datovým modelem pro výměnu dat. Spousta uživatelů XML má spoustu XML formátů popsaných pomocí XML schémat. Pro usnadnění správy několika XML schémat modelujících podobnou realitu byl definován konceptuální model pro XML. S jeho definicí přišly různé výzvy, které bylo třeba dále zkoumat. Tato práce se zaměřuje na dvě z těchto výzev. První výzvou je správa a evoluce tohoto vícevrstvého konceptuálního modelu podle toho, jak se v čase vyvíjí modelovaná realita, XML schémata a aplikace. Druhou výzvou je umožnit většině uživatelů, kteří již XML schémata používají bez konceptuálního modelu, jejich schémata použít k jeho poloautomatické tvorbě. Navíc byl podniknut krok směrem k integraci konceptuálního modelování pro XML a technik sémantického webu.

Klíčová slova: XML schéma, evoluce, integrace, konceptuální modelování, MDA

Title: XML Formats Evolution and Integration

Author: Jakub Klímek

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D.

Abstract: In the past decade XML became a wide-spread information exchange data model. Many XML users have many XML formats described by XML schemas. To ease the management of multiple XML schemas modeling similar reality, the conceptual model for XML was defined. With its definition came various challenges that needed to be further researched. This thesis focuses on two of those challenges. The first challenge is to manage the evolution of the multi-level conceptual model as the modeled reality, XML schemas and applications evolve in time. The second challenge is to allow the majority of users who already use XML schemas in their system without the conceptual model to use their schemas to semi-automatically create one. In addition a step towards integration of the conceptual modeling of XML and semantic web techniques was taken.

Keywords: XML schema, evolution, integration, conceptual modeling, MDA

Contents

Introduction

Recently, XML [126] has become a corner stone of many information systems. It is a de facto standard for data exchange (i.e. Web services [35]) and it is also a popular data model in databases [23]. XML is a metalanguage that allows to specify several XML formats that suit our needs. To prevent chaos, the XML formats are specified by XML schemas expressed in an XML schema language like DTD [126] or XML Schema [128]. The schema can be used, for example, each time an XML document described by the schema is processed to validate the document. Usually, a system does not use only one XML schema, but a set of different XML schemas, each in a particular logical execution part. We can, therefore, speak of a *family of XML schemas*. As the number of possible usages of XML grows, so does the need of easy management of families of XML schemas.

Having a system which exploits a family of XML schemas, we face the problem of *XML schema evolution*. The XML schemas may need to be evolved whenever user requirements or surrounding environment changes. A single change may influence zero or more XML schemas. Without a proper technique, we have to identify the XML schemas affected by the change manually and ensure that they are evolved coherently with each other. When the XML schemas have already been deployed, in the run-time environment there are also XML documents which might became invalid and need to be, therefore, modified appropriately.

In [86], a conceptual model for XML is introduced. It is a technique based on the Model-Driven Development (MDD) [78] methodology. It considers modeling the XML schemas at two MDD levels – *platform-independent* and *platform-specific*. First, the whole application data domain is modeled independently of the XML schemas in the form of a platform-independent schema. Then, each XML schema in the family is designed in the form of a platform-specific schema which is mapped to the platform-independent schema. It may be then automatically translated to an expression in a selected XML schema language, e.g. XSD (XML Schema Definition) [128] or RELAX NG [29]. In this thesis, we mainly focus on two aspects of conceptual modeling for XML.

XML schema evolution First is the coherent *evolution of XML schemas* according to the changing requirements. The mappings of platform-specific schemas to platform-independent schema naturally support evolution management. A change is explicitly expressed as a change to the platform-independent schema or one of the platform-specific schemas. The mappings allow us to propagate the change between platform-independent and platform-specific levels and evolve the whole family of XML schemas coherently. The challenge addressed in this thesis is how to provide a formal background for this kind of operations in the multi-level conceptual model. The formal background is used for evaluation of the proposed set of operations in regard to its minimality, completeness and correctness. Also, we formally describe the mechanism of coherent propagation of changes between the platform-independent and platform-specific level of the conceptual model. This brings the advantage of assurance that when a designer uses the conceptual model and executes operations, no modeled semantics is lost without his knowledge, which is crucial especially when there is a larger number of

XML schemas involved. In addition, complex operations can be created simply by concatenation of the proposed operations without the need to be concerned about correct propagation as it is ensured by the operations proposed in this thesis.

XML schema integration The problem with conceptual modeling for XML is that it is relatively new and it requires that the system designer starts with the platform-independent conceptual model first. Once the designer has the platform-independent level of the model in place, XML formats and, consequently, schemas can be derived from it easily. Unfortunately, in most cases the situation is that designers have their XML schemas designed without the conceptual model, if they have any schemas at all. Therefore, it is paramount that we enable such designers to create the conceptual model based on their pre-existing XML schemas. The same technique can be used when a designer needs to augment an existing conceptual model by including a new part of reality that may be present in an XML schema that was not created using the conceptual model. We call the process *XML schema integration* and it is the second main topic of this thesis.

Additional contributions In addition to handling XML schema evolution and integration using the conceptual model for XML we provide additional usability enhancements for the coneptual model itself. In a supervised master theses we proposed a technique to use the Schematron XML schema language in the conceptual model for XML, which opens an entire research area of using rule-based XML schema languages to describe structural constraints. Moreover, we proposed a technique for an easy transformation of XML data to RDF using ontologies and the conceptual model for XML. Finally, we described our tool which we used for implementation of the proposed approaches called *eXolutio*.

Outline

The rest of the thesis is structured in chapters according to individual contributions as published in our research papers. The order of the chapters is basically chronological, divided in the three problem groups desribed earlier. First group of contributions is *XML schema integration*, which is based on the original conceptual model for XML as described in [86] and Chapter 3. Then we redefine the conceptual model with our current version from [96] in Chapter 6 and we present our contributions in *XML schema evolution*, which are based on this updated conceptual model version. Finally, we present the *additional contributions* to the conceptual modeling for XML in general, which show possible use cases for the proposed techniques.

Most of the content in this thesis was primarily authored by the author of this thesis and supervised by the thesis supervisor Mgr. Martin Nečaský, Ph.D. except for chapters 6,8 and 11. Those were primarily authored by the thesis supervisor with contributions by the author of the thesis and other colleagues from our research group (see citations for complete author lists). However, they are included in

this thesis for completeness and to provide the reader with the related context.

We start with our initial publication of our research intent in both main areas in Chapter 1 as published in the Ph.D. workshop paper *Integration and Evolution of XML Data via Common Data Model* [58], in which we describe the further directions of our research. Then there are three groups of contributions: *XML schema integration* papers, *XML schema evolution* papers and *additional contributions* to conceptual modeling for XML in general.

In the *XML schema integration* group are the following contributions:

- In Chapter 2 we present a related work survey in the area of XML schema integration. Published as a conference paper *Reverse-engineering of XML Schemas: A Survey* [59].

- In Chapter 3 we present our core contribution in the area of XML schema integration, which consists of definition and evaluation of methods for semi-automatic construction of mappings from XML schemas to the platform-independent part of the conceptual model for XML. Published as a conference paper *Semi-automatic Integration of Web Service Interfaces* [60].

- In Chapter 4 we further abstract our approach to XML schema integration and we introduce a framework for XML schema integration methods. Published as a workshop paper *A Framework for XML Schema Integration via Conceptual Model* [57].

- In Chapter 5 we extend the XML schema integration method to take into consideration reusable schema parts in the form of structural representants in our conceptual model for XML. Published as a conference paper *XML Schema Integration with Reusable Schema Parts* [56].

In the *XML schema evolution* group are the following contributions:

- In Chapter 6 we formally describe the updated version of our conceptual model for XML which was considerably refined as part of our research in our impacted journal paper *When conceptual model meets grammar: A dual approach to XML data modeling* [96]. In the paper, the conceptual model is presented from two perspectives - grammatical based on regular tree grammars and conceptual, which is the base of this thesis and is described in the chapter. To maintain consistence of formalisms throughout the thesis, we present our conceptual model definition from [89].

- In Chapter 7 we describe our initial results in the area as we identify basic operations and their granularity needed for formal definition and description of the problem. Published as a workshop paper *Model-Driven Approach to XML Schema Evolution* [95].

- In Chapter 8 we present our contribution to the area of XML schema evolution in a form of a related work survey and a formal definition of atomic and composite operations and their propagation between the abstraction levels of the conceptual model for XML. Published as an impacted journal paper *Evolution and Change Management of XML-based Systems* [89].

- In Chapter 9 we preset the first part of our main contribution to XML schema evolution by extending the refined conceptual model for XML with modeling of inheritance relations. Published as a conference paper *On Inheritance in Conceptual Modeling for XML* [63].
- In Chapter 10 we present the second part of our main contribution to XML schema evolution, which is an extension to the formal model of atomic and composite operations considering our inheritance extension to the conceptual model for XML. Published as a conference paper *Formal Evolution of XML Schemas with Inheritance* [62]. [63].

The *additional contributions* to conceptual modeling for XML in general are:

- In Chapter 11 we present a case report in which we identify methods for measuring quality of XML schemas. Published as a conference paper *When Theory Meets Practice: A Case Report on Conceptual Modeling for XML* [88].
- In Chapter 12 we present a description of *eXolutio*, which is our second generation implementation of our proposed methods (first generation was called XCase and was published as a workshop paper [54]). Published as a conference paper *eXolutio: Tool for XML Schema and Data Management* [55].
- In Chapter 13 we show how to integrate conceptual modeling for XML and semantic web technologies. Published as a workshop paper *Generating Lowering and Lifting Schema Mappings for Semantic Web Services* [61].
- In Chapter 14 we present a method for generating Schematron [49] schemas using the conceptual modeling for XML. Published as a conference paper *Using Schematron as Schema Language in Conceptual Modeling for XML* [15].

1. Aims of the Thesis

In this chapter we define our research topics which we then follow in the rest of the thesis. The contents of this chapter was published as a Ph.D. workshop paper *Integration and Evolution of XML Data via Common Data Model*[1] [58] at the EDBT/ICDT 2010 conference.

1.1 Research topics

There are two main research topics in this doctoral thesis. The first one is evolution of a family of XML schemas that describe a common data model from different views as different applications need only certain subsets of the data. The second one is integration of various XML data sources into one easily manageable system. Those two areas of research are brought closely together by our approach using conceptual modeling as both can be addressed by introducing a common conceptual model to the system. From this model, all the XML schemas used in the system can be derived by the user, creating mappings between the elements of the schemas and the elements of the conceptual model. As any XML data source can be described by a schema, the method works here as well. A disadvantage of this approach is the fact that we first need the conceptual model and then we can derive the XML schemas. Because of that, the second part of our doctoral work is focused on reverse engineering of the mappings between an XML schema and the conceptual model, which can also mean that we need to create the model when there is none.

1.1.1 XML schemas evolution

This research topic addresses the following problem. Let us have a company providing e.g. web services (or generally using various XML messages). Those messages will probably contain some subsets of the company's data depending on the purpose of each message. As an example of the problem of XML schema evolution, let us have a company that receives orders and let us focus on a part of the system that processes purchases as seen on Figure 1.1. Let the messages used in the process be XML messages each with a separate XML schema.

The XML schemas are visualized in Figures 1.2. The figures show graphical representations of XML schemas called PSMs (platform specific models), explained later in Section 1.2. Simply put, they model the elements and attributes of the XML schema and its structure. The operations that use these schemas are:

1. A client process sends an order to the *aggregator*. The order contains a list of items purchased and an address to which to send the order. (Figure 1.2(a))
2. The *aggregator* sends the list of items to the *inventory*. (Figure 1.2(b))
3. The *inventory* checks if the items are available and makes a reservation. Then it sends a response containing a reservation number to the *aggregator*

[1]`http://portal.acm.org/citation.cfm?id=1754239.1754283`

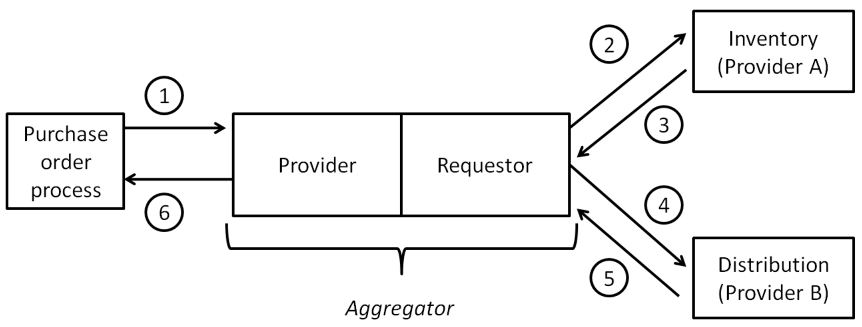
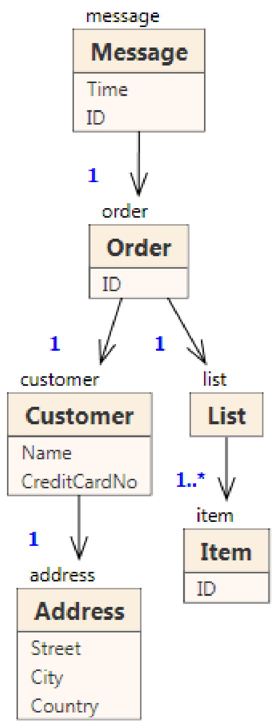


Figure 1.1: Example of a system with 6 XML schemas

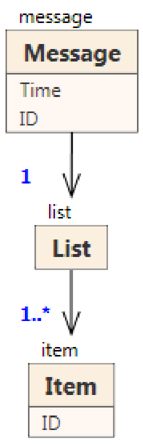
4. If the items are in stock the *aggregator* sends a message containing the reservation number and the address to the *distribution*. (Figure 1.2(c))

5. *Distribution* registers the order and sends a confirmation to the *aggregator*.

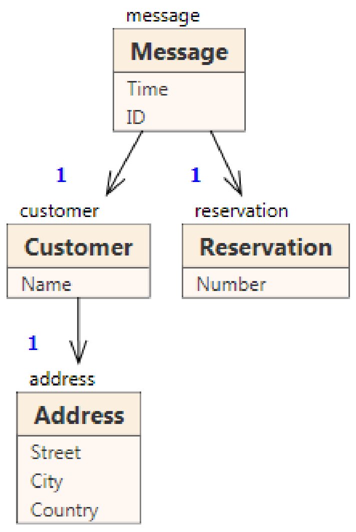
6. The *aggregator* sends a confirmation or an "out-of-stock" message back to the client.



(a) Message 1



(b) Message 2



(c) Message 3

Figure 1.2: Example schemas of messages

This simple example demonstrates the problems we solve in the doctoral thesis. All the messages in the process deal with the same data, although some of them use only a subset of the data. We could say that the XML schemas specify *views* on the data. To be specific, message 1 contains a list of items and an address, message 2 contains only the list of items, but not the address, as it is not needed in the *inventory.* It provides a simplified view on the data that best suits the purpose of the message. These XML schemas need to be designed and maintained, which today is done mostly manually. For example, to change the representation of a customer's name from one value *name* to a pair of values *forename* and *surname*, a domain expert would have to identify the XML formats containing customer names (Messages 1 and 3 in our case) and perform the change in each format manually. Remember that our example is only a small part of a company's infrastructure, so in real life, it could be dozens of schemas instead of two in our case and also the changes made to them can be far more complicated.

1.1.2 Integration of XML data sources

This topic is closely related to the previous one, when viewed from the perspective of mapping different data sources to the common conceptual model. As an example, let us have a company that uses online advertisement through various providers. All the advertisement providers will probably require similar data for the ad service, and for simplicity, let us assume that they use a web service interface (and therefore XML messages) for the management of the ads. Now, we want to add an advertisement to all the ad services at once. Today, this would be done by creating a universal XML message for addition of an advertisement and a set of XSLT scripts [52], one for each ad service. And here we are faced again with the same problem: management of the set of these scripts. What if my data representation of an advertisement changes? We would need to go through every XSLT script manually and adjust it. This situation can again be solved by applying a conceptual model, from which we can derive an XML schema for each ad service and through the mappings of XML schemas to the conceptual model we can manage them all from one place. The two questions are:

1. How do we create the common conceptual model with the knowledge given to us by the XML schemas?

2. How do we create those mappings, when we already have the conceptual model?

These questions are addressed by the second part of our work where we deal with reverse engineering of the conceptual model from the XML schemas, during which the mappings are created, assuring that we can manage the system from one place.

1.2 Our approach idea

In our work, we view the problem of XML evolution and integration as having five levels as it can be seen in Figure 1.3. Components from all levels are connected with components from one level above and one level below. This allows for a change anywhere in the system to be propagated through these connections to all affected places automatically. The levels in the figure can be divided into two groups. The *platform-independent* and *platform-specific* levels are called

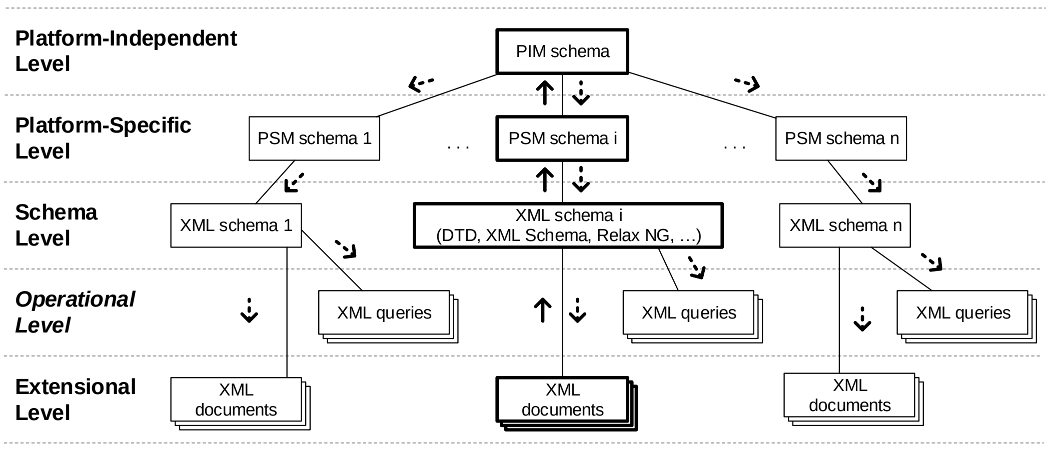


Figure 1.3: Five level XML evolution architecture

conceptual levels. The remaining three levels, the *schema*, *operational* and *extensional*, containing the actual files present in the system, are called *logical levels*. In our work, we focus on the conceptual levels, as they are paramount for the XML data evolution and integration. We already have an automatic algorithm translating our PSMs to XML Schema and we are working on algorithms for automatic translation to other XML schema languages, such as DTD or Relax NG [29]. The following sections describe these levels in detail.

1.2.1 Conceptual levels

The two conceptual levels are platform independent and platform specific. They are based on MDA - *Model-Driven Architecture* [78]. It is a general approach to modeling software systems and can be profitably applied to data modeling as well. MDA distinguishes several types of models that are used for modeling at different levels of abstraction. For our work, two types of models are important.

A *Platform-Independent Model* (PIM, conceptual model) allows modeling data at the conceptual level. A PIM is abstracted from a representation of the data in concrete data models such as relational or XML. An example PIM can be seen in Figure 1.4.

A *Platform-Specific Model* (PSM) models how the data is represented in a target data model (such as XML). For each target data model, we need a special PSM that is able to capture its implementation details. A PSM then models a representation of the problem domain in this particular target data model, it provides a mapping between the conceptual model and a target data model schema. Examples of PSMs can be seen in Figure 1.2.

Because we want to model XML representations of data, we need a conceptual model based on MDA, that would allow us to model data on the PIM level as well as various XML formats on the PSM level. In the following section (1.2.1), we will describe a conceptual model, which is a proper model for our work.

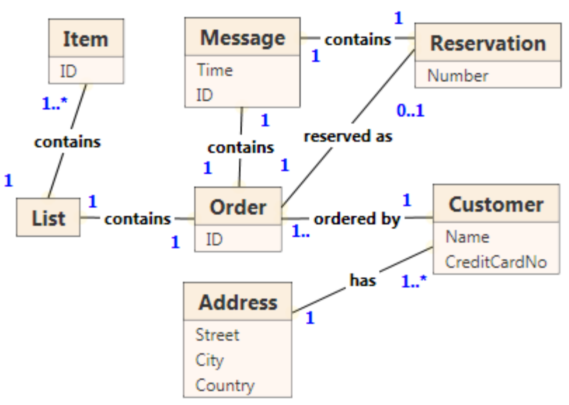


Figure 1.4: Platform independent model

The conceptual model for XML

The conceptual model that we use is described in detail in [86] and then redefined with better formal background in Chapter 6. It utilizes UML class diagrams to apply MDA to model XML data on two levels: PIM and PSM. A PIM can be a description of a company domain, which usually already exists. One PSM models how elements of PIM are mapped to XML and in fact describes a specific type of an XML message used in the company's IT infrastructure. While the PIM is usually only one, there can be any number of PSMs representing different *views* on the same company data as used under different circumstances as can be seen in the introduction.

The main feature of this conceptual model is that all the PSM components are formally interrelated with the components of the PIM level. This allows for describing semantics of the PSM components by components from the PIM level. Using this, a software implementing this conceptual model can maintain connections between corresponding PIM and PSM components. These connections enable a change in a PIM component (class, association, etc.) to be propagated to all the affected PSM components in PSMs[2]. Even more interestingly, a change in a PSM component can be propagated to the PIM level, where all the other derived[3] PSM components can be discovered and updated as well. In other words, when someone decides that a customer's name is to be represented by two values (first and last name) instead of one string, it is possible to automatically find all relevant usages of this value and change all the affected XML schemas and the conceptual model at the same time. So far, these connections between PIM and PSM components are created by the user during the creation of PSMs from the PIM. We have already implemented this approach in our open-source tool called *XCase* described in Section 1.2.1. Part of our work is focused on an semi-automatic algorithm which would ease the process of creating those connections the other way around, from an existing PSM (which we can get automatically from an XML schema) to the PIM, by suggesting probable matches in the PIM for each PSM component.

[2]A PSM directly represents an XML schema, which can be written in any one of XML schema languages, such as DTD, XML Schema, Relax NG, etc.

[3]PSM components connected to the PIM components are *derived* from the PIM components.

XCase - A Tool for XML Data Modeling

XCase[4] is a tool for conceptual XML data modeling implementing the original version of the described conceptual model. Since this was the first tool for conceptual modeling of XML using this model, its main purpose was to examine its possibilities as well as conceptual modeling for XML in general. It provides an automatic propagation of changes from PIM to PSMs and vice versa. This means the user can model his PIM and from there, derive corresponding PSMs, which can be later automatically translated to XML Schema. When, for example, a change in the PIM occurs, it is automatically propagated to all affected PSMs and therefore, XML schemas.

Currently, the conceptual model for XML is described formally in Chapter 6 and its current implementation is called eXolutio and is described in Chapter 12.

1.2.2 Logical levels

Now that the conceptual levels of the XML evolution architecture have been presented, the logical levels will be described briefly as well as the propagation of changes from the conceptual levels to the logical levels.

Schema (a.k.a. logical) level

The schema level contains the actual XML schemas written in an XML schema language such as DTD, XML Schema, Relax NG, etc. Those can be automatically generated from PSMs [86, p. 109-128], which are on the platform-specific conceptual level. They can be generated every time the conceptual model evolves.

Operational level

XML can be used as means to exchange data, but it also can be used to store the data in a database. When we have a database containing XML data, we also use queries to access it. The queries are dependent on the XML schemas describing the data, therefore, need to be changed when the schemas change. The operational level makes it possible for the changes made during the XML schema evolution process to be propagated further, keeping the queries consistent. The propagation of changes to this level is part of our future work.

Extensional level

This level contains the actual XML documents. When the XML schemas evolve, the documents they describe may become invalid in the process. The extensional level makes it possible for a change in the XML schemas to be propagated (i.e. through an XSLT transformation) to the actual documents, changing them to be compatible with the evolved XML schemas. The propagation of changes to this level is also a part of our future work.

The conceptual model on its own is today only useful when taken into account during the design of the data model. However, a common situation is that we have XML data sources not described by this model and we want to use it for their

[4]`http://www.ksi.mff.cuni.cz/xcase`

maintenance and development anyway (XML data sources integration). This means to reverse-engineer the XML schemas describing those data sources and create their mapping to the conceptual model. The problem of XML schema evolution and reverse-engineering was discussed in my master thesis, and my doctoral work will continue in this effort and extend the results achieved there.

2. Reverse-engineering of XML Schemas: A Survey

As approaches to conceptual modeling of XML data become more popular, a need arises to reverse-engineer existing schemas to the conceptual models. They make the management of XML schemas easier as well as provide means for accomplishing integration of various XML data sources. Some methods for reverse-engineering of XML schemas have been proposed and in this chapter, they are compared using various criteria such as used XML schema languages, level of user involvement, number of XML schemas that can be covered by the conceptual model or support for consecutive XML schema evolution. They are also evaluated according to their potential to be used as parts of a system for management, evolution and integration of XML as a whole.

In this chapter, we compare and evaluate approaches for reverse-engineering of XML schemas according to various criteria, including their usefulness as a method that can be integrated into a system for management of evolution of XML schemas. Therefore, this chapter covers the related work for the *XML schema integration* part of this thesis.

The contents of this chapter was published as a conference paper *Reverse-engineering of XML Schemas: A Survey*[1] [59] in Dateso 2010 Annual International Workshop on DAtabases, TExts, Specifications and Objects (DATESO 2010).

2.1 Introduction

As the number of possible usages of XML grows, so does the need of easy management of large numbers of XML data sources and their integration. There we can use conceptual modeling of XML data. It allows a domain expert to model the problem domain independently of the implementation (XML in our case) and then create corresponding XML schemas, which are used to describe a structure of XML documents. A common situation today is that a company uses several XML formats for various purposes and has these formats described by an XML schema. To ease the process of managing those formats and schemas as they evolve in time, the company can use a conceptual model such as [4, 11, 69, 73, 86, 113], to which the schemas would be connected. A problem usually arises when there is a new format that should be connected to the conceptual model, as most of the conceptual models do not support this operation well enough. During the process of connecting the new format to the conceptual model, the model may need to be extended, if the format contains a concept that was not covered by the model. A special case of this problem is, when the company does not have a conceptual model at all, and wants to create it from the schemas which they already have (they extend an empty model).

Therefore, an important aspect of the approaches used for conceptual modeling of XML data is if and how we can create the model from existing XML

[1] `http://ceur-ws.org/Vol-567/paper19.pdf`

schemas (or connect a new schema to an existing model) and once we have it, if we can use it for the management of evolution of our set of schemas. The process of creating a conceptual model from existing XML schemas is what we call *Reverse-engineering of XML schemas*.

Outline The rest of this chapter is structured as follows. In section 2.2, an introduction to the frequently used techniques in this area is given. In section 2.3 we introduce our framework for evolution and integration of XML schemas, against which the approaches will be evaluated. Section 2.4 contains a description of our comparison criteria. In section 2.5, we describe approaches to the problem, which reverse-engineer XML schemas to various user-friendlier models. In section 2.6, approaches to reverse-engineering to ontologies are described. In section 2.7 we summarize our findings and section 2.8 concludes.

2.2 Terms

In this section we provide an introduction to basic techniques used widely in the area of reverse-engineering of XML schemas.

2.2.1 XML schemas

In this paper, by *XML schema language* we mean one of the XML schema languages such as DTD [126], XML Schema [128], Relax NG [29], Schematron [49] etc. We state this because sometimes an XML schema gets confused with the actual XML Schema language.

2.2.2 Model-Driven Architecture

Model-Driven Architecture (MDA) [78] is a general approach to modeling software systems and can be profitably applied to data modeling as well. MDA distinguishes several types of models that are used for modeling at different levels of abstraction. For this paper, two types of models are important. A *Platform-Independent Model* (PIM) allows modeling data at the conceptual level. A PIM diagram is abstracted from a representation of the data in concrete data models such as relational or XML. A *Platform-Specific Model* (PSM) models how the data is represented in a target data model. For each target data model (such as XML), we need a special PSM that is able to capture its implementation details. A PSM diagram then models a representation of the problem domain in this particular target data model, it provides a mapping between the conceptual diagram and a target data model schema.

2.2.3 UML class diagrams

A large number of approaches to reverse-engineering use UML class diagrams as a PIM. Basically, it consists of *classes* representing concepts, *associations* representing relations and *attributes* of classes, representing properties of the concepts. For a more detailed description see [101, 102].

2.2.4 Schema matching

Most of the reverse engineering approaches use some methods of schema matching. They include string comparisons, data type compatibility measurements, structural similarity measurements and linguistic resources like thesauri and dictionaries. These methods are surveyed in detail in [121, 45]. There is one major difference between XML schema matching and reverse-engineering of XML schemas to conceptual models. XML schema matching usually works with two different XML schemas (written in XML schema languages) and the goal is to find mappings of components of one schema to the components of the second schema. On the other hand, reverse-engineering of XML schemas works with one XML schema and optionally a conceptual model. The goal is either to create the conceptual model when there is none, or to find appropriate mappings of the XML schema components to the components of the model, which can be written in e.g. UML, and therefore is of a whole different type.

2.3 Framework for evolution and integration of XML schemas

In this section, we introduce our framework for evolution and integration of XML schemas. It comprises six levels, each representing a different view of an XML system and its evolution. The framework is depicted in Figure 2.1. The lowest level, called *extensional level*, represents XML documents. Its parent level, called *operational level*, represents operations over XML documents, i.e. XML queries. The level above is called *schema level* and represents XML schemas that describe the structure of the XML documents.

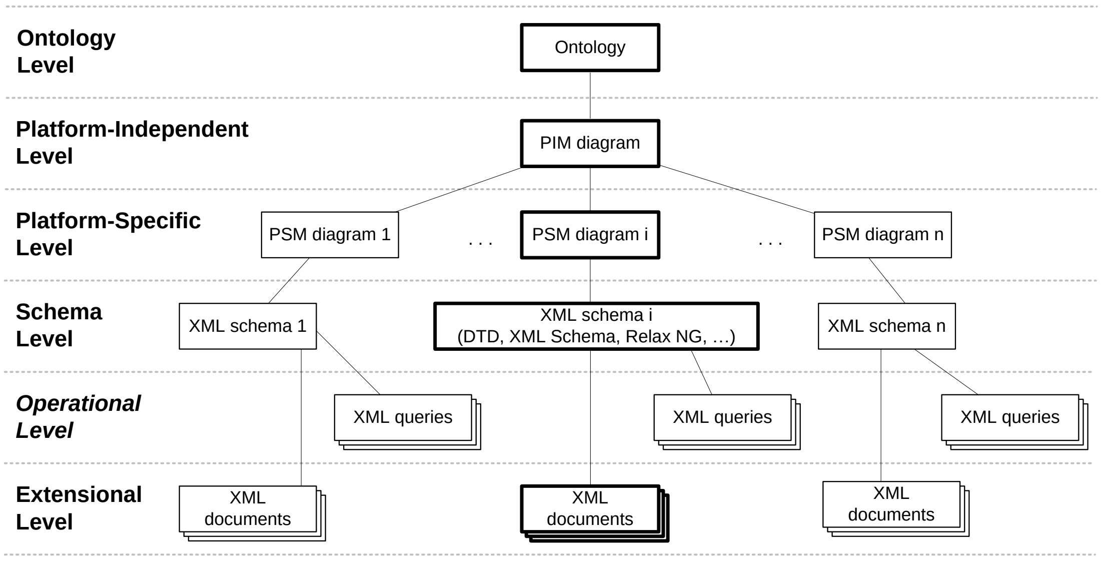


Figure 2.1: Six-level XML evolution and integration framework

The platform-independent and platform-specific levels follow MDA [78] which is based on modeling the problem domain on different levels of abstraction. The *platform-independent* level represents the whole problem domain. It consists of a conceptual model that specifies the problem domain independently of its representation in the XML formats below. We call the conceptual model a *platform-independent model* (*PIM*) of the problem domain. The level below, called *platform-specific* level, represents mappings of the problem domain to particular XML formats. For each XML format it comprises a model of mapping of a selected part of the PIM to XML element and attribute declarations. We call this model *platform-specific model* (*PSM*) of the selected XML format. Recently, a number of approaches translating XML schemas to an ontology have appeared. The ontology level is the topmost in our framework.

In our framework, components in documents on individual levels can be formally binded with components in documents on the neighboring levels. These bindings can then be used for evolution of the conceptual model, XML schemas, XML documents and queries. They provide means for automatic detection of all the places on all the levels where a single change has an impact.

2.4 Comparison criteria

We will firstly introduce several criteria which we will later use to compare current approaches to reverse-engineering of XML schemas. In particular, we will focus on the following criteria:

1. Target model - on what level of our framework does the target model of the reverse-engineering method belong. This criterion is important, because lots of methods only visualize the XML schema in another model (e.g. UML class diagrams) and therefore their target is on the *platform-specific* level (it is a PSM in our framework). A true conceptual model on the *platform-independent* level (a PIM in our framework) should be independent of the target implementation completely. If it is a conceptual model bent for a specific implementation, it is in fact a PSM. Recently, some approaches to mapping an XML schema directly to an ontology (the top level of our framework) have appeared. This criterion distinguishes among these three types of target models.

2. Number of schemas supported by the model - whether the method is limited to only one schema, or whether it can reverse-engineer multiple schemas to one model. This is also very important, because to be able to manage a set of XML schemas, it is not enough to have a separate model for each schema. We need all the schemas to be related to one model.

3. XML schema languages supported - These can be DTD, XML Schema, Relax NG, Schematron, etc. As can be seen from our framework, the conceptual model can be and should be independent of the actual XML schema language used on the *schema* level, because the target data model (which a PSM should represent) is XML itself, not a specific XML schema language.

4. Mapping to an existing model - whether the method can map a schema to an already existing model or whether it can only generate a new model. This criterion is paramount for evaluating the possibility of integrating an approach to a bigger system for evolution and integration of XML schemas. If it only can create a new model for each input, it cannot be used if we

already have the model and only want to add a new XML schema to the system.

5. Level of user involvement - methods can be automatic or semi-automatic (relying on human intervention). It is impossible to infer a conceptual diagram automatically, as so far only a human can determine if two objects represent the same concept. And because we need an exact and reliable match, if we want to use an approach as a part of a system for integration of XML data, we can not rely on an automatic translation (mapping) and we need the user to at least confirm a match.

6. Evolution support - whether the approach is a part of a system that also supports evolving the schema once it is integrated into the system (when the system changes). This criterion indicates, whether the method is developed by itself, or if it already is a part of a system that also helps with the evolution of the mapped schemas (e.g. by preserving the bindings between levels of our framework)

2.5 Mapping to user-friendly models

In this section we evaluate different approaches to reverse-engineering of XML schemas to various more user-friendly models.

2.5.1 Yang Weidong et al.

In [133], there is an algorithm for automatic generation of UML class diagrams (PIMs in our framework) from DTDs according to MDA using a DTD graph as a PSM. The authors also claim that they can generate UML class diagrams from XSDs, but the prototype implementation is not freely available, so it cannot be verified. The main drawback of this approach is that it only serves as an automatic translator from DTD to UML meant to make the schema more understandable to people who do not know DTD or XML Schema.

This approach does not support mapping of multiple XML schemas to the PIM and it does not preserve any mappings between the PIM and the PSM nor between the PSM and the DTD. It is automatic, limited to DTD only and it cannot map a schema to an existing model. The bright side is that it actually uses both PIM and PSM correctly.

2.5.2 Mikael R. Jensen et al.

In [51], another method of automatic conversion of DTDs to UML class diagrams is presented. Again, it is meant for easier browsing of XML data available on the Internet and to ease the work of a data integrator. In contrast with the previous method, the UML diagrams reflect the structure of the DTD an therefore are only on the PSM level.

This approach does not support mapping of multiple XML schemas and it does not preserve any mappings between the PSM and the DTD. It is automatic, limited to DTD only and it cannot map a schema to an existing model.

2.5.3 DIXSE framework

In [116], a semi-automatic method of deriving a semantic model from several DTDs is presented. By default, for each element of every DTD a new element is created in the model. This process is done automatically. If the user wants to create a more meaningful model (e.g. wants all *Address* elements to be mapped to one element of the model), a rule written in their DIXml language extending the element in the DTD must be created manually. The model is a PSM as it still preserves the DTD structure. It uses the Telos [83] metamodeling language.

This approach supports semi-automatic mapping of multiple schemas to a conceptual model, but it does not preserve any mappings between the PSM and the DTDs and it is limited to DTD only. It can map a new DTD to an existing model.

2.5.4 Xyleme

In [115], a project called Xyleme is described. Its focus is to provide a unified view of a large number of heterogeneous XML documents described by DTDs. This enables the user to perform queries on one unified model (called an abstract DTD), to which all the other DTDs describing the XML documents are mapped automatically. The methods for discovery of mappings are mainly language based (thesauri, discovery of synonyms, abbreviations, etc.). A semi-automatic prototype implementation called SAMAG was used to evaluate the approach. In SAMAG, a user needs to validate each syntactic relationship detected.

This approach supports semi-automatic mapping of multiple schemas to a model which is a PSM. It preserves no mapping between the PSM and the DTDs. It is limited to DTD only.

2.5.5 XTM - XML Tree Model

In [42], a conceptual model for XML called XTM - XML Tree Model is proposed, including an algorithm for reverse-engineering of XML Schema into XTM. It also has a strong theoretical background. However, it is still only a PSM in our framework.

This approach automatically visualizes one schema at a time, is limited to XML Schema and does not maintain any mappings between the PSM and the schemas.

2.5.6 Nečaský

In [87], a complex approach to the process of reverse engineering of XML schemas to a conceptual model is presented. The model follows MDA as it uses UML class diagrams as a PIM and their extension as PSMs. It is further described in [86]. Because the model has two levels, the process is divided into two parts. The first part is an automatic translation of an XML schema to a PSM. The PSMs are, however, independent of any specific XML schema language; the approach is presented using XML Schema. The second part is a semi-automatic algorithm for the reconstruction of mappings between the PSM and a PIM, but it has some drawbacks. The most problematic one is the computational cost which is up to m^n, where m is a maximum number of outgoing PIM associations from one PIM class and n is the number of PIM classes in the model. Therefore in practice, the algorithm will not work if the PIM diagram contains a bigger number of associations. Nevertheless, the algorithm uses maximum of information that we can get from a PSM and thus can offer the best results. An implementation in an experimental stage is available in the development version of XCase [54], which is a tool implementing the conceptual model and its evolution.

This approach supports semi-automatic reverse-engineering to PSMs and to a PIM. It supports multiple schemas, is independent of any specific XML schema language and it can also map to an existing conceptual model. It maintains mappings between the PIM and the PSM and therefore support further schema evolution.

2.6 Approaches to mapping to ontologies

Recently, a number of methods of reverse-engineering of XML schemas to ontologies have appeared. The main difference between a PIM and an ontology is that ontologies are more expressive, as the relations they capture can be more complex and they may even involve logical formulae. A frequent language for ontologies is OWL [132].

2.6.1 Canonic Conceptual Models (CCMs)

In [40], a method for integration of XML schemas into an ontology is presented. At first, each input DTD is semi-automatically transformed into a so called CCM - Canonic Conceptual Model. It combines the ER [26] and ORM [46] models. A default transformation is made and a user is then allowed to make adjustments where needed. The CCMs are PSMs in our framework. Then, each CCM is integrated (again semi-automatically - user has to verify/adjust) to form the final ontology. The mappings between the individual CCM components and the components of the ontology are preserved. Although the authors call it an ontology, it is in fact more of a conceptual diagram - a PIM in our framework, because it still is a CCM, only independent of the XML structure. This method only provides a unified view of the integrated data so far. No implementation was mentioned.

This approach supports semi-automatic mapping of multiple DTDs to corresponding PSMs and to a PIM. It is restricted to DTD only. It preserves mappings between PSMs and a PIM.

2.6.2 Xiao et al.

In [135], an algorithm is proposed to match given XML document elements to given ontology concepts to achieve an integrated view of multiple XML documents, when matched to the same ontology. It is based on automatic structural matching of a DTD tree to an ontology tree. However, a precondition is that a domain expert has provided a table of synonyms, i.e. a list of semantically matching strings from the DTD and from the ontology (which seems to be a strong precondition).

This in fact semi-automatic approach maps multiple DTDs, it is limited to DTD only and it does not preserve any mappings between the DTDs and the ontology. It can only map to an existing ontology.

2.6.3 Bedini et al.

In [13], a general architecture of building ontologies from XML schemas is presented. However, no specific methods are suggested, only a sequence of tasks that a tool for ontology building should follow and also a set of rules according to which concepts and their relations can be extracted from a XML Schema. The general architecture takes the need for consecutive schema evolution into account. A semi-automatic prototype implementation called Janus is presented briefly. It requires human assistance for merging of similar concepts and it does not preserve any mappings between the schemas and the ontology.

This approach is semi-automatic, it is limited to XML schema and it does not preserve any mappings between the schemas and the ontology. It also only creates new ontologies.

2.6.4 DTD2OWL

In [129], a method of automatic translation of DTD to OWL ontology is suggested. In addition, this method transforms the actual XML documents into OWL individuals. The authors suggest that the whole web should be transformed this way. This method is pure translation of one DTD to one OWL ontology with no support for schema evolution nor conceptual modeling.

This approach is automatic, it is limited to DTD and it can only map one DTD to one new ontology, not preserving any mappings.

2.7 Summary

In this section, a brief summary of evaluated approaches and the comparison criteria is given.

	Model	**Schemas**	**Languages**	**Automatic**	**Maps to**	**Evolution**
2.5.1	PSM	One	DTD	Yes	New	No
2.5.2	PSM	One	DTD	Yes	New	No
2.5.3	PSM	Multiple	DTD	No	New, Existing	No
2.5.4	PSM	Multiple	DTD	Yes	New, Existing	No
2.5.5	PSM	One	XSD	Yes	New	No
2.5.6	PIM	Multiple	Any[a]	No	New, Existing	Yes
2.6.1	PIM	Multiple	DTD	No	New, Existing	Yes
2.6.2	O	Multiple	DTD	No	Existing	No
2.6.3	O	Multiple	XSD	No	New	No
2.6.4	O	One	DTD	Yes	New	No

[a]This method is not limited to any XML schema language. Currently implemented for XML Schema

Table 2.1: Overview of approaches according to various criteria

Let us review the comparison criteria used (the columns in Table 2.1 correspond to them):

1. Whether the target of the approach is a PIM, PSM or an ontology
2. Whether the approach is limited to only one schema or whether it can handle multiple schemas
3. Which XML schema languages can the approach handle
4. Whether the method is a pure automatic translation or whether the process is semi-automatic - a user is involved
5. Whether the method creates a new target model or whether it can use an existing one
6. Whether the method preserves mappings between the schemas and the target model, which can be used for consecutive schema evolution

The best suitable method for our intention of creating a system for management of XML schema evolution and integration is 2.5.6. However, it has some issues with computational complexity, which need to be resolved before its implementation.

2.8 Conclusion

In this chapter, we have compared and evaluated several approaches for reverse-engineering of XML schemas according to given comparison criteria. Among them, only one was well suited for being a part of a larger system for evolution and integration of XML schemas. Also, this survey showed a severe lack of support for newer XML schema languages like Relax NG or Schematron in the area of conceptual modeling of XML and reverse-engineering of XML schemas.

3. Semi-automatic Integration of Web Service Interfaces

Modern information systems may exploit numerous web services for communication. Each web service may exploit its own XML format for data representation which causes problems with their integration and evolution. Manual integration and management of evolution of the XML formats may be very hard. In this chapter, we present a novel method which exploits a conceptual schema. We introduce an algorithm which helps a domain expert to map the XML formats to the conceptual schema. It measures similarities between the XML formats and the schema and adjusts them on the base of the input from the expert. The result is a precise mapping. The schema then integrates the XML formats and facilitates their evolution - a change can be made only once in the schema and propagated to the XML formats.

The contents of this chapter is published as a conference paper *Semi-automatic Integration of Web Service Interfaces*[1] [60] in 2010 IEEE International Conference on Web Services (ICWS 2010).

3.1 Introduction

We aim at the problem of integration of XML schemas by lifting them to a common conceptual schema. In our previous work (see Chapter 1 and [58]) and [86] a framework for XML schema integration and evolution is introduced. It supposes a set of XML schemas that are conceptually related to the same problem domain. As a problem domain, we can consider, e.g., purchasing products. Sample XML schemas may be XML schemas for purchase orders, product catalogue, customer detail, etc. The central part of the framework is a conceptual schema of the problem domain. Each XML schema is then mapped to the conceptual schema. In other words, the conceptual schema integrates the XML schemas. We then exploit the mappings to evolve the XML schemas when a change occurs. Simply speaking, the change is made only once at the conceptual level and then propagated to the affected XML schemas.

Contributions In practice, a conceptual schema and XML schemas exist separately, i.e. there are no mappings between both levels. This disallows to exploit the integration and evolution capabilities of our framework. In our work [86], we have introduced a method for deriving required XML schemas from the conceptual schema. However, it does not consider an existing XML schema that needs to be somehow mapped to the conceptual schema. In this work, we introduce a reversed method which allows to (1) correctly map a supplied XML schema to the conceptual schema, and (2) adapt the conceptual schema if a particular part of the XML schema can not be mapped.

Our aim is not to develop new methods for measuring schema similarities. These methods have been already intensively studied in the literature. Instead, we exploit the existing ones and combine them together. For this, we provide

[1] `http://doi.ieeecomputersociety.org/10.1109/ICWS.2010.28`

an algorithm skeleton that can be supplemented by various similarity methods. An important contribution of the method, not considered by existing similarity methods, is an active participation of a domain expert. This is necessary, since we need to achieve exact mapping.

Outline The rest of this chapter is organized as follows. In Section 3.2, we present related work. Section 3.4 introduces an algorithm which assists a domain expert during mapping discovery. In Section 3.5, we introduce two kinds of precision measures and present some experimental results and conclude in Section 3.6.

3.2 Related work

Recent literature has been focused on a discovery of mappings of XML formats to a common model. We can identify several motivations. Firstly, XML schemas are hardly readable and a friendly graphical notation is necessary. This motivation has appeared in [42][51] or [133]. A survey of these approaches can be found in [137]. They introduce an algorithm for automatic conversion of a given XML schema to a UML class diagram. The result exactly corresponds to the given XML schema. However, these approaches can not be applied in our case – we need to map an XML schema to an existing conceptual diagram.

Secondly, there are approaches aimed at an integration of a set of XML format into a common XML format. These works include, e.g. the DIXSE framework [116] or Xyleme project [115]. They have a similar idea to derive a common abstract XML format from the existing XML formats. The mappings of the XML format to the abstract format are discovered automatically. The mappings can then be checked by a domain expert who can also specify additional mappings manually in the case of DIXSE framework. These approaches are closer to our work. However, they do not consider mapping of XML formats to a more abstract conceptual diagram which needs to be adapted when necessary. Moreover, they do not consider a domain expert participating directly in the mapping discovery process.

Thirdly, there are approaches that convert or map XML formats to ontologies. DTD2OWL [129] presents a simple method of automatic translation of an XML format with an XML schema expressed in DTD into an ontology. More advanced methods are presented in [40] and [135]. They both introduce an algorithm that automatically maps an XML format to an ontology. This is close to our approach since a conceptual schema can be understood as an ontology. In both cases, the domain expert can edit the discovered mappings but is not involved in the discovery process directly.

Many of these approaches widely exploit research results of the schema matching community. There have been introduced many methods or systems for automatic discovery of mappings between two given schemas based on measuring syntactical and semantical similarity of strings as well as measuring structural similarities, e.g. [70][17][108]. Nice surveys of these approaches can be found in [121][41] or [5]. The purpose of our work is not to introduce new similarity methods. We exploit and adapt existing ones to be applicable when mapping XML formats to the conceptual schema. We also extend these methods with an active participation of the domain expert. We will use only basic similarity methods and show how they can be substituted with advanced methods introduced in the recent literature.

In [87] an algorithm which discovers mappings of XML formats to a conceptual schema was introduced. It was a theoretical approach without implementation. Its time complexity was too high as its search space was very large. It was also complicated for the domain expert to participate in the mapping process.

3.3 Conceptual Model for XML

In this section, we re-introduce our conceptual model for XML. The reverse-engineering approaches in this thesis are designed using the original version of the conceptual model for XML from [86] and implemented in our original tool [54]. While the formalism slightly differs from the redesigned version from Chapter 6, the idea remains the same.

We introduce the notions of PIM and PSM formally in this section. We firstly introduce several symbols. $\mathfrak{L}$ denotes the set of all string labels. $\mathfrak{L}_a$ and $\mathfrak{L}_e$ are two sets s.t. $\mathfrak{L}_a \cup \mathfrak{L}_e = \mathfrak{L}$, $\mathfrak{L}_a \cap \mathfrak{L}_e = \emptyset$ and each label in $\mathfrak{L}_a$ starts with the '@' symbol. $\mathfrak{D}$ denotes the set of all basic data types such as `string`, `integer`, etc. $\mathfrak{C} \subset N \times (N \cup \{*\})$ is a set of *cardinality constraints* where N denotes the set of natural numbers and $(\forall (x,y) \in \mathfrak{C})\ (x \leq y \vee y = *)$. Finally, $\mathfrak{P}^S$ and $\mathfrak{O}^S$ denote the power set of a set S and the set of all ordered sequences of elements of S, respectively.

The *PIM meta-model* is de-facto the model of UML class diagrams. It introduces three modeling constructs: *PIM class*, *PIM attribute* and *PIM binary association.* We provide its formalization in Definition 3.1.

Definition 3.1 *A* PIM *is a 9-tuple* $\mathcal{M} = (\mathcal{C}, \mathcal{A}, \mathcal{R}, name, type, attrs, ends, acard, rcard)$ *where*

- $\mathcal{C}$, $\mathcal{A}$ *and* $\mathcal{R}$ *are sets of* PIM classes, PIM attributes *and* PIM associations*, respectively,*
- $name : \mathcal{C} \cup \mathcal{A} \cup \mathcal{R} \rightarrow \mathfrak{L}$ *is a function which assigns a label to each PIM class, attribute or association; the label is called* name*,*
- $type : \mathcal{A} \rightarrow \mathfrak{D}$ *is a function which assigns a data type to each PIM attribute,*
- $attrs : \mathcal{C} \rightarrow \mathfrak{P}^{\mathcal{A}}$ *is a function which assigns a set of PIM attributes to each PIM class s.t.:*
  - $(\forall A \in \mathcal{A})(\exists C \in \mathcal{C})(A \in attrs(C))$, *and*
  - $(\forall C_1, C_2 \in \mathcal{C})(attrs(C_1) \cap attrs(C_2) = \emptyset)$,
- $ends : \mathcal{R} \rightarrow \binom{\mathcal{C}}{2}$ *is a function which assigns a set of two PIM classes to each PIM association; for* $R \in \mathcal{R}$ *with* $ends(R) = \{C_1, C_2\}$ *we say that* C_1 *and* C_2 participate *in* R*,*
- $acard : \mathcal{A} \rightarrow \mathfrak{C}$ *is a function which assigns a cardinality to each PIM attribute,*
- $rcard : \mathcal{R} \times \mathcal{C} \rightarrow \mathfrak{C}$ *is a function which assigns a cardinality to each PIM class* $C \in \mathcal{C}$ *and PIM association* R *s.t.* C *participates in* R*.*

For a given PIM $\mathcal{M}$, we will use an auxiliary function $class : \mathcal{A} \rightarrow \mathcal{C}$ defined as $(\forall A \in \mathcal{A}, C \in \mathcal{C})\ (class(A) = C \Leftrightarrow A \in attrs(C))$.

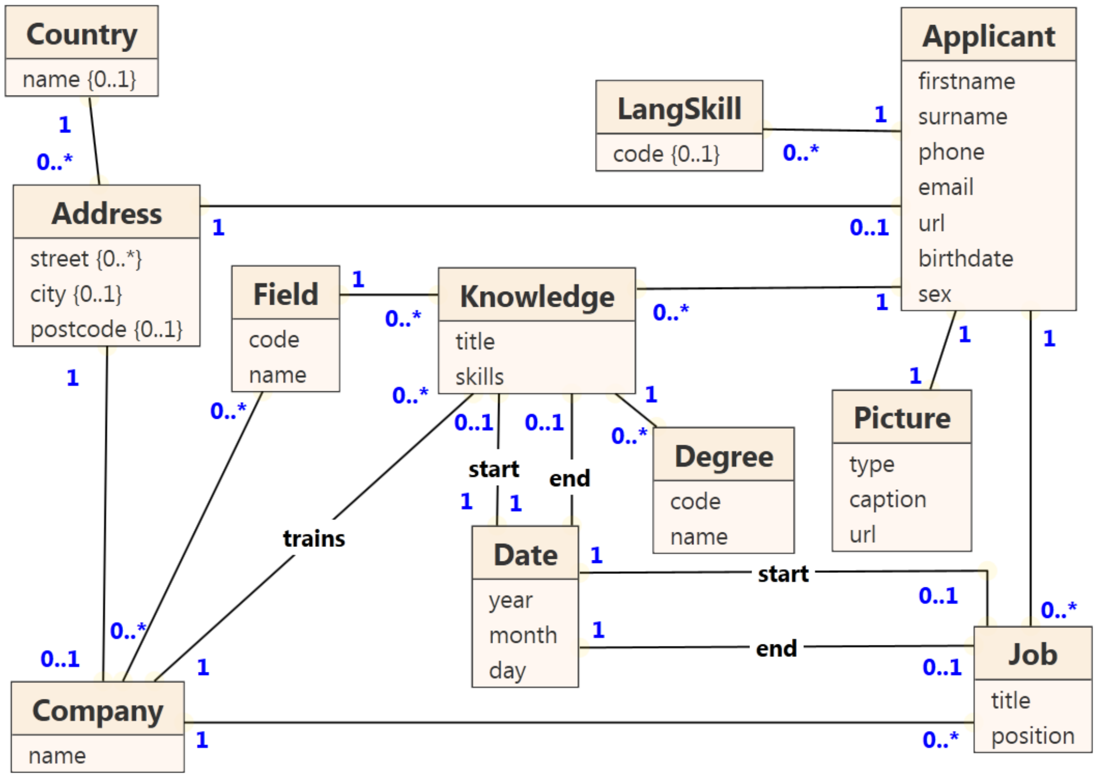


Figure 3.1: Employment PIM

A sample PIM is depicted in Figure 3.1. It models a simple jobs and education domain. We will further need a construct called a *PIM path* which is defined by Definition 3.2.

Definition 3.2 *Let* $\mathcal{M} = (\mathcal{C}, \mathcal{A}, \mathcal{R}, name, type, attrs, ends, acard, rcard)$ *be a PIM. A* PIM path P *is an ordered sequence* $\langle R_1, \ldots, R_n \rangle \in \mathfrak{O}^{\mathcal{R}}$ *where* $(\forall 1 \leq i \leq n))\ (ends(R_i) = \{C_{i-1}, C_i\})$. C_0 *and* C_n *are called* start *and* end *of* P, *respectively.*

We will use auxiliary functions $start, end : \mathfrak{O}^{\mathcal{R}} \rightarrow \mathcal{C}$ defined for each PIM path P which return the start and end of P, respectively.

The *PSM meta-model* consists of three modeling constructs which reflect the PIM constructs: *PSM class*, *PSM attribute*, and *PSM binary association*. Its formalization is in Definition 3.3.

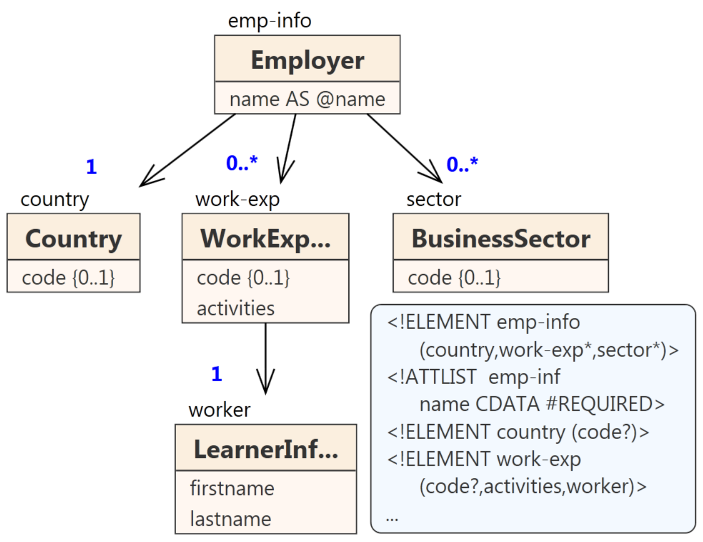
Definition 3.3 *A* PSM *is a 10-tuple* $\mathcal{M}' = (\mathcal{C}', \mathcal{A}', \mathcal{R}', name', type', xml', attrs', content', acard', rcard')$ *where*

- $\mathcal{C}'$, $\mathcal{A}'$ *are sets of* PSM classes *and* attributes, *resp.,*
- $\mathcal{R}' \subseteq \mathcal{C}' \times \mathcal{C}'$ *is a set of oriented* PSM associations; *for* $R' = (C'_1, C'_2) \in \mathcal{R}'$ *we call* C'_1 parent *and* C'_2 child *of* R,
- $name' : \mathcal{C}' \cup \mathcal{A}' \rightarrow \mathfrak{L}$ *is a function which assigns a label to each PSM class or attribute; the label is called* name,

- *$type' : \mathcal{A}' \rightarrow \mathfrak{D}$ is a function which assigns a data type to each PSM attribute,*
- *$xml' : \mathcal{C}' \cup \mathcal{A}' \rightarrow \mathfrak{L}$ is a function which assigns a label to each PSM class or attribute where $(\forall\ C' \in \mathcal{C}')\ (xml'(C') \in \mathfrak{L}_e)$; the label is called* xml label,
- *$attrs' : \mathcal{C}' \rightarrow \mathfrak{S}^{\mathcal{A}'}$ is a function which assigns a sequence of PSM attributes to each PSM class s.t.:*
  - *$(\forall A' \in \mathcal{A}')(\exists C' \in \mathcal{C}')(A' \in attrs'(C'))$, and*
  - *$(\forall C'_1, C'_2 \in \mathcal{C}')(attrs'(C'_1) \cap attrs(C'_2) = \emptyset)$,*
- *$content' : \mathcal{C}' \rightarrow \mathfrak{S}(\mathcal{R}')$ is function which assigns a sequence of PSM associations to each PSM class s.t. $(\forall C' \in \mathcal{C}')\ (\forall R' \in \mathcal{R}')\ (R' \in content'(C') \Rightarrow parent(R') = C')$,*
- *$acard' : \mathcal{A}' \rightarrow \mathfrak{C}$ is a function which assigns a cardinality to each PSM attribute,*
- *$rcard' : \mathcal{R}' \rightarrow \mathfrak{C}$ is a function which assigns a cardinality to each PIM association.*

Moreover, there must be a PSM class $C' \in \mathcal{C}'$ which is not a child of any PSM association in $\mathcal{R}'$. C' is called the root PSM class *of $\mathcal{M}'$. Any other PIM class in $\mathcal{C}' \setminus \{C'\}$ must be a child of exactly one PSM association in $\mathcal{R}'$.*

We use a function $class' : \mathcal{A}' \rightarrow \mathcal{C}'$ defined as $(\forall A' \in \mathcal{A}', C' \in \mathcal{C}')\ (class'(A') = C' \Leftrightarrow A' \in attrs'(C'))$ and functions $parent', child' : \mathcal{R}' \rightarrow \mathcal{C}'$ assigning the parent and child to each $R' \in \mathcal{R}'$, respectively.



```
<!ELEMENT emp-info
    (country,work-exp*,sector*)>
<!ATTLIST  emp-inf
    name CDATA #REQUIRED>
<!ELEMENT country (code?)>
<!ELEMENT work-exp
    (code?,activities,worker)>
...
```

Figure 3.2: Sample PSM

The definition ensures that the graph with a set of nodes $\mathcal{C}'$ and a set of oriented edges $\mathcal{R}'$ is an oriented rooted tree. A sample PSM is depicted in Figure 3.2. PSM components are visualized similarly to PIM components. In addition, for a PSM class C', $xml'(C')$ is depicted above the rectangle of C'. For a PSM attribute A', we depict $name'(A')$ and $xml'(C')$ separated by the word ‘as‘. If $name'(A') = xml'(C')$, we show only $name'(A')$.

We view PSM components from *grammatical* and *conceptual perspective*. From the *grammatical perspective*, a PSM class C' models XML elements with the name specified by $xml'(C')$ and content specified by $attrs'(C')$ and $content'(C')$. A PSM attribute $A' \in attrs'(C')$ models XML elements with a simple content (if $xml'(A') \in \mathfrak{L}_e$) or XML attributes (if $xml'(A') \in \mathfrak{L}_a$). In both cases, the name is specified by $xml'(A')$. A PSM association $R' \in content'(C')$ models hierarchical parent-child relationships between XML elements modeled by $parent(R')$ and $child(R')$. Each PSM can be automatically translated to an XML schema in a particular XML schema language. Conversely an XML schema can be translated to a PSM. See [86] and [87] for details. A partial translation of our sample PSM expressed in DTD is depicted in Figure 3.2.

From the *conceptual perspective*, the PSM maps each XML element and attribute to a PIM class or attribute and each XML parent-child relationship to a PIM association. This mapping is formally expressed as an *interpretation* introduced by Definition 3.4.

Definition 3.4 *An* interpretation I *of a PSM* $\mathcal{M}' = (\mathcal{C}', \mathcal{A}', \mathcal{R}', name', type', xml', attrs', content', acard', rcard')$ *against a PIM* $\mathcal{M} = (\mathcal{C}, \mathcal{A}, \mathcal{R}, name, type, attrs, ends, acard, rcard)$ *is a total function defined as*

$$\textit{if } P' \in \begin{cases} \mathcal{C}' & \textit{then } I(P') \in \mathcal{C}; \\ \mathcal{A}' & \textit{then } I(P') \in \mathcal{A}; \\ \mathcal{R}' & \textit{then } I(P') \in \mathcal{R}. \end{cases}$$

The following conditions must be satisfied:

$$\begin{array}{ll} (1) & (\forall A' \in \mathcal{A}')(class(I(A')) = I(class'(A'))) \\ (2) & (\forall R' \in \mathcal{R}')(ends(I(R')) = \\ & \qquad\qquad \{I(parent'(R')), I(child'(R'))\}) \end{array}$$

A part of an interpretation of our sample PSM against the sample PIM is as follows:

- $\mathcal{C}'$: I(Employee) = Company; I(Country) = Country; I(WorkExperience) = Job; ...
- $\mathcal{A}'$: I(Employer.name) = Company.name; ...
- $\mathcal{R}'$: I((Employer,Country)) = $\langle$\{Company,Address\}, \{Address,Country\}$\rangle$; I((Employer,WorkExperience) = $\langle$\{Company,Job\}$\rangle$; ...

It can be easily verified that it satisfies the conditions given by Definition 3.4.

3.4 Algorithm

We introduce an algorithm which builds an interpretation I of a PSM against a PIM. I must be correct in the formal sense, i.e. it must fulfil Definition 3.4. Moreover, it must be correct in the conceptual sense, i.e. a PSM component and its PIM interpretation must conceptually correspond to the same real-world concept. We ensure the formal correctness. The conceptual correctness is ensured by a domain expert.

The algorithm works in two phases. Firstly, it measures initial similarities between PSM and PIM attributes and classes. Secondly, it builds the interpretation with an assistance of a domain expert. Formally, we will suppose a PSM

$\mathcal{M}' = (\mathcal{C}', \mathcal{A}', \mathcal{R}', name', type', xml', attrs', content', acard', rcard')$ and a PIM $\mathcal{M} = (\mathcal{C}, \mathcal{A}, \mathcal{R}, name, type, attrs, ends, acard, rcard)$ on the input. The output of the algorithm is an interpretation I of $\mathcal{M}'$ against $\mathcal{M}$.

We will need to measure a similarity of two strings $S^{str}(s_1, s_2)$. There are various known string similarity methods, e.g. edit distance, N-grams, etc. We use a simple method measuring the length of their common substring.

3.4.1 Measuring Initial Similarity

Attributes. Let $(A', A) \in \mathcal{A}' \times \mathcal{A}$. The similarity of A' and A is a weighted sum

$$\begin{aligned} S^{init-attr}(A', A) &= w^{init-attr} * S^{type}(A', A) + \\ (1 - w^{init-attr}) &* max\{S^{str}(name'(A'), name(A)), \\ &\qquad S^{str}(xml'(A'), name(A))\} \end{aligned}$$

The first component, $S^{type}(A', A)$, measures the type similarity of both attributes. For this, we use the identity function in this chapter for simplicity. It is possible to employ more advanced techniques reflecting, e.g. sub-typing or attribute cardinalities. The second component is the maximum of two string similarities. $w^{init-attr} \in (0, 1)$ is a weighting factor which is set by the expert.

Classes. Let $(C', C) \in \mathcal{C}' \times \mathcal{C}$. The similarity between C' and C is a weighted sum

$$\begin{aligned} S^{init-class}(C', C) &= w^{init-class} * S^{init-attrs}(C', C) + \\ (1 - w^{init-class}) &* max\{S^{str}(name'(C'), name(C)), \\ &\qquad S^{str}(xml'(C'), name(C))\} \end{aligned}$$

$w^{init-class} \in (0, 1)$ is a weighting factor. $S^{init-attrs}(C', C)$ measures a similarity between $attrs'(C')$ and $attrs(C)$. It is defined as $S^{init-attrs}(C', C) = \sum_{A' \in attrs'(C')} (MAX_{A \in attrs(C)}\ S^{init-attr}(A', A))$. In other words it finds for each PSM attribute $A' \in attrs'(C')$ the most similar PIM attribute A of C and summarizes these similarities.

3.4.2 Building Interpretation

The second part of the algorithm iteratively traverses the PSM classes in $\mathcal{M}'$ in pre-order and helps the domain expert to build the interpretation. Individual steps are shown in Algorithm 1. For an actual PSM class $C' \in \mathcal{C}'$, the algorithm firstly constructs $I(C')$ (lines 2 - 6). Secondly, it constructs $I(A')$ for each $A' \in attrs(C')$ (lines 7 - 19). Finally, it constructs $I(R')$ for each $R' \in content(C')$ (lines 20 - 22). It can be shown that this algorithm runs in $O(N^3)$ where N is the number of PSM classes and in $O(n \times log(n))$ where n is the number of PIM classes.

Building Class Interpretation

To construct $I(C')$, the algorithm firstly computes $S^{class}(C', C)$ for each $C \in \mathcal{C}'$ at line 3. It is a weighted sum of two similarities. The former is the initial similarity $S^{init-class}(C', C)$. The other is a reversed class similarity adjustment $S^{adj-class}(C', C)$ which we discuss in a while. The algorithm then sorts the PIM

Algorithm 1 Interpretation Construction Algorithm

```
 1: for all $C' \in \mathcal{C}'$ in post-order do
 2:    for all $C \in \mathcal{C}$ do
 3:       $S^{class}(C',C) \leftarrow w^{class} * S^{init-class}(C',C) +$
                $(1 - w^{class}) * \frac{1}{S^{adj-class}(C',C)}$
 4:    end for
 5:    Offer the list of PIM classes sorted by $S^{class}$ to the domain expert.
 6:    $I(C') \leftarrow C$ where $C \in \mathcal{C}'$ is the PIM class selected by the domain expert.
 7:    for all $A' \in attrs(C')$ do
 8:       for all $A \in \mathcal{A}$ do
 9:          $S^{attr}(A',A) \leftarrow w^{attr} * S^{init-attr}(A',A) +$
                $(1 - w^{attr}) * \frac{1}{\mu(I(C'),class(A))+1}$
10:       end for
11:       Offer the list of PIM attributes sorted by $S^{attr}$ to the domain expert.
12:       $I(A') \leftarrow A$ where $A \in \mathcal{A}'$ is the PIM attribute depicted by the domain expert.
13:       if $I(class'(A')) \neq class(A)$ then
14:          Create PSM class $D' \in \mathcal{C}'$; $I(D') \leftarrow class(A)$
15:          Put $A'$ to $attrs'(D')$
16:          Create PSM association $R' = (C', D') \in \mathcal{R}'$
17:          Put $R'$ at the beginning of $content(C')$.
18:       end if
19:    end for
20:    for all $R' \in content(C')$ do
21:       $I(R') \leftarrow P$ where $P$ is the PIM path connecting $I(C')$ and $I(child'(R'))$ s.t.
          $\mu(I(C'), I(child'(R')))$ is minimal.
22:    end for
23: end for
```

classes by their similarity with C' and offers the sorted list to the domain expert at line 5. The expert selects a PIM class from the list and the algorithm sets $I(C')$ to this selected class at line 6.

Class similarity adjustment $S^{adj-class}(C',C)$ is computed on the base of the completed part of I. We have already set $I(D')$ for each PSM class D' preceding C'. The situation is depicted in Figure 3.4 (a) with predecessors in the part with grey background. $S^{adj-class}(C',C)$ is a combination of the distances between C and PIM classes which are interpretations of the predecessors of C'. We use $\mu(C,D)$ to denote the distance between PIM classes C and D. The idea is depicted in Figure 3.4 (b).

Algorithm 1 is only a skeleton which needs to be supplemented with particular methods for (1) measuring distances between PIM classes (i.e. suitable metric), (2) combining distances, and (3) selecting predecessors of C'. In this chapter, we supplement the skeleton with basic methods to show that the general idea works. For measuring the distance between two PIM classes C and D, we use the length of the shortest PIM path connecting C and D.

As the distance combination method which results in $S^{adj-class}(C',C)$ we can also choose from various possibilities. In this chapter, we use

$$S^{adj-class}(C',C) = (\sum_{i=1}^{n} \frac{\mu(C, I(D_i'))}{n}) + 1$$

where D_1', …, D_n' are the selected predecessors of C'. $S^{adj-class}(C',C)$ is the average of the lengths of the shortest PIM paths between C and each $I(D_i')$.

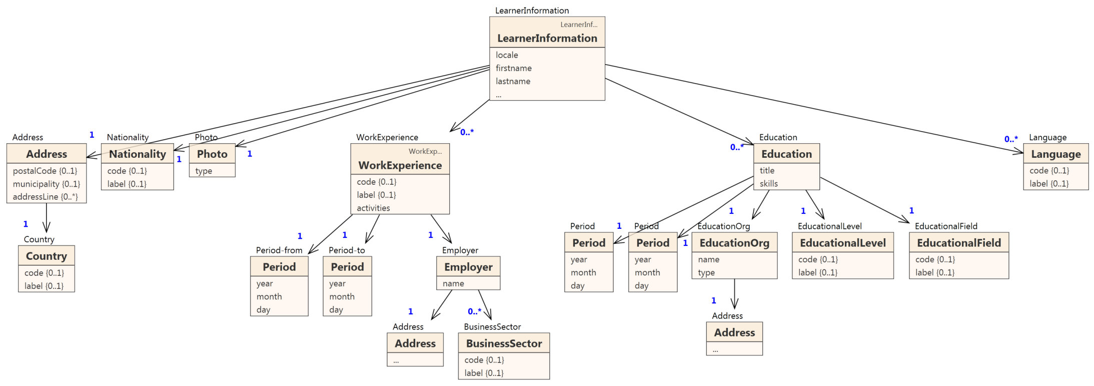


Figure 3.3: Employment PSM

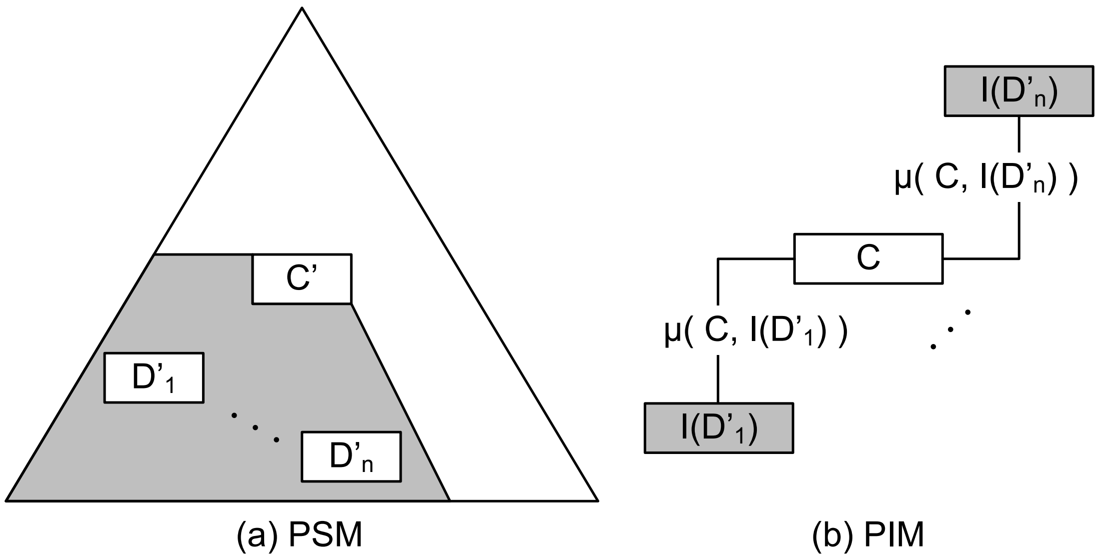


Figure 3.4: Initial Class Similarity Adjustment

Finally, we need to decide what predecessors of C' will be selected to compute $S^{adj-class}(C', C)$. In general, we can select all predecessors. However, this would result in a high time complexity. To restrict the search space, we use a heuristic that the impact of a predecessor D' of C' to the final similarity decreases with the growing distance of D' from C'. Therefore, we consider only the children of C' which are the closest predecessors to C'. This basic selection method can be extended by considering other close predecessors such as previous siblings of C'. Another possibility is to consider leaves of the sub-tree of C'. These possibilities have been recently discussed in [5].

We demonstrate the class similarity adjustment on the PSM class *Employer* from our sample PSM in Figure 3.2. From the previous iterations of the algorithm we already have $I(Country) = Country$, $I(WorkExperience) = Job$, and $I(BusinessSector) = Field$. Suppose the PIM class *Company*. Its distances from the PIM classes *Country*, *Job*, and *Field* are 2, 1, and 1, respectively. On the other hand, the distances of the PIM class *Applicant* are 2, 1, and 2, respectively. Therefore, the adjustment is higher for the PIM class *Company* then for *Applicant*. In this case, the adjustment helps. On the other hand, if the PSM class *BusinessSector* is not present in the PSM, the adjustment can not distinguish between both PIM classes.

Building Attribute Interpretation

Interpretation of PSM attributes of C' is computed at lines 7 - 19. For a PSM attribute $A' \in attrs(C')$, $I(A')$ can be any PIM attribute from $\mathcal{A}$. Therefore, the algorithm measures the similarity between A' and each PIM attribute A at line 9. Again, the similarity is a weighted sum of the initial similarity $S^{init-attr}(A', A)$ and the reversed value of an adjustment to the initial similarity. The adjustment in this case is simply a distance between $I(C')$ and the class of A increased by 1. The algorithm then offers the list of PIM attributes sorted by the computed similarity to the expert who selects a correct interpretation of A' at lines 11 and 12.

If $I(C') \neq class(A)$ then $class(I(A')) = class(A) \neq I(C') = I(class'(A'))$ which is inconsistent with the condition (1) of Definition 3.4. Therefore, we need to create a new PSM class $D' \in \mathcal{C}'$ such that $I(D') = class(A)$ and move A' to $attrs'(D')$ and put D' as the beginning of $content'(C')$ at lines 13 - 18.

Building Association Interpretation

Interpretation of PSM associations in $content'(C')$ is computed at lines 20 - 22. Suppose a PSM association $R' = (C'_1, C'_2) \in \mathcal{R}'$. The algorithm simply puts $I(R')$ $= P$ where P is the PIM path connecting $I(C'_1)$ and $I(C'_2)$ with the minimal distance with respect to the chosen metric. This may of course be inaccurate and therefore a domain expert should check the constructed interpretation and, where necessary, change the represented PIM path.

Evolution of PIM

It may happen that there is no component in $\mathcal{M}$ suitable as an interpretation of a given component in $\mathcal{M}'$. A special case is when $\mathcal{M}$ is empty. There are two possible solutions of this situation. The expert can (1) leave the interpretation of the PSM component unspecified or (2) create a new component of $\mathcal{M}$ which will be the interpretation of the PSM component. The former would result into an inconsistency with Definition 3.4 which requires the interpretation to be a total function. Our full PSM meta-model introduces special constructs that also model XML elements and attributes but have no interpretation and can be therefore used in this case.

The other solution does not violate the consistency. When the PSM component is a PSM class C', the algorithm creates a new PIM class $C \in \mathcal{C}$ with $name(C) = name'(C')$. When it is a PSM attribute A', it creates a new PIM attribute $A \in \mathcal{A}$ with $name(A) = name'(A')$, $class(A) = I(class'(A'))$, and $type(A)$ $= type'(A')$. When the PSM component is a PSM association R', the algorithm creates a new PIM association $R \in \mathcal{R}$ with association ends $ends(R) = \{I(parent'(R'), I(child'(R'))$.

3.5 Experiments

In this section, we briefly present some experimental results on building interpretations of PSM classes. We have implemented the introduced method in our tool

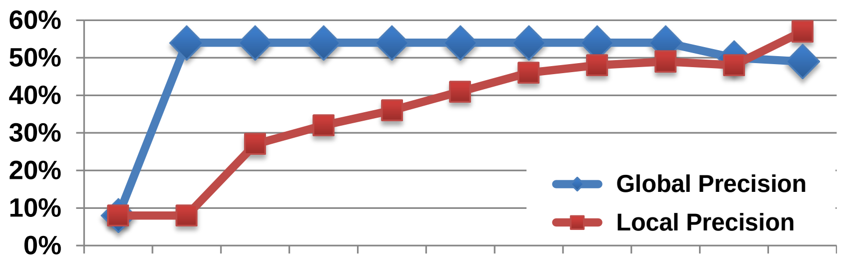


Figure 3.5: $\mathcal{P}_G$ and $\mathcal{P}_L$ for Leaf PSM Classes

XCase[2] which was primarily intended for designing XML schemas from a created PIM. XCase can be downloaded with some experimental XML schemas. These also include the experimental XML schema used in this section.

Let us suppose an actual PSM class C'. Let the domain expert set $I(C')$ to a PIM class C. We measure the precision of the algorithm from two points of view. Firstly, we measure the position of C in the list of PIM classes offered to the expert sorted by their S^{class}. We call this precision a *global precision* $\mathcal{P}_G$:

$$\mathcal{P}_G = ((\sum_{C' \in \mathcal{C}'} 1 - \frac{order(C) - 1}{n})/n') * 100$$

where n denotes the size of $\mathcal{C}$, n' denotes the size of $\mathcal{C}'$, and $order(C)$ denotes the order of C in the list. If there are more PIM classes with the same similarity to C', $order(C)$ is the order of the last one. $\mathcal{P}_G = 0$ (resp. 1) if for each PSM class C', the selected PIM class was the last (resp. first).

The global precision is not sufficient. When C is the first class, there can be other PIM classes before C which have their similarity to C' close to $S^{class}(C', C)$. We therefore propose another metric called *local precision* which measures the amount of PIM classes with their similarity to C' close to $S^{class}(C', I(C'))$. It is defined as

$$\mathcal{P}_L = ((\sum_{C' \in \mathcal{C}'} 1 - \frac{close(C) - 1}{n})/n') * 100$$

where $close(C)$ denotes the number of PIM classes with their similarity to C' close to $S^{class}(C', C)$. The term *close similarity* can be defined in various ways. In this chapter, we say that y is *close* to x if $y \in (x - 0.1, x + 0.1)$.

We used our algorithm to build an interpretation of the PSM depicted in Figure 3.3 against the PIM depicted in Figure 3.1. The PSM was directly constructed from *EuropassSchema* XML schema[3] which is an official EU XML standard for the employment domain. We tested various settings of weighting factors. We show only the results related to PSM classes. We distinguish leaf PSM classes, i.e. PSM classes with empty content, from inner PSM classes, i.e. PSM classes with a non-empty content.

Figure 3.5 shows the global and local precision for leaf PSM classes when $w^{class} = 1$ (it has no sense to consider the similarity adjustment as leaves have no children). Various settings of the weighting factor $w^{init-class}$ are displayed at the horizontal axis. The highest global precision is a little above 0.5, i.e. the average

[2]`http://xcase.codeplex.com`

[3]`http://europass.cedefop.europa.eu/xml/CVLPSchema_V2.0.xsd`

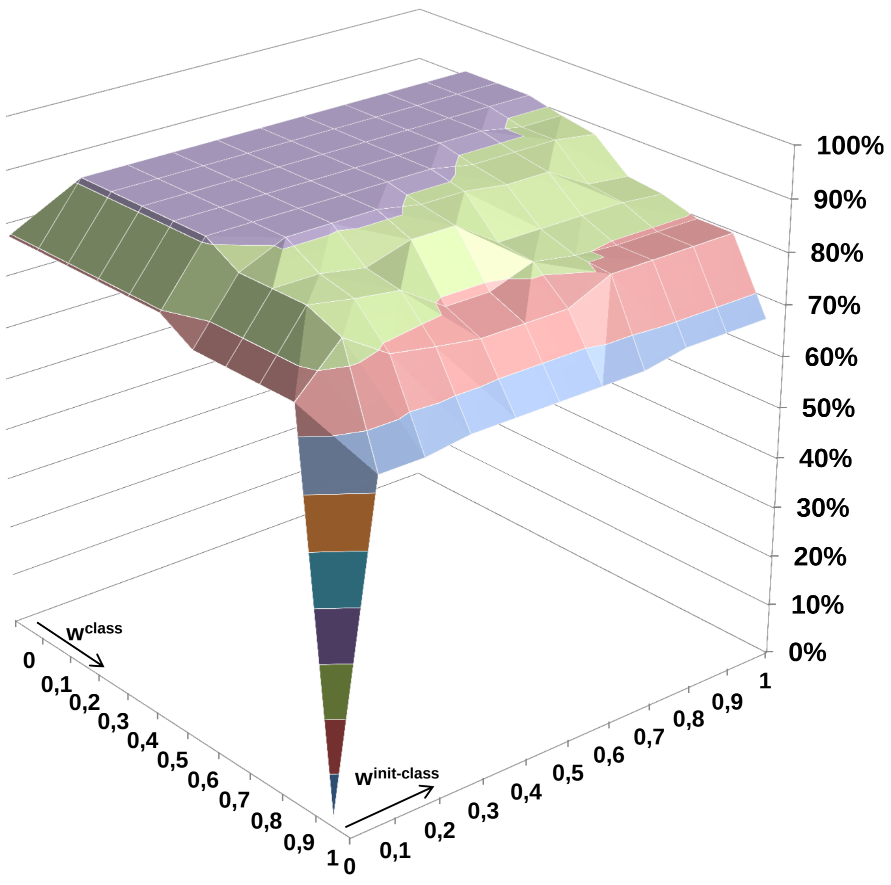


Figure 3.6: $\mathcal{P}_G$ for Inner PSM Classes

position of $I(C')$ for a PSM class C' is somewhere in the middle of the offered list. Moreover, the local precision shows that there were many PIM classes with their similarity to C' close to $S^{class}(C', I(C'))$. The local precision grows with $w^{init-class}$, i.e. measuring similarity of the attributes of C' with the attributes of PIM classes helped. The precision of the algorithm is not very good for leaf PSM classes. This is also because we only used primitive similarity methods.

Figures 3.6 and 3.7 show global and local precision for inner PSM classes, respectively. They show that we can reach much higher precision for inner PSM classes then for leaves. This is because we can exploit the similarity adjustments for the inner PSM classes. The former figure shows that we reached the highest global precision for $w^{class} \in [0.1, 0.3]$ and $w^{init-class} \in [0.1, 1]$. In other words, it shows that the similarity adjustment is important in our sample. This is because PSM classes are not very similar on their names and attributes. The other figure shows that there is only a small range of weighting factors where we can reach a good local precision. This range is around $w^{class} \in [0.2, 0.3]$ and $w^{init-class} \in [0.5, 1]$.

3.6 Conclusion

In this chapter, we studied mapping of XML formats to a conceptual schema. We introduced a basic algorithm which allows to exploit various similarity mea-

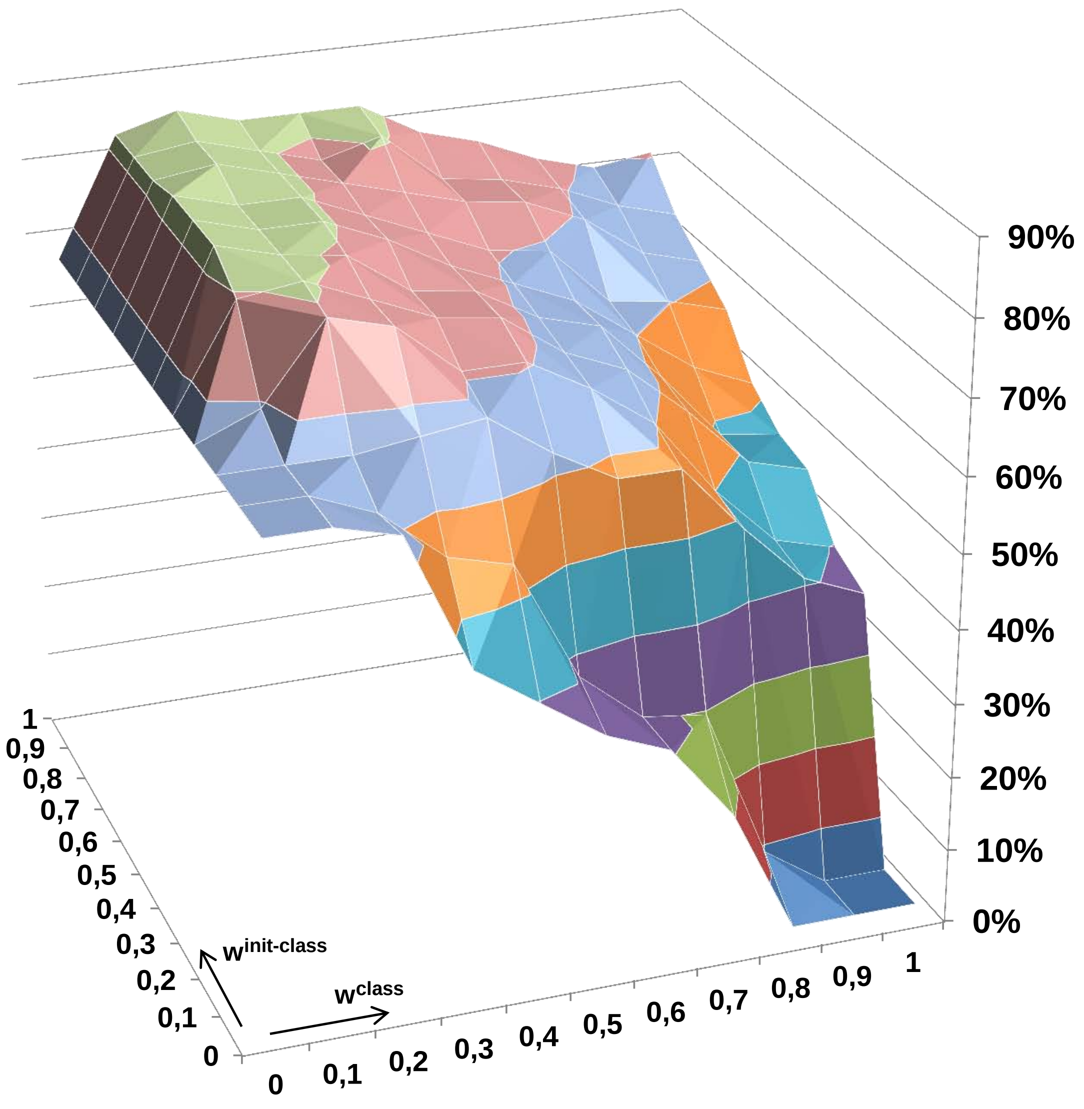


Figure 3.7: $\mathcal{P}_L$ for Inner PSM Classes

surement methods. The algorithm also incorporates a domain expert into the mapping process. In each iteration of the algorithm, the domain expert confirms the discovered mappings. The algorithm exploits this decision to adjust the similarities in following iterations. We have demonstrated on a simple experiment with a real XML format that this idea works.

We have shown that the algorithm can be adapted by the chosen similarity measurement techniques. We can also choose what part of the mapping already approved by the expert is used to compute the similarity adjustments. In our future work, we will experiment with various XML schemas and possible adaptations of the algorithm. We will also consider 1:n mappings and investigate the possibility of incorporation of behavioral aspects of web services. We expect that these adaptations will help in certain cases but will fail in others. Therefore, we will also study if and how it is possible to adapt the algorithm during its runtime on the base of its history.

4. A Framework for XML Schema Integration via Conceptual Model

Modern information systems may exploit numerous XML schemas for communication. Each schema represents a different XML format. This causes problems with their integration and evolution. Manual integration and management of evolution of the XML formats may be very hard. In this chapter, we experiment with our novel method exploiting a conceptual schema and we present our results. We introduce a framework which helps a domain expert to map the XML formats to the conceptual schema. It can be configured to use various similarities of the XML formats and the schema and it can adjust them on the basis of the input from the expert. The result is a precise mapping. The schema then integrates the XML formats and facilitates their evolution.

The content of this chapter was published as a workshop paper *A Framework for XML Schema Integration via Conceptual Model*[1] [57] in Web Information Systems Engineering – WISE 2010 Workshops (WISE 2010).

4.1 Introduction

In our previous work (see Chapter 1 and Chapter 3) and in [86] a framework for XML schema integration and evolution is introduced. It supposes a set of XML schemas that are conceptually related to the same problem domain. As a problem domain, we can consider, e.g., purchasing products. Sample XML schemas may be XML schemas for purchase orders, product catalogue, customer detail, etc. The central part of the framework is a conceptual schema which specifies the problem domain. Each XML schema is then mapped to this schema, i.e. XML schemas are integrated via the conceptual schema. This also facilitates the evolution. A change in the domain is made only once in the conceptual schema and propagated to the XML schemas.

Contributions In practice, a conceptual schema is usually developed during an analysis and the XML schemas are developed separately by designers. Therefore, there are no mappings between both levels. This disallows to exploit the integration and evolution capabilities of our framework. In [86] a method for deriving required XML schemas from the conceptual schema is introduced. However, it does not consider an existing XML schema that needs to be mapped to the conceptual schema. In this chapter, we introduce a framework based on our reverse-engineering algorithm introduced in Chapter 3 which allows to correctly map a supplied XML schema to the conceptual schema semi-automatic way.

An important part of the method is measurement of similarity of the XML schema and conceptual schema. In the recent research literature, there have been

[1] `http://link.springer.com/chapter/10.1007/978-3-642-24396-7_8`

proposed various methods for measuring similarities of XML schemas or ontologies (see [41] for a comprehensive survey). Our aim is not to develop entirely new similarity methods. Instead, we exploit the existing ones and combine them together. We also combine existing similarity methods with an active participation of a domain expert.

Outline In Section 4.2 our framework for integration of XML schemas is presented. Section 4.3 contains metrics used to evaluate our approach. In Section 4.4 we experiment with various configurations of the framework. Section 4.5 summarizes related work and Section 4.6 concludes.

4.2 Integration Framework

In this section, we introduce a framework which constructs an interpretation I of a PSM schema $\mathcal{M}'$ against a PIM schema $\mathcal{M}$. It has three parts:

- P_I measures *initial similarities* of components of $\mathcal{M}'$ and $\mathcal{M}$.
- P_{II} produces *initial interpretation* of a subset of components of $\mathcal{M}'$ on the basis of the initial similarities. The result is usually incomplete and inaccurate.
- P_{III} constructs the final interpretation I of $\mathcal{M}'$ against $\mathcal{M}$ on the basis of interaction with the domain expert. The resulting interpretation is complete and correct.

There are various algorithms for measuring similarities of two schema graphs [41]. Our aim is not to develop new similarity measure. Instead, our framework is configurable by these existing techniques. Possible configurations of the current implementation are described in this section and summarized in the end of the section in Table 4.3.

4.2.1 Similarity Functions in General

The framework exploits various types of similarity functions. Generally, a *similarity function* is a function $S : O \times O' \to [0, 1]$ which assigns a real number from the interval $[0, 1]$ (i.e. including 0 and 1) to a pair of items from sets O and O'. A similarity function may also be defined as a combination of other, simpler similarity functions. Suppose similarity values $s_1, \ldots, s_n \in [0, 1]$. The recent literature [36] considers various combination methods, e.g.

- weighted sum, i.e. $\sum_{i=1}^{n} w_i * s_i$, where $w_1 + \cdots + w_n = 1$ (c-4.2.1.1)
- minimum, i.e. $\min_{i=1}^{n} s_i$ (c-4.2.1.2)
- maximum, i.e. $\max_{i=1}^{n} s_i$ (c-4.2.1.3)

The result of a combination is a *composite similarity function*. Each composite similarity function will take one or more similarity values as an input, but we will not specify their particular combination. Choosing a suitable combination is a part of the configuration of the framework.

	$name(A)$	$type(A)$	$acard(A)$
$name'(A')$	S^{str}	S^{str}	–
$type'(A')$	S^{str}	S^{type}	–
$acard'(A')$	–	–	S^{card}
$xml'(A')$	S^{str}	S^{str}	–

Table 4.1: Basic possibilities for computing $S^{init-attr}$

4.2.2 Auxiliary Similarity Functions

Firstly, we define several auxiliary similarity functions. A *string similarity function S^{str}* assigns a similarity to a pair of strings. To compute S^{str}, we can exploit various methods introduced in the literature such as the longest common substring, edit distance, or N-grams [30]. These basic methods may be extended with semantic similarities based on, e.g. *WordNet* [77]. There have also appeared methods that normalize strings by, e.g. expansion of shortcuts [124], which further improve the precision of string similarity methods. On the other hand, a *data type similarity function S^{type}* assigns a similarity to a pair of data types. To compute S^{type}, we can also exploit various functions [70, 36] based on sub-typing hierarchy, etc. Finally, a *cardinality similarity function S^{card}* assigns a similarity to a pair of association cardinalities (i.e. intervals). To compute S^{card}, we can consider various interval relation functions (e.g. interval inclusion, equivalent lower cardinality, equivalent upper cardinality, etc.).

4.2.3 P_I: Measuring Initial Similarities

P_I measures similarities of each pair of attributes and each pair of classes from $\mathcal{M}'$ and $\mathcal{M}$. This phase is fully automatic. Formally, it introduces two similarity functions. An *initial attribute similarity function $S^{init-attr}$* assigns a similarity to each pair of attributes $(A', A) \in \mathcal{A}' \times \mathcal{A}$. Analogously, an *initial class similarity function $S^{init-class}$* assigns a similarity to each pair of classes $(C', C) \in \mathcal{C}' \times \mathcal{C}$.

For computing these two functions we can exploit various characteristics of attributes or classes, respectively. An attribute from $\mathcal{A}$ is characterized by its name, data type and cardinality. An attribute from $\mathcal{A}'$ is moreover characterized by its XML label. Therefore, $S^{init-attr}$ can be defined in various ways. Basically, for $(A', A) \in \mathcal{A}' \times \mathcal{A}$, it can be defined as the string similarity of the names of both attributes, the type similarity of their types, or the string similarity of the XML label of A' and the name of A. We summarize basic possibilities in Table 1(a). Naturally, they may be further combined into more complex ones.

Similarly, a class from $\mathcal{C}$ is characterized by its name and attributes. Moreover, a class from $\mathcal{C}'$ has its XML label. For $\langle C', C \rangle \in \mathcal{C}' \times \mathcal{C}$, basic possibilities are therefore again the string similarity of the names of both classes or the string similarity of the XML label of C' and the name of C. Again, combinations are possible. We can also combine initial similarities of the attributes of both classes. See Table 1(b) for their list.

To compute initial similarities we could also exploit structural similarities of the neighborhoods of the measured attributes and classes. Such possibilities were discussed, e.g. in [5]. However, one needs to consider the fact that these structural similarities increase the time complexity. Moreover, as we will show, P_{III}

	$name(C)$	$attrs(C)$
$name'(C')$	S^{str}	–
$xml'(C')$	S^{str}	–
$attrs'(C')$	–	combination of $S^{init-attr}$

Table 4.2: Basic possibilities for computing $S^{init-class}$

also operates with structural similarities which are based on more precise inputs. Therefore, we do not consider measuring structural similarities for computing initial similarity.

4.2.4 P_{II}: Initial Interpretation Setup

P_{II} sets initial class interpretations according to the initial class similarities pre-computed in the previous step. It is a simple procedure that takes the most similar pairs of PSM and PIM classes and sets these pairs as initial interpretations. Formally, an *initial class interpretation* is a partial function $I^{init} : \mathcal{C}' \to \mathcal{C}'$ which maps a PSM class to a PIM class. We consider an initial interpretation threshold $t \in [0, 1]$. For each class $C' \in \mathcal{C}'$, the framework takes the class $C \in \mathcal{C}$ with the highest initial class similarity to C'. If the similarity exceeds t, the framework sets $I^{init}(C') = C$. If there are more such classes in $\mathcal{C}$, the framework takes an arbitrary one. The threshold t is set by the domain expert. We present some experiments with setting t in the following section.

4.2.5 P_{III}: Constructing Final Interpretation

The last part P_{III} of the framework traverses the classes in $\mathcal{C}'$ in pre-order and helps the domain expert to build the interpretation. For a given class $C' \in \mathcal{C}'$, it firstly constructs $I(C')$, then it constructs $I(A')$ for each $A' \in attrs(C')$ and, finally, $I(R')$ for each $R' \in content(C')$. In the rest of this section, we describe the algorithm for P_{III}.

Constructing Class Interpretation

To construct $I(C')$, the algorithm offers the list of classes from $\mathcal{C}$ to the domain expert, who selects the optimal class C_0 from the offered list. The algorithm then sets $I(C') = C_0$. Our goal is to sort the offered list so that C_0 is as high as possible in the list. In the optimal case, C_0 is the first offered class. To sort the list, we use the pre-computed initial similarities. Moreover, we adjust the initial similarities by the already constructed part of I and by I^{init}.

Formally, for each class $C \in \mathcal{C}$ the algorithm computes a *class similarity*. It is a combination of the initial similarity of C' and C and so-called *class similarity adjustment* of C' and C. The class similarity of C' and C is defined as follows:

$$S^{class}(C', C) = w^{class} * S^{init-class}(C', C) + (1 - w^{class}) * S^{adj-class}(C', C)$$

where $w^{class} \in [0, 1]$ and $S^{adj-class}$ denotes the class similarity adjustment. The class similarity adjustment $S^{adj-class}(C', C)$ reflects the similarity of neighborhood of C' and C and exploits the results of the previous steps of the algorithm confirmed by the user. In particular, the algorithm has already constructed interpretations of classes in $\mathcal{C}'$ which are before C' in the pre-order traversal. We will use a function $interpreted(C')$ which returns the set of these classes for C'. There are also zero or more classes in $\mathcal{C}'$ which are after C' in the pre-order traversal and the initial class similarity I^{init} was set for them during P_{II} part of the framework. We will use a function $preinterpreted(C')$ which returns the set of these classes for C'.

Let D be a class from the set $\{I(D') : D' \in interpreted(C')\} \cup \{I^{init}(D') : D' \in preinterpreted(C')\}$. We will measure the distance $\mu(C, D)$ between C and D. There exist various ways of computing $\mu(C, D)$. E.g. having a function $paths(C, D)$ which returns the set of all PIM paths connecting C and D, it can be defined as:

$$\mu(C,D) = \min_{\substack{P=\langle R_1,\dots,R_n\rangle\\ \in paths(C,D)}} \begin{cases} n & \text{(c-4.2.5.1)} \\ n/S^{comp}(\{S^{str}(xml'(C'), name(R_i))\}_{i=1}^{n}) & \text{(c-4.2.5.2)} \\ n/S^{comp}(\{\frac{S^{str}(xml'(C'),name(R_i))}{(n-i+1)}\}_{i=1}^{n}) & \text{(c-4.2.5.3)} \end{cases}$$

where (c-4.2.5.1) takes the length of the shortest PIM path. (c-4.2.5.2) and (c-4.2.5.3) consider similarities of the names of the associations along PIM paths and the XML label of C'. (c-4.2.5.3) also considers the position of the respective association in the PIM path – the closer to C the association with a similar name is, the higher is the resulting distance. There are other possibilities as well. For example, we can use name of C' instead of its XML label or combine them together.

$S^{adj-class}(C', C)$ is then computed as a combination of the distances between C and interpretations of classes from $interpreted(C') \cup preinterpreted(C')$. Even though we can consider the whole $interpreted(C') \cup preinterpreted(C')$, we will consider only the near neighbors of C'. It is practical as considering the whole set would increase the time complexity. Moreover, our hypothesis is that only the close neighbors have impact on the similarity and with the growing distance the impact declines. Formally, we will use the following functions:

$$\begin{aligned} nbhr\text{-}children(C') &= \{D' : (\exists R' \in content(C'))(D' = child(R'))\} && \text{(c-4.2.5.4)} \\ nbhr\text{-}psiblings(C') &= \{D' : content(parent(C')) = \langle R'_1, \dots, R'_n\rangle \wedge && \text{(c-4.2.5.5)} \\ &\quad (\exists 1 \leq j < i \leq n)(child(R'_i) = C' \wedge child(R'_j) = D')\} \\ nbhr\text{-}parent(C') &= \{parent(C')\} \cap preinterpreted(C') && \text{(c-4.2.5.6)} \\ nbhr\text{-}fsiblings(C') &= \{D' : content(parent(C')) = \langle R'_1, \dots, R'_n\rangle \wedge && \text{(c-4.2.5.7)} \\ &\quad (\exists 1 \leq i < j \leq n)(child(R'_i) = C' \wedge child(R'_j) = D')\} \\ &\quad \cap preinterpreted(C') \end{aligned}$$

Informally, $nbhr\text{-}children(C')$ returns the child classes of PSM class C' and $nbhr\text{-}psiblings(C')$ returns the previous sibling classes of C'. Classes of both types already have their interpretations because of the pre-order traversal of the PSM. The function $nbhr\text{-}parent(C')$ returns a set containing the parent class of C' if I^{init} of the parent has been set. The function $nbhr\text{-}fsiblings(C')$ returns the following sibling classes of C'. Again, only those with the initial interpretation are considered. Therefore, both $nbhr\text{-}parent(C')$ and $nbhr\text{-}fsiblings(C')$ may return empty sets.

A configuration of the algorithm may choose any combination of the four functions to target classes whose (initial) interpretations will be used to compute

	Purpose	Possible Configurations
S^{str}	String similarity	see Section 4.2.2
S^{type}	Data type similarity	see Section 4.2.2
$S^{init-attr}$	Initial attribute similarity	Basic possibilities in Table 1(a) or their combinations, e.g. (c-4.2.1.1), (c-4.2.1.2), or (c-4.2.1.3)
$S^{init-class}$	Initial class similarity	Basic possibilities in Table 1(b) or their combinations, e.g. (c-4.2.1.1), (c-4.2.1.2), or (c-4.2.1.3)
t	Initial interpretation threshold	$[0, 1]$
μ	Distance between classes from $\mathcal{C}$	e.g. (c-4.2.5.1), (c-4.2.5.2), or (c-4.2.5.3)
$I_{\|C'}$	Selection of suitable neighbors of C' for computing S^{adj}	e.g. various unions of (c-4.2.5.4), (c-4.2.5.5), (c-4.2.5.6), or (c-4.2.5.7)
$S^{comp}_{adj-class}$	Combination of reversed values of distances between a class $C \in \mathcal{C}$ and classes from $I_{\|C'}$	e.g. (c-4.2.1.1), (c-4.2.1.2), or (c-4.2.1.3)
w^{class}	Weighting factor of $S^{init-class}$ and $S^{adj-class}$ for computing S^{class}	$[0, 1]$

Table 4.3: Possible framework configurations

the adjustment $S^{adj-class}(C', C)$. We will use $I_{|C'}$ to denote the set of (initial) interpretations of the classes returned by the selected configuration applied on C'.

We are now ready to formally define $S^{adj-class}(C', C)$:

$$S^{adj-class}(C', C) = S^{comp}_{adj-class}(\{\frac{1}{\mu(C, D)}\}_{D \in I_{|C'}})$$

where $S^{comp}_{adj-class}$ is a composite similarity function.

Constructing Attribute and Association Interpretation

To construct interpretation $I(A')$ of an attribute $A' \in attrs'(C')$, the algorithm offers the list of attributes in $I(C')$ sorted by their initial attribute similarity with A' to the domain expert. The domain expert selects the optimal attribute A and the algorithm sets $I(A') = A$. There may occur a situation when A is from another class than $I(C')$.

For an association $R' \in content(C')$, the algorithm directly sets $I(R') = P$, where P is the PIM path connecting classes $I(parent(R'))$ and $I(parent(R'))$ with the minimal distance according to the selected distance metric μ. This may be inaccurate and therefore a domain expert needs to check the constructed interpretation and, where necessary, change the represented PIM path. However, we do not discuss this process in a more detail in this chapter.

4.3 Measuring Quality

The key aspect of our approach is how to measure the quality of a given configuration of the framework. In particular, we target the quality of building class interpretations in this chapter. We will not measure the quality of building attribute and association interpretations due to the lack of space.

Suppose a class $C' \in \mathcal{C}'$. Let the domain expert set $I(C')$ to a class $C \in \mathcal{C}$. We measure the precision from two points of view. Firstly, we measure the position of C in the list of classes offered to the expert sorted by S^{class}. We call this metric a *global precision* $\mathcal{P}_G$:

$$\mathcal{P}_G = ((\sum_{C' \in \mathcal{C}'} 1 - \frac{order(C) - 1}{|\mathcal{C} - 1|})/|\mathcal{C}'|)$$

where $order(C)$ denotes the order of C in the list. If there are multiple PIM classes with the similarity equivalent to C', $order(C)$ returns the order of the last one. $\mathcal{P}_G = 0$ (resp. 1) if for each class $C' \in \mathcal{C}'$, the selected $I(C')$ was the last (resp. first) one.

However, the global precision is not sufficient. When C is the first class, there can be other classes after C in the sorted list with their similarity with C' "close" to the similarity $S^{class}(C', C)$. Therefore we propose another metric called *local precision* which measures the number of classes with their similarity with C' "close" to $S^{class}(C', C)$:

$$\mathcal{P}_L = ((\sum_{C' \in \mathcal{C}'} 1 - \frac{close(C) - 1}{|\mathcal{C} - 1|})/|\mathcal{C}'|)$$

where $close(C)$ denotes the number of PIM classes with their similarity to C' "close" to $S^{class}(C', C)$. The term *close similarity* can be defined in various ways. In this chapter, we say that y is *close* to x if $y \in (x - 0.1, x + 0.1)$.

4.4 Experimental Evaluation

We have implemented a general framework which is fully configurable as described in Section 4.2. In this section, we present selected interesting experimental results. In particular, we present an experiment which tests various settings of $I_{|C'}$, i.e. it shows the impact of selected neighborhood of a class $C' \in \mathcal{C}'$ on the computed similarities. We will consider three experimental configurations $\mathfrak{C}_1$, $\mathfrak{C}_2$ and $\mathfrak{C}_3$ with the following common settings:

S^{str}	The longest common substring
S^{type}	Identity
$S^{init-attr}$	$0.5 * S^{type}(type'(A'), type(A)) +$ $0.5 * \max\{S^{str}(name'(A'), name(A)),\ S^{str}(xml'(A'), name(A))\}$
$S^{init-class}$	$w^{iclass} * \sum_{A' \in attrs'(C')} \frac{\min_{A \in attrs(C)} S^{attr}(A',A)}{\lvert attrs'(C') \rvert} +$ $(1 - w^{iclass}) * \max\{S^{str}(name'(C'), name(C)),$ $S^{str}(xml'(C'), name(C))\}$ where w^{iclass} is a weighting factor from $[0, 1]$
μ	(c-4.2.5.1)
$S^{comb}_{adj-class}$	(c-4.2.1.1), where all weights are set to $\frac{1}{\lvert I_{\lvert C'} \rvert}$
w^{class}	$[0, 1]$

The configurations differ as follows:

	$\mathfrak{C}_1$	$\mathfrak{C}_2$	$\mathfrak{C}_3$
t	0.5	0.5	0.5 or 0.75
$I_{\lvert C'}$	(c-4.2.5.4)	(c-4.2.5.4) $\cup$ (c-4.2.5.5)	(c-4.2.5.4) $\cup$ (c-4.2.5.5) $\cup$ (c-4.2.5.6) $\cup$ (c-4.2.5.7)

The configuration uses an additional weighting factor w^{iclass} which is not fixed. We also do not fix w^{class} and we will experiment with various settings of both of them.

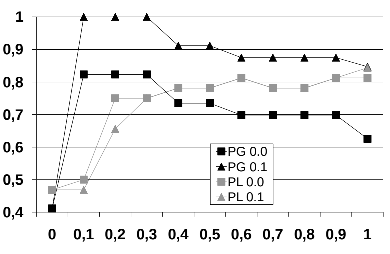
We will present experiments with our sample PIM and PSM and two real world scenarios in the rest of this section. The results are presented in a form of charts in Figures 4.1, 4.5 and 4.6. Each chart shows different settings of w^{class} on the horizontal axis and achieved global and local precisions on the vertical axis. Note that PG shortcut stands for global precision $\mathcal{P}_G$, PL means local precision $\mathcal{P}_L$.

Experiments with Example Our first experiment shows the achieved precisions during the process of building interpretation of our sample PSM in Figure 4.3(b) against the sample PIM in Figure 4.2. The experiment shows that event the classes in the sample schemas do not have very similar names, considering class similarity adjustment allows to achieve good results. The highest precisions were achieved for $w^{iclass} = 0$ and $w^{iclass} = 0.1$ which are depicted in Figure 4.1. This is natural as the class names in the example are not very similar. $w^{iclass} = 0$ (squares in the charts) means that we do not consider S^{str} when we compute $S^{init-class}$. It means that the initial similarity is based only on S^{attr}. $w^{iclass} = 0.1$ (triangles in the charts) means that $S^{init-class}$ is influenced by S^{str} partially.

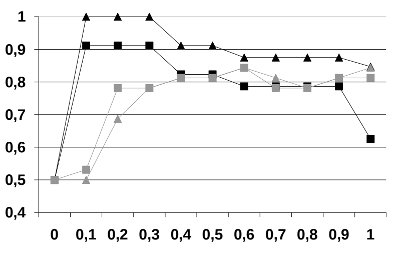
The difference between $\mathcal{P}_G$ and $\mathcal{P}_L$ in Figures 4.1(a) and 4.1(b) shows the positive impact of involving *nbhr-psiblings*(C') in the result of $S^{adj-class}$. Figures 4.1(b) and 4.1(d) are equivalent, because only for PSM classes *Customer* and *ItemInfo* the *initial interpretation* was set and they are neither following siblings, nor parents of any other PSM class, so they can not contribute to $S^{adj-class}$. The benefits of the initial interpretation can be clearly seen in Figure 4.1(c), where $t = 0.5$. In that case, the initial interpretation is set for PSM classes *Customer*, *ItemInfo* and *OrderRequest* which has a positive impact on the local precision.

Experiments with EuroPass. The second experiment is based on *Europass* XML schema[2] – an official EU XML standard for the employment domain. As

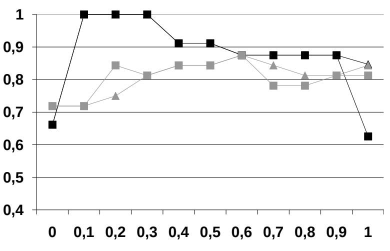
[2]`http://europass.cedefop.europa.eu/xml/CVLPSchema_V2.0.xsd`



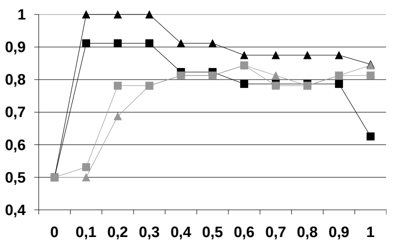
(a) $\mathfrak{C}_1$



(b) $\mathfrak{C}_2$



(c) $\mathfrak{C}_3$ $t = 0.5$



(d) $\mathfrak{C}_3$ $t = 0.75$

Figure 4.1: Experiments with sample PIM and PSM

a PIM we used the schema depicted in Figure 4.4. It was manually constructed from the Europass PSM. We have firstly converted each PSM component to a corresponding PIM component. Then, we edited the resulting PIM to create various name and structural mismatches.

This time $w^{iclass} = 0.6$ was the best option. It means that we consider both attribute and string similarities of PSM and PIM classes. In this case, the various combinations of neighbors of the interpreted class influence $\mathcal{P}_G$ only very slightly. Nevertheless, we can see a significant improvements in $\mathcal{P}_L$ in Figures 4.5(a) – 4.5(c). Figure 4.5(d) is same as Figure 4.5(b) because $t = 0.75$ is too high, so no initial interpretations were set.

Experiments with OpenTravel. Finally, we experimented with the OpenTravel standard [3] which provides a set of XML schemas for the travel community. We have selected a particular XML schema [4] which specifies an XML format for flight details. We derived a PSM from this XML schema. In the experiment, we constructed an interpretation of the PSM against a PIM which was constructed from another standard for flight details taken from *FlightStats.com* portal[5]. Therefore, we obtained independent PSM and PIM and simulated a real situation.

The results of the experiment are depicted in Figure 4.6. The charts show two settings of w^{iclass} where we achieved highest global and local precisions. The squares are for 0.7 and triangles are for 0.8. The charts again show how expansion of $I_{|C'}$ with $nbhr - psiblings(C')$ and initial interpretations, i.e. $nbhr -$

[3]`http://opentravel.org`

[4]`http://opentravel.org/2009A/FS_OTA_AirDetailsRS.xsd`

[5]`https://www.flightstats.com/developers/bin/download/Web+Services/WSDL/FlightAvailabilityService.xsd`

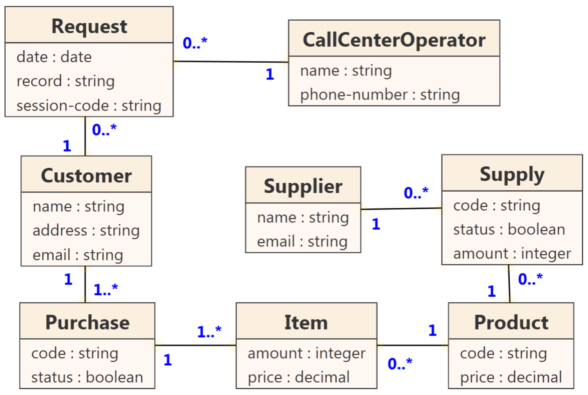
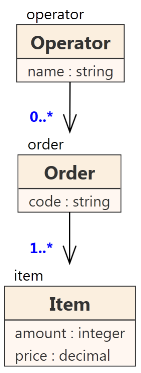
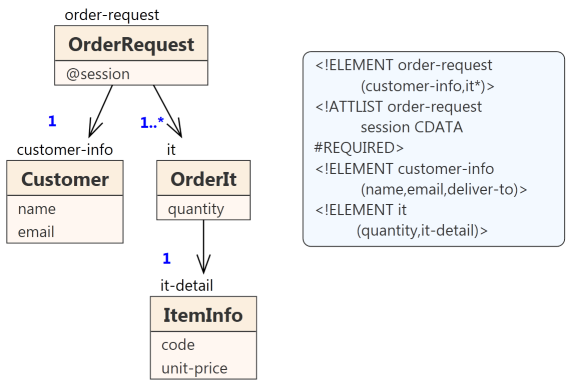


Figure 4.2: A sample PIM



(a) Sample PSM 1



```
<!ELEMENT order-request
        (customer-info,it*)>
<!ATTLIST order-request
        session CDATA
#REQUIRED>
<!ELEMENT customer-info
        (name,email,deliver-to)>
<!ELEMENT it
        (quantity,it-detail)>
```

(b) Sample PSM 2 and its partial DTD translation

Figure 4.3: Sample PSMs

$fsiblings(C')$, and $nbhr - parent(C')$, improves global as well as local precision.

4.4.1 Implementation Issues

In the description of our framework, we did not consider computational complexity which, of course, needs to be taken into account when implementing specific methods. For example, when computing the distance $\mu(C, D)$ of two PIM classes C and D, we cannot get the actual set of all PIM paths between C and D, because their number can be exponential with regard to the number of PIM associations. However, we can still find, e.g., the shortest one (according to a weight function like (c-4.2.5.1)) using, e.g., BFS (breadth-first search) or Dijkstra's algorithm. Unfortunately, this is not the case for functions (c-4.2.5.2) and (c-4.2.5.3), since we need to have a whole PIM path to compute its length and all of them to pick their minimum.

4.5 Related Work

The recent literature has focused on discovery of mappings of XML formats to a common model. We can identify several motivations. Firstly, XML schemas are hardly readable and a friendly graphical notation is necessary [42, 51, 133]. A

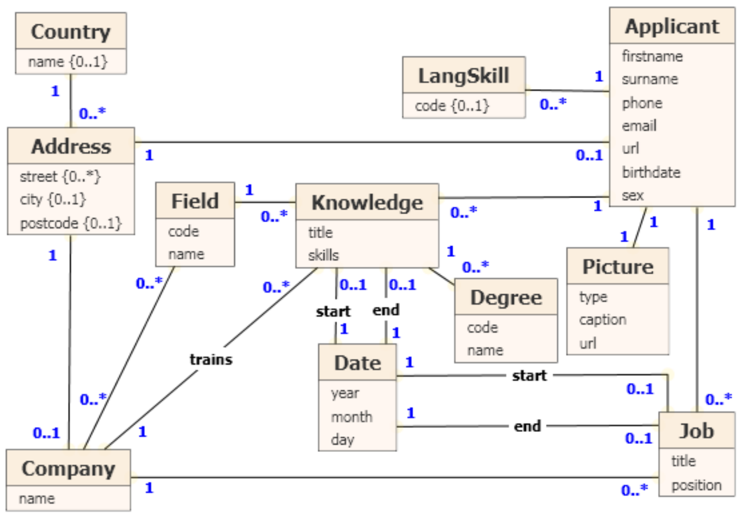


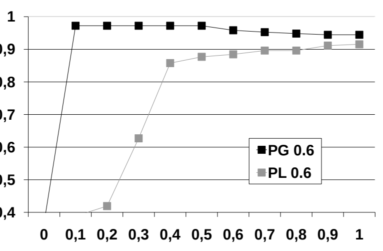
Figure 4.4: Europass PIM

survey of these approaches can be found in [137]. They introduce an algorithm for automatic conversion of a given XML schema to a UML class diagram. The result exactly corresponds to the given XML schema. However, these approaches can not be applied in our case – we need to map an XML schema to an existing conceptual schema.

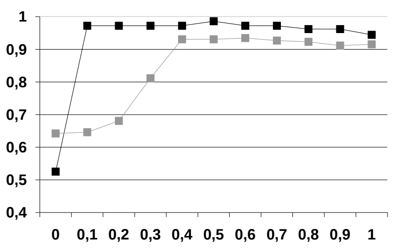
Secondly, there are approaches aimed at an integration of a set of XML format into a common abstract XML format. These works include, e.g. the *DIXSE* framework [116] or *Xyleme* project [115]. The mappings are discovered automatically and can then be checked by a domain expert who can also specify additional mappings manually [116]. These approaches are closer to our work. However, they do not consider mapping of XML formats to a more general conceptual schema and they do not consider a domain expert participating directly in the mapping discovery process.

Last but not least, there are approaches that convert or map XML formats to ontologies. For example, *DTD2OWL* [129] presents a simple method of automatic translation of an XML format with a DTD into an ontology. More advanced methods can be found in [40, 135]. All these approaches are close to ours since a conceptual schema can be understood as an ontology. In the latter two cases, the domain expert can edit the discovered mappings, but is not involved in the discovery process directly.

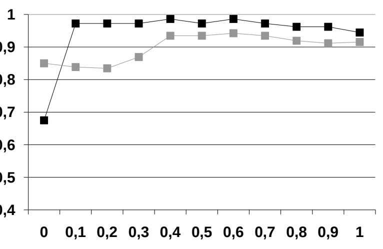
Most of the approaches widely exploit research results of the schema-matching community, where similarity of XML schemas is evaluated using syntactic and semantic similarity of strings, structural similarities etc. [70, 17, 108] Surveys and comparisons of these approaches can be found in [121, 41, 5]. The purpose of our work is not to introduce new similarity methods, but to exploit and adapt existing ones when mapping XML formats to the conceptual schema. We also extend these methods with active participation of the domain expert and we have implemented a framework that enables one to exchange the basic similarity measures with advanced ones. In [87] an algorithm which discovers mappings of XML formats to a conceptual schema was introduced. However, it was a theoretical paper without implementation, the time complexity was too high and user interaction was not possible easily. In this chapter, we introduced its optimization in all the



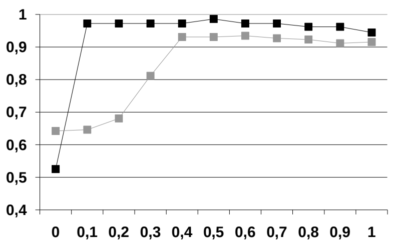
(a) $\mathfrak{C}_1$



(b) $\mathfrak{C}_2$



(c) $\mathfrak{C}_3$ $t = 0.5$



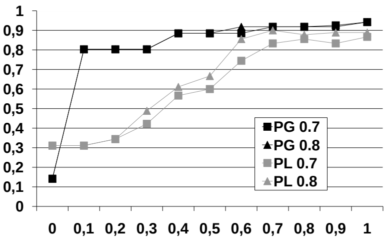
(d) $\mathfrak{C}_3$ $t = 0.75$

Figure 4.5: Experiments with EuroPass XML schema

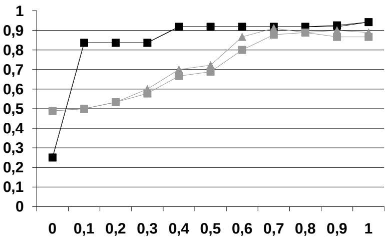
mentioned weak points.

4.6 Conclusion

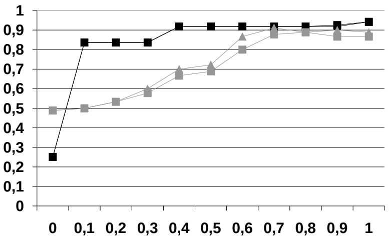
In this chapter we focused on one of the problems of MDA – finding the optimal mapping between PIM and PSM levels. We implemented a general framework which enables a user to find the mappings efficiently using similarity matching which suggest the user the mapping candidates ordered by their relevance. The framework enables one to select from various types of similarity metrics, combine them using several possible strategies and further influence the process using thresholds and continuous decisions. Using a set of experiments we showed that this is a reasonable strategy, since various types of data require different settings.



(a) $\mathfrak{C}_1$



(b) $\mathfrak{C}_2$



(c) $\mathfrak{C}_3$ $t = 0.5$

Figure 4.6: Experiments with OpenTravel XML schema

5. XML Schema Integration with Reusable Schema Parts

In this chapter, we complement our previous work in the area of *XML schema integration* described in Chapter 3 and Chapter 4 with additional methods for schema integration which exploit reusable schema parts that quite often appear in XML schemas. This further helps a domain expert to get a precise mapping to a conceptual schema, which then integrates the XML formats and facilitates their evolution - a change that is made once in the conceptual schema is propagated to the XML formats.

The contents of this chapter was published as a conference paper *XML Schema Integration with Reusable Schema Parts*[1] [56] in Dateso 2011 Annual International Workshop on DAtabases, TExts, Specifications and Objects (DATESO 2011)

5.1 Introduction

Contributions In practice, a conceptual schema and XML schemas exist separately, i.e. there are no mappings between both levels. This disallows to exploit the integration and evolution capabilities of our framework. In our work [96], we have introduced a method for deriving required XML schemas from the conceptual schema and in [60] and [57] we have described a reversed method for mapping of an existing XML schema to the conceptual schema. In this chapter, we extend this method by utilizing inheritance constructs that often appear in XML schemas and that are supported by our conceptual model to get even better results and more comfortable way of integrating them.

Outline The rest of the chapter is organized as follows. Section 5.2 briefly describes the algorithm from Chapter 3 and Chapter 4 which assists a domain expert during mapping discovery and we enhance it with methods for dealing with inheritance. In Section 5.3, we evaluate the presented approach. Finally, Section 5.4 concludes.

5.2 Algorithm

In this section we will enhance our interpretation reconstruction algorithm first introduced in Chapter 3 and extended to a framework in Chapter 4 so that it takes into account for reusable schema parts. These are represented in our conceptual model as *structural representants.*

The algorithm builds an interpretation I of a PSM schema against a PIM schema. I must be correct, it must fulfill Definition 3.4. Moreover, it must be correct in the conceptual sense, i.e. a PSM component and its PIM interpretation must conceptually correspond to the same real-world concept. We ensure the formal correctness. The conceptual correctness is ensured by a domain expert.

[1]`http://ceur-ws.org/Vol-706/paper03.pdf`

5.2.1 Overview

The basic algorithm works in three phases. Firstly, it measures initial similarities between PSM and PIM attributes and classes. Secondly, it creates an *initial interpretation* of PSM classes, whose initial similarity to some PIM class is higher than a given threshold. Becasue this is done automatically, there is a possibility that this initial interpretation is not correct. Therefore, it has to be verified by a domain expert. Nevertheless, the initial interpretation usually helps to avoid confirming lots of obvious mapping matches because the domain expert just needs to confirm a list of pre-mapped classes (or uncheck the incorrect ones). The confirmed initial interpretation now becomes a final interpretation and the algorithm moves to its third phase. It builds interpretation of the unmapped PSM classes with an assistance of a domain expert.

We will suppose a PSM schema $\mathcal{S}'$ and a PIM schema $\mathcal{S}$ on the input. The output of the algorithm is an interpretation I of $\mathcal{S}'$ against $\mathcal{S}$. We will enhance parts of the algorithm where the knowledge of reusable schema parts (structural representants) can help. But first, let us motivate a definiton. Let C' be a structural representant of C''. Due to condition 2 of Definition 3.4, the following must hold: $I(C') = I(C'')$. This means that both C' and C'' need to have the same interpretation in the PIM schema (or both must remain uninterpreted). This also means (from condition 3 of Definition 3.4 and from the definition of $context'(C')$), that PSM attributes of C' and C'' can only have attributes of the same PIM class as interpretation. Intuitively, C' and C'' represent the same concept in the PSM schema and we can suppose that their names also refer to the same concept. Note that the same goes for every PSM class C''', that would be a structural representant of C'. This justifies the follwing definition.

Definition 5.1 *Let a function* $ss' : \mathcal{S}'_c \rightarrow 2^{(\mathcal{S}'_c)}$ *return for each PSM class* C' *a set of PSM classes, which are (transitively) related to* C' *by the* structural representative *(*$repr'$*) relation.*

For example, let C'_1, C'_2, C'_3 and C'_4 be PSM lasses. In addition, let $repr'(C'_1) = \lambda$, $repr'(C'_2) = C'_1$, $repr'(C'_3) = C'_1$ and $repr'(C'_4) = \lambda$. Then $ss'(C'_1) = ss'(C'_2) = ss'(C'_3) = \{C'_1, C'_2, C'_3\}$ and $ss'(C'_4) = \emptyset$.

5.2.2 Measuring Initial Similarity

Attributes. Firstly, the algorithm measures a similarity for each pair of one PIM and one PSM attribute. This is based on their names and datatypes. This phase is not affected by the structural representants and we can skip the detailed description. Suffice to say that results of initial attribute similarity are used in function $S^{init-attrs}(C', C)$ below, which gives us similarity of a PSM class and a PIM class based on their attributes.

Classes. Let $(C', C) \in \mathcal{C}' \times \mathcal{C}$. The similarity between C' and C is customizable, in this chapter it is a weighted sum

$$\begin{aligned} S^{init-class}(C', C) &= w^{init-class} * S^{init-attrs}(C', C) \\ &+ (1 - w^{init-class}) \; * \; max\{S^{str}(name'(C'), name(C)), S^{str}(xml'(C'), name(C))\} \end{aligned}$$

where $w^{init-class} \in (0,1)$ is a weighting factor and $xml'(C')$ is a name of the parent association of C' if any exists. $S^{init-attrs}(C', C)$ is defined as $S^{init-attrs}(C', C)$ $= \sum_{A' \in attributes'(C')} max_{A \in attributes(C)}$ $(S^{init-attr}(A', A))$, i.e. it finds for each PSM attribute $A' \in attributes'(C')$ the most similar PIM attribute A of C and summarizes these similarities.

This is the first place where we can exploit structural representants. For a PSM class C', we can take attibutes of every $C'' \in ss'(C')$, because if those classes have an interpretation, it is the same PIM class for all of them (and similarly for the attributes). Therefore, we define function $attrs_{sr} : \mathcal{S}'_c \rightarrow 2^{(\mathcal{S}'_a)} = \cup_{C'_i \in ss'(C')} attributes'(C'_i)$ and we can redefine:
$S^{init-attrs}(C', C) = \sum_{A' \in attrs_{sr}(C')} max_{A \in attributes(C)}$ $(S^{init-attr}(A', A))$

5.2.3 Initial interpretation

The initial class interpretations are set according to the initial class similarities pre-computed in the previous step. It is a simple procedure that takes the most similar pairs of PSM and PIM classes (with similarity greater than a given threshold) and sets these pairs as initial interpretations. Here is another place were we exploit structural representants. Because of the fact that all PSM classes of $ss'(C')$ need to have the same interpretaion (or none at all), when we initially interpret one of them, we can as well initially interpret all of them and the interpretation will be the same PIM class. And, of course, due to the possibility that this interpretation is incorrect, we can provide the user with the comfort of accepting/rejecting the whole group at once. If the domain expert chose to consider structural representatives in both the attribute similarity and the name similarity, this is an effect of the previous modification. The reason for this is that all of the classes from the group will have the same initial similarities, because when we computed the initial similarities for one class from the group, we included all the other classes as well. If, however, the domain expert chose to ignore structural representants at some stage, the similarities will be different and this adjustment may come in handy.

5.2.4 Final Interpretation

The third part of the algorithm iteratively traverses the PSM classes in $\mathcal{S}'_c$ in pre-order and helps the domain expert to build the final interpretation. Individual steps are shown in Algorithm 2. For an actual PSM class $C' \in \mathcal{S}'_c$, the algorithm firstly constructs $I(C')$ (lines 2 - 6) and also sets the interpretation for all the PSM classes of $ss'(C')$ (lines 7 - 9). This is because all of them must have the same interpretation. Secondly, the algorithm constructs $I(A')$ for each $A' \in attributes(C')$ (lines 10 - 22). Finally, it constructs $I(R')$ for each $R' \in content(C')$ (lines 23 - 25). It can be shown that this algorithm runs in $O(N^3)$ where N is the number of PSM classes and in $O(n \times log(n))$ where n is the number of PIM classes.

Class Interpretation To construct $I(C')$, the algorithm firstly computes similarity $S^{class}(C', C)$ for each $C \in \mathcal{S}'_c$ at line 3. It is a weighted sum of two similarities. The former is the initial similarity $S^{init-class}(C', C)$. The other is a

Algorithm 2 Interpretation Construction Algorithm

```
1: for all $C' \in \mathcal{S}'_c$ in post-order do
2:   for all $C \in \mathcal{S}_c$ do
3:     $S^{class}(C', C) \leftarrow w^{class} * S^{init-class}(C', C) + (1 - w^{class}) * \frac{1}{S^{adj-class}(C',C)}$
4:   end for
5:   Offer the list of PIM classes sorted by $S^{class}$ to the domain expert.
6:   $I(C') \leftarrow C$ where $C \in \mathcal{S}_c$ is the PIM class selected by the domain expert.
7:   for all $C'' \in ss'(C')$ do
8:     $I(C'') \leftarrow C$ {here we set the interpretation for the whole group of PSM classes}
9:   end for
10:  for all $A' \in attributes(C')$ do
11:    for all $A \in \mathcal{S}_a$ do
12:      $S^{attr}(A', A) \leftarrow w^{attr} * S^{init-attr}(A', A) + (1 - w^{attr}) * \frac{1}{\mu(I(C'),class(A))+1}$
13:    end for
14:    Offer the list of PIM attributes sorted by $S^{attr}$ to the domain expert.
15:    $I(A') \leftarrow A$ where $A \in \mathcal{S}'_a$ is the PIM attribute depicted by the domain expert.
16:    if $I(class'(A')) \neq class(A)$ then
17:      Create PSM class $D' \in \mathcal{S}'_c$; $I(D') \leftarrow class(A)$
18:      Put $A'$ to $attributes'(D')$
19:      Create PSM association $R' = (C', D') \in \mathcal{S}'_r$
20:      Put $R'$ at the beginning of $content'(C')$.
21:    end if
22:  end for
23:  for all $R' \in content'(C')$ do
24:    $I(R') \leftarrow P$ where $P$ is the PIM path connecting $I(C')$ and $I(child'(R'))$ s.t. $\mu(I(C'), I(child'(R')))$ is minimal.
25:  end for
26: end for
```

reversed class similarity adjustment $S^{adj-class}(C', C)$ which we discuss in a while. The algorithm then sorts the PIM classes by their similarity with C' and offers the sorted list to the domain expert at line 5. The expert selects a PIM class from the list and the algorithm sets $I(C')$ to this selected class at line 6.

Class similarity adjustment Similarity adjustment $S^{adj-class}(C', C)$ is computed on the base of the completed part of I, which includes confirmed initial interpretation. $S^{adj-class}(C', C)$ is a combination of distances between C and PIM classes D_i which are interpretations of the interpreted neighbors of C'. $\mu(C, D)$ is the distance between PIM classes C and D.

Note that Algorithm 2 is a skeleton which needs to be supplemented with methods for (1) measuring distances between PIM classes, (2) combining distances, and (3) selecting candidates for C' structural similarity adjustment. In this chapter, we use basic methods to show that the general idea works. For measuring the distance between two PIM classes C and D, we use the length of the shortest PIM path connecting C and D. As the distance combination method, which results in the aimed $S^{adj-class}(C', C)$, we can also choose from various possibilities. In this chapter, we use

$$S^{adj-class}(C', C) = (\sum_{i=1}^{n} \frac{\mu(C, I(D'_i))}{n}) + 1$$

where $D'_1, \ldots, D'_n$ are the selected interpreted neighbors of C'. $S^{adj-class}(C', C)$ is the average of the lengths of the shortest PIM paths between C and each $I(D'_i)$.

Finally, we need to decide which mapped neighbors of C' will be selected to compute $S^{adj-class}(C', C)$. We can choose among children of C' or previous siblings of C', as these were already interpreted by the domain expert in this part of the algorithm. Because we have some PSM classes interpreted via the *initial interpretation*, we can use them as another candidates for structural similarity adjustment, if they are close enough. Therefore, we can also select interpreted following siblings, interpreted parent or interpreted ancestors as candidates for structural similarity adjustment. These options are described and experimented with in [57].

Here is another moment where we can exploit reusable schema parts in a form of structural representants. As we choose which interpreted neighbors of C' to use for the structural similarity adjustment, we can also work with the same type of interpreted neighbors of all classes of the group $ss'(C')$. The reasons are the same, because the interpretation of all classes of the group must be the same PIM class.

The rest of the algorithm remains unaffected by the structural representatives, so we describe it only briefly. For details, see [60, 57]. When all the PSM classes have been interpreted or the domain expert decided they should remain uninterpreted, a similar process is performed for PSM attributes of the classes. The possibilities of mapping a PSM attribute in this situation are limited due to the rules that the interpretation must adhere to (see Definition 3.4). Finally, PSM associations are interpreted with respect to the same rules.

5.3 Evaluation

In this section, we briefly evaluate the effect of structural representants on building interpretations of PSM classes. For more detailed experiments with the overall method see [60, 57]. We have implemented the introduced method in our tool XCase[2] which was primarily intended for designing XML schemas from a created PIM schema.

Let us suppose an actual PSM class C'. Let the domain expert set $I(C')$ to a PIM class C either when asked or when confirming the initial interpretation. We measure the precision of the algorithm from two points of view. Firstly, we measure the position of C in the list of PIM classes offered to the expert sorted by their S^{class}. We call this precision a *global precision* $\mathcal{P}_G$:

$$\mathcal{P}_G = ((\sum_{C' \in \mathcal{S}'_c} 1 - \frac{order(C) - 1}{n})/n') * 100$$

where n denotes the size of $\mathcal{S}_c$, n' denotes the size of $\mathcal{S}'_c$, and $order(C)$ denotes the order of C in the list. If there are more PIM classes with the same similarity to C', $order(C)$ is the order of the last one. $\mathcal{P}_G = 0$ (resp. 1) if for each PSM class C', the selected PIM class was the last (resp. first).

The global precision is not sufficient. When C is the first class, there can be other PIM classes before C which have their similarity to C' close to $S^{class}(C', C)$

[2]http://xcase.codeplex.com

and make it harder to distinguish whether C is or is not a good match for C'. We therefore propose another metric called *local precision* which measures the amount of PIM classes with their similarity to C' close to $S^{class}(C', I(C'))$. It is defined as

$$\mathcal{P}_L = ((\sum_{C' \in \mathcal{S}'_c} 1 - \frac{close(C) - 1}{n})/n') * 100$$

where $close(C)$ denotes the number of PIM classes with their similarity to C' close to $S^{class}(C', C)$. The term *close similarity* can be defined in various ways. In this chapter, we say that y is *close* to x if $y \in (x - 0.1, x + 0.1)$.

Intuitively, the effect of using structural representants is a reduction of the number of mapping offers the domain expert needs to go through. This is because when an interpretation of a PSM class C' is constructed, it is automatically constructed for all PSM classes $ss'(C')$ and the domain expert no longer needs to create the interpretation for each one of them.

Additionally, the use of structural representants for class similarity computations may help with global and local precisions. This is, however, dependent on the texts present in the source PSM schema, its structure and selected methods of similarity measurements, which so far can not be determined automatically. Therefore, the experimental results are very complex and their description would not fit into this article.

5.4 Conclusion

In this chapter, we studied the effect of exploiting reusable schema parts on techniques used for mapping of XML formats to a conceptual schema. We briefly described our basic algorithm from [60, 57] which allows to exploit various similarity measurement methods. Then we introduced our enhancements that allow us to take advantage of the reusable schema parts, which are expressed as structural representants in our conceptual model. Finally, we have provided a brief evaluation of the proposed method.

6. Refined Conceptual Model for XML

In this chapter we refine the conceptual model for XML, which is the basis of all the XML schema evolution work in this thesis. It can be described at the conceptual level using a conceptual schema. We follow the MDD principle which is based on modeling the application domain at several levels of abstraction. The most abstract level we adopt in this thesis is the *platform-independent level*. It contains the conceptual schema of the problem domain. The language applied to express the conceptual schema is called *platform-independent model (PIM)* and the conceptual schema is then called *schema in the platform-independent model (PIM schema)*. The level below is the *platform-specific level* which specifies how the whole or part of the PIM schema is represented in a particular platform. In our case, the platform is the XML. The language applied at this level is called *platform-specific model (PSM)* and a schema expressed in this language is called *schema in the platform-specific model (PSM schema)*.

The content of this chapter is based on the refined conceptual model for XML as defined using the Regular Tree Grammars [81] formalism and published in our impacted journal paper *When Conceptual Model Meets Grammar: A Dual Approach to XML Data Modeling*[1] [96] - Data & Knowledge Engineering (DKE). The form present in this chapter was published as a part of another impacted journal paper [89] which is in Chapter 8. Because other contributions use this version of the conceptual model too, we present it as a separate chapter.

6.1 Platform-Independent Model

A schema in the platform-independent model (PIM) models real-world concepts and the relationships between them without any details of their representation in a specific data model (XML in our case). As a PIM, we use the classical model of UML class diagrams [101, 102]. For simplicity, we use only its basic constructs: classes, attributes and binary associations. UML is widely supported by the majority of tools for data engineering and the XMI [105] standard is used for exchanging diagrams between them; it is, therefore, natural to use UML in our approach as well.

Definition 6.1 *A* platform-independent schema (PIM schema) *is a triple* $\mathcal{S} = (\mathcal{S}_c, \mathcal{S}_a, \mathcal{S}_r)$ *of disjoint sets of* classes, attributes*, and* associations*, respectively.*

- Class $C \in \mathcal{S}_c$ *has a name assigned by function* name*.*
- Attribute $A \in \mathcal{S}_a$ *has a name, data type and cardinality assigned by functions* name, type*, and* card*, respectively. Moreover, A is associated with a class from* $\mathcal{S}_c$ *by function* class*.*
- Association $R \in \mathcal{S}_r$ *is a set* $R = \{E_1, E_2\}$*, where* E_1 *and* E_2 *are called* association ends *of R. R has a name assigned by function* name*. Both* E_1

[1] http://dx.doi.org/10.1016/j.datak.2011.09.002

and E_2 *have a cardinality assigned by function* card *and are associated with a class from* $\mathcal{S}_c$ *by function* participant. *We will call* participant(E_1) *and* participant(E_2) participants *of* R. name(R) *may be undefined, denoted by* name$(R) = \lambda$.

For a class $C \in \mathcal{S}_c$*, we will use* attributes(C) *to denote the set of all attributes of* C*, i.e.* attributes$(C) = \{A \in \mathcal{S}_a : \text{class}(A) = C\}$. *Similarly,* associations(C) *will denote the set of all associations with* C *as a participant, i.e.* associations(C) $= \{R \in \mathcal{S}_r : (\exists E \in R)(\text{participant}(E) = C)\}$.

PIM schema components have usual semantics: a class models a real-world concept, an attribute of that class models a property of the concept, and, an association models a kind of relationships between two concepts modeled by the connected classes. A sample PIM schema modeling our sample domain of products being sold is depicted in Figure 6.1. We display PIM schemas as UML class diagrams. We omit displaying data types of class attributes. When a cardinality of a class attribute or association endpoint is not displayed, it is `1..1` by default.

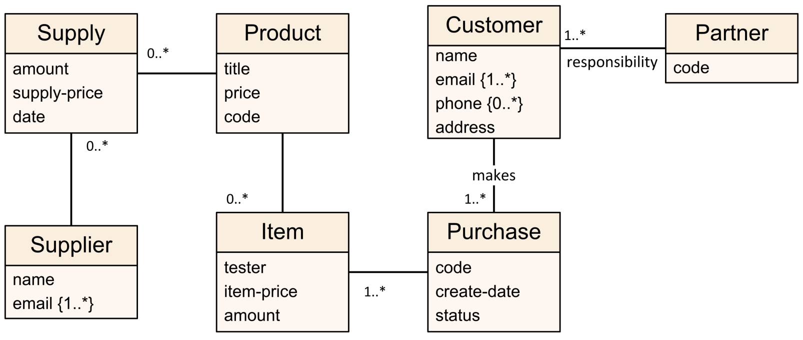


Figure 6.1: PIM schema modeling the domain of selling products

6.2 Platform-Specific Model

A schema in the platform-specific model (PSM) describes how a part of the reality modeled by the PIM schema is represented with a particular XML schema. For each aimed XML schema a separate PSM schema is created. As a PSM we use UML class diagrams extended for the purposes of XML modeling. The extension is necessary because of several specifics of XML (such as hierarchical structure or distinction between XML elements and attributes) which cannot be modeled by standard UML constructs.

Definition 6.2 *A* platform-specific schema (PSM schema) *is a 5-tuple* $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{\mathcal{S}'})$ *of disjoint sets of* classes, attributes, associations, *and* content models, *respectively, and one specific class* $\mathcal{C}'_{\mathcal{S}'} \in \mathcal{S}'_c$ *called* schema class.

- Class $C' \in \mathcal{S}'_c$ *has a name assigned by function* name.
- Attribute $A' \in \mathcal{S}'_a$ *has a name, data type, cardinality and XML form assigned by functions* name, type, card *and* xform, *respectively.* xform(A') $\in \{\text{e}, \text{a}\}$. *Moreover, it is associated with a class from* $\mathcal{S}'_c$ *by function* class *and has a position assigned by function* position *within the all attributes associated with class*(A').

- Association $R' \in \mathcal{S}'_r$ *is a pair* $R' = (E'_1, E'_2)$, *where* E'_1 *and* E'_2 *are called* association ends *of* R'. *Both* E'_1 *and* E'_2 *have a cardinality assigned by function* card *and each is associated with a class from* $\mathcal{S}'_c$ *or content model from* $\mathcal{S}'_m$ *assigned by function* participant, *respectively. We will call* participant(E'_1) *and* participant(E'_2) parent *and* child *and will denote them by* parent(R') *and* child(R'), *respectively. Moreover,* R' *has a name assigned by function* name *and has a position assigned by function* position *within the all associations with the same* parent(R'). name(R') *may be undefined, denoted by* name$(R') = \lambda$.

- Content model $M' \in \mathcal{S}'_m$ *has a content model type assigned by function* cmtype. cmtype$(M') \in \{$`sequence`, `choice`, `set`$\}$.

The graph $(\mathcal{S}'_c \cup \mathcal{S}'_m, \mathcal{S}'_r)$ *must be a forest*[2] *of rooted trees with one of its trees rooted in* $C'_{\mathcal{S}'}$. *For* $C' \in \mathcal{S}'_c$, *attributes* (C') *will denote the sequence of all attributes of* C' *ordered by* position, *i.e.* attributes $(C') = (A'_i \in \mathcal{S}'_a : \text{class}(A'_i) = C' \wedge i = \text{position}(A'_i))$. *Similarly,* content (C') *will denote the sequence of all associations with* C' *as a parent ordered by* position, *i.e.* content $(C') = (R'_i \in \mathcal{S}'_r : \text{parent}(R'_i) = C' \wedge i = \text{position}(R'_i))$. *We will call* content (C') content of C'. *With anc*(X') *we will denote the set of all ancestor classes of a component* X' *in* $\mathcal{S}'$.

To distinguish PIM components from PSM components, we strictly use a notation without the **'** symbol for PIM components (e.g. class `Purchase`) and notation with the **'** symbol for PSM components (e.g. class `Purchase'`). Before showing sample PSM schemas, we explain the semantics of the PSM constructs. We view a PSM schema $\mathcal{S}'$ from two perspectives: *grammatical* and *conceptual*. From each perspective, the constructs have a different semantics.

From the *conceptual perspective*, $\mathcal{S}'$ is mapped to a PIM schema $\mathcal{S}$ and models the same part of the reality as $\mathcal{S}$. More precisely, some classes, attributes and associations of $\mathcal{S}'$ are mapped to some classes, attributes, and associations of $\mathcal{S}$, respectively. These mapped components of $\mathcal{S}'$ model exactly the same part of the reality as do their corresponding counterparts in $\mathcal{S}$. The rest of $\mathcal{S}'$ has no semantics from the conceptual perspective.

From the *grammatical perspective*, $\mathcal{S}'$ models an XML schema. Its components model XML attributes and XML elements, and their structure. We summarize XML constructs modeled by PSM constructs in Table 6.1. Formally, $\mathcal{S}'$ unambiguously models a regular tree language which can be specified by a regular tree grammar [81]. However, this formalism is not a part of this thesis. For the details on the modeled regular tree language and formal proofs of unambiguity we refer to our previous work [92], where we proved that our PSM is equivalent to regular tree grammars. In other words, it can be equivalently used as an XML schema language. We showed how a PSM schema can be unambiguously translated to an expression in a selected XML schema language and vice versa. The important consequence of our previous results for this chapter is that we can abstract our evolution mechanism from particular XML schema languages and work only at the PSM level.

[2]Note that since $\mathcal{S}'$ is a forest, we could model R' directly as a pair of connected components. However, we use association ends to unify the formalism of PSM with the formalism of PIM.

PSM construct	**Modeled XML construct**
$C' \in \mathcal{S}'_c$	Complex content which is a sequence of XML attributes and XML elements modeled by attributes in *attributes* (C') followed by XML attributes and XML elements modeled by associations in *content* (C')
$A' \in \mathcal{S}'_a$, where $xform(A') = a$	XML attribute with name $name(A')$, data type $type(A')$ and cardinality $card(A')$
$A' \in \mathcal{S}'_a$, where $xform(A') = e$	XML element with name $name(A')$, simple content with data type $type(A')$ and cardinality $card(A')$
$R' \in \mathcal{S}'_r$, where $name(R') \neq \lambda$	XML element with name $name(R')$, complex content modeled by $child(R')$ and cardinality $card(R')$. If $R' \in$ *content* $(C'_{\mathcal{S}'})$, R' models a root XML element
$R' \in \mathcal{S}'_r$, where $name(R') = \lambda$	Complex content modeled by $child(R')$
$M' \in \mathcal{S}'_m$ and $cmtype(M') =$ `sequence` (or `choice` or `set`)	Complex content which is a sequence (or choice or set, respectively) of XML attributes and XML elements modeled by associations in *content* (C')

Table 6.1: XML attributes and XML elements modeled by PSM constructs

If we put both perspectives together, the PSM schema $\mathcal{S}'$ specifies how the corresponding part of the PIM schema $\mathcal{S}$ is represented in the XML schema. In other words, it specifies how a part of the real world modeled by $\mathcal{S}$ is represented in XML documents valid against the XML schema. Conversely, it specifies the semantics of the XML schema in terms of $\mathcal{S}$, i.e. the semantics of a particular XML document in terms of the PIM schema.

Three sample PSM schemas are depicted in Figure 6.2. We display PSM schemas as UML class diagrams with some extended notation. First, they are displayed in a tree layout; attributes and associations are sorted in the order given by *position*. Second, attributes with XML form *a* are displayed with the `@` symbol. Third, sequence, choice and set content models are displayed as rounded boxes with an inner symbol ..., | or {}, respectively.

From the conceptual perspective, our sample PSM schemas are mapped to a part of the PIM schema in Figure 6.1. We display the components mapped to the PIM schema in the sea shell color. The mapping is intuitive[3] and we do not display it explicitly. The components which are not mapped are displayed in grey. For example, the PSM class `Purchase'` in Figure 6.2 (a) is mapped to the PIM class `Purchase`. In other words, the semantics of `Purchase'` is the same as the semantics of `Purchase` which models purchases. Similarly, attribute `name'` of class `Customer'` is mapped to `name` of class `Customer`. Association `cust'` connecting classes `Purchase'` and `Customer'` is mapped to association `makes` connecting classes `Purchase` and `Customer`. On the other hand, PSM class `Contact'` is not mapped to the PIM schema. In other words, it has no semantics from the conceptual point of view. Similarly, the association with `Contact'` as child is not mapped. And, attribute `version'` of `Purchase'` is not

[3]The reader may deduce it from their names which intuitively suggest the mapping

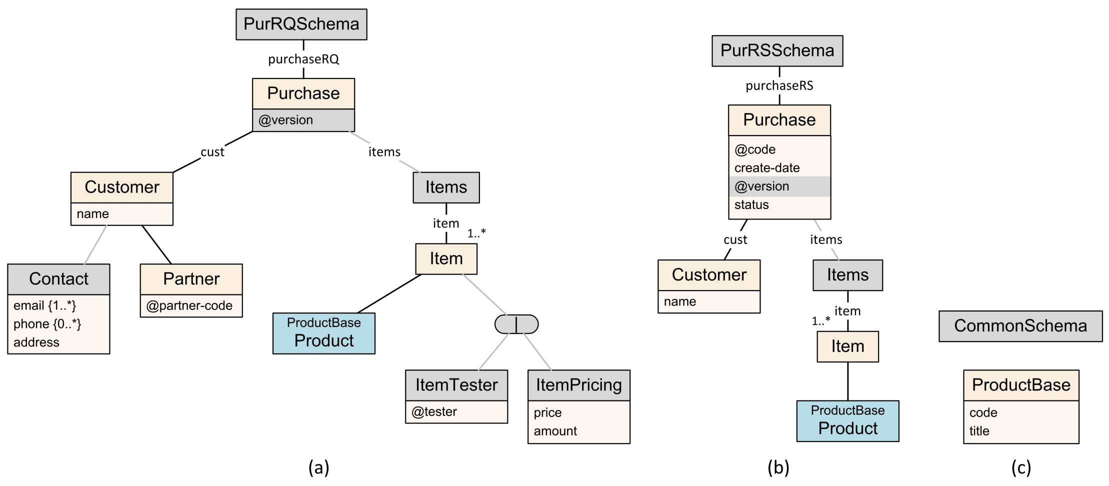


Figure 6.2: PSM schema modeling (a) XML format for purchase requests received from customers, (b) XML format for purchase responses sent to customers, (c) components shared by other PSM schemas

mapped as well. These non-mapped components have no semantic meaning from the conceptual point of view.

From the grammatical perspective, the PSM schema depicted in Figure 6.2 (a) models an XML schema for purchase requests sent by customers to our system. A sample XML document formatted according to this XML schema is depicted in Figure 6.3. The hierarchical structure of the XML schema is modeled by the associations of the PSM schema. As can be seen from the example, each named association models an XML element whose cardinality is given by the child cardinality of the association. For example, association `items’` which connects classes `Purchase’` and `Items’` models XML element `items` with cardinality `1..1`. Association `item’` which connects classes `Items’` and `Item’` models XML element `item` with cardinality `1..*`. Moreover, when such association is in the content of the schema class, it models root XML elements. In our case, association `purchaseRQ’` models XML elements `purchaseRQ` which are root XML elements of the modeled XML format. An association without a name models only the nesting of XML content. For example, association `ItemProduct’` which connects classes `Item’` and `Product’` does not model any XML element. It specifies that the XML content modeled by its child is a part of the XML content modeled by its parent. An attribute models an XML element or XML attribute depending on its XML form. An attribute with XML form = `a` models XML attribute and is depicted by the additional symbol `@`. An attribute with XML form = `e` models XML element and is depicted without any additional symbol. Again, cardinality is given by the attribute cardinality. For example, attribute `version’` of class `Purchase’` models a mandatory XML attribute `version`. Attribute `name’` of class `Customer’` models a mandatory XML element `name` which can be repeated.

Sometimes, classes in one or more PSM schemas may share the same attributes and/or part of their content. Instead of repeating them at several places, we introduce *structural representatives* which allow for attribute and content reuse. If a class C' in a PSM schema is a structural representative of another class D' from the same or another PSM schema, C' "inherits" the attributes and content of D'. From the grammatical perspective, C' models the same XML attributes

```
<purchaseRQ version="1.0">
  <cust partner-code="PA1">
    <name>Martin Necasky</name>
    <email>necasky@...</email><address>Malostranske nam. 25, Praha, Czech Republic</address>
  </cust>
  <items>
    <item tester="true"><code>P001</code><title>Sample for testing</title></item>
    <item><code>P002</code><title>Umbrella</title><price>100</price><amount>2</amount></item>
  </items>
</purchaseRQ>
```

Figure 6.3: Sample purchase request represented in the XML format modeled by the PSM schema depicted in Figure 6.2 (a)

as D' followed by its own modeled XML attributes followed by XML elements modeled by D' and, finally, followed by its own modeled XML elements.

Definition 6.3 *Let $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{\mathcal{S}'})$ be a PSM schema and C' be a class from $\mathcal{S}'_c$. C' may be a* structural representative *of another class D' in $\mathcal{S}'_c$ which is assigned to C' by function* repr *($\text{repr}(C') = D'$). If repr(C') is undefined, denoted by $\text{repr}(C') = \lambda$, we say that C' is not a structural representative of any class. Let $\text{repr}^*(\lambda) = \{\}$ and $\text{repr}^*(C') = \{\text{repr}(C')\} \cup \text{repr}^*(\text{repr}(C'))$ where $C' \neq \lambda$. It must hold that $C' \neq \text{repr}^*(C')$.*

A structural representative C' of *repr*(C') is displayed as a class with a blue background and the name of *repr*(C') above its own name. For example, class `Product'` from Figure 6.2 (a) and class `Product'` from Figure 6.2 (b) are both structural representatives of class `ProductBase` from the PSM schema depicted in Figure 6.2 (c). From the grammatical perspective they both model the same XML fragment as the latter one. Note that the PSM schema in Figure 6.2 (c) does not model any XML documents (because it does not have any named association going from the schema class and, therefore, does not model any root XML elements). It acts as an auxiliary PSM schema which contains components shared by other PSM schemas via the mechanism of structural representatives.

In the rest of this section we further formalize the conceptual perspective. A formal model of the grammatical perspective is provided in [92] and we omit it in this chapter.

6.3 Formal Model of Conceptual Perspective

Formally, the conceptual perspective of a PSM schema is expressed as a mapping of the PSM schema to the PIM schema. Before we introduce the mapping, we introduce an auxiliary notion of a directed image of an association from a PIM schema which we use in the following definitions.

Definition 6.4 *Let $R = \{E_1, E_2\}$ be an association in a PSM schema $\mathcal{S}$. The* directed images *of R are $R^{E_1} = (E_1, E_2)$ and $R^{E_2} = (E_2, E_1)$. We will denote the set of all directed images of $\mathcal{S}$ as $\overrightarrow{\mathcal{S}_r}$, i.e. $\overrightarrow{\mathcal{S}_r} = \{R^{E_1}, R^{E_2} : R = \{E_1, E_2\} \in \mathcal{S}_r\}$.*

Now, we are ready to introduce the formalism of mappings. We call the mapping of the PSM schema to the PIM schema *interpretation of the PSM schema against the PIM schema.*

Definition 6.5 *An* interpretation *of a PSM schema $\mathcal{S}'$ against a PIM schema $\mathcal{S}$ is a partial function $I : (\mathcal{S}'_c \cup \mathcal{S}'_a \cup \mathcal{S}'_r) \rightarrow (\mathcal{S}_c \cup \mathcal{S}_a \cup \overrightarrow{\mathcal{S}_r})$ which maps a class, attribute or association from $\mathcal{S}'$ to a class, attribute or directed image of an association from $\mathcal{S}$, respectively. For $X' \in (\mathcal{S}'_c \cup \mathcal{S}'_a \cup \mathcal{S}'_r)$, we call $I(X')$* interpretation of X'. *$I(X') = \lambda$ denotes that X' does not have an interpretation. In that case we will also say that X' has an empty interpretation.*

An arbitrary interpretation of a PSM component would lead to inconsistencies between the semantics of the PIM schema and the semantics of the PSM schema given by the interpretation. This would result in ambiguities in the semantics of PSM schemas. For example, suppose the class `Product'` and its attribute `code'` from our sample PSM schema depicted in Figure 6.2 (a). Let the interpretation of `Product'` be the PIM class `Product`. Therefore, `code'`, from the conceptual perspective, belongs to `Product`. On the other hand, suppose that `code'` is mapped to the PIM attribute `code` of PIM class `Purchase`. From this, `code'` belongs to `Purchase` which is in contradiction with the previous conclusion. We, therefore, need the interpretation to meet certain rules which prevent these ambiguities.

Before we introduce the rules, let us define the notion of *interpreted context* of a PSM component.

Definition 6.6 *Let X' be a component of a PSM schema $\mathcal{S}'$. Let I be an interpretation of $\mathcal{S}'$ against a PIM schema $\mathcal{S}$. The* interpreted context *of X' with respect to I is denoted* intcontext(X') *and*

- intcontext$(X') = X'$ *when $X' \in \mathcal{S}'_c$ and $I(X') \neq \lambda$*
- intcontext$(X') = C'$ *when $X' \notin \mathcal{S}'_c$ or $I(X') = \lambda$, where C' is the closest ancestor class to X' s.t. $I(C') \neq \lambda$.*

As the definition shows, the interpreted context of each PSM component X' is X' itself if it is a class with an interpretation. In other cases, it is the closest ancestor class to X'. Let us demonstrate the notion of interpreted context on our sample PSM schema depicted in Figure 6.2 (a). The interpreted context of class `Customer′` is class `Customer′` itself (*intcontext*(`Customer′`) = `Customer′`), because I(`Customer′`) $\neq \lambda$. The interpreted context of attribute `name′` of class `Customer′` is class `Customer′` as well (*intcontext*(`name′`) = `Customer′`), because `Customer′` is the closest ancestor class to `name′` which has an interpretation. And, for the same reason, the interpreted context of association connecting classes `Customer′` and `Partner′` is class `Customer′`. On the other hand, class `Contact′` does not have an interpretation (I(`Contact′`) $= \lambda$). The closest ancestor class with an interpretation is class `Customer′`. Therefore, *intcontext*(`Contact′`) = `Customer′`. Similarly, *intcontext*(`ItemTester′`) = *intcontext*(`ItemPricing′`) = `Item′`. And the same is for attributes, for example *intcontext*(`tester′`) = `Item′`.

Note that *intcontext*(X') may be empty, i.e. *intcontext*$(X') = \lambda$. In that case we will say that X' does not have an interpreted context. Thus, having the notion of interpreted context, we are ready to introduce the rules.

We now define the notion of *consistent interpretation* of a PSM schema against a PIM schema. Consistency ensures that the semantics of the PSM schema determined by the interpretation is consistent with the semantics modeled by the PIM schema.

Definition 6.7 *Let I be an interpretation of a PSM schema $\mathcal{S}'$ against a PIM schema $\mathcal{S}$. We say that I is* consistent *if the following rules are satisifed:*

$$(\forall C' \in \mathcal{S}'_c \text{ s.t. } repr(C') \neq \lambda \wedge I(C') \neq \lambda)(I(C') = I(repr(C'))) \tag{6.1}$$

$$(\forall A' \in \mathcal{S}'_a \text{ s.t. } I(A') \neq \lambda) \tag{6.2}$$
$$(intcontext(A') \neq \lambda \wedge I(A') \in attributes(I(intcontext(A'))))$$

$$(\forall R' \in \mathcal{S}'_r \text{ s.t. } I(child(R')) = \lambda \vee I(intcontext(R')) = \lambda)(I(R') = \lambda) \tag{6.3}$$

$$(\forall R' \in \mathcal{S}'_r \text{ s.t. } I(child(R')) \neq \lambda \wedge intcontext(R') \neq \lambda) \tag{6.4}$$
$$(I(R') = (E_1, E_2) \text{ s.t. } participant(E_1) = I(intcontext(R'))$$
$$\wedge\, participant(E_2) = I(child(R')))$$

Condition (1) requires that a structural representative C' of a class $repr(C')$ has the same interpretation as $repr(C')$. This is because C' acquires the attributes and content of $repr'(C')$. To ensure consistency, the attributes and associations in the content must semantically remain with C.

Condition (2) requires that when an interpreted attribute A' has an interpreted context C', then $I(A')$ must be an attribute of $I(C')$. In other words, A' must semantically belong to the interpretation of its interpreted context.

Conditions (3) and (4) ensure consistency of associations. Condition (3) requires that only an association with an interpreted child and interpreted context may have an interpretation. This is because the semantics of an association specifies how instances of the child of the association are connected to their interpreted context. For associations with interpretation, condition (4) is applied. It is similar to (2). If an association R' has an interpreted context with interpretation C and its child has an interpretation D, the interpretation of R' must be an ordered image of an association connecting C and D.

Let us demonstrate conditions (2)-(4) on the PSM schema depicted in Figure 6.2 (a). First, suppose attribute $\texttt{tester}'$. Its interpreted context is class $\texttt{Item}'$ with $I(\texttt{Item}') = \texttt{Item}$. Condition (2) requires that $I(\texttt{tester}') \in attributes(\texttt{Item})$. This is satisfied in our case because $I(\texttt{tester}') = \texttt{tester}$. Second, suppose the association connecting classes $\texttt{Customer}'$ and $\texttt{Contact}'$. Since $I(\texttt{Contact}') = \lambda$, condition (3) requires that the association does not have an interpretation. This is natural, because both $\texttt{Contact}'$ represents a part of class `Customer` from the conceptual perspective and, therefore, it is meaningless to specify the semantics of the association. On the other hand, the association connecting classes $\texttt{Customer}'$ and $\texttt{Partner}'$ must have an interpretation, because both classes have an interpretation and it is necessary to specify the semantics of the connection between them. The interpretation must be an association connecting `Customer` and `Partner` according to condition (3). In our case it is the association `responsibility` which is correct.

The following lemma shows that Definition 6.7 is correct.

Lemma 6.1 *Let I be a consistent interpretation of a PSM schema $\mathcal{S}'$ against a PIM schema $\mathcal{S}$. The semantics of each component of $\mathcal{S}'$ specified by I is unambiguous.*

Proof 6.1 *We will show that the semantics of each PSM class, attribute or association in $\mathcal{S}'$ is specified unambiguously by I. Without loss of generality, we will consider components of $\mathcal{S}'$ which are semantically related to a PIM class $C \in \mathcal{S}_c$.*

First, let $C' \in \mathcal{S}'_c$ *s.t.* $I(C') = C$. *The semantics of* C' *is specified by* I *unambiguously from Definition 6.5. There is no way to use* I *to deduce that the semantics of* C' *is a class* $C_0 \neq C$.

Second, let $A' \in \mathcal{S}'_a$ *s.t.* $I(A') \neq \lambda$. *Let* $C'_A \in \mathcal{S}'_c$ *s.t.* $C'_A = \mathit{class}(A')$ *or* $\mathit{repr}(C'_A) = \mathit{class}(A')$. *Let* $I(\mathit{intcontext}(C'_A)) = C$. *If* $I(A') \notin \mathit{attributes}(C)$, *the semantics of* A' *is ambiguous. From the conceptual perspective,* A' *semantically belongs to* C *on one hand and it does not on the other. However,* $I(A') \notin \mathit{attributes}(C)$ *cannot happen because of conditions (1) and (2).*

Third, let $R' \in \mathcal{S}'_r$ *s.t.* $I(R') \neq \lambda$ *is a directed image of an association* $R \in \mathcal{S}_r$. *Let* $C'_R \in \mathcal{S}'_c$ *s.t.* $C'_R = \mathit{parent}(R')$ *or* $\mathit{repr}(C'_R) = \mathit{parent}(R')$ *or* $C'_R = \mathit{child}(R')$ *(in this last case, condition (4) ensures that* $I(C'_R) \neq \lambda$ *and, therefore,* $\mathit{intcontext}(C'_R) = C'_R$*). Let* $I(\mathit{intcontext}(C'_R)) = C$. *If* $R \notin \mathit{associations}(C)$, *the semantics of* R' *is ambiguous. From the conceptual perspective,* R' *is an association connected to* C *on one hand and it is not on the other. However,* $R \notin \mathit{associations}(C)$ *cannot happen because of conditions (1) and (3).*

In the rest of this thesis, each interpretation considered will be consistent; we do not consider inconsistent interpretations.

7. Model-Driven Approach to XML Schema Evolution

In this chapter, we briefly present a novel approach to evolution of families of XML schemas. It is based on the conceptual model for XML (see Chapter 6). The designer performs a change only once in the conceptual schema and our introduced mechanism propagates the change to all affected XML schemas. Propagation from the XML schema to the conceptual level is also supported. For a detailed description of the approach see Chapter 8.

Together with Chapter 1, this is the preliminary work in the area of *XML schema evolution*. It provides an overview of our approach. In Chapter 8 we go into greater detail.

The contents of this chapter was published as a workshop paper *Model-Driven Approach to XML Schema Evolution*[1] [95] in On the Move to Meaningful Internet Systems: OTM 2011 Workshops.

7.1 Introduction

In conceptual modeling for XML [86], the aim is at the problem of designing XML schemas of XML vocabularies. The introduced technique exploits the fact that a vocabulary is usually related to a common data domain, e.g. travel, health care or public procurement. Therefore, a conceptual schema of the domain is firstly designed. Each XML schema is then modeled as a specific view of the conceptual schema. In this chapter, we aim at the problem of evolving the XML schemas as user requirements change. A new user requirement may have an impact on several XML schemas in the vocabulary. The designer, therefore, has to identify the impacted XML schemas and determine how they must be evolved. It is a widespread practice today to deal with this task at the level of separate XML schemas expressed in an XML schema language such as DTD or XML Schema. However, this may be very difficult in case of complex XML vocabularies. Also manual identification of the affected parts is not easy in case of tens or even hundreds of schemas. In this chapter we introduce a technique based on describing the required changes at the common conceptual level and semi-automatic propagation of the changes to the affected XML schemas.

The chapter is structured as follows. Section 7.2 is motivating. In Section 7.3, we introduce atomic operations for schema evolution. In Section 7.4, we introduce the propagation mechanism. In Section 7.5, we evaluate the introduced approach. Section 7.6 concludes.

7.2 Motivation

Let us discuss the evolution problem on two scenarios. First, a designer creates a new family of XML schemas which has not been deployed in a run-time environment yet. He iterates in several iterations before an acceptable version of the

[1]http://link.springer.com/chapter/10.1007/978-3-642-25126-9_63

XML schemas is prepared to be deployed. He needs a mechanism which shows an impact of each change to the unfinished XML schemas and helps to propagate the change to the XML schemas. Because the XML schemas have not been deployed yet, there are no XML documents. Therefore, it is not necessary to propagate the changes to the XML document level. The second scenario is *adapting an existing family of XML schemas* which have already been deployed in a run-time environment. In this scenario it is necessary to consider XML documents as well. Such scenario usually occurs when new or changed requirements need to be implemented in the running system (e.g. a change in legislation). A technique for propagation of evolution changes from XML schemas to XML documents using XSLT scripts generated using the approach described in this chapter is published in [71]. In this chapter, we aim at propagation of changes between the conceptual schema and XML schemas bounded to that conceptual schema. For this, we consider a set of atomic operations which are incrementally applied by the designer to the schemas and appropriately propagated by our mechanism between the conceptual and XML schema levels. The current literature considers addition, removal, migratory, and sedentary operations for schema evolution. However, they are not sufficient, because they do not keep the semantic relationships between existing and newly created schema components. We demonstrate the problem on an example. Let us have a class `Customer`. Its attributes `line1` and `line2` represent a customer's address. Later, the users require a precise differentiation of street, city and country. Therefore, the designer creates new respective attributes `street`, `city` and `country`, and removes the old ones. It is clear that the semantic relationship between the old and new attributes is lost when using only the creation and removal operations. It results in loosing of the respective data that should be transformed accordingly. Therefore, we need an operation which enables to explicitly specify the semantic equivalence between two sets of schema components. There are two possible ways of doing that. The simpler one is to denote that the sets are semantically equivalent without specifying how. The more complex way is to moreover describe the equivalence at the data level in a form of a query expression in a suitable language. The second approach is necessary when adapting XML documents. Since we are now interested only in adapting XML schemas, we adopt the first approach and we introduce a new kind of operations called *synchronizing operations*.

7.3 Operations

In this section, we introduce atomic operations for PIM and PSM schema evolution. We provide their examples and describe the most interesting ones. Note that the atomic operations serve as a formal basis for creating more user-friendly operations composed of the atomic ones. Full details of all atomic operations and how composite operations can be constructed can be found in [90]. In the next section, we describe how atomic operations performed at one level (PIM or PSM) are propagated to the atomic operations at the other level (PSM or PIM, respectively). Any operation composed of the atomic ones is automatically propagated as a sequence of corresponding atomic operations.

Atomic operations for PIM schema evolution provide a formal way of changing the PIM schema. Their examples (affecting classes and attributes) are listed in

Tab. 7.1. There are similar operations for associations. The definitions contain preconditions for some operations (**p:**). An addition operation creates a new component and sets its name, type and cardinality to default values configured by the designer. A sedentary operation updates a name, data type, etc. The precondition of the class removal operation ensures that a class is not removed when it has attributes or connected associations.

Operation	**Kind**	**Effect**
$C = \alpha_c()$	Addition	Adds a new class C.
$A = \alpha_a(C)$	Addition	Adds a new attribute A to an existing class C.
$\delta_c(C)$	Removal	Removes an existing class C. **p:** $\mathit{attributes}(C) = \emptyset \wedge \mathit{associations}(C) = \emptyset$
$\delta_a(A)$	Removal	Removes an existing attribute A.
$v_a^{name\|type\|card}(A, v)$	Sedentary	Updates the name, type, or cardinality, respectively of an attribute A to a new value v.
$v_a^{class}(A, C_v)$	Migratory	Moves an attribute A to a class C_v. **p:** $\mathit{associations}(\mathit{class}(A), C_v) \neq \emptyset$
$\sigma_a(\mathcal{X}_1, \mathcal{X}_2)$	Synchronizing	Synchronizes two sets of attributes $\mathcal{X}_1$ and $\mathcal{X}_2$. **p:** $(\exists C \in \mathcal{S}_c)(\mathcal{X}_1, \mathcal{X}_2 \subseteq \mathit{attributes}(C))$

Table 7.1: Examples of atomic operations for PIM schema adaptation

The migratory operation $v_a^{class}(A, C_v)$ allows for moving an attribute . Its precondition requires that there must be an association connecting the current class C_u of A with C_v. This is natural since when we move an attribute, there is typically some semantic relationship between C_u and C_v. This relationship may be modeled in the PIM schema by an association connecting C_u and C_v. It may also be modeled by a path of associations. Or, it can even be not modeled in the PIM schema at all and, instead, only considered by the designer implicitly in his/her mind. Note that the atomic operation only considers the former case. The other cases may be implemented by composing the creation, removal and migratory operations. See [90] for details. The synchronizing operation $\sigma_a(\mathcal{X}_1, \mathcal{X}_2)$ allow for denoting that two sets of attributes are semantically equivalent. They have no direct effect on the structure of the edited PIM schema. The precondition of this operation requires the synchronized attributes to be within the same class. For an example of how these operations can cover changes, let us get back to our motivational example, where we change a representation of a part of customer information from attributes `name`, `line1` and `line2` to attributes `name`, `street`, `city` and `country`. First of all, we add the new attributes: $A_{\texttt{street}} = \alpha_a(\texttt{Customer})$, $A_{\texttt{city}} = \alpha_a(\texttt{Customer})$, $A_{\texttt{country}} = \alpha_a(\texttt{Customer})$. Next, we specify the semantic equivalence of the new and the old set of attributes: $\sigma_a(\{A_{\texttt{name}}, A_{\texttt{line1}}, A_{\texttt{line2}}\}, \{A_{\texttt{name}}, A_{\texttt{street}}, A_{\texttt{city}}, A_{\texttt{country}}\})$. Finally, we remove the old attributes: $\delta_a(A_{\texttt{line1}}), \delta_a(A_{\texttt{line2}})$.

Atomic operations for PSM schema evolution are similar to the previous ones. Their examples are listed in Tab. 7.2. Only the synchronization of two sets of associations is more complicated. We cannot require that the associations connect

Operation	**Kind**	**Effect**
$C' = \alpha'_c()$	Addition	Adds a new class C'.
$R' = \alpha'_r(C'_1, C'_2)$	Addition	Adds a new association R' going from an existing class C'_1 to another existing class C'_2. **p:** $(\forall R' \in \mathcal{S}'_r)(child'(R') \neq C'_2)$.
$\delta'_c(C')$	Removal	Removes an existing class C' from $\mathcal{S}'$. **p:** $attributes'(C') = \emptyset \wedge associations'(C') = \emptyset$
$\delta'_r(R')$	Removal	Removes an existing association R'.
$\upsilon_r^{name'\|card'}(R', v)$	Sedentary	Updates the name or cardinality of an association R' to a new value v.
$\upsilon_{c\|a\|r}^{int'}(X', X)$	Sedentary	Updates an interpretation of a class, attribute, or association X' to a class, attribute or association X in the PIM schema, respectively.
$\upsilon_r^{participant'}(E', C'_v)$	Migratory	Reconnects an endpoint E' to a class C'_v. **p:** $participant'(E) = parentclass'(C'_v) \vee C'_v = parentclass'(participant'(E'))$
$\sigma'_r(\mathcal{X}'_1, \mathcal{X}'_2)$	Synchronizing	Synchronizes two sets of associations $\mathcal{X}'_1$ and $\mathcal{X}'_2$. **p:** $(\exists C'_1 \in S'_c, C_2 \in S_c)(\forall \mathcal{X}'_1 \cup \mathcal{X}'_2)$ $(parent'(R') = C'_1 \vee child'(R') = C'_2)$

Table 7.2: Examples of atomic operations for PSM schema adaptation

the same classes because the PSM schema is a forest of trees. Instead, we require that the associations have only one class in common. The other classes must have the same PIM class as their interpretation.

7.4 Propagation of Atomic Operations

An interpretation I of a PSM schema $\mathcal{S}'$ against a PIM schema $\mathcal{S}$ must satisfy the conditions given by Def. 6.5. When $\mathcal{S}$ or $\mathcal{S}'$ is modified, the conditions may be violated and the other must be modified accordingly. We call this process *propagation*. When an atomic operation is executed on $\mathcal{S}$, it must be propagated to all PSM schemas with an interpretation against $\mathcal{S}$. Vice versa, when an atomic operation is executed on $\mathcal{S}'$, it must be propagated to $\mathcal{S}$ and, from here, to the other PSM schemas. This process is called *joint adaptation* of multiple XML schemas related by the PIM schema. Here we discuss only the most interesting parts of propagation mechanism. See [90] for details.

The propagation works for the addition, removal and sedentary operations as follows: An addition is not propagated by our mechanism, because a created component models a new part of reality which is not represented in the other schemas. The removal operations are propagated from the PIM to the PSM level. Removal of a PIM component X has an impact on each PSM component X'

s.t. $I(X') = X$. There are two propagation options: X' may be removed as well or $I(X')$ is set to λ.The removal operations in the other direction are not propagated because, according to Def. 6.5, the existence of a PIM component does not depend on any PSM component. The sedentary operations are propagated straightforwardly. Types and cardinalities are propagated automatically due to type and cardinality compatibility. Names are propagated only optionally. Finally, the migratory operations propagation must move the interpreted attributes and associations to the PSM schema counterparts of their new PIM classes. This may require creating the PSM classes where they were not before.

Propagation of Synchronizing Operations. Synchronizing of a set $\mathcal{X}_2$ with a set $\mathcal{X}_1$ does not violate Def. 6.5. But, our propagation mechanism ensures that the equivalence is preserved in PSM schemas. An existence of an equivalent to $\mathcal{X}_1$ implies an existence of an equivalent to $\mathcal{X}_2$ in a given PSM schema, and vice versa. We show its propagation only from PIM to PSM. The reversed propagation is only a technical modification. To demonstrate propagation of *attribute synchronization* suppose a PIM schema in Fig. 7.1 (a) and two PSM schemas in Fig. 7.1 (b,d) with interpretations against the PIM schema. Suppose that the designer needs to replace attributes `line1`, `line2` with attributes `street`, `city`, `country` in the PIM schema. One part of this operation is the synchronization of set {`street`, `city`, `country`} with set {`line1`, `line2`}. It means that whenever there are attributes with interpretations `line1` and `line2` in the same interpreted class context C' there must also be attributes with interpretations `street`, `city` and `country` in the same interpreted class context, and vice versa.

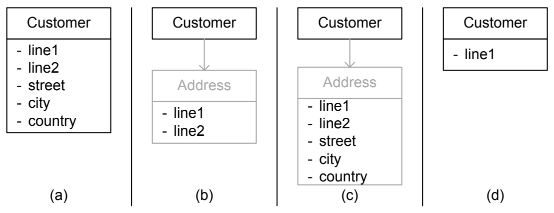


Figure 7.1: Attribute synchronization

This is the case of class `Customer`$'$ in Fig. 7.1 (b). There are attributes `line1`$'$ and `line2`$'$ with interpretations `line1` and `line2`, respectively, and with the interpreted class context `Customer`$'$. Therefore, the propagation creates the attributes `street`$'$, `city`$'$, and `country`$'$ with interpretations `street`, `city`, and `country` with the interpreted class context `Customer`$'$, respectively as shown in Fig. 7.1 (c). On the other hand, there is only a single attribute with an interpretation `line1` in Fig. 7.1 (d) and no attribute with interpretation `line2`. Therefore, the synchronization is not propagated in this case. Due to lack of space we do not show the propagation of synchronization of associations. However, the process is very similar to the synchronization of attributes.

7.5 Evaluation

The conceptual modeling language for XML proposed in our previous works was implemented in a case tool *eXolutio*[2] including the system of atomic and composite operations described in this chapter. We used the implementation to evaluate our approach on XML vocabulary standardized by Czech public authorities. It is used in a public procurement system and its PIM schema is simple. It has 4 classes interconnected by 9 associations. The vocabulary comprises 17 XML schemas each modeled by a separate PSM schema. Figure 7.2(a) shows the num-

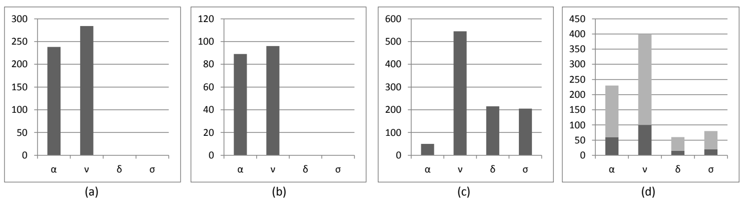


Figure 7.2: Statistics

ber of atomic operations performed to create the PIM and PSM schemas. Only creation and sedentary operations were necessary. First, there was a requirement to model 6 new XML schemas as 6 additional PSM schemas. The number of performed atomic operations is depicted in Figure 7.2(b). Only creation and sedentary operations were performed. Second, there was a requirement to make the existing XML schemas more readable for developers. This required renaming some existing XML elements and attributes in the XML schemas and their reconnecting. Also merges of various components into single ones were done. In both cases, only local changes in PSM schemas were done and the PIM schema was not affected. The number of performed atomic operations is depicted in Figure 7.2(c). All kinds of operations were necessary in this step. Third, various changes to the PIM schema needed to be made. These changes resulted from a new legislation which is currently implemented in Czech republic. In this case, the changes were correctly propagated by our introduced mechanism to the PSM schemas. The number of performed atomic operations is depicted in Figure 7.2(d). The study showed us that all introduced atomic operations are necessary. Also, they are sufficient for all kinds of changes performed. The light gray columns in Figure 7.2(d) show the number of atomic operations automatically performed by our propagation mechanism. Without the mechanism, they would have to be performed manually by the designer. However, the amount of manual work saved is much higher. Even in the previous steps, it saves a significant amount of work. When the designer designs a new PSM schema on the base of the PIM schema, our technique ensures that he works consistently with the PIM schema and, therefore, with other PSM schemas. The designer does not need to check this consistency manually which saves him a great deal of work and prevents from design errors.

[2]`http://www.eXolutio.com`

7.6 Conclusions

In this chapter we have identified several problems of design and evolution of a family of XML schemas and showed how to solve them using the strategy of MDA. We showed that a common conceptual schema in a PIM may be designed and the XML schemas may be then derived in a form of visual schemas of PSM. We defined the sets of atomic operations for the two levels and demonstrated their propagation. We show on preliminary experiments that our strategy apparently enables to save both manual effort and errors of evolution management.

8. Evolution and Change Management of XML-based Systems

XML is de-facto a standard language for data exchange. Structure of XML documents exchanged among different components of a system (e.g. services in a Service-Oriented Architecture) is usually described with XML schemas. It is a common practice that there is not only one but a whole family of XML schemas each applied in a particular logical execution part of the system. In such systems, the design and later maintenance of the XML schemas is not a simple task.

In this chapter we aim at a part of this problem – evolution of the family of the XML schemas. A single change in user requirements or surrounding environment of the system may influence more XML schemas in the family. A designer needs to identify the XML schemas affected by a change and ensure that they are evolved coherently with each other to meet the new requirement. Doing this manually is very time consuming and error prone. In this chapter we show that much of the manual work can be automated. For this, we introduce a technique based on the principles of Model-Driven Development. A designer is required to make a change only once in a conceptual schema of the problem domain and our technique ensures semi-automatic coherent propagation to all affected XML schemas (and vice versa). We provide a formal model of possible evolution changes and their propagation mechanism. We also evaluate the approach on a real-world evolution scenario.

This chapter is based on the refined conceptual model for XML in Chapter 6, extends Chapter 7 and continues in the research direction stated in Chapter 1.

This chapter is the base of the *XML schema evolution* part of this thesis. The contents of this chapter was published as an impacted journal paper *Evolution and Change Management of XML-based Systems*[1] [89] in Journal of Systems and Software (JSS).

8.1 Introduction

In this chapter we focus only on a part of the problem described – coherent evolution of XML schemas according to changing requirements. (See our recent work [71] where we discuss the other part of the problem – adaptation of underlying XML documents when their XML schemas evolve). We propose a technique based on the Model-Driven Development (MDD) [78] methodology. We consider modeling the XML schemas at two MDD levels – *platform-independent* and *platform-specific*. First, the whole application data domain is modeled independently of the XML schemas in the form of a platform-independent schema. Then, each XML schema in the family is designed in the form of a platform-specific schema which is mapped to the platform-independent schema. It may be then automatically translated to an expression in a selected XML schema

[1]`http://dx.doi.org/10.1016/j.jss.2011.09.038`

language, e.g. XSD (XML Schema Definition) [128] or RELAX NG [29]. The mappings of platform-specific schemas to platform-independent schema naturally support evolution management. A change is explicitly expressed as a change to the platform-independent schema or one of the platform-specific schemas. The mappings allow us to propagate the change between platform-independent and platform-specific levels semi-automatically and evolve the whole family of XML schemas coherently.

Contributions The key contributions of this chapter are as follows:

- formal models for designing XML schemas at platform-independent and platform-specific MDD levels and a set of atomic operations for their evolution,
- proof of minimality and correctness of the set,
- mechanism for propagating changes invoked by the atomic operations between the MDD levels,
- specification of operations composed of the atomic ones and their propagation between the MDD levels,
- implementation of the proposed framework called *eXolutio* [55], and
- experimental demonstration of the completeness of the set of atomic operations and correctness of the propagation mechanism by applying eXolutio in a real-world case study.

Although there is other existing work in the area of schema evolution (as we will show in Section 8.7), the evolution problem has not yet been adequately solved [47]. We will show that the current approaches omit some important kinds of operations and provide an insufficient solution to the problem of the propagation of changes. And, last but not least, they do not introduce operations as a formal set of simple atomic operations which would allow authors of tools for schema evolution to build various more-user friendly operations as compositions of the atomic operations. In this work, we introduce such formalism. Its main advantage is that it enables one to specify a new operation as a sequence of the atomic operations without the details of how the new operation is propagated to the other parts of the system. Our propagation mechanism ensures its correct propagation automatically.

In this chapter we combine and, in particular, extend previous work in this area. A technique for designing XML schemas at platform-independent and platform-specific levels was firstly proposed in [86] and later generalized in [94]. A basic implementation of a modeling tool based on the models was introduced in [54]. In this text we describe it in detail including its evolution extension and show its usage in real-world use cases. In [91, 93] is a five-level XML evolution framework which presents a general overview of the problem of XML schema evolution in the context of a whole software system consisting of various parts. In this chapter we lay the theoretical basis of our approach. We provide a formal and detailed description of evolution operations and their propagation, prove minimality and the correctness of our approach and extend it with explanatory examples. In general, this chapter expands on the results of our recent research with an emphasis on formal specification.

In [138] the authors discussed two kinds of evolution approaches – *incremental* and *change-based approaches*. An incremental approach enables a clear formal basis which ensures correctness and allows for simple evolutionary steps made by a designer. A change-based approach is suitable for cases when we are provided with two versions of the schema without the incremental evolutionary steps and we need to manage evolution of the data efficiently. In this work, we introduce an incremental approach based on a set of atomic operations. A designer incrementally performs particular atomic operations or operations comprising the atomic ones. Our technique continuously propagates the changes to affected schemas.

Outline The rest of the chapter is structured as follows: In Section 8.2 we provide a motivating and running example. In Section 8.3 we describe the problem of XML schema evolution in the context of a whole software system and specify the selected part of the problem solved in this chapter. In Section 8.4 we extend the PIM and PSM levels of our conceptual model for XML (see Chapter 6) with a set of atomic operations. In Section 8.5 we describe the propagation mechanism of the atomic operations between the levels and show that the atomic operations together with the propagation mechanism form a minimal and correct evolution formalism. In Section 8.6 we show how the atomic operations form realistic composite operations. In Section 8.7 we compare our proposal with current related works. In Section 8.8 we introduce the implementation of the introduced evolution formalism called *eXolutio* and its application in a real-world case study. We also evaluate our approach on the basis of this case study. Finally, in Section 8.9 we conclude and outline possible future work.

8.2 Motivating and Running Example

As a demonstration of the problem of evolution management of XML schemas, let us consider a company that receives purchase orders and let us focus on a part of the system that processes purchases. Let the messages used in the system be XML messages formatted according to a family of different XML schemas. Consider the two sample XML documents in Figure 8.1. The former one is formatted according to an XML schema specifying a list of customers. The latter one is formatted according to a different XML schema specifying purchase requests. There are also other XML schemas in the family (e.g. customer details, purchase responses, purchase transport details, etc.). All the XML schemas share the same data domain (purchasing goods). On the other hand, the same part of the domain may be represented in different XML schemas in different ways. For example, the concept of *customer* is represented in each of our sample XML schemas in a different way. On the right hand side, elements `name` and `email` are present for a customer. On the left hand side, kinds of customers are distinguished (private and corporate customers). For private customers, elements `name`, `address` and `phone` are present. For corporate customers, elements `name`, different addresses (`headquarters`, `storage` and `secretary`), and `phone` are present.

```
<custList version="1.3">
  <cust>
    <name>Martin Necasky</name>
    <address>Vaclavske nam. 123, Prague</address>
    <phone>123 456 789</phone>
  </cust>
  <cust>
    <name>Department of Software Engineering,
      Charles University</name>
    <hq>Malostranske nam. 25, Prague</hq>
    <storage>Ke Karlovu 3, Prague</storage>
    <secretary>Ke Karlovu 5, Prague</secretary>
    <phone>111 222 333</phone>
  </cust>
</custList>
```

```
<purchaseRQ version="1.0">
  <bill-to>Malostranske nam. 25, Prague</bill-to>
  <ship-to>Ke Karlovu 3, Prague</ship-to>
  <cust>
    <name>Department of Software Engineering,
      Charles University</name>
    <email>ksi@mff.cuni.cz</email>
  </cust>
  <items>
    <item>
      <code>P045</code>
    </item>
    <item>
      <code>P332</code>
    </item>
  </items>
</purchaseRQ>
```

Figure 8.1: Sample XML documents represented in a single XML system

Let us consider a new user requirement that an address should no longer be represented as a simple string. Instead, it should be divided into elements `street`, `city`, `zip`, etc. Such a situation would require a skilled domain expert to identify all the schemas in the system which involve an address and correct them respectively. Apparently, in a complex system comprising tens or even hundreds of schemas, this is a difficult and error-prone task. Even identifying the affected parts of the schema is not an easy and straightforward process. For example, we may need to make the modification only for addresses that represent a place to ship the goods (which are the elements `address` and `storage` in the XML schema instantiated on the left-hand side of the figure and element `ship-to` on the right-hand side). We do not want to modify addresses that represent headquarters, etc.

In the following text we show in detail that evolution management is a complex process that can be solved semi-automatically and, hence, efficiently and precisely if we provide a rigorous theoretical background and preserve nontrivial relations and meta-data.

8.3 XML Evolution Framework

In our previous work [91], we introduced a framework for managing evolution of a software system which exploits XML technologies at different levels. An extended version of the framework is depicted in Figure 8.2. As we can see, the framework can be partitioned both horizontally and vertically; in both cases its components are closely related and interconnected. The relations form the key concept of the evolution management, since they invoke the needs for change propagation.

If we consider the vertical partitioning, we can identify multiple views of the system. In the framework we have depicted the three most common and representative views. The blue (leftmost) part covers an *XML view* of the data processed and exchanged in the system. The green (middle) part represents the *storage view* of the system, e.g. a relational view of the processed data which need to be persistently stored. Finally, the yellow (rightmost) part represents a *processing view* of the data, e.g. processing by sequences of Web Services described using BPEL scripts [100] or various proprietary formats (e.g. [110]).

If we consider the horizontal partitioning, we can identify five levels, each representing a different view of an XML system and its evolution. The lowest

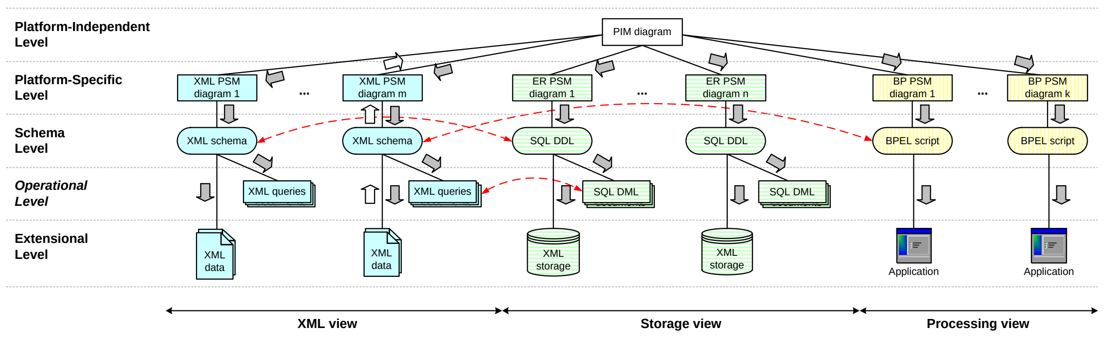


Figure 8.2: Five-level XML evolution architecture

level, called *extensional level*, represents the particular instances that form the implemented system such as, e.g., XML documents, relational tables or Web Services that are components of particular business processes. Its parent level, called *operational level*, represents operations over the instances, e.g. XML queries over the XML data expressed in XQuery [20] or SQL/XML [48] queries over relations. The level above, called *schema level*, represents schemas that describe the structure of the instances, e.g. XML schemas or SQL/XML Data Definition Language (DDL).

Even these three levels indicate problems related to XML evolution. For instance, when the structure of an XML schema changes, its instances, i.e. XML documents, and related queries must be adapted accordingly so that their validity and correctness is preserved respectively. In addition, if we want to preserve optimal query evaluation over the stored data, the storage model also needs to adapt respectively. What is more, as we have mentioned, in practice there are usually multiple XML schemas (families of XML schemas) applied in a single system, e.g. XML schemas for purchases, invoices, product catalogues, etc., i.e. multiple views of the common problem domain. Hence, such a change can influence multiple XML schemas, XML documents and queries. In general, a change at one level can trigger a cascade of changes at other levels. We call such sequences of adaptations *change propagation*.

Considering only the three levels leads to evolution of each affected schema separately. However, this is a highly time-consuming and error-prone solution since we need a domain expert who is able to identify all the affected schemas and propagate the changes. Therefore, we introduce two additional levels, which follow the MDD [78] principle, i.e. modeling of a problem domain at different levels of abstraction. As we have mentioned, the topmost one is the *platform-independent level* which comprises a *schema in a platform-independent model* (*PIM schema*). The PIM schema is a conceptual schema of the problem domain. It is independent of any particular data (e.g. XML or relational) or business process (e.g. Web Services) model. The level below, called *platform-specific level*, represents mappings of the selected parts of the PIM schema to particular data or business process models. For each model it comprises *schemas in a platform-specific model* (*PSM schemas*) such as, e.g., XSEM schemas [86] which model XML schemas, ER [27] schemas which model relational schemas, etc. Each PSM schema can be then automatically translated to a particular language used at the schema level and vice versa. Note that the latter direction allows for integration of incoming formats/applications into the given evolution framework.

As we can see in Figure 8.2, there are not only vertical relations between the levels, but the components of the system can also be horizontally related across the vertical partitions. A few examples are denoted by the red dashed arrows. For instance, there is a relation between an XML schema and its respective storage in a relational database. Similarly, an XML query can be evaluated by translation into an SQL query. And, last but not least, a BPEL script can specify how an input SOAP message, i.e. an XML schema, is processed.

In all the three cases a change in one of the ends of the relation influences the other. So, since we are considering completely different formats involving different constructs which do not have to correspond mutually using one-to-one relationship, the change propagation becomes a complex problem. But, having a hierarchy of models which interconnect all the applications and views of the data domain using the common PIM level, it can be done semi-automatically and much more easily. We do not need to provide a mapping from every PSM to all other PSMs, but only from every PSM to the PIM which is, in addition, quite natural. Hence, the vertical change propagation is realized using this common point. For instance, if a change occurs in a selected XML document, it is first propagated to the respective XML schema, PSM and, finally, PIM. We speak about an *upwards propagation*, in Figure 8.2 represented by white arrows. It enables one to identify the part of the problem domain that was affected. Then, we can invoke the *downwards propagation.* It enables one to propagate the change of the problem domain to all the related parts of the system. In Figure 8.2 it is denoted by grey arrows.

8.3.1 Selected Part of the Problem

Apparently, the change propagation problem is not an easy task and cannot be covered in a single paper. In this chapter we aim at one particular problem – XML schema evolution. As we have shown in the motivating example, there is usually a whole family of XML schemas which are conceptually related to the problem domain of the system. When a designer needs to make a change in one of the XML schemas, the other XML schemas may be affected as well. We introduce an approach which is based on modeling the changes at PIM and PSM XML levels as highlighted in Figure 8.3. It ensures that whenever a change is performed in a PIM schema, it is correctly propagated to the PSM schemas and vice versa. So it ensures consistency between the schemas when they are changed.

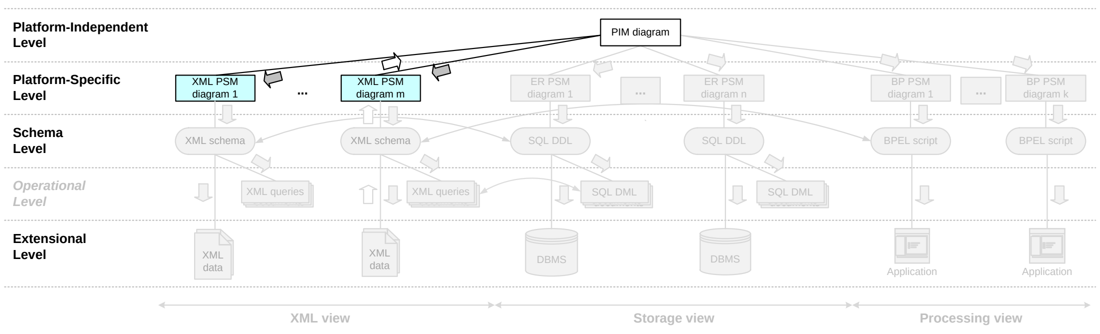


Figure 8.3: Five-level XML evolution architecture – data representation

In practice, this problem appears in two scenarios. The first scenario is when a designer creates *new XML schemas* which have not been deployed in a run-time environment yet. There are neither XML documents formatted according to the XML schemas, nor other developers or applications which would somehow use the XML schemas. In other words, there is no extensional level and no operational level. Because of the complexity of the task, the designer does not create XML schemas in a single linear process. Instead, (s)he iterates in several cycles before an acceptable version of the XML schemas is prepared to be deployed. (S)he starts each iteration with a selected part of the requirements and incorporates them into the XML schemas. For that, (s)he needs a mechanism which shows an impact of a next change to the unfinished XML schemas and which helps to adapt the XML schemas according to this change. No propagation to extensional or operational levels is necessary at this stage.

The most frequent modifications to the XML schemas in this scenario will be, intuitively, creating new parts of the XML schemas. However, updating existing parts with their more detailed and elaborate variants will be frequent as well. This is because the designer will cover some of the requirements only briefly in the XML schemas in early iterations and will return to them in later iterations to finish them. In simpler cases, updating means changing properties of existing XML schema components (e.g. data type). In more complex cases, updating means removing old parts and replacing them with new but semantically equivalent and more elaborate parts. No backward compatibility of the new version needs to be preserved since there is neither extensional, nor operational level.

The second scenario is *adapting existing XML schemas* which have already been deployed in a run-time environment. In this scenario it is necessary to consider the extensional and operational level as well, because there exist XML documents and applications which use the XML schemas. Such scenario usually occurs when new or changed requirements need to be implemented in the system (e.g. a legislative change). Due to backward compatibility the designer will probably not remove the existing parts of the XML schemas. If some part needs to be detailed (or, conversely, simplified), it will be extended with a new version, not replaced.

The approach we introduce in the following sections is fully sufficient for the first scenario and partly also for the second scenario. For the second scenario, propagation to the extensional and operational level is also necessary. We described the propagation to the extensional level in [71]. The technique introduced generates an XSLT script which transforms XML documents from the old version of each affected XML schema to the new version. The propagation to the operational level is the matter of our future work.

A careful reader might notice that we omitted the schema level in the above paragraphs. Our approach allows the designer to work only at the PIM and PSM levels and not to consider the schema level. This is because our introduced PSM level is equivalent to the schema level from the syntactical point of view. The PSM level has two purposes in addition to the schema level – it provides a more user-friendly presentation of the XML schemas to the designer and extends the XML schemas with mappings to the conceptual schema at the PIM level. In [92], the equivalence was proven formally. We also showed how a PSM schema may be automatically translated to an XML schema expressed in some XML schema language and vice versa via the formalism of *regular tree grammars* [96]. However, this is beyond the scope of this chapter. We will just keep this fact in mind. We will present a set of operations for changing PSM schemas. Because of the equivalence, a change operation at the PSM level unambiguously and correspondingly describes a change in the modeled XML schema and it is not, therefore, necessary to explicitly convert it to a change specific for an XML schema expressed in some XML schema language.

8.4 Atomic Operations

In this section, we introduce *atomic operations* for editing PIM and PSM schemas. They are not intended to be used directly by the designer, because they are too primitive and using them would be too laborious and clumsy for the designer. However, they will serve us as a formal basis for describing more user-friendly operations composed of these atomic operations. In Section 8.6 we will describe *composite operations*. In Section 8.5 we will describe how operations are propagated between PIM and PSM levels to ensure the consistency of corrupted interpretations.

Formally, we suppose a PIM schema $\mathcal{S} = (\mathcal{S}_c, \mathcal{S}_a, \mathcal{S}_r)$ and a set of PSM schemas $\mathcal{PSM} = \{\mathcal{S}'_1, \ldots, \mathcal{S}'_n\}$, where each $\mathcal{S}'_i$ has an interpretation I_i against $\mathcal{S}$. We also consider one specific PSM schema $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{\mathcal{S}'})$ from this set with an interpretation I against $\mathcal{S}$. For each atomic operation, we specify its input parameters together with a precondition and postcondition. If a precondition is not satisfied, the operation cannot be performed. The postcondition describes the effect of the operation. When an operation is executed on $\mathcal{S}$ or $\mathcal{S}'$, we say that the schema *evolved to a new version*. This is denoted $\mathcal{S}^+$ or $\mathcal{S}'^+$, respectively. The new version of the interpretation will be denoted I^+. Initially, we suppose a single empty PIM schema and empty $\mathcal{PSM}$. The PIM schema cannot be removed. On the other hand, a new PSM schema with an interpretation against the PIM schema may be created and later removed.

We classify atomic operations into 4 categories: *creation* (denoted by the Greek letter α), *update* (denoted by the Greek letter υ), *removal* (denoted by the Greek letter δ) and *synchronization* (denoted by the Greek letter σ). While the creation, update and removal operations are common in the literature, the synchronization operations have not been considered and are novel in our approach. They are crucial for the evolution.

A synchronization operation allows for the specification that two sets of components are *semantically equivalent*. Consider a simple scenario with a class *Customer* which models a concept of customer. Customer's address is modeled with attribute *address*. Later, users require a more precise specification of address including street, street number and city. Therefore, the designer needs to replace *address* with new attributes *street*, *streetno* and *city*. According to existing approaches, this means creating the new attributes and removing the old one. However, this leads to loosing the information that the old attribute is semantically equivalent to the new set. Without this information, the performed change cannot be correctly propagated as we will show later. This is the reason why we propose synchronization operations. We use them to specify that *address* is semantically equivalent to the set $\{$*street*, *streetno*, *city*$\}$.

Definition 8.1 *Let $\mathcal{X}_1$ and $\mathcal{X}_2$ be two sets of components from the same PIM or PSM schema. We use predicate equiv($\mathcal{X}_1$, $\mathcal{X}_2$) to denote that $\mathcal{X}_1$ and $\mathcal{X}_2$ are* semantically equivalent. *It means that $\mathcal{X}_1$ models the same information as $\mathcal{X}_2$.*

8.4.1 Atomic Operations for PIM Schema Evolution

We start with atomic operations for evolution of PIM schemas. The operations for creating new components are summarized in Table 8.1: their semantics is clear and so we provide no further description. Let us just note that the name, data type, and cardinality of created components are set to default values configured by the schema designer.

Notation	Description	Precond.	Postcondition
$C = \alpha_c()$	Create class C with default name l_c	`true`	$C \in (\mathcal{S}_c^+ \setminus \mathcal{S}_c) \wedge name^+(C) = l_c$
$A = \alpha_a(C)$	Create attribute A with default name, type and cardinality l_a, l_t, and l_c	$C \in \mathcal{S}_c$	$A \in (\mathcal{S}_a^+ \setminus \mathcal{S}_a) \wedge class^+(A) = C \wedge name^+(A) = l_a \wedge type^+(A) = t_a \wedge card^+(A) = c_a$
$R = \alpha_r(C_1, C_2)$	Create association R with default name and cardinalities l_r and c_r	$C_1, C_2 \in \mathcal{S}_c$	$R = \{E_1, E_2\} \in (\mathcal{S}_r^+ \setminus \mathcal{S}_r) \wedge name^+(R) = l_r \wedge participant^+(E_1) = C_1 \wedge participant^+(E_2) = C_2 \wedge card^+(E_1) = c_r \wedge card^+(E_2) = c_r$

Table 8.1: Atomic operations for creating new PIM components

The operations for updating components are summarized in Table 8.2. There are two update operations which merit a more detailed explanation – moving an attribute A from its current class to another class (υ_a^{class}) and reconnecting an association end E from its current class to another class (υ_r^{class}). The preconditions of both operations require that the current and new class are connected by an association ($associations(C_1, C_2)$ denotes all associations connecting classes C_1 and C_2). Therefore, it is not possible to move an attribute or reconnect an association end between classes which are only connected by a path of associations or are not connected at all. However, it is possible to create a composite operation from the atomic operations which allows for moving the attributes and reconnecting association ends freely. It can perform atomic moves or reconnections in case where there is a connecting path. And, it can create a temporary association connecting the classes in case there is no connection at all.

The operations for removing components are summarized in Table 8.3; however, the class removal operation (δ_c) requires the removed class to have no attributes and connected associations, so all attributes and associations connected to the class must be removed before removing the class itself.

And, finally, the operations for synchronizing components are summarized in Table 8.4. We introduce two operations – synchronization of two sets of attributes and synchronization of two sets of associations. The precondition of the former synchronization operation requires the attributes from both sets to belong to the same class. It is not restrictive. It is possible to have two synchronized sets of attributes, where each attribute is in a different class. However, we need to perform a sequence of atomic operations – this consists of moving the attributes to

Notation	Description	Precondition	Postcondition
$v_c^{name}(C, v)$	Update name of class to v	$C \in \mathcal{S}_c$	$name^+(C) = v$
$v_a^{name \mid type \mid card}(A, v)$	Update name, type, or cardinality of attribute to v	$A \in \mathcal{S}_a$	$name^+(A) = v$, $type^+(A) = v$ or $card^+(A) = v$
$v_a^{class}(A, C)$	Move attribute to class C	$A \in \mathcal{S}_a \wedge C \in \mathcal{S}_c \wedge$ $associations(class(A), C) \neq \emptyset$	$class^+(A) = C$
$v_r^{name}(R, v)$	Update name of association to v	$R \in \mathcal{S}_r$	$name^+(R) = v$
$v_r^{class}(E, C)$	Reconnect association end to class C	$C \in \mathcal{S}_c \wedge (\exists R \in \mathcal{S}_r)(E \in R) \wedge$ $associations(participant(E), C) \neq \emptyset$	$participant^+(E)$ $= C$
$v_r^{card}(E, v)$	Update cardinality of association end to v	$(\exists R \in \mathcal{S}_r)(E \in R)$	$card^+(E) = v$

Table 8.2: Atomic operations for updating PIM components

Notation	Description	Precondition	Postcondition
$\delta_c(C)$	Remove class C	$C \in \mathcal{S}_c \wedge attributes(C) = \emptyset \wedge$ $associations(C) = \emptyset$	$C \notin \mathcal{S}^+$
$\delta_a(A)$	Remove attribute A	$A \in \mathcal{S}_a$	$A \notin \mathcal{S}^+$
$\delta_r(R)$	Remove association R	$R \in \mathcal{S}_r$	$R \notin \mathcal{S}_c^+$

Table 8.3: Atomic operations for removing PIM components

the same class, synchronization and moving them back to their original classes. Similarly, the precondition of the other operation needs the associations to connect the same two classes. Again, it is not restrictive, because other cases may be achieved by performing a sequence of atomic operations.

The reader might notice that we do not provide an operation for synchronizning classes. An operation for synchronizing a mixture of classes, attributes and associations is missing as well. Our preliminary case studies (one of the provided in Section 8.8) show that class synchronization is not necessary as classes do not model data but only encapsulate them. Synchronization of a mixture of components would be, theoretically, necessary, but too complex and unnatural for common designers. Therefore, in the current version of our technique we try to manage the evolution without these advanced synchronization operations. We leave this scientifically interesting issue to our future work.

A sample evolution is depicted in Figure 8.4. Figure 8.4(a) shows a starting PIM schema. It is a fragment of the PIM schema depicted in Figure 6.1. It contains two classes `Customer` and `Partner` which model customers and partners, respectively. Partners are responsible for customers which is modeled by the relationship `responsibility`. First, there is a requirement to not further consider partners. Therefore, class `Partner` needs to be deleted by operation δ_c(`Customer`). It is necessary to perform δ_a(`code`) and δ_r(`responsibility`), which delete the attribute `code` and also the association `responsibility`, prior to δ_c(`Customer`). The result is depicted in Figure 8.4(b).

Second, there is a requirement to consider customer's addresses in more detail. Currently, it is modeled by attribute `address` of class `Customer`. The aim

Notation	Description	Precondition	Postcondition
$\sigma_a(\mathcal{X}_1, \mathcal{X}_2)$	Synchronize set of attributes $\mathcal{X}_2$ with set of attributes $\mathcal{X}_2$	$\mathcal{X}_1 \subseteq \mathcal{S}_a \wedge \mathcal{X}_2 \subseteq \mathcal{S}_a$ $\wedge\ (\exists C \in \mathcal{S}_c)(\mathcal{X}_1, \mathcal{X}_2 \subseteq \mathit{attributes}(C))$	$\mathit{equiv}^+(\mathcal{X}_1, \mathcal{X}_2)$
$\sigma_r(\mathcal{X}_1, \mathcal{X}_2)$	Synchronize set of associations $\mathcal{X}_2$ with set of associations $\mathcal{X}_2$	$\mathcal{X}_1 \subseteq \mathcal{S}_r \wedge \mathcal{X}_2 \subseteq \mathcal{S}_r$ $\wedge\ (\exists C_1, C_2 \in \mathcal{S}_c)(\mathcal{X}_1, \mathcal{X}_2 \subseteq \mathit{associations}(C_1, C_2))$	$\mathit{equiv}^+(\mathcal{X}_1, \mathcal{X}_2)$

Table 8.4: Atomic operations for synchronization of PIM components

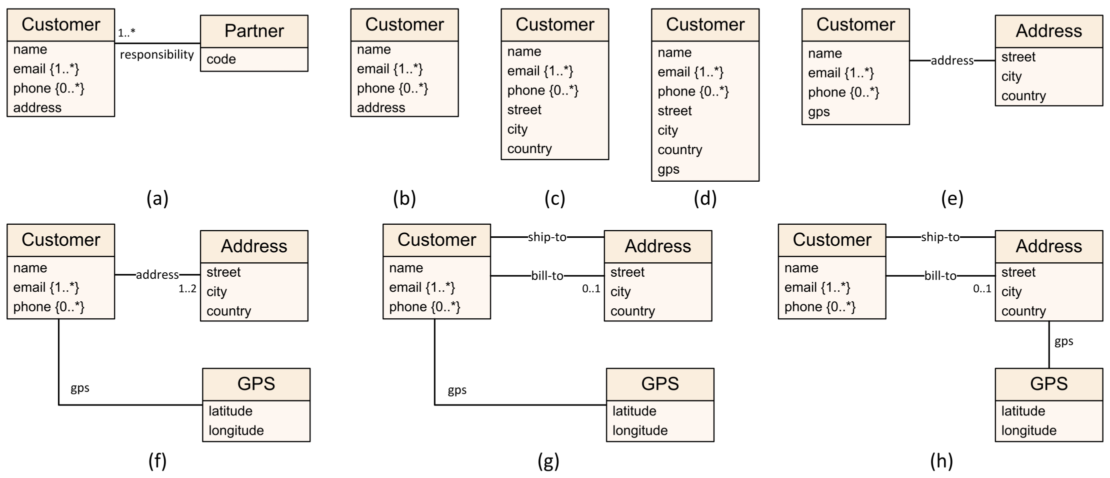


Figure 8.4: Evolution of a sample PIM schema demonstrating the introduced creation, update, removal and synchronization atomic operations

is to model addresses as depicted in Figure 8.4(h). The evolution is iterative. Particular iterations are depicted in Figures 8.4(c)-(g). The designer starts with modeling addresses with three separate attributes `street`, `city` and `country` instead of the original attribute `address`. For this, (s)he creates the attributes ($\texttt{street} = \alpha_a(\texttt{Customer}), \ldots$), changes the default values of their names and data types when necessary ($\upsilon_a^{name}(\texttt{street}, \text{“}\mathit{street}\text{“}), \ldots$), synchronizes them with the original attribute ($\sigma_a(\{\texttt{address}\}, \{\texttt{street}, \texttt{city}, \texttt{country}\})$) and, finally, removes the original attribute ($\delta_a(\texttt{address})$). The synchronization is important. It specifies that the new attributes are semantically equivalent with the old one. The whole sequence of performed atomic operations can be viewed as splitting the original attribute into the three new ones. Note that the precondition for synchronization is satisfied (all attributes are in the same class). The result is depicted in Figure 8.4(c).

Later, the designer notices that (s)he forgot to include GPS information. (S)he needs to extend the three attributes `street`, `city` and `country` with a new attribute `gps`. For this, (s)he creates the new attribute and synchronizes the original set of attributes modeling address with the new set which is the original extended with `gps` ($\sigma_a(\{\texttt{street}, \texttt{city}, \texttt{country}\}, \{\texttt{street}, \texttt{city}, \texttt{country}, \texttt{gps}\})$). The result is depicted in Figure 8.4(d).

Now, class `Customer` contains too much information and the designer wants to make it more transparent. Therefore, (s)he decides to move attributes `street`, `city` and `country` to a separate class `Address`. However, this class is not present. Therefore, the designer needs to create it and update its name ($\texttt{Address} = \alpha_c()$, $\upsilon_c^{name}(\texttt{Address}, \text{“}\mathit{Address}\text{“})$). (S)he also needs to connect it with `Customer` by creating a new association `address` ($\texttt{address} = \alpha_r(\texttt{Customer}, \texttt{Address}), \ldots$). Then, (s)he can move the attributes to the new class ($\upsilon_a^{class}(\texttt{street}, \texttt{Address})$, $\ldots$). The old and new class are connected by an association and, therefore, the precondition for moving the attributes is satisfied. The result is depicted in Figure 8.4(e). (S)he also needs to detail `gps` to `latitude` and `longitude` and move them to a separate class `GPS`. Therefore, she performs a similar sequence of operations as for the former `address` attribute. And, finally, (s)he needs to extend customers to have one or two addresses instead of one. (S)he, therefore, changes the cardinality of association `address` to `1..2` ($\upsilon_r^{card}(\texttt{address2}, 1..2)$), where `address2` is the endpoint associated with `Address`. The result is depicted in Figure 8.4(f).

In the following step, the designer gets a requirement to explicitly distinguish the semantics of the two addresses to a mandatory shipping address and optional billing address. Therefore, (s)he splits the association `address` to two new associations `shipto` and `billto`. As with the splitting of attributes, this entails creating two new associations ($\texttt{shipto} = \alpha_r(\texttt{Customer}, \texttt{Address}), \ldots$), changing their default names and cardinalities ($\upsilon_r^{name}(\texttt{shipto}, \text{“}\textit{shipto}\text{“}), \ldots$), synchronizing the old association with the new ones ($\sigma_r(\{\texttt{address}\}, \{\texttt{shipto}, \texttt{billto}\})$), and removing the old association ($\delta_r(\texttt{address})$). Note that the synchronized associations connect the same classes and, therefore, preconditions for the synchronization are satisfied. The result is depicted in Figure 8.4(g).

Finally, there appears a requirement to record GPS information for each address instead of a customer. For this, the designer reconnects the association `gps` from class `Customer` to class `Address` ($\upsilon_r^{class}(\texttt{gps1}, \texttt{Address})$), where `gps1` is the endpoint associated with `Customer`. Again, the precondition for the reconnection is satisfied, because the classes are connected by an association. The result is depicted in Figure 8.4(h).

8.4.2 Atomic Operations for PSM Schema Evolution

In this section, we introduce atomic operations for evolution of PSM schemas. The operations for creating new components are summarized in Table 8.5: there is also an operation for creating PSM schemas themselves. Again, names, data types, XML forms and cardinalities of new components are set to default values which are configured by the designer. All components are created with an empty interpretation against the PIM schema.

The operations for updating components are summarized in Table 8.6. Similar to the operations for updating PIM components, there are two interesting operations – moving an attribute (υ'^{class}_a) and reconnecting an association end (υ'^{class}_r). Both are similar to their PIM equivalents but there are some differences. An attribute can be moved to the nearest ancestor or descendant class of its current class ($\textit{parentclass}(C')$ denotes the nearest ancestor class to C'). It can also be moved to a structural representative of its current class or, conversely, to a class which is a structural representative of its current class. For an association, only its parent association end can be reconnected to the parent or to any child of its current parent. When its current parent is a class, it can also be reconnected to a structural representative of the current parent or, conversely, to a class which is a structural representative of its current parent.

Notation	Description	Precondition	Postcondition
$(\mathcal{S}', I) = \alpha'_s(\mathcal{S})$	Create new PSM schema $\mathcal{S}'$ with interpretation I against $\mathcal{S}$	`true`	$\mathcal{S}' = (\{\mathcal{C}'_{\mathcal{S}'}\}, \emptyset, \emptyset, \emptyset, \mathcal{C}'_{\mathcal{S}'}) \wedge \mathcal{S}' \in (\mathcal{PSM}^+ \setminus \mathcal{PSM}) \wedge I = \{(\mathcal{C}'_{\mathcal{S}'}, \lambda)\}$
$C' = \alpha'_c()$	Create new class C' with default name l_c	`true`	$C' \in (\mathcal{S}'^+_c \setminus \mathcal{S}'_c) \wedge I^+(C') = \lambda \wedge name^+(C') = l_c$
$A' = \alpha'_a(C')$	Create new attribute A' with default name, type, XML form and cardinality l_a, t_a, x_a, and c_a	$C' \in \mathcal{S}'_c$	$A' \in (\mathcal{S}'^+_a \setminus \mathcal{S}'_a) \wedge class^+(A') = C' \wedge I^+(A') = \lambda \wedge name'^+(A') = l_a \wedge type^+(A') = t_a \wedge xform^+(A') = x'_a \wedge card^+(A') = c_a$
$R' = \alpha'_r(X'_1, X'_2)$	Create new association R' with default name and cardinalities l_r and c_r	$X'_1 \in (\mathcal{S}'_c \cup \mathcal{S}'_m) \wedge X'_2 \in (\mathcal{S}'_c \cup \mathcal{S}'_m) \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge (\nexists R'_0)(child(R'_0) = X'_2)$	$R' \in (\mathcal{S}'^+_r \setminus \mathcal{S}'_r) \wedge parent(R') = X'_1 \wedge child(R') = X'_2 \wedge I^+(R') = \lambda \wedge name^+(R') = l_r \wedge card^+(R') = c_r \wedge position(R')^+ = \|content(C')\|$
$M' = \alpha'_m()$	Creates new sequence content model M'	`true`	$M' \in (\mathcal{S}'^+_m \setminus \mathcal{S}'_m) \wedge cmtype(M') =$ `sequence`

Table 8.5: Atomic operations for creating PSM schemas and their components

The operations for updating interpretations are summarized in Table 8.7. Their preconditions ensure that the consistency of interpretation is not violated. Concretely, the operation for updating class interpretation (υ'^{int}_c) could violate any of the conditions necessary for consistency. However, its precondition requires that the interpretation of any attribute or association, whose consistency would be corrupted by the update, must be empty. This includes all attributes and associations which have the same interpreted context as the class ($anc(X')$ used in the precondition which denotes all ancestor classes of X'). Also, the class can not be a structural representative and, conversely, it cannot have a structural representative. Therefore, the conditions can not be violated.

The operation for updating attribute interpretation (υ'^{int}_a) could affect condition (2) and the operation for updating association interpretation (υ'^{int}_r) could affect conditions (3) and (4). Their preconditions prevent any violations. (They are directly rewritten from the definition.)

The operations for removing components of PSM schemas are listed in Table 8.8. Their functionality is quite clear. Let us note that we can only remove classes and content models that are empty and are roots of their PSM schema. Also, we can only remove associations, whose removal does not violate Definition 6.7. When there are attributes or associations in the subtree of R' with the same interpreted class context as R' and with non-empty interpretations, we cannot remove R'. To correct the schema, we would need to set empty interpretations to these attributes and associations, which is not an atomic operation. Note that when we remove an association going to a class or content model, this class or content model becomes a root.

Notation	**Description**	**Precond.**	**Postcond.**
$v'^{name}_{c}(C',v)$	Update name of class C' to v	$C' \in \mathcal{S}'_c$	$name^+(C') = v$
$v'^{repr}_{c}(C',C'_r)$	Set class C' as structural representative of C'_r	$C' \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \land (C'_r = \lambda \lor (C'_r \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \land I(C') = I(C'_r) \land C' \notin repr^*(C'_r)))$	$repr^+(C') = C'_r$
$v'^{cmtype}_{m}(M',t)$	Update type of content model M'	$M' \in \mathcal{S}'_m \land t \in \{\texttt{sequence}, \texttt{choice}, \texttt{set}\}$	$cmtype^+(M') = t$
$v'^{name\|type}_{a}(A',v)$ $v^{card\|xform}_{a}(A',v)$	Updates name, type, cardinality, or XML form of attribute to v	$A' \in \mathcal{S}'_a$	$name^+(A') = v$, $type^+(A') = v$, $card^+(A') = v$ or $xform^+(A') = v$
$v'^{pos}_{a}(A')$	Changes position of attribute A' by -1	$position(A') > 1$	$position^+(A') = position(A') - 1$
$v'^{class}_{a}(A',C')$	Move attribute A' to class C'	$A' \in \mathcal{S}'_a \land C' \in \mathcal{S}'_c \land (repr(class(A')) = C' \lor class(A') = repr(C') \lor class(A') = parentclass(C') \lor C' = parentclass(class(A')))$	$class^+(A') = C'$
$v'^{name\|card}_{r}(R',v)$	Update name or cardinality of association R' to v	$R' \in \mathcal{S}'_r$	$name^+(R') = v$ or $card^+(R') = v$
$v'^{pos}_{r}(R')$	Change position of association R' by -1	$position(R') > 1$	$position^+(R') = position(R') - 1$
$v'^{class}_{r}(R',P')$	Reconnect parent association end of association R' to new parent P'	$R' \in \mathcal{S}'_r \land P' \in \mathcal{S}'_c \cup \mathcal{S}'_m \land (repr(parent(R')) = P' \lor parent(R') = repr(P') \lor$ $\exists R'_p \in \mathcal{S}'_r$ which connects $parent(R')$ and P')	$parent^+(R') = P'$

Table 8.6: Atomic operations for updating PSM components

And, finally, the operations for synchronizing two sets of PSM components are listed in Table 8.9. Similar to their PIM equivalents, they allow for synchronization of two sets of attributes and two sets of associations. The operation for attributes corresponds to its PIM equivalent. The operation for associations is also similar. However, it is not possible to require the associations to have the same participants (because of the tree nature of PSM schemas). Instead, we require that they have one of their participants in common: that is the child of none or one of the associations and the parent of the others. The other participants must be different classes but with the same non-empty interpretation. In other words, these other participants are semantically equivalent (they have the same class in the PIM schema as their interpretation). Therefore, the operation also corresponds to its PIM equivalent.

Similar to synchronization in a PIM schema, the expression $equiv^+(\mathcal{X}'_1, \mathcal{X}'_2) =$ `true` in the postconditions of both operations denotes that $\mathcal{X}'_1$ and $\mathcal{X}'_2$ are synchronized in the new version of the PSM schema.

Notation	Description	Precondition	Post...
$\upsilon'^{int}_c(C', C)$	Update interpretation of class C' to class C	$C' \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge (C = \lambda \vee C \in \mathcal{S}_c) \wedge$ $(\forall A' \in \mathcal{S}_a$ s.t. $\mathit{intcontext}(A') = \mathit{intcontext}(C') \wedge C' \in \mathit{anc}(A'))(I(A') = \lambda)$ $\wedge$ $(\forall R' \in \mathcal{S}'_r$ s.t. $(\mathit{intcontext}(R') = \mathit{intcontext}(C') \wedge C' \in \mathit{anc}(R')) \vee \mathit{child}(R') = C')$ $(I(R') = \lambda) \wedge$ $(\forall C'_0 \in \mathcal{S}'_c)(\mathit{repr}(C'_0) \neq C') \wedge \mathit{repr}(C') = \emptyset$	$I^+(C') = C$
$\upsilon'^{int}_a(A', A)$	Update interpretation of attribute A' to attribute A	$A' \in \mathcal{S}'_a \wedge (\ A = \lambda \vee (A \in \mathcal{S}_a \wedge \mathit{class}(A) = I(\mathit{intcontext}(A')))\ )$	$I^+(A') = A$
$\upsilon'^{int}_r(R', O)$	Update interpretation of association R' to directed image O of association R	$R' \in \mathcal{S}'_r \wedge \mathit{child}'(R') \in \mathcal{S}'_c \wedge (\ O = \lambda \vee ($ $O = (E_1, E_2) \wedge$ $\mathit{participant}(E_1) = I(\mathit{intcontext}(R')) \wedge \mathit{participant}(E_2) = I(\mathit{child}(R'))$ $))$	$I^+(R') = O$

Table 8.7: Atomic operations for updating interpretations

We demonstrate the operations in Figure 8.5. Figure 8.5(a) shows a starting PSM schema. It has an interpretation against the PIM schema depicted in Figure 8.4(a). The PIM schema evolves as we have demonstrated, so the consistency of the interpretation of the PSM schema is broken. In this example, we show how the PSM schema and its interpretation can be adapted using the introduced atomic operations to ensure the consistency. Figures 8.5(b)-(h) show particular evolutionary steps which result from changes at the PIM level demonstrated in Figure 8.4.

First, the designer needs to remove class `Partner'` (δ'_c(`Partner'`)) to reflect the first change in the PIM schema (removing class `Partner`). Prior to this, (s)he removes attribute `code'` (δ'_a(`code'`)) and both associations `partner'` and `customer'` (δ'_r(`partner'`), δ'_r(`customer'`)). Then, the designer creates a new association `customer'` connecting the schema class and class `Customer'` (*customer'* = α'_r(`CustomerDetailSchema'`, `Customer'`)) and sets its name using the appropriate operation (υ'^{name}_r(`customer'`, “*customer*“)). The association has an empty interpretation. The result is depicted in Figure 8.5(b).

Second, `address` was split into three new attributes in the PIM schema. The designer correspondingly needs to split attribute `address'` into three new attributes `street'`, `city'`, and `country'` in the PSM schema. (S)he creates attributes (`street'` = α'_a(`Contact'`), ...) and sets their names using the rename operation (υ'^{name}_a(`street'`, “*street*“), ...) and sets the interpretations accoridingly (υ'^{int}_a(`street'`, `street`), ...). Then, (s)he synchronizes the new attributes with the original attribute (σ'_a({`address'`}, {`street'`, `city'`, `country'`})). Note that the preconditions of both setting interpretations and synchronization is satisfied – the attributes are in the respective interpreted context and are within the same class. Finally, (s)he removes the old attribute `address'` (δ'_a(`address'`)). The result is depicted in Figure 8.5(c).

The designer then proceeds with extending the three new attributes with a

Notation	Description	Precondition	Postcond.
$\delta'_s(\mathcal{S}')$	Remove existing PSM schema $\mathcal{S}'$ and its interpretation I against $\mathcal{S}$	$\mathcal{S}' \in \mathcal{PSM} \wedge \mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{\mathcal{S}'}) \wedge \mathcal{S}'_a = \mathcal{S}'_r = \mathcal{S}'_m = \emptyset \wedge \mathcal{S}'_c = \{\mathcal{C}'_{\mathcal{S}'}\}$	$\mathcal{S}' \notin \mathcal{PSM}^+$
$\delta'_c(C')$	Remove class C'	$C' \in \mathcal{S}'_c \wedge \mathit{attributes}(C') = \mathit{content}(C') = \emptyset \wedge (\nexists C'_0 \in \mathcal{S}_c)(\mathit{repr}(C'_0) = C')$	$C' \notin \mathcal{S}'^+$
$\delta'_a(A')$	Remove attribute A'	$A' \in \mathcal{S}'_a$	$A' \notin \mathcal{S}'^+$
$\delta'_r(R')$	Remove association R'	$R' \in \mathcal{S}'_r \wedge$ $(\forall X' \in (\mathcal{S}_a \cup \mathcal{S}_r : \mathit{intcontext}(X') = \mathit{intcontext}(R') \wedge R' \in \mathit{anc}(X'))(I(X') = \lambda)$	$R' \notin \mathcal{S}'^+$
$\delta'_m(M')$	Remove cont. model M'	$M' \in \mathcal{S}'_m \wedge \mathit{content}(M') = \emptyset$	$M' \notin \mathcal{S}'^+$

Table 8.8: Atomic operations for removing PSM schemas and their components

Notation	Description	Precondition	Postcondition
$\sigma'_a(\mathcal{X}'_1, \mathcal{X}'_2)$	Synchronize set of attributes $\mathcal{X}'_2$ with set of attributes $\mathcal{X}'_1$	$\mathcal{X}'_1 \subseteq \mathcal{S}'_a \wedge \mathcal{X}'_2 \subseteq \mathcal{S}'_a$ $\wedge$ $(\exists C' \in \mathcal{S}'_c)$ $(\mathcal{X}'_1, \mathcal{X}'_2 \subseteq \mathit{attributes}(C'))$	$\mathit{equiv}^+(\mathcal{X}'_1, \mathcal{X}'_2)$
$\sigma'_r(\mathcal{X}'_1, \mathcal{X}'_2)$	Synchronize set of associations $\mathcal{X}'_2$ with set of associations $\mathcal{X}'_1$	$\mathcal{X}'_1 \subseteq \mathcal{S}'_r \wedge \mathcal{X}'_2 \subseteq \mathcal{S}'_r \wedge$ $(\exists C'_1 \in \mathcal{S}'_c, C_2 \in \mathcal{S}_c \cup \{\lambda\})(\forall R' \in \mathcal{X}'_1 \cup \mathcal{X}'_2)($ $(C'_1 = \mathit{parent}(R') \wedge \mathit{child}(R') \in \mathcal{S}'_c \wedge$ $(I(\mathit{child}(R')) = C_2)) \vee$ $(C'_1 = \mathit{child}(R') \wedge \mathit{parent}(R') \in \mathcal{S}'_c \wedge$ $(I(\mathit{parent}(R')) = C_2)))$	$\mathit{equiv}^+(\mathcal{X}'_1, \mathcal{X}'_2)$

Table 8.9: Atomic operations for synchronization of components of PSM schemas

new attribute `gps'`. (S)he creates the attribute ($\texttt{gps}' = \alpha'_a(\texttt{Contact}'), \ldots$) and sets its name ($\upsilon'^{name}_a(\texttt{gps}', \text{“}gps\text{“}), \ldots$) and interpretation ($\upsilon'^{int}_a(\texttt{gps}', \texttt{gps}), \ldots$). Then, (s)he specifies that the three original attributes are semantically equivalent to the extension ($\sigma'_a(\{\texttt{street}', \texttt{city}', \texttt{country}'\}, \{\texttt{street}', \texttt{city}', \texttt{country}', \texttt{gps}'\})$). The result is depicted in Figure 8.5(d).

Third, the designer moves the attributes `street'`, `city'`, and `country'` to a new class `Address'`. (S)he creates it ($\texttt{Address}' = \alpha'_c()$) and sets its name ($\upsilon'^{name}_c(\texttt{Address}', \text{“}Address\text{“})$) and interpretation ($\upsilon'^{int}_c(\texttt{Address}', \texttt{Address})$). The designer (s)he connects the new class with `Contact'` by creating a new association ($\texttt{address}' = \alpha'_r(\texttt{Contact}', \texttt{Address}')$). The designer sets the name of the association ($\upsilon'^{name}_r(\texttt{address}', \text{“}address\text{“})$) and interpretation ($\upsilon'^{int}_r(\texttt{address}', \texttt{address})$). Now the preconditions allow for moving the attributes from `Contact'` to `Address'` ($\upsilon'^{class}_a(\texttt{street}', \texttt{Address}'), \ldots$). The designer moreover needs to specify that a customer has one or two addresses ($\upsilon'^{card}_r(\texttt{address}', 1..2)$). The result is depicted in Figure 8.5(e). Later, (s)he similarly moves the attribute `gps'` to a new class `GPS'` and splits it into two new attributes `longitude'` and `latitude'` as depicted in Figure 8.5(f).

Fourth, the PIM schema now distinguishes two different addresses - shipping and billing. The designer needs to reflect this change in the PSM schema by

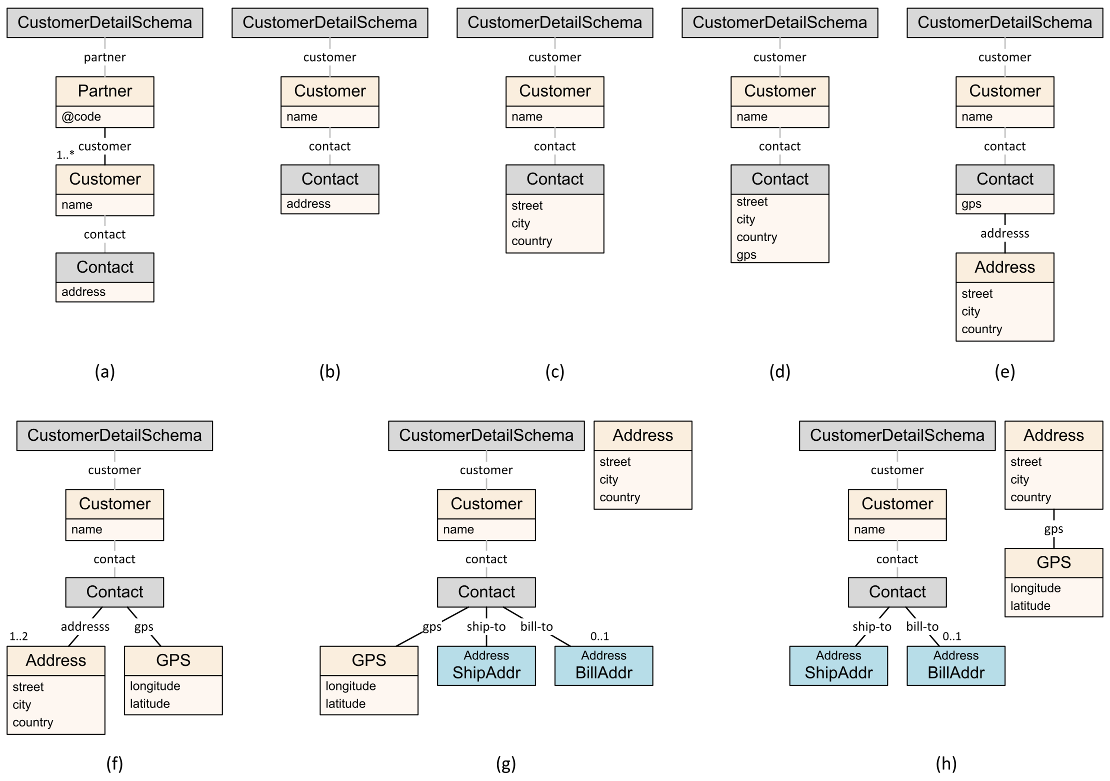


Figure 8.5: Evolution of a sample PSM schema demonstrating the introduced creation, update, removal and synchronization atomic operations

splitting the association `address’` correspondingly. However, the change is more complex than in case of the PIM schema, because the resulting PSM schema must be a tree. (S)he first needs to create new classes `ShipAddr’` and `BillAddr’`, set their names, and set their interpretation to PIM class `Address`. Now (s)he may split `address’`. (S)he creates two new associations `shipto’` and `billto’` connecting `Contact’` with `ShipAddr’` and `BillAddr’`, respectively. (S)he sets their names and interpretations to PIM associations `shipto` and `billto`, respectively. (S)he also sets cardinality of `billto’` to `0..1`. Then, (s)he synchronizes the original association with the new ones ($\sigma'_r(\{\texttt{address}'\}, \{\texttt{shipto}', \texttt{billto}'\})$) and removes the original one ($\delta'_r(\texttt{address}')$). (S)he wants both new addresses to model the same XML fragments as the original one and (s)he, therefore, sets `ShipAddr’` and `BillAddr’` as structural representatives of `Address’` ($\upsilon'^{repr}_c(\texttt{ShipAddr'}, \texttt{Address'}), \ldots$).

In the final step, the designer needs to reflect in the PSM schema reconnecting the association `gps` in the PIM schema. The impact of this change to the PSM schema is that both, shipping and billing address have GPS information. Therefore, the designer needs to reconnect `gps’` association to class `Address’`. This requires two atomic reconnections of `gps’`. First, from class `Contact’` to class `ShipAddr’` ($\upsilon'^{class}_r(\texttt{gps}', \texttt{ShipAddr}')$) and then to `Address’` ($\upsilon'^{class}_r(\texttt{gps}', \texttt{Address}')$). Note that both reconnections are allowed by the operation precondition. In the first case the reconnection is between classes connected by an association. In the other case, the reconnection is between a structural representative and its referenced class.

8.5 Propagation of Atomic Operations

According to Section 6.3, an interpretation of a PSM schema $\mathcal{S}'$ against a PIM schema $\mathcal{S}$ must be consistent. When $\mathcal{S}$ or $\mathcal{S}'$ is modified by an atomic operation, one or more conditions necessary for consistency may be violated and, consequently, the interpretation or the other schema must be adapted accordingly. We call the process which ensures the adaptation *propagation of the atomic operation*. In the example in the previous section we showed how a designer can solve this issue manually (our designer performed a sequence of operations in the PIM schema and then (s)he needed to perform similar steps in the PSM schema). In this section, we show how the propagation can be automated. If we consider the fact that there may be many PSM schemas affected, automation is very helpful.

8.5.1 Propagation from PIM to PSM Level

In this section, we describe how introduced atomic operations executed on the PIM schema $\mathcal{S}$ are propagated to each PSM schema $\mathcal{S}' \in \mathcal{PSM}$ and its interpretation I against $\mathcal{S}$. We will demonstrate the propagation on our sample PIM and PSM schema evolution depicted in Figure 8.4 and 8.5, respectively. We suppose that the designer manually changes the PIM schema in the steps depicted in Figure 8.4. In Section 8.4.2 we showed in Figure 8.5 how the designer manually adapts the PSM schema according to the changes in the PIM schema. In this section we show that our propagation mechanism is able to adapt the PSM schema automatically which reduces the designer's manual work.

Creating PIM Components Let us start with propagating the creation operations. Creating a new component X in $\mathcal{S}$ does not automatically imply the existence of any component in $\mathcal{S}'$. This is because the creation does not violate Definition 6.5. Moreover, X models a new part of the reality which has no representation in the PSM schemas, where its creation could be propagated. It is up to the designer, whether to create new components in the PSM schemas which represent this new part of the reality, or not. Therefore, the creation operations are not propagated.

Let us consider the evolution of our sample PIM schema depicted in Figure 8.4(e). Here, the designer first created a new class `Address` and association `address`. These operations on their own do not automatically result in creating new classes and associations in the PSM schemas. It is up to the designer whether to propagate them, or not. For example, (s)he later decides to move some attributes from `Customer` to `Address`. In that case it is necessary to create new classes with `Address` as their interpretation in the PSM schemas, because we need to correspondingly move attributes in the PSM schemas.

Updating PIM Components An update of a component X of $\mathcal{S}$ may have an impact on each component X' in the PSM schema with $I(X') = X$ and its propagation may be necessary. More specifically, an update of the name of X is propagated to an optional update of the name of X'. This is because X and X' do not necessarily need to share the same name. On the other hand, an update of the type or cardinality of X is propagated to a mandatory update of the type or cardinality of X'.

In our sample PIM schema evolution depicted in Figure 8.4(f), the cardinality of the association endpoint of association `address` connected to class `Address` was updated from 1..1 to 1..2. This is automatically propagated by our mechanism to all associations in the PSM schemas with `address` as an interpretation. For example, it is propagated to association `address`$'$ depicted in Figure 8.5(f).

The propagation of the two remaining update operations, i.e. moving an attribute and reconnecting an association end, is more complex. Both operations modify the structure of $\mathcal{S}$ which may break the consistency of the interpretation. The impact on the structure of $\mathcal{S}'$ may be quite extensive and it would be almost impossible for the designer to manage the impact manually. The idea of propagation is similar for both operations even though reconnecting an association end is technically more complicated. However, we will discuss only the first one.

Suppose that $v_a^{class}(A, D)$ was performed. In other words, an attribute A in the PIM schema was moved from its current class C to another class D. Consider an attribute A' in the PSM schema s.t. $I(A') = A$. Since the interpretation is consistent, we see that $class(A') = C'$ s.t. $intcontext(C') = C = class(A)$. By executing $v_a^{class}(A, D)$ we get $class(A) = D$. We see that, on one hand, A' is semantically an attribute of C. On the other hand, we see that the semantics is A which is an attribute of $D \neq C$.

Therefore, the move of A must be propagated to a corresponding move of A'. Concretely, we have to move A' to a class with an interpretation D to make the interpretation consistent. The move and its propagation is illustrated in Figure 8.6. Figure 8.6(a) contains a PIM schema fragment before executing the operation (on the left hand side of the thick arrow) and the fragment after the move (on the right hand side). Figures 8.6(b)-(c) contain three PSM schema fragments before and after the propagation. They illustrate three basic situations which may occur.

Suppose class C' in the PSM schema with interpretation C. Let A' be an attribute of C' with an interpretation A. The first situation is depicted in Figure 8.6(b). Here, C' contains an association R' with an interpretation R. Its child is class D' with an interpretation D. In this case the propagation means moving A' to D' which makes the interpretation consistent. The second situation is depicted in Figure 8.6(c). Here, there is an association R' with an interpretation R which goes to C'. Its parent is class D' with an interpretation D. Again, this case means moving A' to D'. The last situation is depicted in Figures 8.6(d). Here, there is no association connected to C' and with an interpretation R. In this case, propagation means creating a new association R' with an interpretation R connecting C' and a new class D' with an interpretation D. A' may be again moved to D'. Also there are some other situations which differ from the three demonstrated only in technical details. This includes situations with content models or classes without an interpretation on the path between C' and D'. We have solved these situations in our implementation but do not specify them in this chapter.

In a general case, there can be more and different associations $R_1, \ldots, R_n$ connecting C and D. There are associations $R'_1, \ldots, R'_n$ connected to C' with directed images of $R_1, \ldots, R_n$ as interpretations, respectively. If some R'_i is

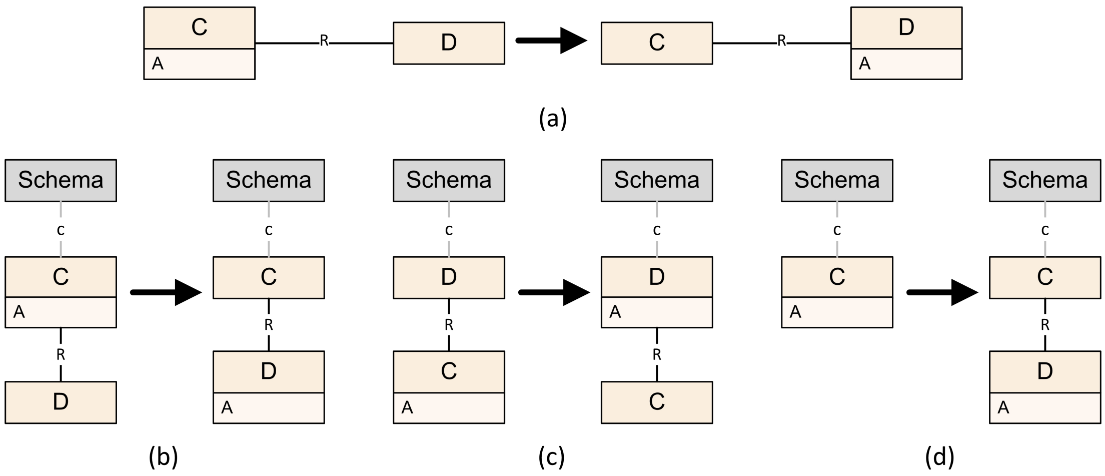


Figure 8.6: Visualization of the mechanism for propagating the operation for moving PIM attributes

missing, we ask a designer if it should be created[2]. If we apply the previous idea, we get up to $C'_{v,1}$, …, $C'_{v,n}$ classes, where A' should be moved. However, such move is not possible. Instead, we make a copy A'_i of A' for each $C'_{v,i}$ and move the copy to $C'_{v,i}$. Making a copy means the following sequence of atomic operations: (1) creating A'_i, (2) synchronizing it with A' (it is important since it specifies that A' and A'_i model the same information), (3) setting the properties of A'_i to the same values as A' and (4) moving A'_i.

In our sample evolution depicted in Figure 8.4(e), the designer moved attributes `street`, `city` and `country` from class `Customer` to class `Address`. This makes the interpretation of the PSM schema depicted in Figure 8.5(d) inconsistent. There are attributes `street`′, `city`′ and `country`′ having the moved PIM attributes as their interpretation. Our propagation mechanism ensures automatically that the attributes are moved correspondingly so that the interpretation is consistent again as depicted in Figure 8.5(e). First, the mechanism automatically creates a new class `Address`′ which was not present in the PSM schema and connects it with class `Contact`′ by a new association. Then, it automatically moves the attributes.

Removing PIM Components Removing components of $\mathcal{S}$ must be propagated by removing corresponding components of $\mathcal{S}'$ or setting their interpretations to λ to keep the interpretation consistent. More specifically, removing an attribute A leads to removing each attribute A' in $\mathcal{S}'$ s.t. $I(A') = A$ or setting $I(A') = \lambda$. Both solutions are correct (i.e. they do not break the consistency of interpretation) and, therefore, the designer has to decide. Removing an association leads mandatorily to removing each association R' in $\mathcal{S}'$ s.t. $I(R')$ is a directed image of R. We cannot set $I(R') = \lambda$. This is because condition (3) of Definition 6.7, R' with a non-empty interpretation has a child with an non-empty interpretation and vice versa. Setting $I(R')$ to λ would break this condition. And, finally, removing a class C leads to removing each class C' in $\mathcal{S}'$ s.t. $I(C') = C$ or setting

[2]If all of them are missing and the designer decides not to create any, no propagation is performed.

$I(C')$ to λ. Both possibilities are correct. From the precondition of the operation for removing a class, C has no attributes and there are no associations connected to C. Because of conditions (2) and (3) of Definition 6.7, there is no attribute or association in $\mathcal{S}'$ in the interpreted context of C' with an non-empty interpretation. Therefore, it is possible to set $I(C') = \lambda$. It is also possible to remove C'. However, it may have attributes and there may be associations connected to C' with empty interpretations. These must be removed first. There are also some technical details we do not discuss further. For example, parent ends of the associations going from C' may be reconnected to the parent of C' in certain cases etc.

In our sample evolution depicted in Figure 8.4(b), the designer removed association `responsibility` and class `Partner` with its attribute `code`. The propagation mechanism ensured that the corresponding components in our sample PSM schema were removed after a dialogue with the designer as depicted in Figure 8.5(b).

Synchronizing PIM Components Synchronizing two sets $\mathcal{X}_1$ and $\mathcal{X}_2$ of components of $\mathcal{S}$ means that the existence of both sets must be synchronized at all levels. Whenever there is an equivalent to $\mathcal{X}_1$ in the PSM schema $\mathcal{S}'$ there must also be an equivalent to $\mathcal{X}_2$ and vice versa.

The operation for synchronization of attributes, i.e. $\sigma_a(\mathcal{X}_1, \mathcal{X}_2)$, only enables one to synchronize two sets of attributes which have a common class C. Let C' be a class s.t. $I(C') = C$ which contains attributes whose interpretations are all attributes from $\mathcal{X}_1$. Since $\mathcal{X}_1$ and $\mathcal{X}_2$ are semantically equivalent, our propagation mechanism interprets this as a fact that C' must be supplemented with new attributes so that it contains attributes whose interpretations are all attributes from $\mathcal{X}_2$ as well (and conversely). (There are some technical details we do not discuss in more detail.) First, we have to consider also all attributes of $repr(C')$ if $repr(C') \neq \lambda$. Second, we have to consider all attributes with C' as an interpreted context, not only the attributes of C'.

In our sample evolution depicted in Figure 8.4(c), the designer split the original attribute `address` into three new attributes `street`, `city` and `country`. For this, after the creation of the new attributes (which is not propagated to the PSM level as we have already discussed), the designer synchronized the original attribute with the new ones. The synchronization is automatically propagated by our mechanism as follows: wherever there is an attribute `address`′ with interpretation `address` in a PSM schema, create three new attributes `street`′, `city`′ and `country`′ and synchronize them with `address`′. The result of this automatic propagation is depicted in Figure 8.5(c). Note that after the synchronization, the designer removed the original attribute `address`. This was propagated by our mechanism to removing the attribute `address`′ in the PSM schema after the decision of the designer.

The operation for synchronization of associations, i.e. $\sigma_r(\mathcal{X}_1, \mathcal{X}_2)$, is very similar to the previous case. Again, when their common class C' contains associations whose interpretations are directed images of all associations from $\mathcal{X}_1$, it must be supplemented so that it contains associations whose interpretations are directed images of all associations from $\mathcal{X}_2$, and vice versa. Again, there are technical details we omit for space limitations, i.e. we must not forget those associations that are implicitly in the content of C' – it is a structural representative and we must consider not only associations which have C' as a parent but all which have C' as their interpreted context.

Synchronization of associations is demonstrated in Figure 8.4(g), where the designer split association `address` into two new associations `shipto` and `billto`. The result of automatic propagation is depicted in Figure 8.5(g).

8.5.2 Propagation from PSM to PIM Level

In this section, we describe the opposite direction of propagation, i.e. how operations executed on a PSM schema $\mathcal{S}' \in \mathcal{PSM}$ are propagated to the PIM schema $\mathcal{S}$. Again, we will demonstrate the propagation on our sample PIM and PSM schema evolution depicted in Figure 8.4 and Figure 8.5, respectively. Now, we will, however, suppose that the designer manually changes the PSM schema according to the steps depicted in Figure 8.5. We show how our propagation mechanism ensures that the PIM schema is adapted automatically.

Creating PSM Components Creating a new component in $\mathcal{S}'$ does not directly imply an existence of any component in $\mathcal{S}$ and, therefore, creation operations are not propagated from PSM to the PIM level. For example, when creating a new class `Address′` and association `address′`, connecting `Contact′` and `Address′` in Figure 8.5(e) does not imply creating a corresponding class and association in the PIM schema. The creation will be performed in the PIM schema only when it is explicitly required by the designer. The propagation mechanism then also ensures that interpretations of the class and association in the PSM schema are set correctly. The result is depicted in Figure 8.4(e).

Updating PSM Components Updating components in $\mathcal{S}'$ has an effect on corresponding components in $\mathcal{S}$ with some exceptions; there are updates with no effect on the PIM schema $\mathcal{S}$, because the updated properties have no equivalent in $\mathcal{S}$. This includes updating a structural representative of a class, updating the position or XML form of an attribute and updating the position of an association. There are also updates which are only optionally propagated to $\mathcal{S}$. This is similar to the other direction; for example, changing a name of an attribute. And, there are operations which are propagated mandatorily. The simple case is, for example, updating a cardinality which is propagated in a straightforward manner. And, as in the other direction, there are two operations whose propagation is mandatory and more complex: moving an attribute and reconnecting an association end.

When the interpretation of a moved attribute or reconnected association is empty, the change is not propagated at all to $\mathcal{S}$. This is because the updated component has no equivalent in $\mathcal{S}$ and, therefore, consistency of interpretation is not broken. Similarly, no propagation is necessary when the interpreted context of the updated component has not changed. In that case, there is no change from the conceptual point of view.

For example, suppose the PSM schema in Figure 8.5(b). Let the designer move attribute `address′` from class `Contact′` to `Customer′`. The move is within the same interpreted context (which is class `Customer′`) and, therefore, the attribute was not moved from the conceptual perspective and its interpretation remains consistent. No propagation is necessary in this case.

In other cases, propagation is necessary. However, except for the technical details, the principles of the propagation are similar to the other direction and so, we omit their detailed description. For example, suppose that the designer moves attributes `street′`, `city′` and `country′` from class `Contact′` to class `Address` as depicted in Figure 8.5(e). The interpreted context is changed (from class `Customer′` to class `Address′`). Our propagation mechanism automatically ensures that the interpretations of the three attributes (i.e. `street`, `city` and `country`) are moved correspondingly in the PIM schema. The resulting PIM schema is depicted in Figure 8.4(e).

Removing PSM Components Similar to the updates, removing a component from $\mathcal{S}'$ is not propagated to $\mathcal{S}$ when the removed component has an empty interpretation. Removing a PSM component X' with an interpretation X may imply removing X when there are no other PSM components with interpretation X. However, even when there are no PSM components with interpretation X, we do not remove X' automatically. This is because PSM schemas are only views of the whole domain modeled by the PIM schema. Absence of a given concept modeled by X in the views does not imply the necessity of removing X from the PIM schema. The removal of X is, therefore, only optional.

For example, when the designer removes class `Partner′` as depicted in Figure 8.5(b), the propagation mechanism asks the designer whether the corresponding class `Partner` in the PIM schema should be removed as well, or not. In our case, the designer decides to remove `Partner` as depicted in Figure 8.4(b).

Synchronization of PSM Components Synchronization of two sets $\mathcal{X}_1'$ and $\mathcal{X}_2'$ is propagated from $\mathcal{S}$ to $\mathcal{S}'$ very similarly as in the opposite direction. The only difference is that there are components in $\mathcal{X}_1'$ and $\mathcal{X}_2'$ with and without an interpretation. If $\mathcal{X}_1'$ (or $\mathcal{X}_2'$) contains only components with an interpretation, its semantic equivalent exists in $\mathcal{S}$ and each component X' of $\mathcal{X}_2'$ (or $\mathcal{X}_1'$), respectively, which does not have an interpretation is, therefore, propagated to $\mathcal{S}$. Propagation means creating a new component X corresponding to X' and setting $I(X') = X$. Otherwise, the synchronization is not propagated to the PIM level, because an equivalent of $\mathcal{X}_1'$ or (or $\mathcal{X}_2'$, respectively) does not exist in $\mathcal{S}$.

Sample synchronization operations are demonstrated in Figure 8.5(c) (attribute synchronization) and Figure 8.5(g) (association synchronization). They are automatically propagated by our mechanism to the PIM schema as depicted in Figure 8.4(c) and Figure 8.4(g), respectively.

8.5.3 Minimality and Correctness of Atomic Operations

Important properties of any set of atomic operations are their minimality and correctness. Minimality means that there is no atomic operation which could be expressed as a sequence of other atomic operations. Correctness means that the proposed operations are correct. In our specific case it means not only that an atomic operation transforms a schema from a consistent state to another consistent state but also that the propagation mechanism preserves the consistency of interpretations of PSM schemas to PIM schemas.

Theorem 8.1 *The set of atomic operations is minimal.*

Proof 8.1 *Assume the operations for evolution of classes in the PIM schema, i.e.* α_c, δ_c *and* υ_c^{name}. *Without* α_c *we are not able to create any class. Similarly, without* δ_c *we are not able to remove any class. Finally, without* υ_c^{name} *we are not able to change the class name. It cannot be set during the creation, because* α_c *sets a default name. The proof for other atomic operations for creating, removing and updating PIM and PSM components is similar. The operations for synchronizing two sets of attributes or associations are clearly atomic as well. No other operation allows for synchronization.*

Theorem 8.2 *The set of atomic operations together with the propagation mechanism is correct.*

Proof 8.2 *We have already proved the correctness in the previous text. In Section 8.4.2 we have shown that the preconditions of operations for updating interpretations of PSM components ensure that the consistency of interpretation can not be broken. In Sections 8.5.1 and 8.5.2 we have shown that the propagation mechanism repairs the consistency of interpretation when broken by moving attributes, reconnecting associations ends and removing components. We have also shown that the other operations do not touch the consistency at all. And, finally, we have shown in these sections that whenever the propagation mechanism needs to perform a sequence of atomic operations to repair the consistency of interpretation, the preconditions of these operations are always satisfied so that the sequence may be performed in any time.*

8.5.4 Completeness of Atomic Operations

Sometimes completeness is understood as a property which ensures that for any two given schemas there always exists a sequence of atomic operations which transform one of the schemas to the other. The sequence usually removes all components of the former schema and creates the components of the other. This is not a correct proof of completeness, because it does not consider possible semantic relationships between the components of both schemas. The old components are simply removed and the new ones are created without preserving the semantic relationships. However, this only covers the structural part of the schema. What we also aim for is preserving the semantic part of the schemas. This is largely dependent on the user and his/her interpretation of the meaning of the schemas.

However, even if semantic relationships are considered (e.g. semantic equivalence in our case) it is not easy to prove general completeness formally. Even though such proof would be interesting from the theoretical point of view, it is beyond the scope of this chapter. Instead, our aim is to demonstrate completeness practically. In this chapter we provide a case study of a real world system, where we applied our approach. It experimentally shows that the proposed set of atomic operations is complete. The case study can be found in Section 8.8.

8.6 Composite Operations

The atomic operations introduced formally in the previous sections were proposed so that they form minimal and correct set as proven above. Naturally, they are not supposed to be used directly by the user in all cases and it is not the whole set of available operations. In this section we show examples how the atomic operations can form more user-friendly and realistic *composite* operations.

Formally, a composite operation is a sequence of two or more atomic operations. As we have shown in the previous text, propagation mechanism ensures that any atomic operation does not corrupt the consistency of affected interpretations. Therefore, composition of atomic operations preserves consistency as well and it is not necessary to extend the propagation mechanism with specifics of the composition.

Creation with Parameters A simple composite operation necessary in every system is creating a particular component with pre-set values. We show such case in the operation `createPIMAttr`(C, n, t, c) which allows for creating of a PIM attribute in a class C with name n, data type t and cardinality c. It consists of the following steps:

$$A = \alpha_a(C); \quad \upsilon_a^{name}(A, n); \quad \upsilon_a^{type}(A, t); \quad \upsilon_a^{card}(A, c)$$

The propagation mechanism optionally creates corresponding PSM components.

Splitting of a PIM Attribute This operation is a typical example of *drill-down modeling*, i.e. creating more and more precise data structures. An example of such an operation is shown in Figure 8.4(c), where the designer needs to detail a single-valued address of a customer to street, city and country. In general, the composite operation `splitPIMAttr`(A, $\{n_1, n_2, ..., n_k\}$) for splitting a PIM attribute A of a class C to a set of attributes with names $\{n_1, n_2, ..., n_k\}$ consists of the following steps:

$$\begin{aligned}
&A_1 = \texttt{createPIMAttr}(C, n_1, type(A), card(A));\\
&\ldots;\\
&A_k = \texttt{createPIMAttr}(C, n_k, type(A), card(A));\\
&\sigma_a(\{A\}, \{A_1, A_2, ..., A_k\}); \quad \delta_a(A)
\end{aligned}$$

The propagation mechanism, in particular in case of synchronization, ensures that all the PSM attributes representing A are replaced with PSM attributes representing A_1, A_2, ..., A_k.

In our sample depicted in Figure 8.4(c), the designer would execute a single composite operation `splitPIMAttr` (`address`, {“*street*“, “*city*“, “*country*“}).

Removing a PSM Tree In the previous case we have shown an example of a composite operation which consists of a sequence of atomic operations and a composite operation which consists of atomic and other composite operations. Operation `removePSMtree`(C') for removing a PSM tree rooted at class C' is an example of a recursive composite operation, i.e. it calls itself if necessary. The operation consists of the following steps:

1. $(\forall R' \in content(C'))$ `removePSMtree`$(child(R'))$;
2. $(\forall A' \in attributes(C'))\ \delta_a(A')$;
3. $(\forall R'_p \in \mathcal{S}'_r \text{ s.t. } child(R'_p) = C')(\delta_r(R'))$;
4. $\delta_c(C')$;

Naturally, we cannot provide the full list of possible composite operations, as the particular set depends on the choice of the vendor of a particular system and the requirements of users. Our aim was to demonstrate that the proposed mechanism can be used in real-world situations.

8.7 Related Work

The current approaches towards evolution management can be classified according to distinct aspects [74, 34]. The changes and transformations can be expressed [104, 21] as well as divided [28] variously too. However, to our knowledge there exists no general framework comparable to our proposal in Section 8.3; particular cases and views of the problem have previously only been solved separately, superficially and mostly imprecisely without any theoretical or formal basis. In this section we describe the closest and most advanced approaches related to our proposal.

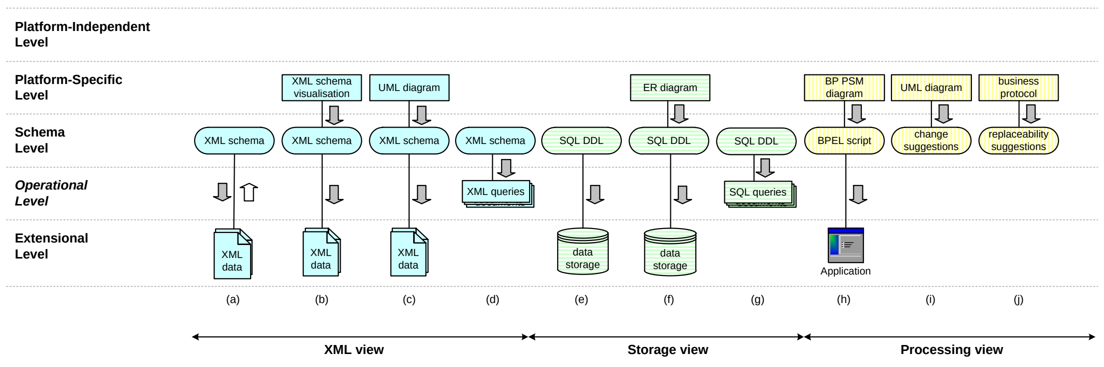


Figure 8.7: Current XML evolution approaches

XML View We can divide the current approaches to XML schema evolution and change management into several groups as depicted in Figure 8.7. Approaches in the first group (a) consider changes at the schema level and differ in the selected XML schema language, i.e. DTD [2, 31] or XML Schema [127, 24]. In general, the transformations can be variously classified. For instance, paper [127] proposes *migratory* (e.g. movements of elements/attributes), *structural* (e.g. adding/removal of elements/attributes) and *sedentary* (e.g. modifications of simple data types). The changes are expressed variously and more or less formally. For instance in [24] a language called *XSUpdate* is described. The changes are then automatically propagated to the extensional level to ensure validity of XML data. There also exists an opposite approach that enables one to evolve XML documents and propagate the changes to their XML schema [22]. Approaches in the second (b) and third (c) group are similar, but they consider changes at an abstraction of logical level – either visualization [53] or a kind of UML diagram [39]. Both cases work at the PSM level, since they directly model XML schemas with their abstraction. No PIM schema is considered. All approaches consider only a single separate XML schema being evolved.

Another open problem related to schema evolution is adaptation of the respective XML queries, i.e. propagation to the operational level (Figure 8.7(d)). Unfortunately, the amount of existing works is relatively low. Paper [80] gives recommendations on how to write queries that do not need to be adapted for an evolving schema. On the other hand, in [44] the authors consider a subset of XPath 1.0 constructs and study the impact of XML schema changes on them.

In all the papers cited the authors consider only a single XML schema. In [111] multiple *local* XML schemas are considered and mapped to a *global* object-oriented schema. Then, the authors discuss possible operations with a local schema and their propagation to the global schema. However, the global schema does not represent a common problem domain, but a common integrated schema; the changes are propagated just upwards and the operations are not defined rigorously. The need for well defined set of simple operations and their combination is clearly identified in Section 6 of a recent survey of schema matching and mapping [14].

Storage View The idea of evolution and change management in XML storage strategies is currently focused particularly on data updates and, usually, joined with *XQuery Update Facility* [25]. However, this is not the area we are dealing with since the updates are mostly considered within the respective XML schema. As depicted in Figure 8.7(e), in the area of evolution of general database schemas we can find approaches that focus on evolution of (object-)relational schemas [32, 33] as well as object-oriented schemas [12, 68]. Similar to the case of XML schema evolution, there are also approaches that deal with propagation from an ER schema, i.e. PSM level, to a relational schema [8], i.e. schema level (Figure 8.7(f)) or propagation to an operational level [32] (Figure 8.7(g)).

In the purely XML-related approaches we need to consider *schema-driven* storage strategies. As surveyed in [122], the amount of the respective approaches is not high. We can find first attempts of change propagation in the current leading object-relational database management systems – *Oracle DB*[3], *IBM DB2*[4] and *Microsoft SQL Server*[5]. In this case we can differentiate two types of schema evolution – whether *backward compatibility* of the changes, i.e. preservation of data validity, is required, or not. Both the DB2 and SQL Server require the backward compatibility. Oracle DB also supports change propagation regardless backward compatibility; however, it is not done automatically; a data expert must provide an XSLT script which re-validates the stored XML documents. To ease this approach we have recently proposed an algorithm that enables one to provide such transformation script semi-automatically [71].

[3]`http://www.oracle.com/us/products/database/`

[4]`http://www-01.ibm.com/software/data/db2/`

[5]`http://www.microsoft.com/sqlserver/2008/en/us/`

Processing View Since we are considering the area of evolution of XML applications, we cannot omit the most popular application of XML format – Web Services. Currently we can find several approaches that deal with evolution of Web Service; however, again they solve just part of the issues described [1]. In [123] the authors describe a plugin to IBM *Rational Software Architect* (RSA)[6] which enables semi-automatic propagation of changes from business process model of Web Services (Figure 8.7(h)) to respective BPEL scripts and thus respective applications. It is one of the frameworks that are very close to our proposal described in Section 8.3; however, the authors do not provide any theoretical background on the allowed changes or details on the propagation mechanisms. A different approach (Figure 8.7(i)) is used in system *Morpheus* [114], also based on IBM RSA. At the platform-specific level it considers three UML artifacts – use cases, sequence diagrams and service specifications – and the change propagation among them. The output of the propagation is a set of change suggestions for the respective execution part which should be then done manually by an expert. Similarly, in [118] (Figure 8.7(j)) the authors deal with change propagation of business protocols of Web Services, i.e. a kind of activity diagrams. The output of the system is a set of recommendations detailing when affected parts are replaceable/migrateable and under what circumstances. Again, the migration is expected to be done by a system expert; however, the system advises how to perform it correctly.

In [9] the authors solve the problem using a completely different strategy. They provide an *abstract service definition model* (ASD) which enables us to model all related concepts of a Web Service, i.e. data structures, behavior and policies at a conceptual level using UML class diagrams. Both ASD and the related operations are defined formally and the completeness and correctness of the operations is proven. On the other hand, change propagation to respective PSMs is not considered and the ASD itself is relatively unnatural. And, considering even more formal approaches and model, in [10] the authors model the Web Services using Formal Concept Analysis and, in particular, lattices or in [125] using lenses and monoids of edits. However, though the approaches are theoretically very interesting, our aim is to provide less complex and more user-friendly formal background and tools.

8.8 Case Study and Evaluation

As has already been mentioned, we have implemented the proposed technique in a tool called *eXolutio* [55]. In general, it is a proof-of-concept desktop application for conceptual XML data modeling. It implements the PIM and PSM modeling languages and operations for evolution of the PIM and PSM schemas described in this chapter. It provides a designer with a set of operations which are composed of the atomic operations described in Section 8.4. It implements the propagation mechanism introduced in Section 8.5. At the highest level, eXolutio is based on a well known Model View Controller (MVC) design pattern.

Currently, for the purpose of this chapter, the atomic operations are implemented in the exact same way they are described here. We use the implementation

[6]`http://www-01.ibm.com/software/awdtools/architect/swarchitect/`

to experimentally demonstrate that the proposed set of atomic operations is *complete*, i.e. that the atomic operations are sufficient for real-world situations. As we have already discussed in Section 8.5.4, we do not prove completeness formally in this chapter. As to *performance* and *scalability*, it is a fact that a single atomic operation on a PIM schema can lead to a large number of operations in each of the affected PSM schemas. This number can be reduced by some optimizations, improving both performance and scalability. So far, our implementation is strictly based on our *formal model* and focuses on the clear demonstration of our ideas. The issues of performance and scalability will be addressed in later stages of development. It is now clear that some complex operations are far more efficient if they are implemented from scratch, rather than by combining the individual atomic operations. This also holds true for some cases of change propagation. Still, it will always be necessary to prove that the optimized version of the operation has the same formal properties as the non-optimized version would have, which is possible again thanks to our formal model. In addition, many operations are in fact interactive. For example, the designer will choose to which PSM schemas a change will be propagated, in which case the actual time spent by performing the operation will always be comparatively negligible.

In the concluding part of this section we show how the developed technique for designing a family of XML schemas and their evolution on a real-world system was applied. And, finally, we evaluate our technique on the basis of this case study and compare it with other known techniques for XML schema evolution.

8.8.1 Case Study: National Register for Public Procurement

Our case study is the *National Register for Public Procurement* (NRPP)[7]. It is a governmental information system intended for publishing data about public contracts by public authorities in the Czech Republic. Publishing a contract is only obligatory when the contracted price exceeds a level given by the current legislation; otherwise, it is optional. Authorities send contract information formatted according to one of the 17 XML formats accepted by the NRPP. This includes, e.g. XML format for contract notifications, supplier selection notifications, etc.

Currently, the NRPP only provides a textual documentation for the XML formats and a set of sample XML documents. There are neither XML schemas for the XML formats, nor a conceptual schema of the problem domain. Therefore, our first goal was to design not only the XML schemas but also the conceptual schema in a form of a PIM schema and derive PSM schemas for the XML formats from the PIM schema. The resulting PIM schema is depicted in Figure 8.8 (a). Two of the resulting PSM schemas are depicted in Figures 8.9 (a) and 8.9 (b). The PSM schemas are mapped to the PIM schema. The mapping is intuitive and we do not describe it here. The PSM schemas were created exactly according to the textual documentation and XML examples. Let us note that the original schemas we created are more extensive. Due to space limitations, we present here only those parts that bear on our work.

The PIM schema contains classes which model public contracts (class *Contract*) and their procurers and suppliers (class *Organization*). There are also some

[7]`http://www.isvz.cz` (in Czech only)

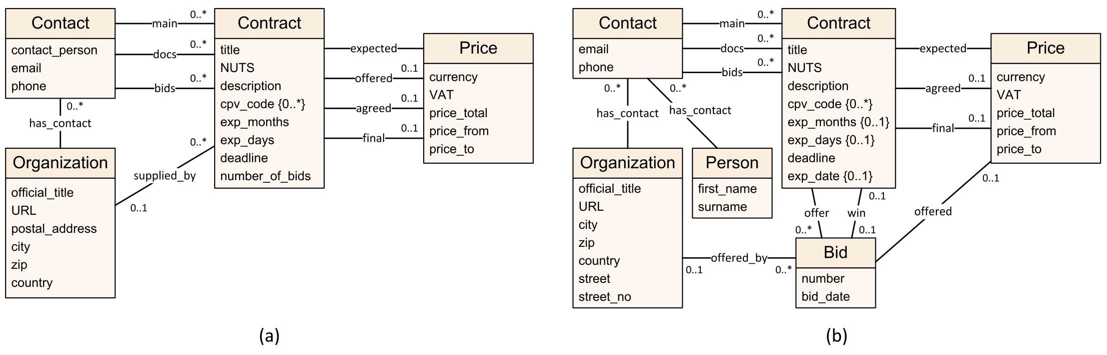


Figure 8.8: (a) PIM schema modeling the NRPP domain, (b) PIM schema evolved according to new requirements.

additional concepts modeled – prices (class *Price*) and contact information (class *Contact*). There are several relationships modeled with associations. A supplier is associated with a contract by *supplied_by* association. A procurer is associated with a contract by a path of associations *has_contact* and *main*. Each contract has additional contact information – where documentation for the contract is provided (association *docs*) and where bids to the contract are collected (association *bids*). Finally, there are four different prices – expected price (association *expected*), the best offered price (association *offered*), price agreed by a selected supplier and procurer (association *agreed*), and a final real price known after finishing the contract (association *final*).

The PSM schema depicted in Figure 8.9 (a) models an XML format for notifications about a new public contract. When a public authority issues a new contract, it must send a notification about the contract to the the NRPP using this format; it should contain contact information and basic information about the contract. The other PSM schema depicted in Figure 8.9 (b) models an XML format for notifications about the supplier selected for the contract; it contains the main contract contact, information about the number of offered bids, selected supplier and offered and agreed price.

The numbers of atomic operations executed to create the PIM and PSM schemas are depicted in Figure 8.10(a). It shows that only creation and update operations were used.

There were several issues to solve in this case study. First, the NRPP provides only XML formats which are used by public authorities to send data about their contracts to the NRPP. There are no XML formats for providing information back to the public authorities and other users, e.g. procurer or supplier detail. We show how our approach may be used to easily design such XML formats in a form of PSM schemas on the basis of the existing PIM schema. One such PSM schema which models XML formats for public procurer details is depicted in Figure 8.9 (c). The numbers of the atomic operations executed at this step are depicted in Figure 8.10(b). Again, only the creation and update operations were performed. Even though the designer needs to design the PSM schemas for the new XML formats manually, the experiment clearly showed that our approach saves him/her a great deal of work and prevents him/her from making unnecessary errors. This is because our technique enables us to create the PSM schemas on the basis of the PIM schema (which is quicker than creating PSM schemas separately)

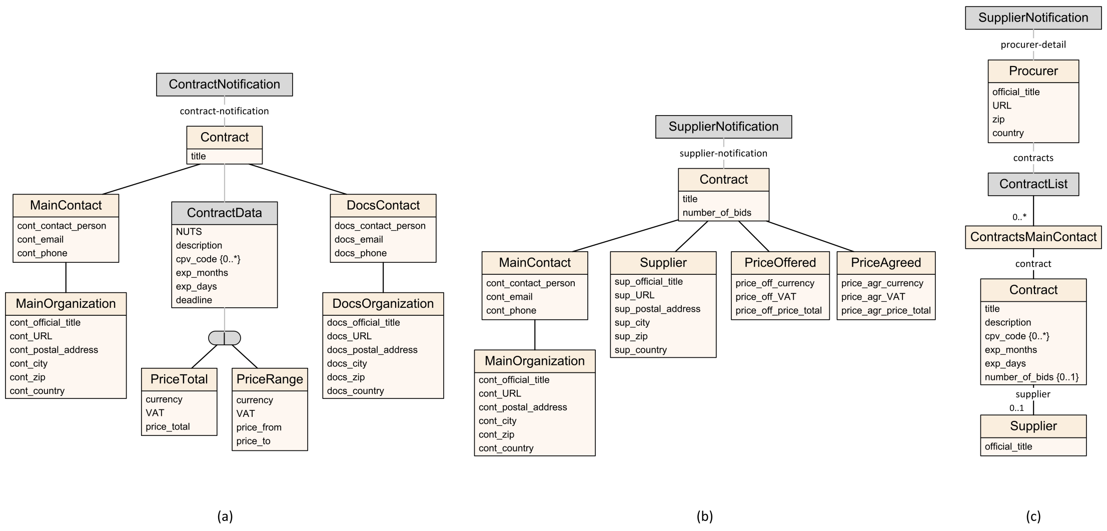


Figure 8.9: PSM schemas modeling XML formats for (a) sending contract notifications to NRP, (b) reporting on contract supplier selection to the NRPP, and (c) representing procurer detail

and ensures that the designer creates the PSM schemas coherently with the PIM schema (as it preserves the consistency of the interpretation). The designer needs not check whether the PSM schema is semantically correct, or not.

Second, as the reader may have noticed, the quality of the XML formats is low. The designers of the XML formats did not follow basic XML design principles (e.g. exploiting the hierarchical nature of XML); for example, contact information is modeled by XML elements with names prefixed with `cont_`, `docs_`, etc. It would have been better to remove the prefixes and enclose the semantically related XML elements into separate XML elements (e.g. enclose contact XML elements to XML element `contact` structured to `main`, `doc`, etc. or enclose all information related to the supplier into XML element `supplier`). We have made these adaptations in the present XML formats. Some PSM schema components also appeared which had the same content and we, therefore, used structural representatives to declare the shared content only once. The numbers of the executed atomic operations are depicted in Figure 8.10(c). In this step, synchronization and removal operations were also used, because some of the old parts of the PSM schemas were replaced by new ones. Again, the experiment demonstrated that our approach saves a lot of work as it preserves the consistency of PSM schemas against the PIM schema. If the designer makes a change which affects the PIM schema and, possibly, other PSM schemas, our propagation mechanism will notify him/her. We depict the evolved PSM schema from Figure 8.9 (b) in Figure 8.11(b) (it also includes changes described in the following steps). The other PSM schema was evolved similarly. As the reader may see, contact information is now represented hierarchically. Also, the PSM schema is simplified by using structural representatives referring to shared classes contained in a new separate PSM schema depicted in Figure 8.11(a).

Third, we implemented various changes which resulted from new requirements on the NRPP functionality and from new legislation. In both cases, changes to the PIM schema needed to be done. The new functionalities required us to model

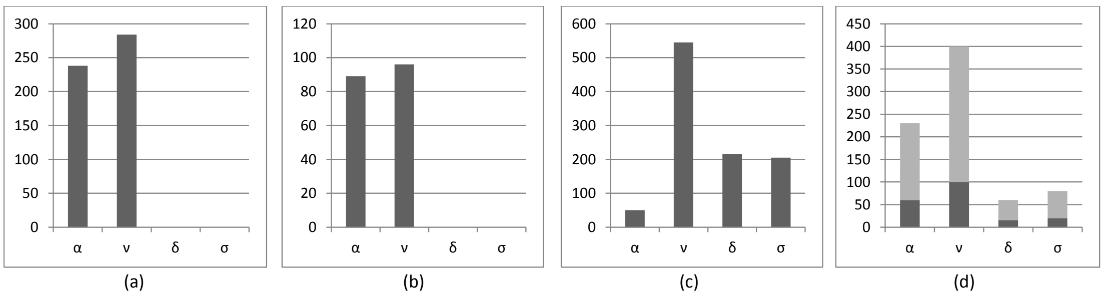


Figure 8.10: Numbers of atomic operations performed manually by the designer (dark gray) and automatically by the propagation mechanism (light gray)

contact persons as a special class instead of attribute *contact_person*. Therefore, we evolved the attribute to a new class *Person* associated with *Contact* and with two new attributes *first_name* and *surname* using our evolution operations.

The new legislation required to report not only the number of bids received for each contract, but also particular bids including the bidding supplier and offered price. Therefore, we replaced the attribute *number_of_bids* with a new class *Bid* with several new attributes. We changed the semantics of *supplied_by* and *offered* associations by reconnecting them from *Contract* class to the new *Bid* class. Finally, we distinguished the winning bid from the other bids by splitting the association connecting *Bid* and *Contract* classes into two associations *offer* and *win*. The evolved PIM schema is depicted in Figure 8.8 (b). Since the PIM schema changed, the PSM schemas needed to be adapted accordingly. This was ensured by our propagation mechanism. Figures 8.11(b) and 8.11(c) show how PSM schemas depicted in Figures 8.9 (b) and 8.9 (c) were automatically adapted by the propagation mechanism, respectively.

Finally, there was a requirement to update the XML format for contract notifications (Figure 8.9 (a)) so that it is possible to give notification not only on the expected months and days in which the contract should be finished, but also on the exact date. Therefore, we added a new attribute *exp_date* which can be used equivalently instead of two present attributes *exp_months* and *exp_years*. This change was correctly propagated to the PIM schema, because it is a conceptual change (see Figure 8.8 (b)). From here, it was propagated to the other PSM schemas (see Figure 8.11(c)).

The numbers of the atomic operations executed during the last two steps are depicted in Figure 8.10(d). The darker part shows the numbers of manually executed operations. The lighter part shows the numbers of operations executed automatically by the propagation mechanism.

8.8.2 Evaluation and Comparison to Other Approaches

The following conclusions are drawn from the above case study:

- All proposed atomic operations are necessary for real-world scenarios as summarized in Figure 8.10. The necessity for the creation and updating of atomic operations is clear. The case study showed that we also need removal operations even though we do not want to directly remove parts of

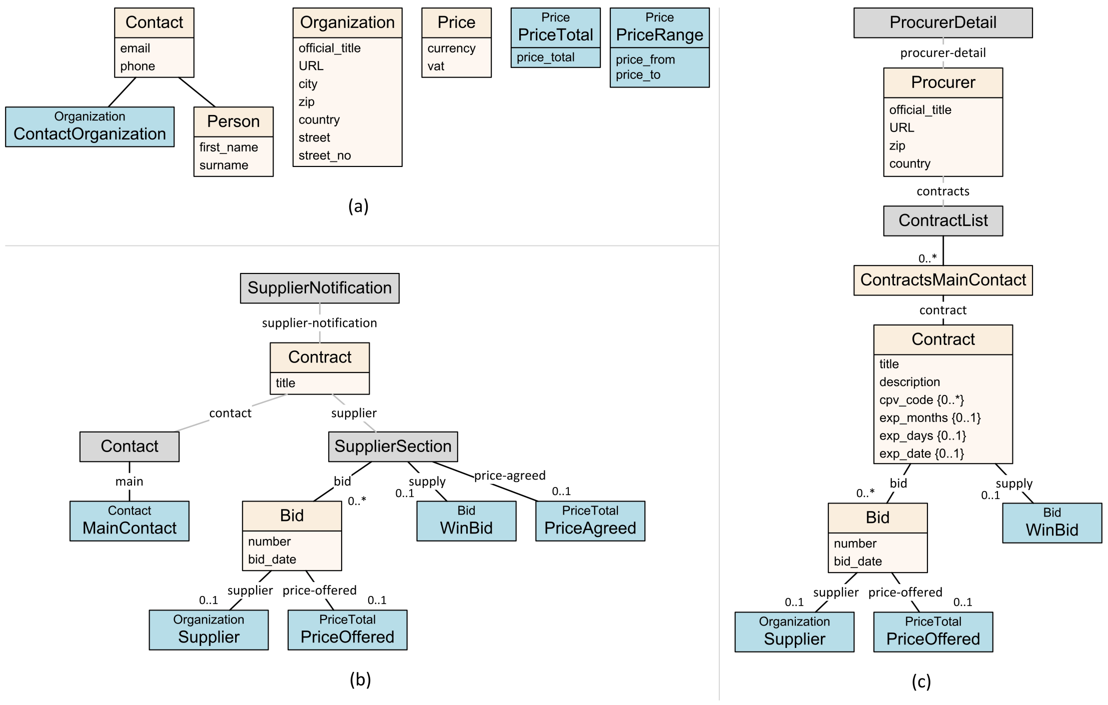


Figure 8.11: (a) PSM schema with common components shared between other PSM schemas, (b) evolved PSM schema for reporting on contract supplier selection to the NRPP, (c) evolved PSM schema for representing procurer detail

data but represent them in more (or less) detailed structures (e.g. splitting attributes). For this, we also need synchronization operations.

- The case study also demonstrates the completeness of the proposed set of atomic operations. Most real-world scenarios we target in our work will be similar to the presented case study (i.e. extending existing schemas with new parts and replacing their existing parts with more (or less) detailed alternatives). For this kind of scenarios our proposed set of operations is complete. On the other hand, there are some limitations. For example, when synchronizing two sets of attributes, we can not exactly specify a function which would transform values between both sets. However, we are not interested in data transformations in this chapter but only PIM and PSM schema evolution.

- The existence of the PIM schema and interpretations of PSM schemas against the PIM schema is beneficial when the designer performs creation and update operations for building new PSM schemas or new parts of existing PSM schemas. Our technique ensures that the designer creates new PSM components consistently with the PIM schema (from the conceptual perspective). This ensures semantical coherence between the modeled family of XML schemas. All XML schemas in the family, even those designed by different developers, are consistent with the PIM schema. The designers need not check this coherence manually which saves them a great deal of work and prevents design errors.

- Sometimes the designer may want to optimize the structure of an XML schema but avoid changes to the semantics of the XML schema. When the designer works with the PSM schema, our mechanism is able to prevent these changes. It can automatically check whether a change to the PSM schema needs to be propagated to the PIM schema, or not. This also saves the designer a lot of work, because (s)he does not need to check this manually.

- Finally, when the designer needs to change the PIM schema, our mechanism automatically propagates the changes to the PSM schemas and vice versa. Again, this saves work and prevents errors, because the designer does not need to propagate the changes manually.

Figure 8.10(a)-(d) shows the number of atomic operations performed by the designer in our case study. In comparison to existing approaches to XML schema evolution, our technique saves the designer a great deal of manual work. This is because we consider the PSM schemas interpreted against a single common PIM schema. As we have shown this saves work and prevents errors when the designer needs to check the semantical consistency of his/her new or evolved part of a PSM schema and when making changes to PIM schema or PSM schemas and their propagation to the other schemas. The amount of work saved in comparison to other approaches is demonstrated by Figure 8.10. The darker columns show the amount of atomic operations performed manually by the designer. These operations are assisted by our technique which ensures that the consistency between the created XML schemas is preserved. The designer does not need to check consistency manually which saves a lot of time. This consistency check is not provided by existing approaches, where the designer has to do the check manually. The lighter columns show the amount of atomic operations performed automatically by our propagation mechanism. Again, propagation is not supported by existing approaches and these operations would have to be done manually by the designer.

We can also see a fundamental problem in the current approaches, because they do not consider synchronization operations or their equivalent. Without this operation a correct propagation between PIM and PSM schemas is not possible. As we have shown, this is necessary in various practical situations when a part of a PIM or PSM schema is split into more detailed parts. It is also useful in extending an existing part with new components, as well as in a reversed process when more parts of a schema are merged together.

On the other hand, our approach is more laborious in the initial phases, because the PIM schema and PSM schemas modeling the XML schemas must be created. This is not the case of the other approaches which work directly with an XML schema or its direct translations to a conceptual schema. Therefore, the other approaches are more suitable in situations, where the designer works only with a single XML schema. When a family of XML schemas needs to be managed, our approach is more beneficial.

Finally, let us note that the approach presented deals only with PIM and PSM schemas and propagation of changes between both levels. It does not solve the problem of propagation of changes to the data, i.e. XML documents. As we have shown, this has been solved by other approaches. We have also worked on this problem in our previous work. In [71] we show how XML documents need to be adapted when a PSM schema which models their XML schema evolves.

8.9 Conclusions

In this chapter we focused on two of the main challenges of model driven development [43] – evolution and its formal specification. In particular, we were interested in model driven XML schema evolution and concentrated on the PIM and PSM levels of our previously proposed five-level evolution framework. We defined PIM and PSM schemas for modeling XML schemas formally and extended them with atomic and composite operations for their modification. We then identified minimal set of atomic operations, proved its correctness and specified the respective mechanism for automatic propagation of changes between PIM and PSM levels. The formal basis of the operations enables us to ensure that the framework is designed correctly. Next we introduced implementation of the framework and depicted the advantages of the system in a real-world use case.

Key Contributions If we compare the proposed system with the current approaches, we can identify several key contributions and innovations it brings:

- *Global View of the Evolution Problem:* As mentioned in Section 8.7, the existing approaches towards the evolution and change management of XML schemas consider only a single XML schema. Our proposed technique considers a family of XML schemas applied in a system.

- *Formal Basis of the Proposal*: Similar to the current work being done, we exploit the idea of a platform-independent conceptual schema (PIM schema) of the problem domain which allows for abstraction from technical details of particular XML schemas. We also consider a platform-specific (PSM) schema for each targeted XML schema. Each PSM schema is mapped to the PIM schema and can be automatically translated to an expression in a selected XML schema language such as XSD or RELAX NG. We defined PIM and PSM schemas and mappings between them formally, which enables us to effectively manage the evolution of XML schemas. When a change to the PIM schema is made, we can precisely identify all the parts of XML schemas affected by this modification and, conversely, when a change to the PSM schema is made, we can identify whether the PIM schema is affected and how, or not.

- *Hierarchy of Operations*: Naturally, the idea of change management is based on a set of operations. As we have mentioned, they can be classified variously and current approaches utilize different sets. In our work we defined a set of atomic operations and proved its minimality and correctness. Having this concept, we could restrict ourselves to this set and define the respective change propagation precisely and correctly. Last but not least, we showed that using the set of correctly defined atomic operations and the respective change propagation we can define any composite and, hence, more user-friendly operation. The respective change propagation is then defined implicitly and its correctness is ensured as well. A system of operations similar to this was identified in a recent survey [14] (Section 6), but has not yet been researched properly.

- *Experimental Implementation of the Proposal:* The final contribution of this chapter is not only the proposal itself, but also its experimental and open-source implementation *eXolutio*. Even though it currently does not cover all the proposed aspects (it is still under intense development), a user may test the key features for his/hers real-world examples. For instance, recently it has been tested in real-world use-cases by the *Fraunhofer Institut*[8].

[8]`http://www.isst.fraunhofer.de/dasinstitut/`

9. Inheritance in Conceptual Modeling for XML

In this chapter we extend the conceptual model for XML from Chapter 6 with modeling of inheritance relations that we previously omitted for simplicity. This is the first part of our core and most important contribution to XML schema evolution.

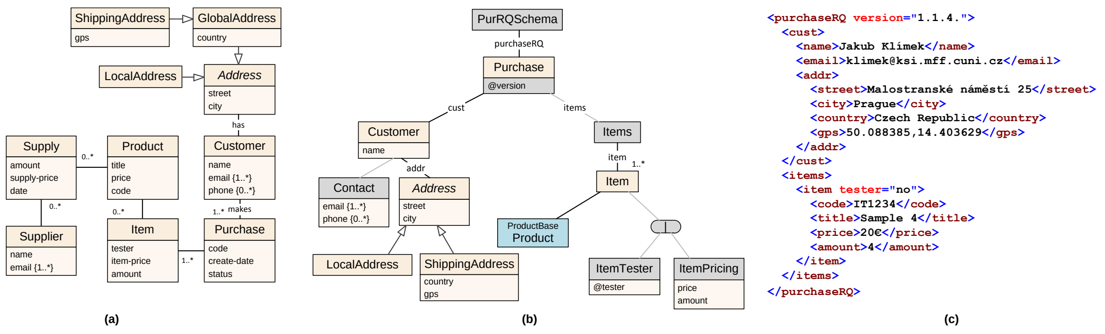
The contents of this chapter was published as a conference paper *On Inheritance in Conceptual Modeling for XML*[1] [63] in the 3rd International Conference on Ambient Systems, Networks and Technologies (ANT 2012).

9.1 Introduction

Motivation In general, we can distinguish two types of inheritance. The first one is a so called *structural* inheritance. This means that we only want to reuse a part of a schema for two different concepts. For example, we can have an `address` containing `street_name` and `country` attributes. We want to use these two attributes among others within a description of a `customer` and within a description of a `letter`. The concept of a customer is in no conceptual relationship to the concept of a letter, except they both have an address. This is similar to the concept of *interfaces* in modern programming languages like Java or C#. The second type of inheritance is a so called *conceptual* inheritance. With this type of inheritance, the child also inherits all the characteristics of the parent, but also they are in a conceptual relationship. For demonstration, let us use a classical example from biology. As a parent, we have the concept of a `mammal`. As its children, we can have a `cat` and a `human`. In this type of inheritance, being an instance of the child (a cat) also implies being an instance of the parent (a mammal). This is in contrast with the structural inheritance, where being a child (a country or a customer) does not imply being a parent (an address). Typically, in conceptual modeling languages like UML, we find support for the conceptual inheritance (generalizations) and in data modeling languages like XML Schema we find the structural inheritance (type extensions). Because our model's goal is to bridge the gap, we support both types of inheritance in an intuitive manner. The reason for bridging the gap is that we use our model as a part of a larger framework for data and schema evolution [71] incorporating more than just XML as a target platform. Because of this we chose UML class diagrams as a platform-independent modeling language. However, on the platform-specific level, we need do be able to model whatever the target language offers. In XML, the XML Schema language offers type extension (structural inheritance) in contrast to the conceptual inheritance present in UML class diagrams. Therefore, we need to support both types in out conceptual model.

Outline The rest of the chapter is organized as follows. In Section 9.2, we summarize our conceptual model for XML and we extend it with the constructs

[1] `http://dx.doi.org/10.1016/j.procs.2012.06.011`



```xml
<purchaseRQ version="1.1.4.">
  <cust>
    <name>Jakub Klímek</name>
    <email>klimek@ksi.mff.cuni.cz</email>
    <addr>
      <street>Malostranské náměstí 25</street>
      <city>Prague</city>
      <country>Czech Republic</country>
      <gps>50.088385,14.403629</gps>
    </addr>
  </cust>
  <items>
    <item tester="no">
      <code>IT1234</code>
      <title>Sample 4</title>
      <price>20€</price>
      <amount>4</amount>
    </item>
  </items>
</purchaseRQ>
```

Figure 9.1: Sample PIM and PSM schema and an XML document modeld by the PSM schem

for modeling of inheritance. Section 9.3 contains examples of translation from our conceptual model to the XML Schema language. Section 9.4 surveys related work. Finally, Section 9.5 contains a brief description of our evaluation and concludes.

9.2 Conceptual Model with Inheritance

In this section, we will simplify our conceptual model for XML and focus on its inheritance extension. We follow the Model-Driven Architecture (MDA) principle which is based on modeling data at several levels of abstraction. The most abstract level contains a conceptual schema of the problem domain. The language applied to express the conceptual schema is called *platform-independent model (PIM)*. The level below is the *platform-specific level* which specifies how the whole or a part of the PIM schema is represented in a particular platform. In our case, the platform is XML.

9.2.1 Platform-Independent Model

A PIM schema is based on UML class diagrams and models real-world concepts and relationships between them. It contains three types of components: classes, attributes and associations. A sample PIM schema is in Figure 9.1(a).

Definition 9.1 *A* PIM schema *is a triple* $\mathcal{S} = (\mathcal{S}_c, \mathcal{S}_a, \mathcal{S}_r)$ *of disjoint sets of* classes, attributes, *and* associations, *respectively.*

- Class $C \in \mathcal{S}_c$ *has a name assigned by function* name. *For inheritance purposes, function* isa *assigns a parent class to a child class and the relation must not form a cycle. Furthermore, functions abstract and final determine whether the class can have instances in data and whether this class can be inherited from, respectively.*

- Attribute $A \in \mathcal{S}_a$ *has a name, data type and cardinality assigned by functions* name, type, *and* card, *respectively. Moreover, A is associated with a class from $\mathcal{S}_c$ by function* class.

- Association $R \in \mathcal{S}_r$ *is a set* $R = \{E_1, E_2\}$, *where E_1 and E_2 are called* association ends *of R. R has a name assigned by function* name. *Both E_1 and E_2 have a cardinality assigned by function* card *and are associated with* *a class from $\mathcal{S}_c$ by function* participant. *We will call* participant(E_1) *and* participant(E_2) participants *of R.* name(R) *may be undefined, denoted by* name$(R) = \lambda$.

For a class $C \in \mathcal{S}_c$, we will use attributes (C) *to denote the set of all attributes of C, i.e.* attributes $(C) = \{A \in \mathcal{S}_a : \text{class}(A) = C\}$. *Similarly,* associations (C) *will denote the set of all associations with C as a participant, i.e.* associations (C) $= \{R \in \mathcal{S}_r : (\exists E \in R)(\text{participant}(E) = C)\}$.

9.2.2 Platform–Specific Model

The *platform-specific model (PSM)* enables to specify how a part of the reality is represented in a particular XML schema in a UML-style way. We introduce it formally in Definition 9.2. We view a PSM schema in two perspectives. From the *grammatical perspective*, it models XML elements and attributes. From the *conceptual perspective*, it delimits the represented part of the reality. Its advantage is clear – the designer works in a UML-style way which is more comfortable then editing the XML schema. A sample PSM schema is in Figure 9.1(b) and a corresponding valid XML document is in Figure 9.1(c).

Definition 9.2 *A* PSM schema *is a tuple $\mathcal{S}' = (\mathcal{S}'_c, \mathcal{S}'_a, \mathcal{S}'_r, \mathcal{S}'_m, \mathcal{C}'_{\mathcal{S}'})$ of disjoint sets of* classes, attributes, associations, *and* content models, *respectively, and one specific class $\mathcal{C}'_{\mathcal{S}'} \in \mathcal{S}'_c$ called* schema class.

- Class $C' \in \mathcal{S}'_c$ *has a name assigned by function* name.
- Attribute $A' \in \mathcal{S}'_a$ *has a name, data type, cardinality and XML form assigned by functions* name, type, card *and* xform, *respectively.* xform(A') $\in \{\text{e}, \text{a}\}$. *Moreover, it is associated with a class from $\mathcal{S}'_c$ by function* class *and has a position assigned by function* position *within the all attributes associated with class(A').*
- Association $R' \in \mathcal{S}'_r$ *is a pair $R' = (E'_1, E'_2)$, where E'_1 and E'_2 are called* association ends *of R'. Both E'_1 and E'_2 have a cardinality assigned by function* card *and each is associated with a class from $\mathcal{S}'_c$ or content model from $\mathcal{S}'_m$ assigned by function* participant, *respectively. We will call* participant(E'_1) *and* participant(E'_2) parent *and* child *and will denote them by* parent(R') *and* child(R'), *respectively. Moreover, R' has a name assigned by function* name *and has a position assigned by function* position *within the all associations with the same* parent(R'). name(R') *may be undefined, denoted by* name$(R') = \lambda$.
- Content model $M' \in \mathcal{S}'_m$ *has a content model type assigned by function* cmtype. cmtype$(M') \in \{$`sequence`, `choice`, `set`$\}$.

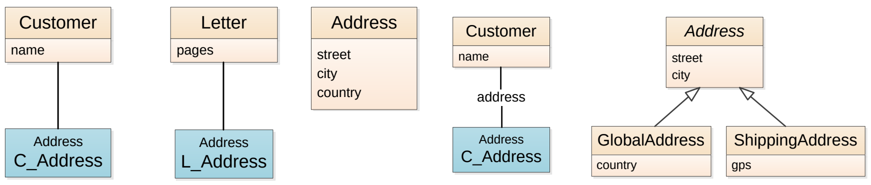
The graph $(\mathcal{S}'_c \cup \mathcal{S}'_m, \mathcal{S}'_r)$ must be a forest[2] *of rooted trees with one of its trees rooted in $\mathcal{C}'_{\mathcal{S}'}$. For $C' \in \mathcal{S}'_c$, attributes(C') will denote the sequence of all attributes of C' ordered by* position, *i.e.* attributes $(C') = (A'_i \in \mathcal{S}'_a : \text{class}(A'_i) =$

[2]Note that since $\mathcal{S}'$ is a forest, we could model R' directly as a pair of connected components. However, we use association ends to unify the formalism of PSM with the formalism of PIM.

$C' \wedge i = \text{position}(A'_i))$. *Similarly,* content (C') *will denote the sequence of all associations with* C' *as a parent ordered by* position*, i.e.* content $(C') = (R'_i \in \mathcal{S}'_r : \text{parent}(R'_i) = C' \wedge i = \text{position}(R'_i))$. *We will call* content (C') content of C'.

A sample PSM schema is depicted in Figure 9.1(b). As can be seen from the definition, PSM introduces similar constructs to PIM: classes, attributes and associations. The PSM-specific constructs have precisely defined semantics. Briefly, a class models a complex content. The complex content is specified by the attributes of the class and associations in its content (their ordering is given by functions *attributes* and *content*). An attribute models an XML element declaration with a simple content or XML attribute declaration depending on its XML form (function *xform*). An association models an XML element declaration with a complex content if it has a name. Otherwise, it models only that the complex content modeled by its child is nested in the complex content modeled by its parent.

Sometimes, classes in one or more PSM schemas may share the same attributes and/or part of their content. Instead of repeating them at several places, we can use inheritance. We need to be able to specify that a class can reuse an already modeled part of a PSM schema. We distinguish two types of inheritance, the structural and the conceptual.

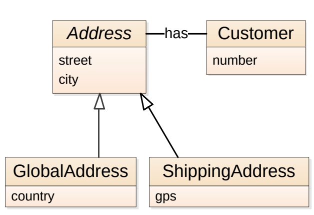


(a) Structural inheritance (b) Conceptual inheritance

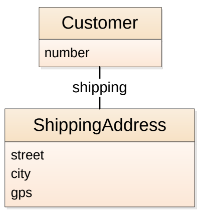
Figure 9.2: Two inheritance types in PSM schemas

Structural inheritance For structural inheritance, we introduce a *structural representative (repr)* function. This function specifies that a PSM class C' has a reference to another PSM class C'_r. It means that the complex content modeled by C' contains all the attributes and content of C' and all the attributes and content of C'_r. However, because this is *structural* inheritance, C' can not be used where C'_r is used. In our visualization we write the name of the referenced class on top of the name of the referencing class. An example can be seen in Figure 9.2(a). The example is from our motivation. In the PSM schema we model a `Customer` and a `Letter`. Both of them use attributes of `Address`.

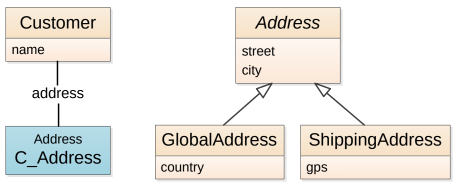
Conceptual inheritance Because of the nature of the *structural* inheritance which does not allow the usage of a child where its parent is used, we need another construct for expressing the *conceptual* inheritance in a PSM schema. For that, we introduce an *isa* function. This function specifies that a PSM class C' is an conceptual inheritance child of another PSM class C'_p. This also means that the complex content modeled by C' contains all the attributes and content of C' and all the attributes and content of its parent, C'_p, but in addition, it means that wherever the content modeled by C'_p is used, the content modeled by C' can also be used. In our visualization we use the usual inheritance arrow known from UML. An example can be seen in Figure 9.2(b). In this example, we model a



(a) PIM schema for interpretation with inheritance examples



(b) Implicit use of conceptual inheritance in PSM schema



(c) Explicit use of conceptual inheritance in PSM schema

Figure 9.3: PSM schemas for interpretation with inheritance examples

`Customer` who has an `Address`, but we do not mind which particular address it is. Because there is conceptual inheritance between `Address` and `BillingAddress` and between `Address` and `ShippingAddress`, the actual address used in the content of `Customer` (here we again use the structural inheritance) can be any one of them. Formally:

Definition 9.3 *Let* $C', D' \in \mathcal{S}'_c$*, let* $\text{repr}^*(\lambda) = \{\}$ *and* $\text{repr}^*(C') = \{\text{repr}(C')\} \cup \text{repr}^*(\text{repr}(C'))$ *where* $C' \neq \lambda$*. It must hold that* $C' \notin \text{repr}^*(C')$*. Let* $\text{isa}^*(\lambda) = \{\}$ *and* $\text{isa}^*(C') = \{\text{isa}(C')\} \cup \text{isa}^*(\text{isa}(C'))$ *where* $C' \neq \lambda$*. It must hold that* $C' \notin \text{isa}^*(C')$*. In addition,* repr *and* isa *must not form a cycle when combined (e.g.* isa $(C') = D' \Rightarrow \text{repr}(D') \neq C'$ *in the simplest case). Formally, let* $\text{isarepr}^*(\lambda) = \{\}$ *and* $\text{isarepr}^*(C') = \{\text{isa}(C')\} \cup \text{isa}^*(\text{isa}(C')) \cup \{\text{repr}(C')\} \cup \text{repr}^*(\text{repr}(C'))$ *where* $C' \neq \lambda$*. It must hold that* $C' \notin \text{isarepr}^*(C')$*.*

9.2.3 Interpretation of PSM schema against PIM schema

A PSM schema represents a part of a PIM schema. A class, attribute or association in the PSM schema may be mapped to a class, attribute or association in the PIM schema. In other words, there is a mapping which specifies the semantics of classes, attributes and associations of the PSM schema in terms of the PIM schema. The mapping must meet certain conditions to ensure consistency between PIM schemas and the specified semantics of the PSM schema. The interpretation of a PSM schema against a PIM schema is what we call the mapping. It is the core feature of our conceptual model. It interconnects constructs on the platform-specific level with those on the platform-independent level and allows for interesting use cases for the conceptual model like XML schema evolution and integration [58, 89, 61]. An arbitrary interpretation of a PSM component would, however, lead to inconsistencies between the semantics of the PIM schema and the semantics of the PSM schema given by the interpretation. An example of such inconsistency can be a class C' such that $I(C') = C$ and its attribute A' such that $I(A') = A$ in the PSM schema, while in the PIM schema the attribute A would belong to a class other than C. Before we introduce the rules that prevent those inconsistencies, let us define the notion of *interpreted context* of a PSM component which we need for the rules. The PSM schema can contain some uninterpreted classes, attributes and associations. This means that those have no meaning on the platform-independent level, but they are used in the XML format. Definition 9.4 says that if a PSM component is a class and has an interpretation, then it is its own *interpreted context*. The other components exist in a semantic *context* of the nearest ancestor class which has an interpretation (semantic equivalent on the PIM level) and that class is their *interpreted context*.

In the next definition, we will use $\overrightarrow{\mathcal{S}_r}$ to denote the set of all PIM associations with their direction specified (normally, they do not have direction).

Definition 9.4 *An* interpretation *of a PSM schema $\mathcal{S}'$ against a PIM schema $\mathcal{S}$ is a partial function $I : (\mathcal{S}'_c \cup \mathcal{S}'_a \cup \mathcal{S}'_r) \rightarrow (\mathcal{S}_c \cup \mathcal{S}_a \cup \overrightarrow{\mathcal{S}_r})$ which maps a class, attribute or association from $\mathcal{S}'$ to a class, attribute or directed image of an association from $\mathcal{S}$, respectively. Let X' be a component of a PSM schema $\mathcal{S}'$. We call $I(X')$* interpretation of X'. *Let I be an interpretation of $\mathcal{S}'$ against a PIM schema $\mathcal{S}$. The* interpreted context *of X' with respect to I is denoted $intcontext(X')$ and*

- *$intcontext(X') = X'$ when $X' \in \mathcal{S}'_c$ and $I(X') \neq \lambda$*
- *$intcontext(X') = C'$ when $X' \notin \mathcal{S}'_c$ or $I(X') = \lambda$, where C' is the closest ancestor class to X' s.t. $I(C') \neq \lambda$.*

Note that $intcontext(X')$ may be empty, i.e. $intcontext(X') = \lambda$. In that case we will say that X' does not have an interpreted context. This can happen when no class in a particular tree of the PSM schema forest has an interpretation.

Definition 9.5 *Let I be an interpretation a PSM schema $\mathcal{S}'$ against a PIM schema $\mathcal{S}$. Let $C_1 \preceq C_2$ be a shortcut for $C_1 = C_2$ or C_1 is an ancestor of C_2 regarding inheritance in a PIM schema. We say that I is* consistent *if the following rules are satisfied:*

$$(\forall A' \in \mathcal{S}'_a \text{ s.t. } I(A') \neq \lambda) \quad (intcontext(A') \neq \lambda \wedge class(I(A')) \preceq I(intcontext(A'))) \tag{9.1}$$

$$(\forall R' \in \mathcal{S}'_r \text{ s.t. } I(child(R')) \neq \lambda \wedge intcontext(R') \neq \lambda) \quad (I(R') = (E_1, E_2) \text{ s.t. } participant(E_1) \preceq I(intcontext(R')) \wedge participant(E_2) \preceq I(child(R'))) \tag{9.2}$$

$$(\forall R' \in \mathcal{S}'_r \text{ s.t. } I(child(R')) = \lambda \vee I(intcontext(R')) = \lambda) \quad (I(R') = \lambda) \tag{9.3}$$

$$(\forall C' \in \mathcal{S}'_c \text{ s.t. } repr(C') \neq \lambda \wedge I(C') \neq \lambda)(I(C') \preceq I(repr(C'))) \tag{9.4}$$

$$(\forall C' \in \mathcal{S}'_c \text{ s.t. } isa(C') \neq \lambda)(I(isa(C')) \preceq I(C')) \tag{9.5}$$

$$(\forall C' \in \mathcal{S}'_c \text{ s.t. } abstract(C') = \texttt{true})(abstract(I(C')) = \texttt{true}) \tag{9.6}$$

$$(\forall C' \in \mathcal{S}'_c \text{ s.t. } final(C') = \texttt{true})(final(I(C')) = \texttt{true}) \tag{9.7}$$

The first rule says that the interpretation (a PIM class) of an interpreted context (which is a PSM class) of each attribute A' that has an interpretation is the same as the attributes interpretations (a PIM attribute) class (a PIM class). The second rule is similar, but for associations. The third rule says that if an association does not have an interpreted context or its child is not interpreted, it is not interpreted either. Rule 9.4 states that two classes involved in a structural inheritance relationship must have the same interpretation (as they share the same content). Besides that, we need to take into account the possibility that the class from which we inherit content and attributes may have conceptual inheritance children that may replace it. Finally, Rule 9.5 states that if two classes are in an conceptual inheritance relationship in the PSM schema, their interpretations must be the same or the parent's interpretation must be an ancestor of the child's interpretation in the PIM schema. This is what we call an *explicit* inheritance, as we see the inheritance function in the PSM schema.

9.2.4 Conceptual model summary

In summary, the usefulness of our conceptual model for XML can be clearly seen when we, for example, ask questions like "In which of our hundred XML schemas used in our system is the concept of a `customer` represented?" and "What impact on my XML schemas would this particular change on the conceptual level have?". Even better, with our extensions for evolution of XML schemas [58, 89] we can make changes to the PIM schema (e.g. change the representation of a customer's `name` from one string to `firstname` and `lastname`) and those changes can be automatically propagated to all the affected PSM schemas. Thanks to automated translations from PSM schemas to, for example, XML Schema and back [96], we can easily manage a whole system of XML schemas from the conceptual level all thanks to the interpretations. These extensions, however, are not trivial and are not in the scope of this chapter, where we only deal with modeling of inheritance. In addition, the conceptual model can serve as a quality documentation of the XML schemas, because it is clear to which concepts individual schema parts relate. Also, it would be possible to generate a clickable HTML documentation of a system modeled in eXolutio. With the model, it is also much easier and faster to grasp a system of multiple XML schemas when, for example, negotiating interfaces between two information systems.

9.3 Translation of inheritance to XML Schema

In this section, we will very briefly sketch how the new inheritance constructs can be translated to XML Schema language using a few examples. This is, of course, very much dependent on the chosen style of writing an XML schema. In our future work, we will focus on enhancing our PSM schema with user-defined markers which will help us to automatically determine the desired XML schema style. We also show some limitations when using the XML Schema language. There are several scenarios. As a first example, we use the structural inheritance as in Figure 9.2(a), class `C_Address` to reference the `Address` class. In this case, `Address` would be translated to the *attributeGroup* construct. This is because it is a root of the PSM schema and it has no named association leading to it, so it does not model an element. The *attributeGroup* would then be referenced from the complexType modeled by `Customer`. In our second example, we use the conceptual inheritance as in Figure 9.3(c). In this case, the child complexTypes `GlobalAddress` and `ShippingAddress` will use the *extension* construct to extend the `Address` complexType. Now the element modeled by the named association `address` will have the `Address` type, which makes it possible to substitute it with `GlobalAddress` or `ShippingAddress` types. The problem with XML Schema language and inheritance is that for our situation to work, we must explicitly mention the used datatype directly in the data using the *XML Schema Instance* construct *xsi:type*, which is very inconvenient because we would like to keep the data unchanged. The solution could be for example to use another XML schema language like Relax NG. Note that the *abstract* and *final* constructs have their respective counterparts in the XML Schema language and their translation is straightforward.

9.4 Related work

There exist various XML schema languages, e.g. DTD, XSD or Relax NG for specification of XML schemas. However, these languages are not very user-friendly and each of them has a different level of expressiveness. In particular, support for inheritance is only present in XML Schema. Therefore, various approaches for designing XML schemas at more abstract, conceptual level were introduced. In comparison to our introduced approach, none of the approaches considers a formal binding between XML schemas and conceptual schemas like our interpretation. They only show how a conceptual schema is translated to an XML schema or vice versa but they do not consider the case when more XML schemas need to be designed which is the motivation for our conceptual model. For the same reason the approaches have limited capabilities when it comes to modeling conceptual and structural inheritance. In majority of cases the authors do not consider inheritance at all or they do it only superficially. The top-down approaches are based on designing first a conceptual schema and then its translation to one or more XML schemas. There are approaches based on the ER model (see our survey [85] or a more recent survey [137]) and then there are approaches based on the model of UML class diagrams (see surveys [18, 38]). The bottom-up approaches consider existing XML schemas and use them to derive a conceptual schema. Again there are approaches considering the ER model [112] or UML class diagrams [136]. There is also a recent survey of them in [137].

The common characteristic is that the approaches force a designer to model an XML schema directly in (or import it to) the ER/UML diagram. This can be a disadvantage because the designer must concentrate on XML specific implementation details at the conceptual level. Another problem is that these approaches consider design of only a single separate XML schema but not a set of XML schemas that describe XML representations of the same data in different types of XML documents.

The approaches briefly considering inheritance are [119, 37] but only in [3] the authors go into more detail and also show some limitations of the XML Schema language. However, they still model XML schema structure and semantics in one schema, which is not good enough as we showed in [96].

9.5 Evaluation and Conclusion

We implemented the inheritance extension in our tool eXolutio [55] and evaluated it as we modeled a medium-sized family of XML schemas of the *Data Standard for eHealth in the Czech Republic (DASTA)*[3]. We only summarize our results here. The PIM schema contained more that 70 classes, 100 associations including conceptual inheritance relations and hundreds of attributes. Mapped to the PIM schema were 12 XML formats (PSM schemas). Our approach improved the speed with wich programmers were able to orientate themselves in the schemas and it also helped in revealing inconsistencies in the XML schemas.

In this chapter, we summarized our previous work on a conceptual model for XML and we introduced an inheritance extension to it. We showed how inheritance can be modeled on a platform-independent and platform-specific levels and we show how the constructs can be translated to the XML Schema language. We surveyed work related to conceptual modeling of XML in general.

[3]`http://ciselniky.dasta.mzcr.cz/` (in Czech only)

10. Formal Evolution of XML Schemas with Inheritance

In Chapter 9, we formally extended our conceptual model for XML with inheritance modeling, which we omitted before for simplicity. In this chapter, we extend our approach with evolving schemas with inheritance on both PIM and PSM levels and we bridge the gap between the conceptual modeling world (*conceptual* inheritance) and the schema design world (*structural* inheritance). This is the second part of our core and most important contribution to XML schema evolution.

The content of this chapter was published as a conference paper *Formal Evolution of XML Schemas with Inheritance*[1] [62] in 2012 IEEE International Conference on Web Services (ICWS 2012).

10.1 Introduction

Motivation Let us shortly discuss the two main inheritance types. The structural inheritance means that we want to reuse a part of a schema for different concepts. For example, we can have an *address* containing *street_name* and *country* attributes. We want to use these among others within a description of a *customer* and within a description of a *letter*. A customer is in no conceptual relationship to a letter, except they both have an address. With conceptual inheritance, the child inherits all characteristics of the parent too, but there is also a conceptual relationship. An example from biology: As a parent, we have a *mammal* and as its children we can have a *cat* and a *human*. In this type of inheritance, being an instance of the child also implies being an instance of the parent. This is in contrast with the structural inheritance, where being a child does not imply being a parent. In conceptual modeling languages like UML we find support for the conceptual inheritance and in data modeling languages like XML Schema we find the structural inheritance. Because our model's goal is to bridge the gap, we support both types of inheritance in an intuitive manner. The reason for bridging the gap is that we use our model as a part of a larger framework [71] incorporating more than just XML as a target platform and we chose UML class diagrams as a universal platform-independent modeling language. On the platform-specific level, we need do be able to model whatever the target language offers and in XML and the XML Schema language it is type extension (structural inheritance) in contrast to the conceptual inheritance present in UML class diagrams. For the model to be usable, it needs well defined operations that assure the model consistency during its evolution in time.

Outline In Section 10.2 we complement the model with a set o operations for inheritance management and in Section 10.3 we describe their propagation. Section 10.4 describes our implementation of the approach. Section 10.5 contains

[1] `http://dx.doi.org/10.1109/ICWS.2012.11`

evaluation. Section 10.6 contains a brief survey of related work. Section 10.7 concludes.

10.2 Atomic Operations

In this section, we extend *atomic operations* for editing PIM and PSM schemas which we introduced in our previous work [89] with inheritance evolution, which is the main contribution of this paper. The atomic operations serve as a formal basis for describing user-friendly operations composed of them. Formally, we suppose a PIM schema $\mathcal{S}$ and a set of PSM schemas $\mathcal{PSM} = \{\mathcal{S}'_1, \ldots, \mathcal{S}'_n\}$, where each $\mathcal{S}'_i$ has an interpretation I_i against $\mathcal{S}$. We also consider one specific PSM schema $\mathcal{S}'$ from this set with an interpretation I against $\mathcal{S}$. For each atomic operation we specify input parameters together with a precondition and postcondition. If a precondition is not satisfied, the operation cannot be performed. The postcondition describes the effect of the operation. When an operation is executed on $\mathcal{S}$ or $\mathcal{S}'$, we say that the schema *evolved to a new version*. This is denoted $\mathcal{S}^+$ or $\mathcal{S}'^+$, respectively. The new version of the interpretation will be denoted I^+. In [89] we classified atomic operations into 4 categories: *creation* of classes, attributes and associations (denoted by the Greek letter α), their *update* (denoted by the Greek letter υ) and *removal* (denoted by the Greek letter δ) and we introduced a special *synchronization* operation (denoted by the Greek letter σ). For their detailed description we refer the reader to our previous work [89]. In this paper we extend them with operations for inheritance management.

10.2.1 Atomic Operations for PIM Schema Inheritance Evolution

In [89] we introduced basic operations for creation, change, movement and deletion of classes, attributes and associations. Now we extend these operations with new ones required for proper modeling and evolution of inheritance. The formal commands for the operations and their preconditions and postconditions formalized according to our model are in Table 10.1. Note that so far, we do not describe propagation from the PIM level to the PSM level.

There is basically one atomic operation which changes the *isa* function on PIM classes $\upsilon_c^{isa}(C_s, C_g)$. However, there are four distinct cases of its use, so we describe each case individually, as it makes the description easier to formalize and understand. First, we have the simple addition and removal of generalizations. Formally, this means setting the *isa* function to a class when it was λ (addition) and setting it to λ when it was set to some class (removal). Next is resetting generalizations – setting it to a different class than it was originally set to. The preconditions state that these operations can be done only if there is a generalization between the source and the target general class. This means moving the specific class higher or lower in the inheritance hierarchy.

In addition, there are operations which change the *abstract* and *final* functions and that is all we need for the management of the inheritance on the PIM level. These operations are trivial so we do not provide examples for them.

Then there are special operations for movement of attributes and association ends between classes which are in an inheritance relation. These are different from

Notation	Description	Precondition	Postcondition
$v_c^{isa}(C_s, C_g)$	Sets conceptual inheritance relation. C_g is the general class and C_s is the specialized class	$C_g, C_s \in \mathcal{S}_c \wedge C_g \neq C_s \wedge \mathit{final}(C_g) = \texttt{false} \wedge C_s \notin \mathit{isa}^*(C_g) \wedge \mathit{isa}(C_s) = \lambda \wedge C_g \neq \lambda$	$\mathit{isa}^+(C_s) = C_g$
$v_c^{isa}(C_s, \lambda)$	Unsets conceptual inheritance relation.	$C_s \in \mathcal{S}_c \wedge \mathit{isa}(C_s) \neq \lambda$	$\mathit{isa}^+(C_s) = \lambda$
$v_c^{isa}(C_s, C_g)$	Resets conceptual inheritance relation to a more general class. C_g is the more general class, C_{g0} is the original general class and C_s is the specialized class	$C_g, C_{g0}, C_s \in \mathcal{S}_c \wedge C_g \neq C_s \wedge \mathit{final}(C_g) = \texttt{false} \wedge C_s \notin \mathit{isa}^*(C_g) \wedge \mathit{isa}(C_s) = C_{g0} \neq \lambda \wedge C_g \neq \lambda \wedge \mathit{isa}(C_{g0}) = C_g$	$\mathit{isa}^+(C_s) = C_g$
$v_c^{isa}(C_s, C_g)$	Resets conceptual inheritance relation to a less general class. C_g is the less general class, C_{g0} is the original general class and C_s is the specialized class	$C_g, C_{g0}, C_s \in \mathcal{S}_c \wedge C_g \neq C_s \wedge \mathit{final}(C_g) = \texttt{false} \wedge C_s \notin \mathit{isa}^*(C_g) \wedge \mathit{isa}(C_s) = C_{g0} \neq \lambda \wedge C_g \neq \lambda \wedge \mathit{isa}(C_g) = C_{g0}$	$\mathit{isa}^+(C_s) = C_g$
$v_c^{abstract}(C, b)$	Sets the *abstract* property of a class C to b, which is either `true` or `false`		$\mathit{abstract}^+(C) = b$
$v_c^{final}(C, b)$	Sets the *final* property of a class C to b, which is either `true` or `false`	$(b = \texttt{true} \wedge \nexists D \in \mathcal{S}_c(\mathit{isa}(D) = C)) \vee b = \texttt{false}$	$\mathit{final}^+(C) = b$
$v_a^{gen}(A, C_g)$	Move attribute to a general class C_g	$A \in \mathcal{S}_a \wedge C_0, C_g \in \mathcal{S}_c \wedge \mathit{class}(A) = C_0 \wedge \mathit{isa}(C_0) = C_g$	$\mathit{class}^+(A) = C_g$
$v_a^{spec}(A, C_s)$	Move attribute to a specific class C_s	$A \in \mathcal{S}_a \wedge C_0, C_s \in \mathcal{S}_c \wedge \mathit{class}(A) = C_0 \wedge \mathit{isa}(C_s) = C_0$	$\mathit{class}^+(A) = C_s$
$v_r^{gen}(E, C_g)$	Reconnect association end to a general class C_g	$C_0, C_g \in \mathcal{S}_c \wedge \mathit{participant}(E) = C_0 \wedge \mathit{isa}(C_0) = C_g$	$\mathit{participant}^+(E) = C_g$
$v_r^{spec}(E, C_s)$	Reconnect association end to a specific class C_s	$C_0, C_s \in \mathcal{S}_c \wedge \mathit{participant}(E) = C_0 \wedge \mathit{isa}(C_s) = C_0$	$\mathit{participant}^+(E) = C_s$

Table 10.1: Atomic operations for PIM inheritance management

the operations for simple attribute or association end movement, their semantics and therefore their propagation is different.

10.2.2 Atomic Operations for PSM Schema Inheritance Evolution

In this section we present operations for conceptual inheritance in PSM schemas. The operations are similar to those on the PIM level but there are differences in preconditions as on the PSM level we need do coordinate conceptual and structural inheritance. In Table 10.2 we summarize notation, description, preconditions and postconditions of operations working with the *isa*, *abstract* and *final* functions and operations for moving attributes and associations through the inheritance hierarchy.

Notation	Description	Precondition	Postcond.
$\upsilon_c'^{isa}(C'_s, C'_g)$	Sets conceptual inheritance relation. C'_g is the general class and C'_s is the specialized class	$C'_g, C'_s \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge \mathcal{C}'_{\}} \neq \mathcal{C}'_{\int} \wedge \mathcal{C}'_{\int} \notin isarepr^*(\mathcal{C}'_{\}}) \wedge isa(\mathcal{C}'_{\int}) = \lambda \wedge \mathcal{C}'_{\}} \neq \lambda$	$isa^+(C'_s) = C'_g$
$\upsilon_c'^{isa}(C'_s, \lambda)$	Unsets conceptual inheritance relation.	$C'_s \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge isa(\mathcal{C}'_{\int}) \neq \lambda$	$isa^+(C'_s) = \lambda$
$\upsilon_c'^{isa}(C'_s, C'_g)$	Resets conceptual inheritance relation to a more general class. C'_g is the more general class, C'_{g0} is the original general class and C'_s is the specialized class	$C'_g, C'_{g0}, C'_s \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge \mathcal{C}'_{\}} \neq \mathcal{C}'_{\int} \wedge \mathcal{C}'_{\int} \notin isarepr^*(\mathcal{C}'_{\}}) \wedge isa(\mathcal{C}'_{\int}) = \mathcal{C}'_{\}'} \neq \lambda \wedge \mathcal{C}'_{\}} \neq \lambda \wedge isa(\mathcal{C}'_{\}'}) = \mathcal{C}'_{\}}$	$isa^+(C'_s) = C'_g$
$\upsilon_c'^{isa}(C'_s, C'_g)$	Resets conceptual inheritance relation to a less general class. C'_g is the less general class, C'_{g0} is the original general class and C'_s is the specialized class	$C'_g, C'_{g0}, C'_s \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge \mathcal{C}'_{\}} \neq \mathcal{C}'_{\int} \wedge \mathcal{C}'_{\int} \notin isarepr^*(\mathcal{C}'_{\}}) \wedge isa(\mathcal{C}'_{\int}) = \mathcal{C}'_{\}'} \neq \lambda \wedge \mathcal{C}'_{\}} \neq \lambda \wedge isa(\mathcal{C}'_{\}}) = \mathcal{C}'_{\}'}$	$isa^+(C'_s) = C'_g$
$\upsilon_c'^{abstract}(C', b)$	Sets the *abstract* property of a class C' to b, which is either `true` or `false`		$abstract^+(C') = b$
$\upsilon_c'^{final}(C', b)$	Sets the *final* property of a class C' to b, which is either `true` or `false`	$(b = \texttt{true} \wedge \nexists D' \in \mathcal{S}'_c(isa(D') = C')) \vee b = \texttt{false}$	$final^+(C') = b$
$\upsilon_a'^{gen}(A', C'_g)$	Move attribute to a general class C'_g	$A' \in \mathcal{S}'_a \wedge C'_0, C'_g \in \mathcal{S}'_c \wedge class(A') = C'_0 \wedge isa(C'_0) = C'_g$	$class^+(A') = C'_g$
$\upsilon_a'^{spec}(A', C'_s)$	Move attribute to a specific class C'_s	$A' \in \mathcal{S}'_a \wedge C'_0, C'_s \in \mathcal{S}'_c \wedge class(A') = C'_0 \wedge isa(C'_s) = C'_0$	$class^+(A') = C'_s$
$\upsilon_r'^{gen}(R', C'_g)$	Reconnect parent association end of association R' to a general class C'_g	$C'_0, C'_g \in \mathcal{S}'_c \wedge parent(R') = C'_0 \wedge isa(C'_0) = C'_g$	$parent^+(R') = C'_g$
$\upsilon_r'^{spec}(R', C'_s)$	Reconnect parent association end of association R' to a specific class C'_s	$C'_0, C'_s \in \mathcal{S}'_c \wedge parent(R') = C'_0 \wedge isa(C'_s) = C'_0$	$parent^+(R') = C'_s$

Table 10.2: Atomic operations for PSM inheritance management

We also need to update some operations from [89] to take into account new requirements imposed by our conceptual inheritance extension to the interpretation definition. The redefined operations are in Table 10.3. For example, we redefine the operation for setting a class as a structural representative so it respects the rules of Definition 9.3. Next, we need to assure that when we delete a PSM class, it is not part of any conceptual inheritance relation. The same goes for setting a PSM classes interpretation. When we do that, we do not want that class to be a part of a conceptual inheritance relation. Finally, we need to allow the *implicit* inheritance when setting interpretation of attributes and associations. The most complicated redefinition is $\upsilon_c'^{int}(C', C)$ where we make sure that when we update and interpretation of a class, there are no attributes nor associations whose interpretation relies on the one of C' and that C' does not participate in any inheritance relations. In the precondition, *anc* is an ancestor in the PSM tree.

Notation	Description	Precondition	Postcond.
${v'}_c^{repr}(C', C'_r)$	Set class C' as structural representative of C'_r	$C' \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge (\mathcal{C}'_\nabla = \lambda \vee (C'_r \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge \mathcal{I}(\mathcal{C}') = \mathcal{I}(\mathcal{C}'_\nabla) \wedge \mathcal{C}' \notin \mathit{isarepr}^*(\mathcal{C}'_\nabla)))$	$\mathit{repr}^+(C') = C'_r$
$\delta'_c(C')$	Remove class C'	$C' \in \mathcal{S}'_c \wedge \mathit{attributes}(C') = \mathit{content}(C') = \emptyset \wedge (\nexists C'_0 \in \mathcal{S}'_c)(\mathit{repr}(C'_0) = C') \wedge (\nexists C'_1 \in \mathcal{S}'_c)(\mathit{isa}(C'_1) = C')$	$C' \notin \mathcal{S}'^+$
${v'}_c^{int}(C', C)$	Update interpretation of class C' to class C	$C' \in \mathcal{S}'_c \setminus \{\mathcal{C}'_{\mathcal{S}'}\} \wedge (C = \lambda \vee C \in \mathcal{S}_c)$ $\wedge (\forall A' \in \mathcal{S}'_a \text{ s.t. } \mathit{intcontext}(A') = \mathit{intcontext}(C') \wedge C' \in \mathit{anc}(A'))$ $(I(A') = \lambda) \wedge (\forall R' \in \mathcal{S}'_r \text{ s.t. } (\mathit{intcontext}(R') = \mathit{intcontext}(C') \wedge C' \in \mathit{anc}(R')) \vee \mathit{child}(R') = C')$ $(I(R') = \lambda) \wedge (\nexists C'_0 \in \mathcal{S}'_c)(\mathit{repr}(C'_0) = C') \wedge \mathit{repr}(C') = \lambda \wedge (\nexists C'_1 \in \mathcal{S}'_c)(\mathit{isa}(C'_1) = C') \wedge \mathit{isa}(C') = \lambda$	$I^+(C') = C$
${v'}_a^{int}(A', A)$	Update interpretation of attribute A' to attribute A	$A' \in \mathcal{S}'_a \wedge (A = \lambda \vee (A \in \mathcal{S}_a \wedge \mathit{class}(A) \preceq I(\mathit{intcontext}(A'))))$	$I^+(A') = A$
${v'}_r^{int}(R', O)$	Update interpretation of association R' to ordered image O of association R	$R' \in \mathcal{S}'_r \wedge \mathit{child}(R') \in \mathcal{S}'_c \wedge (O = \lambda \vee (O = (E_1, E_2) \wedge \mathit{participant}(E_1) \preceq I(\mathit{intcontext}(R')) \wedge \mathit{participant}(E_2) \preceq I(\mathit{child}(R'))))$	$I^+(R') = O$

Table 10.3: Basic atomic operations with inheritance update

10.3 Propagation of Atomic Operations

An interpretation of a PSM schema $\mathcal{S}'$ against a PIM schema $\mathcal{S}$ must be consistent. When $\mathcal{S}$ or $\mathcal{S}'$ is modified by an atomic operation, one or more conditions necessary for consistency may be violated and, consequently, the interpretation or the other schema must be adapted accordingly. We call the process which ensures the adaptation *propagation of the atomic operation.* The propagation of the basic atomic operations was described in detail in our previous work [89] and therefore in this paper we focus on the new operations for inheritance evolution. We only allow changing the inheritance hierarchies on the PIM level as we view the conceptual inheritance as belonging to the PIM level. We therefore restrict the operations on the PSM level to the boundaries set by the PIM schema.

Here we describe how the introduced atomic operations executed on the PIM schema $\mathcal{S}$ are propagated to each PSM schema $\mathcal{S}' \in \mathcal{PSM}$ and its interpretation I against $\mathcal{S}$. Creation of a generalization $(\mathit{isa}(C_s) = \lambda \wedge v_c^{isa}(C_s, C_g \neq \lambda))$ is not propagated at all. When the generalization is set, it is new and therefore it does not have an equivalent on the PSM level yet. Then there is setting of the *abstract* function saying that a class cannot have instances in data $(v_c^{abstract}(C, b))$. This is a constraint that we can not check on the modeling level and therefore we propagate it straight to the PSM schemas. We set the *abstract* function to the same value for each PSM class C' that has C as an interpretation. Formally, $(\forall C' : I(C') = C)(\mathit{abstract}(C') = b)$. From the PSM schemas this constraint is propagated to the actual PSM schema languages and it is up to their validators to check this constraint. Setting the *final* function is propagated in the same way as the setting of the *abstract* function. The propagation of resetting a generalization (*isa* function) to a more general or a more specific class is almost 1:1, which means that we perform similar operations on the PSM level to maintain consistency. There is one exception when moving generalization to a more general class.

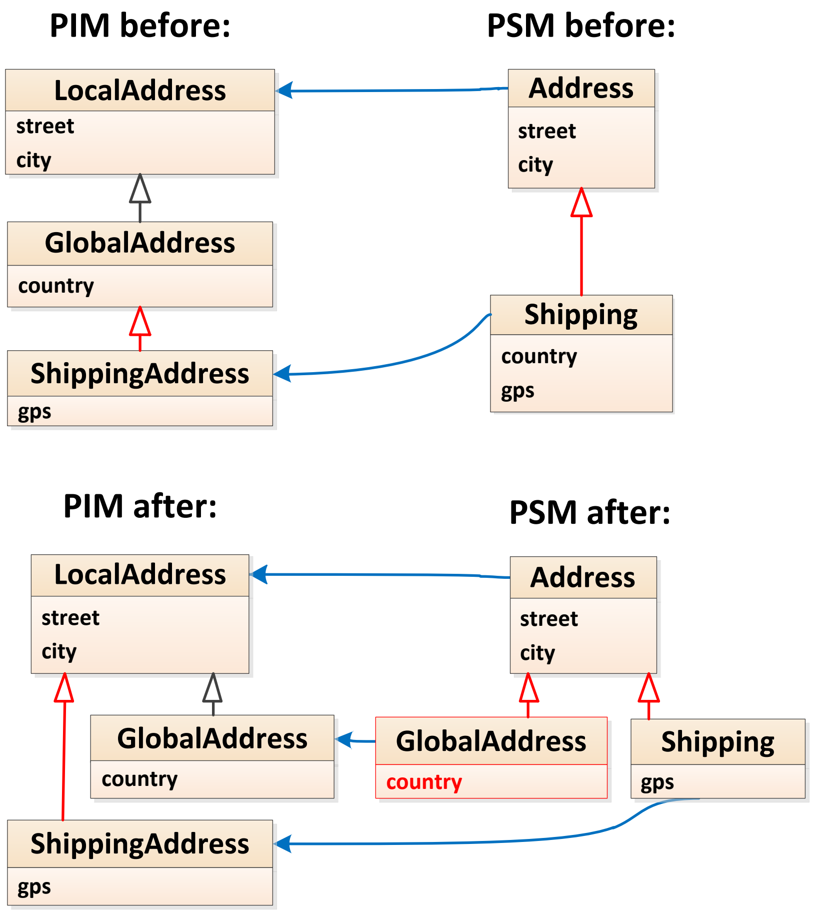


Figure 10.1: Propagation of moving a generalization to a more general class.

The example is in Figure 10.1. The blue lines represent the interpretation of PSM classes against PIM classes. In the example of resetting the *isa* function to a more general class, the new general class for `ShippingAddress` is `LocalAddress`. On the PSM level, we already have the `Address` class a general class to `Shipping`, so it would seem that no propagation is necessary. However, we have an attribute `Shipping'.country'` and $I(\texttt{Shipping}'.\texttt{country}') = \texttt{GlobalAddress.country}$. This is the use of *implicit* inheritance (Definition 9.5, rule 9.2) and that rule would be broken. Therefore, we need to create `GlobalAddress'` on the PSM level and keep the `country` attribute there. The opposite direction - moving a generalization to a more specific class is really a 1:1 propagation - we simply do the PSM counterpart operation. Another easy operation is removal of a generalization, which is in fact setting the *isa* function to λ ($v_c^{isa}(\texttt{ShippingAddress}, \lambda)$). When the PIM generalization is removed, we simply remove its PSM counterpart. Formally, $(\forall C' \in \mathcal{S}'_c : I(C') = C_s)(isa(C') = \lambda)$. Next are the operations for movement of attributes and associations through the inheritance hierarchy. Because the cases for attributes and associations are similar, we show here only the cases for the attributes. We start with moving an attribute to a more general class. In the first case the more general class is also present in the PSM schema and the movement of the affected attribute follows the movement in the PIM schema. In the second case, the schema remains unchanged, because the more general class is not present in the PSM schema and the movement does not violate rule 9.2 of Definition 9.5. Finally, there is moving an attribute to a more specific

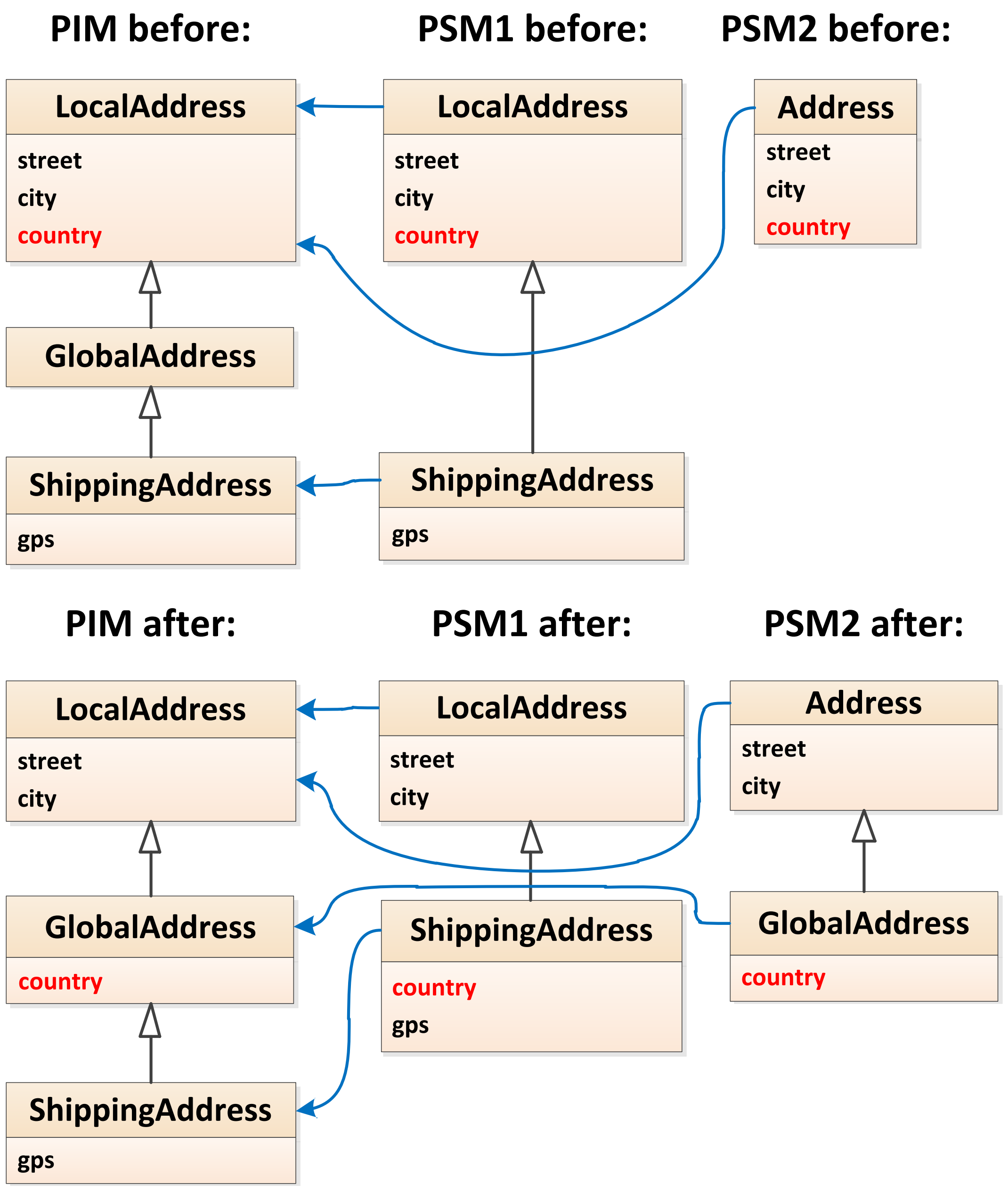


Figure 10.2: Propagation of moving an attribute to a specific class.

class. This is demonstrated in Figure 10.2. Again, we have two cases. In PSM1, there is a more specific class `ShippingAddress'`, whose interpretation inherits the target, more specific class `GlobalAddress` through interpretation. Formally, $\texttt{GlobalAddress} \preceq \texttt{ShippingAddress} = I(\texttt{ShippingAddress}')$. Therefore we move the corresponding attribute there. It there was no such class, we would have to create one, as can be seen in PSM2.

10.4 Implementation

We have implemented the proposed inheritance extension in a tool called *eXolutio* (see Chapter 12). It is a proof-of-concept desktop application for conceptual XML data modeling (screenshot in Figure 10.3). It implements the PIM and PSM modeling languages described in [96] and operations for evolution of the PIM and PSM schemas described in [89]. It provides a designer with a set of operations which are composed of the atomic operations described earlier and it implements their propagation. At the highest level, eXolutio is based on the Model View Controller (MVC) design pattern. For the purpose of this chapter, the atomic operations are implemented in the exact same way they are described here. We use the implementation to experimentally demonstrate that the proposed set of atomic operations is *complete*, i.e. that the atomic operations are sufficient for real-world situations. We do not prove completeness formally in this chapter.

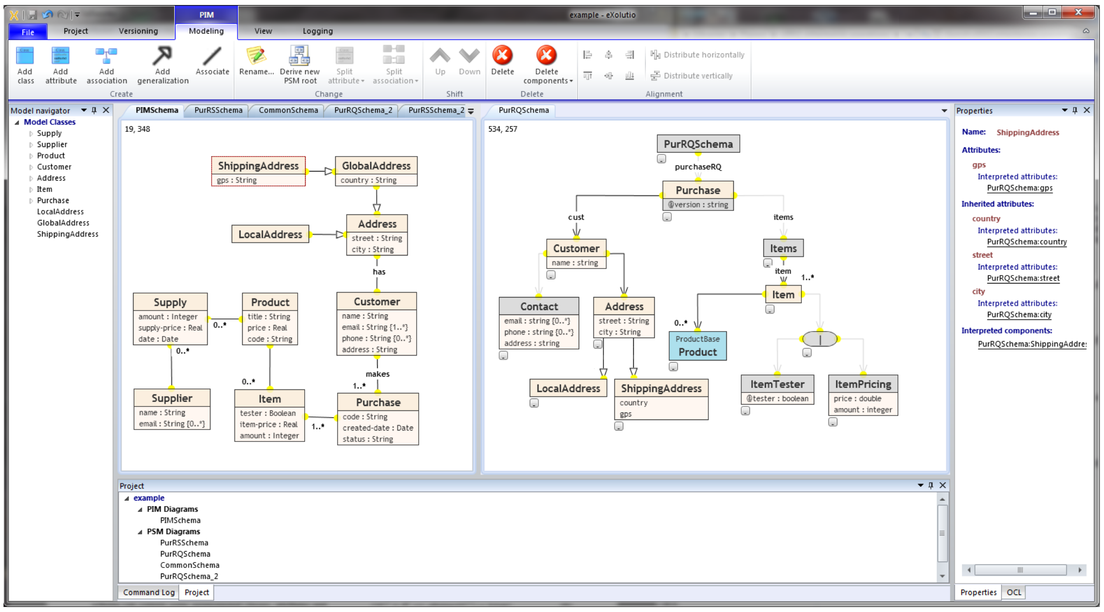


Figure 10.3: *eXolutio* screenshot

10.5 Evaluation

We have evaluated the inheritance extension to our conceptual model as we modeled a medium-sized family of XML schemas of the *Data Standard for eHealth in the Czech Republic (DASTA)*[2]. Due to lack of space we only summarize our results. The PIM schema contained more that 70 classes, 100 associations including conceptual inheritance relations and hundreds of attributes. Mapped to the PIM schema were 12 XML formats (PSM schemas). Since 2006, the format has 24 versions since it is evolved approximately four times a year. Our approach improved the speed with wich programmers were able to orientate themselves in the schemas and it also helped in revealing inconsistencies in the XML schemas. As to the evolution operations, we picked one of the evolution steps and performed

[2]`http://ciselniky.dasta.mzcr.cz/` (in Czech only)

the changes in *eXolutio* using our operations formalism and we confirmed that up to 60% of operations that would have to be done manually by a domain expert can be done automatically using our approach.

10.6 Related Work

The XML schema languages for specification of XML schemas are not very user-friendly. Therefore, approaches for designing XML schemas at a conceptual level were introduced. In comparison to our approach, none of the approaches considers a formal binding between XML schemas and conceptual schemas like our interpretation. They only show how a conceptual schema (ER or UML) is translated to an XML schema or vice versa but they do not consider the case when more XML schemas need to be designed which is the motivation for our conceptual model and schema evolution. Therefore, the other approaches are limited when it comes to modeling multiple schemas and in particular the conceptual and structural inheritance hierarchies. For work related to conceptual modeling of XML in general see [96, 97, 137, 18].

Evolution management The current approaches towards evolution management can be classified according to distinct aspects [74, 34]. The changes and transformations can be expressed [104, 21] as well as divided [28] variously too. To our knowledge there exists no general framework comparable to our conceptual model; particular cases and views of the problem have previously only been solved separately, superficially and mostly imprecisely without any theoretical or formal basis. For a full survey of work related to schema evolution management refer to [89]. The need for simple, well-defined operations for change management has been identified for example in [127].

Evolution of inheritance The problem of evolution in XML schemas with inheritance has been also identified in [24] as their future work. In [39], inheritance in UML to XML Schema translation is mentioned, however, the authors do not say how they translate UML generalization change to a change in XML Schema. To our best knowledge, there is no other approach focusing on inheritance evolution management in XML. In [120], the authors propose a metric for improving modifiability of class inheritance hierarchies.

10.7 Conclusions

In this chapter we described our approach to evolution of XML schemas described by the conceptual model for XML that can be applied to the common case of multiple web service interfaces with common data domain. We extended our formal model of operations with the ones regarding inheritance management and their propagation. We did this formally and on examples. We briefly mentioned implementation, evaluation and related work.

11. When Theory Meets Practice: A Case Report on Conceptual Modeling for XML

Modern information systems usually exploit numerous XML formats for communication with other systems. There are, however, many potential problems hidden. This includes the degree of readability, integrability and adaptability of the XML formats. In the first part of this chapter we demonstrate the problems on a real-world application – the National Register of Public Procurement in the Czech Republic. In the second part we show how we can improve readability, integrability and adaptability of the XML formats of this system with a conceptual model for XML we have developed in our previous works. Finally, we generalize our experience gained into a methodology which can be applied in any other problem domain.

This chapter shows an aspect of conceptual modeling for XML which can be used to improve various qualities of a family of XML schemas. The contents of this chapter was published as a conference paper *When Theory Meets Practice: A Case Report on Conceptual Modeling for XML*[1] [88] in 2011 Sixth International Conference on Digital Information Management (ICDIM 2011).

11.1 Introduction

In this chapter we pursue a concrete portal of Czech government – the *National Register for Public Procurement* (NRPP)[2]. We show that its XML formats are not very readable, integrable and adaptable and show how we used our previously developed conceptual model for XML and, especially, its implementation to improve these drawbacks. We show that a conceptual schema which is created in addition to XML schemas specifying the XML formats provided by NRPP may be used to significantly increase readability, integrability and adaptability of the XML formats. This work is a case study of a selected real-world governmental portal where we apply our previous theoretical results in the area of conceptual modeling, integration and evolution of XML schemas. In the end of the chapter we generalize the case study into a methodology which can be applied in other cases with similar problems.

The chapter is organized as follows: In Section 11.2, we introduce NRPP and discuss its drawbacks regarding readability, integrability and adaptability of the provided XML formats. In Section 11.3, we briefly introduce sample PIM and PSM schemas. In Section 11.4, we introduce our solution of the drawbacks based on the proposed conceptual model for XML. In Section 11.5 we generalize our experience with NRPP into a methodology which shows how our approach may be applied in any other problem domain. In Section 11.6 we discuss related work and evaluate our approach in comparison with the related work. Finally, we conclude in Section 11.7.

[1] `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6093322`

[2] `http://www.isvz.cz` (in Czech only)

11.2 Public Contracts in the Czech Republic

NRPP is intended for publishing data about public contracts by various public authorities in the Czech Republic. Publishing a contract is obligatory when the contracted price exceeds a level given by the current legislation. Otherwise, it is only optional. An authority may send contract information formatted as an XML document to NRPP. NRPP provides various XML formats intended for different situations, e.g. contract notification, supplier selection or contract finalization. Fig. 11.1 shows a sample XML document with a concrete contract notification in an XML format currently accepted by NRPP.

```xml
<contract_notification>
 <cont_oficial_title>Charles Univ.</cont_oficial_title>
 <cont_postal_address>Ovocný trh 3</cont_postal_address>
 <cont_city>Praha 1</cont_city>
 <cont_zip>11636</cont_zip>
 <cont_country>CZ</cont_country>
 <title>Úklid vybraných objektů ...</title>
 …
 <price_total>13000000</price_total>
 <currency>CZK</currency>
 <VAT>20</VAT>
 <docs_oficial_title>OTIDEA a.s.</docs_oficial_title>
 <docs_postal_address>Na Příkopě</docs_postal_address>
 <docs_city>Praha 1</docs_city>
 <docs_zip>110 00</docs_zip>
 <docs_country>CZ</docs_country>
</contract_notification>
```

Figure 11.1: Sample XML document which demonstrates low readability, integrability and adaptability of NRPP XML formats.

We made a detailed study of the XML formats used by NRPP from three different points of view: *readability*, *integrability* and *adaptability* of the XML formats. We describe the viewpoints in detail in the rest of this section. In the end, we summarize their properties from these three viewpoints (see Tab. 11.1). For our purposes, by the term *XML format* we mean one of the XML formats supported by NRPP and by *XML document* we mean an XML document formatted according to one of the XML formats, if not specified otherwise.

Readability of XML Formats To be able to process XML documents, developers need to understand the syntax and also the semantics of the XML formats. They need to know what part of the reality each particular XML element or XML attribute represents and, vice versa, how a selected part of the reality is represented in the XML formats. For example, they need to know that XML element `title` in our sample XML format represents contract title. They also need to know that `docs_postal_address` represents a postal address, where an interested supplier might get a documentation for the contract while that `cont_postal_address` represents the main contact address for the contract. And, it is also important to know that `price_total` represents a total price expected by the contractor and not a final contract price or another price (there are 4 kinds of price considered).

By the term *XML formats readability* we denote the described ability of developers to understand the XML formats. First, readability may be increased by exploiting the hierarchical nature of XML which allows for using separate XML elements to nest XML content representing different objects. However, this is not the case of the studied XML formats of NRPP. For example, instead of nesting different addresses in separate XML elements, the XML document depicted in Fig. 11.1 uses prefixes "`cont_`" and "`docs_`" which is not very transparent. Second, XML schemas of the XML formats (expressed in, e.g., DTD [126] or XSD [128] languages) should be provided. However, XML schemas only allow for describing the syntax. The semantics is not expressed explicitly and must be intuitively deduced by developers. A simple solution is to provide the developers with a textual documentation of the XML formats. A more advanced solution is to also provide them with a conceptual schema of the problem domain which describes the semantics widely and more precisely.

Integrability of XML Formats Another problem arises when the data needs to be converted from the XML formats to another data representation (e.g. a local database inside an authority's information system or other XML formats) and vice versa. Ideally, both representations are the same and, therefore, no export/import scripts are necessary. However, this is a rare case and transformation scripts need to be developed by developers. These scripts might be, e.g., SQL/XML [48] scripts for integration with a relational database or XSLT [52] scripts for integration with other XML formats.

By the term *XML formats integrability* we denote the measure of how easy it is to develop such scripts. To increase the integrability, it is useful when the same concept (e.g. address) is represented in the same way in the XML formats (e.g. by XML elements `street`, `city`, `zip`, etc.). In that case, parts of the scripts might be reused for those parts of the XML formats. However, this is not always possible (e.g. at some places of the XML formats only `zip` itself is present). In other cases, the name of an XML element or XML attribute needs to be different (e.g. XML elements `supplier-city` and `contractor-city`, both representing city, in an XML format which we use to report on distribution of suppliers and contractors across the cities in the Czech Republic). In these cases, a mapping of XML elements and XML attributes of the XML formats to the conceptual schema (e.g. mapping `supplier-city` and `contractor-city` to a shared attribute *city* in the conceptual schema) might be useful. This would allow the developers to map their local data representation schema (e.g. relational database schema) to the conceptual schema instead of mapping the schema to many XML formats. By composing the mapping it would be possible to generate the transformation scripts automatically or, at least, to help the designers with their development.

Adaptability of XML formats The last but not least problem arises when a change needs to be made in one or more XML formats. We distinguish two kinds of changes. First, it may be necessary to make a change in an existing XML format such as adding an XML element or XML attribute or moving it from its current location in the XML format to another. Second, there may appear a requirement to create a completely new XML format or to remove an existing one. Changes to the XML formats may be required because of various technical reasons (e.g. to increase their readability or integrability) or because of changes in the domain (i.e. at the conceptual level). The second case is what is currently happening in the Czech Republic. A new legislation is currently being prepared and it will result into some changes in the XML formats. Not only new

Property	Evaluation of NRPP
Readability	– hierarchical nature of XML not exploited – XML schema definitions missing – conceptual schema of public procurement domain missing – documentation of XML formats provided (in form of Excel sheet)
Integrability	– same concepts (e.g. address) represented in different ways – integration via a common schema not possible – automatic generating of transformation scripts is not possible
Adaptability	– adaptability via a common schema and automatic propagation to XML schemas not possible – adaptability of integration and transformation scripts not possible

Table 11.1: Summarization of readability, integrability and adaptability properties of NRPP

kinds of documents will be required to be sent to NRPP (which means creating completely new XML formats). There will also be legislative changes which will result into changes to existing XML formats. For example, a complete list of bids for a contract will be mandatorily published instead of their total number which is required by the current legislative.

By the term *XML formats adaptability* we denote the measure of how easy it is to change the XML formats and react on these changes. For example, let us consider a requirement to have a street name and number for each address in two separate strings instead of a less detailed postal address in one string. In our sample XML format depicted in Fig. [126], this means replacing XML element `cont_postal_address` with more detailed `cont_street` and `cont_street_no`. Similarly, XML element `docs_postal_address` must be replaced. And other XML elements representing postal address in other XML formats need to be adapted accordingly as well. A more complex change is the one which will be caused by the new legislation. Wherever `number_of_bids` attribute from the PIM schema is represented in the XML formats, there will be necessary to include the detailed information about each particular bid and the bidder and distinguish the winning bid.

To increase adaptability in this case, it is useful to have only one XML representation for a given concept (e.g. one sequence of XML element declarations for addresses) and share it across all XML formats. In that case a change may be implemented only at this one place instead of a repetitive change at various places of the XML schemas. However, this is not always possible (similarly to the case of integrability). In these cases, the mappings of the XML schemas to the conceptual one would help. A change could be then made only at one place of the conceptual schema (e.g. in a class *Address*) and propagated to all XML schemas which involve XML representations of addresses.

The adaptability is also important for the developers of information systems which communicate with NRPP. First, when a new XML format appears, they need to develop scripts which export/import their data from/to the XML format. This is a problem similar to the integrability described above. Second, when an existing XML format is changed, they need to adapt their scripts and, possibly, adapt their internal data representation (e.g. relational database). For example, replacement of XML element `cont_postal_address` with `cont_street` and `cont_street_no` may result into a corresponding change of local relational storage and also SQL/XML scripts which translate the relational representation into the XML formats. Similarly, replacing the number of bids with details of particular bids may lead to changes in the local storage and scripts. As in the previous case, adaptability in this case could be increased when a mapping of the XML formats required by NRPP and local database schema to the conceptual schema would exist. This could inform the developer about what parts of the local storage are affected by the change or whether the local storage needs to be extended. This could also help with adapting the transformation scripts.

We summarize the properties of NRPP in Tab. 11.1 from the three viewpoints. In the following sections, we demonstrate how to improve them by adding a common conceptual schema.

11.3 Conceptual Model for Public Contracts

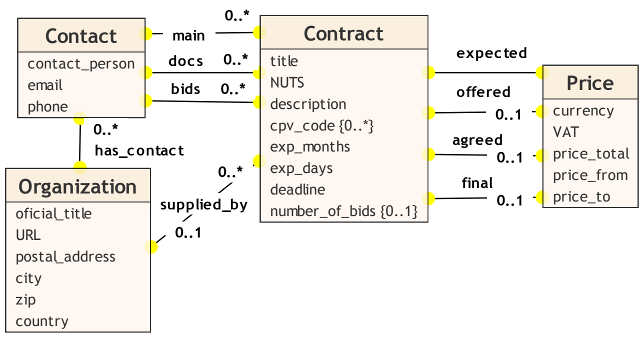


Figure 11.2: PIM Schema of Public Procurement Domain

Platform-Independent Model A sample PIM schema of a part of the public procurement domain is depicted in Fig. 11.2. We describe only a selected part of the diagram for the reader (we kindly ask the reader to interpret the rest intuitively). There is a class *Contract* which models public contracts. Further, there is a class *Contact* which models contact information. It is associated with *Organization* class which models organizations. Each contact is associated with one and only one organization and with many contracts. For a contract there is the main contact, contacts where documentation may be acquired, and contact where bids should be sent. These relationships between contracts and contacts are modeled by associations *main*, *doc*, and *bids*, respectively.

Platform-Specific Model Two sample PSM schemas are depicted in Figures 11.3(a) and 11.3(b). They model two particular XML formats. From the conceptual perspective, components of both PSM schemas are mapped to the components of the PIM schema depicted in Fig. 11.2. The interpretation (mapping) specifies the semantics of the mapped components of both PSM schemas in terms of the single PIM schema. We do not display the mapping explicitly. In our example it can be deduced intuitively (but not in general). Mapped PSM components are shown in the brown color and the others in the grey color. There is also a special kind of class displayed in the blue color. These are called *structural representatives*. They have a special meaning from the grammatical perspective and we explain them later.

Let us discuss the PSM schema depicted in Fig. 11.3(b). The PSM class *Contract* is mapped to the PIM class *Contract*. The PSM class *Contact* is mapped to the PIM class *Contact*. And, the PSM classes *Contractor* and *Supplier* are mapped to the PIM class *Organization*. The other PSM classes are not mapped to any PIM class, namely *Geography* and *Contracts*. It means that they have no semantics from the conceptual perspective.

The attributes and associations are mapped as well in an intuitive manner. For example, the PSM association connecting *Contract* and *Contact* is mapped to the PIM association *main*. It specifies that the PSM association associates each contract with a main contact, not with a contact which provides documents nor with a contact which accepts bids for the contract.

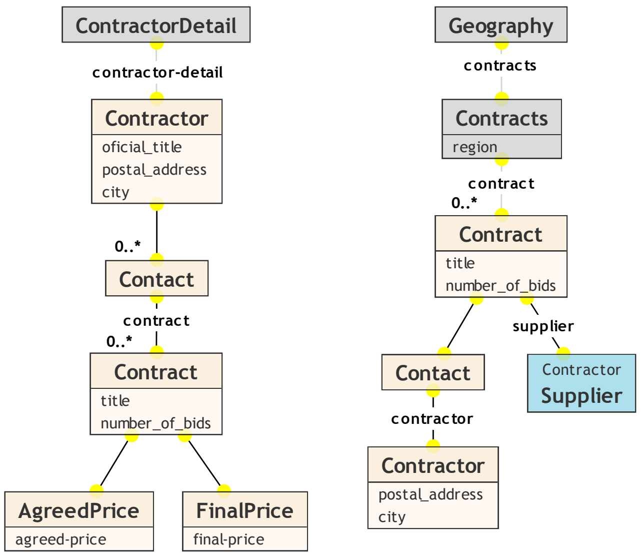


Figure 11.3: Two sample PSM schemas of (a) XML format for contracts viewed by regions, and (b) XML format for contractor details

11.4 Improving Quality

Having introduced the conceptual model for XML, we are now ready to show how we exploited it for increasing the readability, integrability and adaptability of the XML formats for public procurement in the Czech Republic. In this section we suppose the public procurement domain modeled by the PIM schema depicted in Fig. 11.2. NRPP uses 17 different XML formats for communication with other

```
<contractor_detail>
 <oficial_title>
  Charles Uni
 </oficial_title>
 <postal_address>
  Ovocný trh 3/5
 </postal_address>
 <city>Praha 1</city>
 <contract>
  <title>Úklid ...</title>
  <number_of_bids>
   4
  </number_of_bids>
  <price_total>
   12000000
  </price_total>
 </contract>
 <contract>...</contract>
 <contract>...</contract>
</contractor_detail>
```

```
<contracts>
 <region>PRG</region>
 <contract>
  <title>Úklid ...</title>
  <number_of_bids>
   4
  </number_of_bids>
  <contractor>
   <postal_address>
    Ovocny trh 3
   </postal_address>
   <city>Praha 1</city>
  </contractor>
  <supplier>
   <postal_address>
    Kladenska 43
   </postal_address>
   <city>Kladno</city>
  </supplier>
 </contract>
 <contract>...</contract>
</contracts>
```

Figure 11.4: Two sample XML documents valid against the XML format specified by PSM schemas depicted in 11.3(a) and 11.3(b), respectively

information systems. Therefore, we have created 17 PSM schemas specifying these XML formats and mapped them to the PIM schema depicted in Fig. 11.2. An example of an XML document in one of the XML formats is depicted in Fig. 11.1 as we have already discussed. One more XML format is discussed later in this section. For modeling we used our experimental implementation *eXolutio* which may be freely downloaded at `http://exolutio.com` and tested.

11.4.1 XML Format Readability

As we explained in the introduction, the problem of readability comes into play when an XML format is shown to a developer for the first time. If it is only described by an XML schema, it may be quite hard and time consuming to get a basic orientation in the format, especially when the semantics description is missing and/or the XML schema or the XML tags themselves are not fromatted correctly or, in the worst case scenario, when there are no XML schemas and the developer is supposed to learn the format from document instances. Firstly, we have solved the problem of missing XML schemas by inferring them from a set of instance XML documents. For this, we used our tool *jInfer* (`http://jinfer.sourceforge.net`) which is freely available for download. However, inferring XML schemas solved the problem only partially because the XML formats themselves are not very well structured. They do not suitably exploit the hierarchical nature of XML as we have already discussed. Therefore, we created a conceptual schema in the form of the PIM schema partly depicted in Fig. 11.2 and used our reverse-engineering method [60] which allowed us to semi-automatically convert the XML schemas to PSM schemas and derive their mapping to the PIM schema. The PSM schema in Fig. 11.5(a) is the result of the reverse-engineering method applied to the XML document in Fig. 11.1.

The result is much more readable for the developers because it helps them to understand not only the syntax but also semantics of the XML formats (because of mappings to the conceptual PIM schema). For example, our tool can highlight PSM components mapped to a selected PIM component or, vice versa, highlight a PIM component (e.g. by a different color) which is the target of the mapping of the selected PSM component. Or, we can display the list of PIM components at one side and list of PSM components at the other side and display the full mapping on a single screen. For this, we exploit the interpretation of a PSM schema against a PIM schema.Each PSM schema is bounded with its corresponding XML schema. Therefore, the designer may easily switch between the PSM schema and respective XML schema.

11.4.2 XML Format Integrability

The PSM schemas of the XML formats and their mappings to the PIM schema created in the previous step could also be very helpful for integrability of the XML formats. When a developer maps PSM schemas of own legacy XML formats to the PIM schema (possibly from some legacy XML schemas using our reverse-engineering method [60]), we can use the mappings to automatically generate XSLT scripts which transform XML documents from the legacy XML formats to the XML formats of NRPP. This is, of course, possible only when the legacy XML formats cover the same part of the procurement domain as the XML formats of the register. We demonstrate how XSLT scripts are created from the mappings later in Section 11.4.3, where we describe how XSLT scripts transforming XML documents from one version of an XML format to another version may be generated.

Similarly, the developer can map a schema of a local relational database to the PIM schema. This possibility is, however, not covered by our current methods and is the matter of our ongoing work.

11.4.3 XML Format Adaptability

Despite the importance of readability and integrability, we are mainly interested in increasing adaptability. In this section, we show how our conceptual model for XML facilitates (1) creating a new XML format, (2) adapting an existing XML format to increase its readability, and (3) adapting the XML formats on the basis of changes caused by a new legislation in public procurement.

Our adaptability methods (formally described in [91, 89]) are based on a set of atomic operations which allow for editing individual PIM and PSM schemas. When executed, the changes propagate from the PIM schema to sequences of atomic operations over the mapped PSM schemas and vice versa. We have developed a formal model behind the atomic operations which ensures that the operations are propagated correctly. It means that the PIM and mapped PSM schemas remain consistent with each other after the propagation. Due to the lack of space, we do not introduce the formalism in this chapter. The current literature distinguishes four types of operations for schema editing: *creation*, *removal*, *sedentary* and *migratory*. The former two allow for creating and removing components (i.e. classes, attributes, etc.). The sedentary operations allow for modification of component properties such as names, data types, cardinalities, etc. The migratory operations move components in schemas, e.g. move attributes between classes. These operations are indeed different in meaning from simply creating a new attribute or association and deleting the old one and are handled differently in the propagation mechanism.

Our atomic operations are supplemented with a propagation mechanism. For example, removing the PIM attribute *contact_person* of the PIM class *Contact* from our sample PIM in Fig. 11.2 is mandatorily propagated to removing each PSM attribute mapped to *contact_person* or removing their mapping to *contact_person*. Similarly, moving the PIM attribute *email* of *Contact* to *Organization* is mandatorily propagated to moving mapped PSM attributes respectively or removing their mapping. The final decision remains for the developer.

We use one additional type of atomic operations necessary for preservation of semantics and for the propagation mechanism. This type declares that two sets of attributes or two sets of associations are semantically equivalent. We call these operations *synchronization operations*. These operations do not have any effect on the schema on which they are performed, but are necessary for correct propagation. For example, splitting an attribute *postal_address* into more detailed attributes *street* and *street_no* in the PIM schema does not mean only creating the two new attributes and removing the old one. It also includes a specification that the old attribute is semantically equivalent to the new ones.

The atomic operations are not intended to be used directly by an XML schema developer. They are too primitive and, therefore, not user friendly. For example, they do not include the already mentioned operation for splitting an attribute. They are meant to be used in various combinations, so-called *composite operations*, which are more user friendly. For example, we may define the operation for splitting an attribute to two new attributes as a sequence of atomic operations which create the new attributes, synchronize them with the old attribute and, finally, remove the old attribute.

As a small demonstration of how the formalism looks like see Table 11.2 containing formal descriptions of some operations regarding attributes.

Operation	**Kind**	**Effect**
$A = \alpha_a(C)$	Addition	Adds a new attribute A to an existing class C.
$\delta_a(A)$	Removal	Removes an existing attribute A.
$\sigma_a(\mathcal{X}_1, \mathcal{X}_2)$	Synchronizing	Synchronizes two sets of attributes $\mathcal{X}_1$ and $\mathcal{X}_2$. **p:** $(\exists C \in \mathcal{S}_c)(\mathcal{X}_1, \mathcal{X}_2 \subseteq$ *attributes*$(C))$

Table 11.2: Example atomic operations for PIM schema adaptation

The example composite operation of splitting an attribute would be described as $A_1 = \alpha_a(C), A_2 = \alpha_a(C), \sigma_a(\{A\}, \{A_1, A_2\}), \delta_a(A)$.

The advantage is that we can quite easily describe the impact each individual atomic operation has on the schemas and we can describe how they should be propagated between the two levels. When atomic operations are combined into a composite operation, the propagation of the resulting operation is the same as a concatenation of the individual atomic operations. Our formalism ensures correctness of such propagation. This makes the creation of composite operations versatile as their designer does not need to be concerned about the propagation, which is handled by this mechanism.

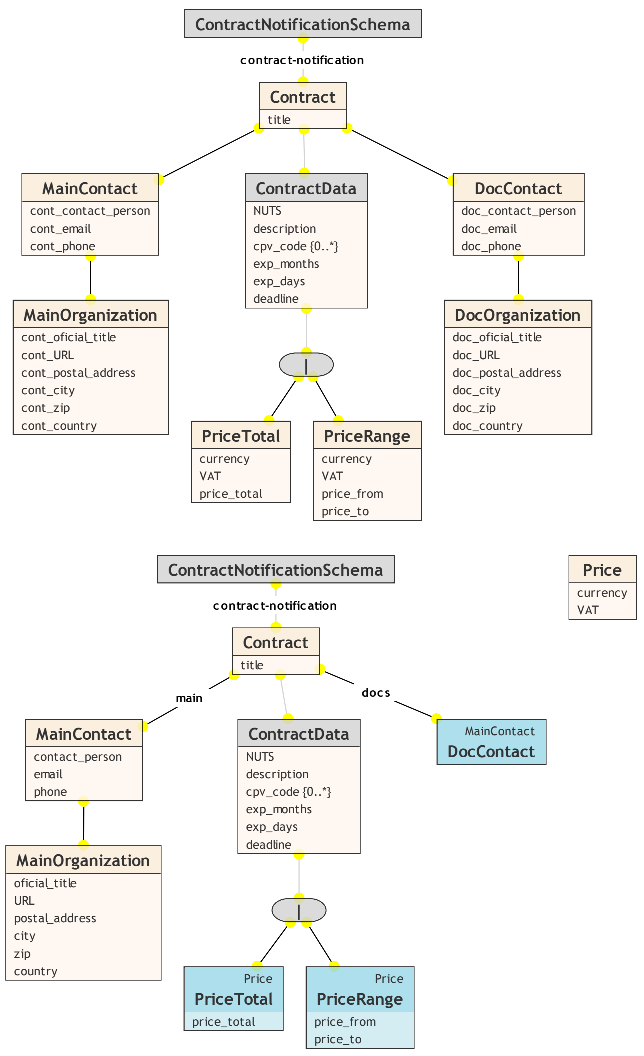


Figure 11.5: (a) PSM schema reverse-engineered from the XML document depicted in Fig. 11.1 and (b) its adaptation which models a better readable XML format

Creating an XML Format We first studied the adaptability problem of creating new XML formats. We instructed a developer to specify an XML format intended for details of public procurers and another developer to specify an XML format intended for reporting on distribution of public procurers or suppliers. Both developers were instructed to use our tool to create a PSM schema on the basis of the PIM schema from Fig. 11.2. The resulting PSM schemas are depicted in Fig. 11.3(a) and 11.3(b). The PSM schemas were then automatically translated to XML schemas. The developers also tried to create the XML schemas manually. The experiment proved that designing the XML formats in the form of PSM schemas is easier.

The tool provides several operations for creating PSM schemas on the basis of a PIM schema. For example, when one of the developers needs to design attributes of the PSM class *Organization*, which is mapped to the PIM class *Organization*, (s)he does not need to create them manually. The tool offers him/her the set of PIM attributes of the PIM class *Organization* and (s)he selects those which should be represented in the PSM schema. The tool then creates the corresponding PSM attributes. Similarly, when the developer needs to design the child PSM associations of the PSM class *Organization*, (s)he does not create them arbitrarily on his/her own. The tool offers him/her the set of associations connected to the PIM class *Organization* and the developer selects which ones should be represented as the child PSM associations. The tool then creates the corresponding PSM associations with correct mappings automatically.

Such process of designing PSM schemas on the basis of the PIM schema is not only fast and easy. It also prevents from errors or mismatches that would undoubtedly occur if the developers would have to create the XML formats manually only on the basis of some informal verbose specification. For example, one of the developers could forget to include some required part of the domain (e.g. organization city) because (s)he missed it in the verbose specification. Or, (s)he could omit that there are two possible kinds of price information – total price and price range. (S)he could also model a particular part of the reality at the different level of granularity. The tool prevents from such mismatches as it forces the developers to work with the shared PIM schema. On the other hand, it provides possible semantics of each component in the PSM schema, e.g. possible semantics of PSM associations connecting PSM classes mapped to PIM classes *Contract* and *Contact* (there are three possible PIM associations connecting these classes and, therefore, three possible semantics). All this ensures that the developers have the same view of the problem domain and, therefore, they create the XML formats coherently (from the conceptual point of view).

Adapting an XML Format to Increase Readability Second, we studied the adaptability problem when an existing XML format needs to be augmented to increase its readability. Such adaptation was required, for example, in the case of the XML format whose PSM schema is depicted in Fig. 11.5(a). As we have already noted, the main problem of the XML format is its flat structure. The primary authors of the XML format tried to increase the readability by indicating XML elements which conceptually belong to the same concept by prefixing their name with a common prefix (e.g. "*cont_*" and "*docs_*" prefixes). However, this is not convenient for the reasons described below.

Only after studying additional documentation, it is possible to identify distinct concepts related to the XML elements. The owner concept is only in some cases indicated by the common prefix (e.g. *cont_* or *docs_*). This can be partly solved by providing a PIM and a PSM schema, which considerably increase readability. Moreover, common prefixes introduce redundancy into the document, when the same prefix is repeated again and again. From the point of view of the designer of an application communicating with the register, the XML format is also not suitable. The application that is required to accept input data in this format, needs to rely heavily on the position and names of the elements, whereas the integral feature of XML as a language – its hierarchical nature – can not be utilized at all.

We were asked to increase the readability of the XML format by removing these common prefixes and replacing them by additional XML elements wrapping the original XML elements with the same prefix. We achieved this in our tool by augmenting the PSM schema with application of a few atomic operations for renaming PSM components. This also allowed us to introduce a structural representative to the PSM schema to further simplify it. The result is depicted in Fig. 11.5(b).

All the proposed changes neither extend, nor narrow the semantics of the XML format. However, they change its structure. Therefore, if there are some XML documents valid against the old XML format, then they need to be transformed so that they are valid against the new XML format. Similarly, various information systems which communicate with the public procurement register may still need to communicate using the old version. Therefore, we need an adapter which transforms XML documents for the purposes of communication with these partners. In [71] we have developed a method which generates such transformation script expressed in the XSLT language automatically when two versions of the same XML format in a form of versioned PSM schemas are supplied. As the PSM schemas evolve from one version to the other using our atomic operations described earlier, the tool keeps a record of what the components transformed to and how. From this record and from both of the PSM schemas (the old and the augmented one), we can generate an XSLT script which transforms the XML document instances accordingly.

Adapting an XML Format to Suit New Legislation Finally, there is the problem of adaptability when a change in the legislation occurs. In our case, the new legislation demands details about individual bids instead of a simple information about the number of bids. This means not only augmenting an individual XML format but, primarily, augmenting the PIM schema because the change is a conceptual change. The change to the PIM schema may be easily done by a developer in our tool using a set of operations which allow for converting the attribute *number_of_bids* of the PIM class *Contract* to a new class *Bid* associated with *Contract*. Two associations connecting *Bid* and *Contract* are created by the developer. An association *winner* associates the winning bid with the contract while an association *offer* associates the other bids. The new class *Bid* moreover contains new attributes which model additional information about particular bids. Also, *Bid* is associated with *Organization* and *Price*. The respective associations associate the bidder and offered price by the bidder for each bid. The changes to the PIM schema are depicted in Fig. 11.6(a). Conceptually, the change means that the original information about the number of bids was replaced by particular bids associated with the contract where one of them is listed as the winning bid. Instead of the original winning supplier and his offered price, a particular bidder and offered price is considered now.

After the PIM schema is changed, it becomes inconsistent with the original PSM schemas which still contain PSM attributes *number_of_bids* mapped to the removed PIM attribute *number_of_bids*. Therefore, these PSM schemas must be adapted accordingly. The adaptation is not done by the designer manually. The tool helps him/her to solve the inconsistencies. It shows all affected places and offers possible propagations. This includes (1) removing the affected part of the PSM schema, (2) removing the mapping of the affected part to the PIM schema, or (3) augmenting the the affected part so that it is consistent with the PIM schema. After the designer decides, the tool automatically performs the selected steps. The result of propagation to the PSM schema depicted in Fig. 11.3(b) is depicted in Fig. 11.6(b) (the figure only shows the changed parts

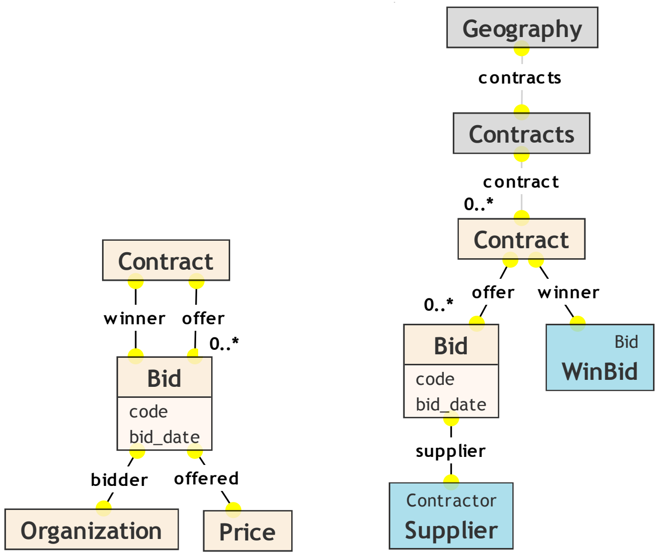


Figure 11.6: (a) Adaptation to PIM schema cause by new legislation, and (b) PSM schema adapted by propagation mechanism on base of changes to PIM schema

of the PSM schema). The designer selected the third option, i.e. to augment the PSM schemas. We can see that *number_of_bids* was removed. Instead, all offered bids and one distinguished winning bid are present. For each bid a supplier is present as well. This corresponds to the changes performed in the PIM schema. Similarly, the change is propagated to the PSM schema depicted in Fig. 11.3(a) and other PSM schemas which contained the information about the offered number of bids.

11.5 Methodology

The solution presented in the previous section is not specific to NRPP. In this section, we generalize it to a methodology which can be applied to any system with similar problems. In general, we suppose a family of XML formats which are conceptually interrelated by a common problem domain (e.g. public procurement, medical records, justice, etc.). The family may describe interfaces of web services of a particular information system or it may be just a standard created by some standardization organization without any relationship to a particular system (e.g. OpenTravel.org [107]). In any case, providing only XML schemas specifying the XML formats might bring the discussed problems with readability, integrability and adaptability. As we have shown the problems may be solved by adding a conceptual schema of the problem domain in a form of a PIM schema and mapping each XML schema expressed as a PSM schema to the PIM schema. For this, the following steps need to be performed by a designer of the XML formats:

1. Create a PIM schema that describes the problem domain at the conceptual level.
2. Convert an XML schema of each XML format in the family to a PSM schema.

3. Map each PSM schema to the PIM schema.

The first step must be done completely manually by the designer. The second step is performed automatically by our tool. The last step is done semi-automatically. The designer creates the mapping but our tool offers possible mappings. The tool exploits component similarity based on string similarity, lexical similarity and structural similarity and combines them by using adjustable weights to one value. The possible mappings are offered sorted by this value, so the most probable mapping is on the top. The details of this method are covered in [60]. It may happen that the PIM schema does not completely cover the semantics of all PSM schemas. In that case the PIM schema needs to be extended by the designer which is also assisted by the tool.

The steps result into a set of PSM schemas mapped to the common PIM schema. As we described in the previous steps it greatly increases the readability of the XML formats. When an existing XML format (e.g. from a system we need to communicate with) needs to be integrated into the existing solution, the steps 1 to 3 are performed again for this particular format and XSLT scripts which transform data from or to this XML format may be generated automatically by our tool as we have already discussed.

Another problem arises when a new XML format needs to be created directly by the designer (e.g. a new view on the data processed by the system needs to be implemented). This is a special case of adaptability. For this, the designer needs to perform the following steps:

1. Identify a part of the PIM schema describing the part of the reality that is going to be represented by the XML format.

2. Shape the selected part of the PIM schema into a new PSM schema that models the XML format.

3. Convert the PSM schema to a corresponding XML schema expression.

The first step is done manually. The second step is also done manually but our tool assists the designer by ensuring that the created PSM schema is consistent with the PIM schema. As we have already discussed, creating a PSM schema on the base of the PIM schema is much easier for the designer than simply writing the XML schema expression manually. The last step is performed automatically.

There are two more adaptability problems that we have discussed in the previous text which includes adapting existing XML formats when a change occurs. The designer needs to perform the following steps:

1. Identify the location where the change must be performed:

2. When the PIM schema is identified:

    (a) Perform the change in the PIM schema.

    (b) Propagate the change to the all affected PSM schemas.

3. When a PSM schema is identified:

    (a) Perform the change in the PSM schema.

(b) Propagate the change to the PIM schema.

(c) When the PIM schema is affected, propagate the change to the all affected PSM schemas.

The first step must be done manually. The designer must also manually perform the change in the PIM or in a PSM schema. For this, our tool offers a set of various operations (e.g. adding/removing components, splitting/merging/moving attributes and associations, etc.). The propagation (in PIM to PSM as well as PSM to PIM direction) is performed semi-automatically. The tool automatically identifies where the change needs to be propagated because the consistency between the PIM and PSM schemas is affected. It also automatically offers possible solutions of the inconsistencies. The designer only selects the required solutions which are then performed by the tool automatically.

11.6 Evaluation and Related Work

Our experiments showed that the proposed approach increases the readability of XML formats and makes the designer more effective when she is integrating or adapting the XML formats. In this section, we briefly compare the proposed solutions with existing commercial as well as academic solutions.

Various approaches aim at increasing readability of XML formats. First, there are tools such as Altova [6] which visualize XML schemas. However, visualization is only the first step towards increasing readability. It does not provide additional conceptual level we used in our work. Second, there are academical approaches which aim at providing the conceptual level. We surveyed them in [97]. There are approaches based on the well-known ER model [11, 69, 73] and approaches which extend UML [117, 119]. However, we could not use them for our purposes, because they all consider an individual conceptual schema for each particular XML format. In our case we have a whole family of different XML formats used in a single system and we need to bind them with a single conceptual schema. This is what makes our approach unique – we consider the whole family of XML formats and a shared conceptual schema each XML format is mapped to. For more details, please refer to [92].

Regarding integrability of XML formats, again there are tools such as Altova which allow for mapping XML formats on each other. However, again, they are not suitable for our approach because we need to integrate any XML format with any other XML format if it is possible and required by the designer. Therefore, it is much easier to map the XML formats to a shared conceptual schema instead of mapping on each other. It is then possible to test whether two XML formats which are going to be integrated share the same part of the reality (i.e. whether they are mapped to the same part of the conceptual schema) and derive the mapping automatically. We use various existing solutions based on schema matching surveyed in detail in [121]. We combined these various approaches in our own method [60].

And finally, there is the adaptability. The current approaches to adaptation of XML formats were surveyed in [47]. They can be divided into several groups. Approaches in the first group consider changes at the schema level and differ in the selected XML schema language, i.e. DTD [2, 31] or XML Schema [127, 24].

In general, the transformations can be variously classified. For instance, chapter [127] proposes *migratory*, *structural* and *sedentary* changes. The changes are expressed variously and more or less formally. For instance in [24] a language called *XSUpdate* is described. The changes are then automatically propagated to the extensional level to ensure validity of XML data. There also exists an opposite approach that enables one to evolve XML documents and propagate the changes to their XML schema [22]. Approaches in the second and third group are similar, but they consider changes at an abstraction of logical level – either visualization [53] or a kind of UML diagram [39].

However, none of these approaches is appropriate for our problem. They all work at the platform-specific level (according to our terminology introduced in Section 11.3), since it directly models the components of the XML. They do not consider the XML formats mapped to the conceptual schema as we do in our approaches. In all the cases only a single separate XML schema can be processed. Therefore, we could not use these approaches in our work, because we need to process multiple different XML schemas at once when a change at the conceptual level appears.

11.7 Conclusions

In this chapter we studied XML formats used by the National Register of Public Procurement in the Czech Republic (NRPP) for communication with its partners. We studied the XML formats of the register from three viewpoints: the degree of readability, integrability and adaptability. We showed that the degrees are low and they can be improved by using a conceptual schema of the problem domain with XML schemas of the XML formats mapped to the conceptual schema. We aimed mainly at increasing adaptability and showed that the conceptual schema is useful when creating a new XML format, increasing the readability of an existing XML format as well as making complex changes to various XML formats on the basis of some legislative changes.

Using the experience with NRPP we proposed a general methodology which shows how to apply our approach in any other domain as well. In the end, we evaluated our approach by comparing it with other solutions in the area of XML format readability, integrability and adaptability and showed the drawbacks tha recent approaches have.

12. eXolutio: Tool for XML Schema and Data Management

In this chapter we describe a tool for evolution and change propagation of XML applications called *eXolutio*, which has been developed and improved in our research group during last few years. The tool implements all the approaches described in this thesis in the area of *evolution of XML schemas*. The text should help the reader to get acquainted with the tool and its theoretical background.

The contents of this chapter was published as a conference paper *eXolutio: Tool for XML Schema and Data Management*[1] [55] in Dateso 2012 Annual International Workshop on DAtabases, TExts, Specifications and Objects (DATESO 2012).

12.1 Introduction

In our research group we have focused on the area of efficient and correct management of a family of XML formats for several recent years. Starting with a simple idea of propagation of changes among related XML formats, we have gradually extended our effort towards a robust framework and its implementation in a tool called *eXolutio*. It currently supports the original idea of designing XML formats using the principles of *Model Driven Architecture* (MDA) [78], their evolution, and integration of new XML formats into the framework.

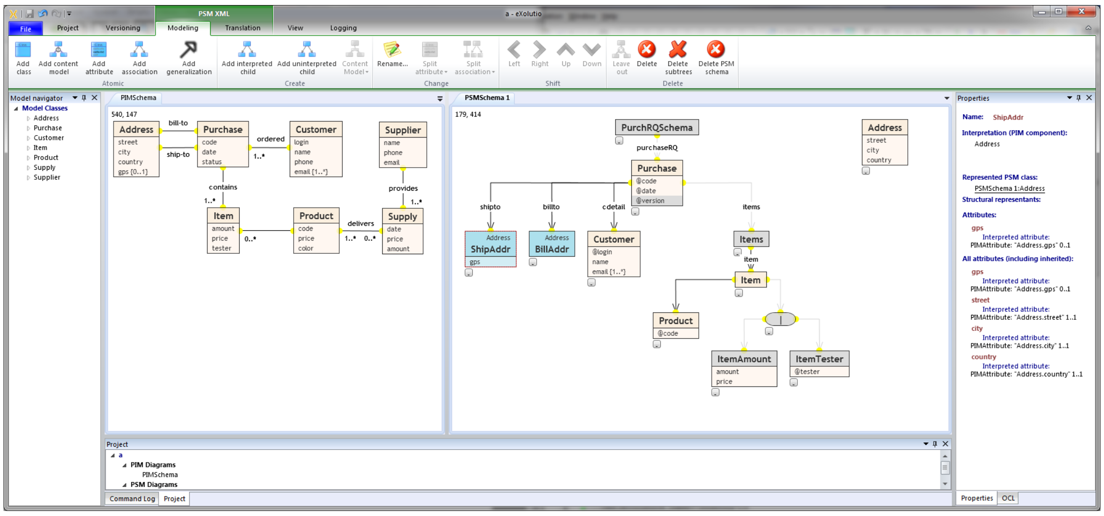
Contributions The aim of this chapter is to provide a covering overview of our research in the area of XML evolution, to describe architecture and implementation of the *eXolutio* tool, to present results of our experiments with the tool proving the concept and efficiency and a comparison of the tool with similar tools.

Outline The rest of the chapter is structured as follows: In Section 12.2 we introduce *eXolutio*, our tool in which we implement our research results. Experiments with our tool are described in section 8.8. In Section 12.3 we discuss the related work. Finally, in Section 12.4 we conclude.

12.2 eXolutio architecture

The implementation of our research results is a tool called *eXolutio* [55]. There exists also an older version of our conceptual model and its implementation called *XCase* [54], which is the predecessor of *eXolutio*. For simplicity, we will stick to the current name. *eXolutio* allows the user to model a PIM schema and multiple PSM schemas with interpretations against the PIM schema. The user can then evolve the whole set of schemas coherently, because his operations are propagated to all affected places by a mechanism described in [89].

[1] `http://ceur-ws.org/Vol-837/paper19.pdf`



Name: ShipAddr

Interpretation (PIM component):
Address

Represented PSM class:
PSMSchema 1:Address

Structural representants:

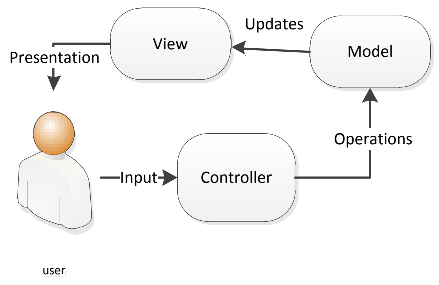
Attributes:
gps
Interpreted attribute:
PIMAttribute: "Address.gps" 0..1

All attributes (including inherited):
gps
Interpreted attribute:
PIMAttribute: "Address.gps" 0..1
street
Interpreted attribute:
PIMAttribute: "Address.street" 1..1
city
Interpreted attribute:
PIMAttribute: "Address.city" 1..1
country
Interpreted attribute:
PIMAttribute: "Address.country" 1..1

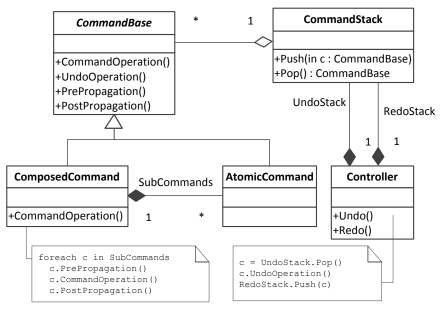
Figure 12.1: *eXolutio* screenshot

The architecture of *eXolutio* is based on the Model–View–Controller (MVC) design pattern (Figure 12.2(a)). This means that we hold all the project data in the model part, neither mixing it with operations, nor visualization. Whenever a user issues a command, it is handled by the controller part. The controller makes all the necessary changes in the model. The view part observes that the model has changed and updates the visualization. The connections between individual parts are loose enough so it is possible to, e.g., use multiple visualizations. In particular, we have a Windows Presentation Foundation [76] visualization (a desktop application) a Silverlight [75] visualization (a web application) and a no-visualization (a console application) versions of *eXolutio* which all share the same model and the same controller.

Model The model part of the tool based on our conceptual model [96] consists of classes for each modeled component, such as a class, an association or an attribute on each of the modeled levels (PIM and PSM), a class for PIM and PSM schemas and a class for the whole project. Besides the obvious properties of components like a name or a collection of attributes of a class, each component class implements methods for serialization and deserialization of the component to and from XML. Therefore, when we save and load a project, we simply call a serialization or a deserialization method on all found objects in a certain order. Finally, each schema contains lists of all the components of individual types in that schema, so we can easily go through, e.g., all associations in a certain schema. Since one of the main features of our tool is the visualization of connections between the two levels of abstraction, one of the most common queries is "Give me all PSM classes which have this PIM class as their interpretation". We basically go through each PSM schema in the project and through each PSM class in that schema and check whether its interpretation is the given PIM class. In addition, the model contains methods for easy traversal of both the PIM and PSM schemas. An example can be a method for retrieval of all attributes of a PSM class including those inherited by the structural representative constructs. Another example can be a method that gets all uninterpreted descendants of an interpreted PSM class. When a certain method representing a query over the model is needed by the controller more than once, we make it a model method so that everyone can use



(a) Overall architecture



```
foreach c in SubCommands
  c.PrePropagation()
  c.CommandOperation()
  c.PostPropagation()
```

```
c = UndoStack.Pop()
c.UndoOperation()
RedoStack.Push(c)
```

(b) Controller

Figure 12.2: eXolutio – main MVC components

it.

View The view component serves for two purposes: it visualizes the model for the user and provides user-friendly interface to run the controller commands. PIM schemas are depicted as UML diagrams and the layout of the diagram is left up to the user preference, for PSM schemas we use automatic hierarchical layouting to emphasize the fact that a PSM schema is a tree/forest. Besides the visualizations of the schemas, view component provides several windows and controls that help the user to navigate the modeled project, see the connections between individual concepts and follow the various links (e.g. find interpretation of an attribute or a class referred from a structural representant). The view component can be run either as a desktop application or inside a web browser using Silverlight plugin technology. This browser view can be used to accompany a documentation of published XML schema standards (e.g. [107]). An interactive visualization of a family of schemas joined by a common model can benefit greatly every system designer, who wants to adopt a third party standard and needs a clear overview of the whole problem domain and its individual schemas.

Controller The controller is the core of the tool. It contains all the operations and algorithms that make the tool unique. Also, it contains the usual command and undo/redo management. Whenever a user issues a command from view, it is handled by the controller. The controller (depicted in Figure 12.2(b)) gets all the necessary parameters from view such as what command is requested, the currently selected components, the new name for a component, etc. The controller creates the appropriate command, which in most cases will be one of our composite operations (described later in this section) and passes all the required parameters. The operation executes and updates the model accordingly. Then it places the command on the undo stack. The command itself contains all the information it needs to change the model back to the state it was in before the command was executed. In other words, we can simply call undo and the command knows what it needs to do and whether it is possible. This way, we can stack the executed commands and perform undo and redo operations as needed and as usual. In [89] we have described a theoretical background for atomic (simple, well defined) and composite (user-friendly) operations, which we will now describe from the implementation point of view. One of our goals was also to make the two levels of abstraction (PIM and PSM) work as independently of each other as possible while maintaining consistency when there are connections between the levels. Therefore, the operations need to work at their respective levels and be propagated only when there is an interpretation. Since we have a quite complex system of operations, we had to break it down into simpler parts. This means that among our atomic operations one can find for example an operation that creates an attribute. But it does not do anything else than that. Specifically, it does not give a name to the attribute, it does not set its datatype, etc. For that, we have other atomic operations. Having the atomic operations, we can compose more complex and user-friendly ones. A basic composite operation can be the already mentioned creation of an attribute, which this time is user friendly. It is composed of the creation of the attribute, renaming the attribute, setting its cardinality and its datatype. If it was a PSM attribute, the operation would also set its *xform*. So this is basically a predefined sequence of 4 or 5 operations, which is quite simple. Another simple example can be deletion of an attribute. This means setting its cardinality, name and datatype to default values and then deleting it. The reason for this is that when we undo this operation, we want the name and the other values of the attribute to recover, so it is not correct to just delete the attribute. Let us have a look at a more advanced example.

So far we have described how to compose atomic and composite operations. However, these worked on their respective levels of abstraction. Now we have to make sure that when there is an interpretation of a PSM schema against a PIM schema, we keep the model in a consistent state and save the user's time by propagating the changes between the levels. This is achieved by the propagation. Before each atomic operation is executed, a method implementing its propagation to the other level is called. It determines whether there is an interpretation and therefore the need to propagate. If so, it creates a (possibly) composite operation on the other level of abstraction and integrates it to the currently running operation. Only when the propagation succeeds, the original atomic operation that caused it is executed. This way, the propagation actually becomes a part of the currently running operation. This is convenient because when it finishes, it can be undone and redone like any other operation.

12.3 Related work

The current approaches towards evolution management can be classified according to distinct aspects [74, 34]. The changes and transformations can be expressed [104, 21] as well as divided [28] variously too. However, to our knowledge there

exists no general framework comparable to our proposal; particular cases and views of the problem have previously only been solved separately, superficially and mostly imprecisely without any theoretical or formal basis.
XML View We can divide the current approaches to XML schema evolution and change management into several groups. Approaches in the first group consider changes at the schema level and differ in the selected XML schema language, i.e. DTD [2, 31] or XML Schema [127, 24]. The changes are expressed variously and more or less formally. Approaches in the second and third group are similar, but they consider changes at an abstraction of logical level – either visualization [53] or a kind of UML diagram [39]. Both cases work at the PSM level, since they directly model XML schemas with their abstraction. No PIM schema is considered. All approaches consider only a single separate XML schema being evolved.

In all the papers cited the authors consider only a single XML schema. In [111] multiple *local* XML schemas are considered and mapped to a *global* object-oriented schema. Then, the authors discuss possible operations with a local schema and their propagation to the global schema. However, the global schema does not represent a common problem domain, but a common integrated schema; the changes are propagated just upwards and the operations are not defined rigorously. The need for well defined set of simple operations and their combination is clearly identified in Section 6 of a recent survey of schema matching and mapping [14].

12.4 Conclusion

In this chapter, we introduced *eXolutio*, our tool for XML schema and data management. We surveyed related work and we showed the theoretical background behind our tool. Its evaluation on real-world XML schemas is in section 8.8.

13. Generating Lowering and Lifting Schema Mappings for Semantic Web Services

With the introduction of the SAWSDL[131] W3C recommendation, the possibility of enriching web service interfaces with semantic model references surfaced as a foundation for semantic web services. However, the recommendation says neither what the semantic model should be nor what to do with the actual XML data. In this chapter, we exploit our conceptual model for XML to generate SAWSDL enriched XML schemas, but mainly to automatically generate the so called Lifting and Lowering schema mappings in a form of XSLT scripts. These scripts can be used to transform the XML data produced by the web service into RDF data (lifting) and vice versa (lowering). In the RDF data state the data can be manipulated using a knowledge given by a corresponding ontology mapped to our model. Also the reasoning power granted by the ontology description can be exploited.

The content of this chapter was published as a workshop paper called *Generating Lowering and Lifting Schema Mappings for Semantic Web Services*[1] [61] in 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications (AINA 2011).

13.1 Introduction

Today, the use of web services is wide-spread. Companies use them to share data and provide and consume services. A typical web service uses XML or XML-based languages such as SOAP as a format for data transfer along with other XML-based languages like WSDL to describe its interface. Furthermore, web services can be *orchestrated* to form more complex web services. But as always when using various services and data sources, heterogeneity becomes a problem. It can be partly solved by a simple XSLT stylesheet, but some format adaptations need a more sophisticated approach. Recently, ontologies have emerged as a possibility of describing complex relations among entities. The languages used for describing ontologies are RDF, RDFS and OWL. The question was how to exploit the power of ontologies for the use in web services. One part of the answer is a recent standard SAWSDL, which describes how to enrich WSDL and XML Schema element and type definitions with an association to a *semantic model* (such as an ontology) and two links to so called *lifting* and *lowering schema mappings*. There are basically two ways of how to exploit this approach. The first one is that a web services produces XML data and the data is transformed (or *lifted*) using the lifting schema mappings to semantic data (e.g. RDF). Then the data can be manipulated using the power of ontologies - in addition to syntax described by conventional means they also describe semantics of the data. Then the data can be transformed back (*lowered*) to XML and transported. The second idea is to

[1] http://doi.ieeecomputersociety.org/10.1109/WAINA.2011.13

have two parties using ontological data and wanting to communicate using web services and thus, XML. The XSLT scripts are then used in the reverse order. First we *lower* the data for transport and then we *lift* it back to ontological data. SAWSDL does not specify what the semantic model is nor what the schema mappings should look like, so up to now all the lowering and lifting schema mappings needed to be created and maintained manually.

Contributions In this chapter we ease the problem of creating lifting and lowering schema mappings identified e.g. in [64] by introducing a method for automatic generation of lifting and lowering schema mappings. The mappings are in a form of XSLT stylesheets. The stylesheets are generated using our conceptual model for XML [86]. We assume that a conceptual schema of all XML schemas in the system is created during information system analysis so we can use it for the automatic generation of the XSLT stylesheets. This is a valid assumption, because the conceptual model should be used for management of evolution and integration of XML schemas anyway as shown in our previous work Chapter 1. It can be created by a designer during the analysis of the system [86] and also it can be created by a semi-automatic reverse-engineering process from the XML schemas as we describe in Chapter 4. The contribution of this chapter is an extension of the conceptual model toward semantic web services. The method is experimentally implemented in our tool called XCase [54].

Outline The structure of the rest of this chapter is as follows. In Section 13.2 we briefly introduce our conceptual model for XML. In Section 13.3 we introduce support algorithms. Section 13.4 contains a description of our algorithms for generation of lifting and lowering XSLT stylesheets. Section 13.5 provides a brief overview of the implementation of our approaches. Section 13.6 contains a survey of related work and Section 13.7 concludes.

13.2 Conceptual Model for XML

In this section, we briefly summarize our original conceptual model for XML from Chapter 3 and extend it with function *ontologyEquivalent* on the platform-independent level.

We firstly introduce several symbols. $\mathfrak{L}$ denotes the set of all string labels. $\mathfrak{L}_a$ and $\mathfrak{L}_e$ are two sets s.t. $\mathfrak{L}_a \cup \mathfrak{L}_e = \mathfrak{L}$, $\mathfrak{L}_a \cap \mathfrak{L}_e = \emptyset$ and each label in $\mathfrak{L}_a$ starts with the '@' symbol. $\mathfrak{D}$ denotes the set of all basic data types such as `string`, `integer`, etc. $\mathfrak{C} \subset N \times (N \cup \{*\})$ is a set of *cardinality constraints* where N denotes the set of natural numbers and $(\forall(x, y) \in \mathfrak{C})\ (x \leq y \vee y = *)$. $\mathfrak{P}^S$ and $\mathfrak{O}^S$ denote the power set of a set S and the set of all ordered sequences of elements of S, respectively.

The *PIM* is de-facto the model of UML class diagrams. It introduces three modeling constructs: *PIM class*, *PIM attribute* and *PIM binary association*. We provide its formalization in Definition 13.1.

Definition 13.1 *In this chapter, a* PIM *schema is a 10-tuple* $\mathcal{M} = (\mathcal{C}, \mathcal{A}, \mathcal{R}$, *name*, *ontologyEquivalent*, *type*, *attrs*, *ends*, *acard*, *rcard*$)$ *where*

- $\mathcal{C}$*,* $\mathcal{A}$ *and* $\mathcal{R}$ *are sets of* PIM classes*,* PIM attributes *and oriented* PIM associations*, respectively,*
- *name* : $\mathcal{C} \cup \mathcal{A} \cup \mathcal{R} \to$ *string assigns a label to each PIM class, attribute or association; the label is called* name*,*
- *ontologyEquivalent* : $\mathcal{C} \cup \mathcal{A} \cup \mathcal{R} \to$ *string is a function which assigns an RDF ID to each PIM class, attribute or association; the ID is called* ontologyEquivalent
- *type* : $\mathcal{A} \to \mathfrak{D}$ *is a function which assigns a data type to each PIM attribute,*
- *attrs* : $\mathcal{C} \to \mathfrak{P}^{\mathcal{A}}$ *is a function which assigns a set of PIM attributes to each PIM class s.t.:*
  - $(\forall A \in \mathcal{A})(\exists C \in \mathcal{C})(A \in attrs(C))$*, and*
  - $(\forall C_1, C_2 \in \mathcal{C})(attrs(C_1) \cap attrs(C_2) = \emptyset)$*,*
- *ends* : $\mathcal{R} \to \mathcal{C} \times \mathcal{C}$ *is a function which assigns a sequence of two PIM classes to each PIM association; for* $R \in \mathcal{R}$ *with* $ends(R) = \{C_1, C_2\}$ *we say that* C_1 *and* C_2 participate *in* R *and* R *goes from* C_1 *to* C_2
- *acard* : $\mathcal{A} \to \mathfrak{C}$ *is a function which assigns a cardinality to each PIM attribute,*
- *rcard* : $\mathcal{R} \times \mathcal{C} \to \mathfrak{C}$ *is a function which assigns a cardinality to each PIM class* $C \in \mathcal{C}$ *and PIM association* R *s.t.* C *participates in* R*.*

The *PSM* consists of three modeling constructs which reflect the PIM constructs: *PSM class*, *PSM attribute*, and *PSM binary association.* Its formalization is unchanged form the one in Definition 3.3.

13.3 PIM and Ontology relations

In this section we will describe support transformations that allow us to create an OWL ontology from our PIM schema and vice versa. These algorithms are not needed for the actual generation of *lowering* and *lifting* XSLT stylesheets. However, they can ease the process when there is no PIM or no ontology present and these would have to be created manually. In the case when we already have a PIM and an ontology and they were not created by these algorithms, we need to connect them manually by setting the *ontologyEquivalent* function for each PIM construct to an appropriate ontology counterpart. This process can be eased by using techniques of schema matching similar to those described in [121]. In all the algorithms presented, we omit technical details regarding XML namespace manipulation, as it would only cloud the basic idea.

13.3.1 OWL to PIM

This simple algorithm allows us to create a PIM schema from an OWL ontology. Basically, we take the *OWL classes* and create corresponding PIM classes in a PIM schema. This can include inheritance relations. Next we take all instances of *DatatypeProperty* and create corresponding PIM attributes. The class to which an attribute belongs to is determined be the *domain* of a DatatypeProperty. The datatype of an attribute is determined by the *range* of a DatatypeProperty. Lastly, we take all the instances of *ObjectProperty* and create corresponding

PIM associations connecting PIM classes determined by the *range* and *domain* of the ObjectProperty. For each PIM class, attribute and association we set the *ontologyEquivalent* to the id (value of *rdf:about*) of the ontology construct from which they came from. Lastly, we take the attribute and association cardinalities which in OWL are described by *owl:Restriction* constructs and modify the *acard* and *rcard* functions accordingly.

These constructs are the only ones that concern us as they are the only ones representable in our PIM.

13.3.2 PIM to OWL

This algorithm allows us to create an OWL ontology from a PIM schema. The process is quite straightforward. For each PIM class an OWL class is generated. For each PIM attribute, an *OWL DatatypeProperty* construct is generated. It has two sub-elements. *Domain* indicates to which OWL class this property belongs. *Range* indicates the datatype of the attribute. Next, for each PIM association an *OWL ObjectProperty* construct is generated. Again, it has the *domain* and *range* sub-elements, indicating the source and target class of an association respectively. Lastly, we express the cardinality constraints represented by our *acard* and *rcard* functions as *owl:Restriction* constructs in the created OWL classes. To keep our propositions simple, we do not mention inheritance relations even though they are present in our conceptual model and can be expressed in OWL by the *rdfs:subClassOf* construct.

13.4 Lifting and Lowering XSLT stylesheets

In this section, we describe how the XSLT stylesheets that transform the XML data to RDF (*lifting*) and back from RDF to XML (*lowering*) work and how are they created. Each XSLT stylesheet is automatically generated from a PSM schema and transforms data corresponding to this schema (e.g. valid against the generated XML Schema). For easy handling by XSLT, the RDF/XML representation of ontological data was chosen. In all the algorithms presented, we omit technical details regarding XML namespace manipulation. Also we omit the generation of an XSLT stylesheet header (the *xsl:stylesheet* and *xsl:output* elements), as it is always the same for all stylesheets. In the following algorithms, the function *oE()* is a shortcut for the function *ontologyEquivalent()* and the function *TNCT(X)* returns the *template name corresponding to X*.

13.4.1 Lifting XSLT

A *lifting XSLT stylesheet* allows us to transform produced XML data to its RDF equivalent, adding references to the describing ontology. A user then can perform various operations over the RDF data using additional manipulation power and reasoning granted by its ontology description. What the stylesheet does is depicted in Algorithm 3.

Note 1 *In Algorithm 3 the function ID() returns an RDF ID for an XML element E. This is done for example by concatenating the ID of the correspond-*

Algorithm 3 Lifting XSLT algorithm

```
1: Generate the *rdf:RDF* root element
2: for all XML elements $E$ do
3:     $C$ is the PSM class corresponding to $E$
4:     Create *rdf:Description* element $d$ in the *rdf:RDF* element
5:     Create *rdf:about* attribute of $d$ and set its value to $ID(E)$
6:     for all XML attributes $A_x$ of $E$ corresponding to PSM attributes $A$ of $C$
       do
7:         Create an XML subelement of $d$ named $oE(I(A))$ with the value of $A_x$
8:     end for
9:     for all XML subelements $S_x$ corresponding to the children of $C$ do
10:        $R$ is the PSM association connecting $C$ and the current child
11:        Create a subelement $s$ of $d$ named $oE(I(R))$
12:        Create *rdf:resource* attribute of $s$ and set its value to $ID(S_x)$
13:    end for
14: end for
```

ing PSM class like `"http://www.example.org/Customer"` *and the ID of E like* `"#JohnDoe"`. *Also, the IDs can be generated by the XSLT* generate-id() *function.*

In Algorithm 4 we describe how we can generate the lifting XSLT script. First we need a template to match the root of an XML document using XPath query "/" (lines 2 - 6). From this template we call the template corresponding to the root PSM class of the PSM schema (line 4). Because we assume that we will have a well-formed XML document with only one root as the input, the *xsl:for-each* construct (line 3) should only match this one root. Next we need to go through every PSM class of the PSM schema (lines 7 - 29). For each PSM class we generate a named template. Inside the template, we generate the *rdf:Description* element (line 10) which will represent the current XML element processed by the XSLT in the RDF/XML representation of the RDF data. Next we generate the content of the *rdf:Description* element. We create its *rdf:about* attribute containing an ID (line 14). Then we generate the link to the corresponding ontology class (lines 17 - 19). Here we exploit the interpretation of PSM against the PIM and the *ontologyEquivalent()* function on the PIM. Next we process the PSM attributes of the current PSM class (lines 21 - 25) and transform them to XML elements with textual content. Lastly, we process all the children of the PSM class. For simplicity this process was extracted to a separate algorithm - Algorithm 5. For each child we will need a new named template in the *xslt:stylesheet* element (lines 7 - 13) which creates a "reference" XML element in the *rdf:Descritpion*, representing a PSM association and therefore XML nesting relation. We need to call the template from an *xsl:for-each* construct (line 3) to process all the XML elements corresponding to the child PSM class. Finally, we call the templates corresponding to the child PSM classes from outside the *rdf:Description* element (line 16) so that we avoid nesting, because in RDF/XML, all the *rdf:Description* elements are on the first level directly in the *rdf:RDF* root element.

Algorithm 4 Lifting XSLT generation

```
 1: Generate a template matching the XML document root element:
 2: <xsl:template match="/">
 3:    <xsl:for-each select="$xml'(C_r)$">
 4:       <xsl:call-template name="$TNCT(C_r)$"/>
 5:    </xsl:for-each>
 6: </xsl:template>
 7: for all PSM classes $C$ do
 8:    Create a named template $T$ corresponding to $C$:
 9:    <xsl:template name="$TNCT(C)$">
10:       <rdf:Description/>
11:    </xsl:template>
12:    Inside the <rdf:Description/> element add:
13:    <xsl:attribute name="rdf:about">
14:       <xsl:value-of select="concat($oE(I(C))$, @id)">
15:    </xsl:attribute>
16:    <rdfs:Class>
17:       <xsl:attribute name="rdf:resource">
18:          <xsl:text>$oE(I(C))$</xsl:text>
19:       </xsl:attribute>
20:    </rdfs:Class/>
21:    for all PSM Attributes $A$ of $C$ do
22:       <xsl:element name="$IDtoXML(oE(I(A)))$">
23:          <xsl:value-of select="@ + $xml'(A)$" />
24:       </xsl:element>
25:    end for
26:    for all Children $H$ of $C$ do
27:       Call Algorithm 5 to process current child
28:    end for
29: end for
```

Algorithm 5 Lifting XSLT generation - Child processing

```
 1: Call an additional template handling "nesting" relations - has e.g. "ref" at the
    end of its name
 2: <xsl:for-each select="$xml'(H)$">
 3:    <xsl:call-template name="$TNCT(H)\ +\ ref$"/>
 4: </xsl:for-each>
 5: Create the additional template in the <xsl:stylesheet> element:
 6: $R = PSM\ association\ connecting\ C\ and\ H$
 7: <xsl:template name="$TNCT(H)\ +\ ref$">
 8:    <xsl:element name="$IDtoXML(oE(I(R)))$">
 9:       <xsl:attribute name="rdf:resource">
10:          <xsl:value-of select="concat($oE(I(H))$, @id)"/>
11:       </xsl:attribute>
12:    </xsl:element>
13: </xsl:template>
14: Call templates corresponding to children (add to $T$ after <rdf:Description/>):

15: <xsl:for-each select="$xml'(H)$">
16:    <xsl:call-template name="$TNCT(H)$"/>
17: </xsl:for-each>
```

13.4.2 Lowering XSLT

A *lowering XSLT stylesheet* allows us to transform the (possibly) modified RDF data back to its XML representation described by a PSM schema (Algorithm 6).

Formally, we search for the *rdf:Description* representing a root element (line 1) like this: Let C_{psm} be the root PSM class and C_{pim} the PIM class represented by C_{psm}, such that $C_{pim} = I(C_{psm})$. Then we search for an element *rdf:Description*, which has a subelement *rdfs:Class* pointing to $ontologyEquivalent(C_{pim})$. This ensures type consistency. Also, this element cannot be referenced from any *rdf:resource* attribute so it is a root.

Algorithm 6 Lowering XSLT algorithm

```
 1: $D$ = The rdf:Description element corresponding to a root element
 2: Start of recursion - $D$ is current rdf:Description element
 3: $C = PSM\ class\ corresponding\ to\ D$
 4: Create an XML element named $xml'(C)$
 5: for all XML sub-elements $E$ of $D$ corresponding to PSM attributes $A$ do
 6:    Create an XML attribute named $xml'(A)$, set its value to the content of $E$
 7: end for
 8: for all XML sub-elements $F$ of $D$ corresponding to PSM associations $R$ do
 9:    Process (recursively) the rdf:Description element, whose rdf:about attribute
       value is equal to the value of the rdf:resource attribute of $F$
10: end for
```

Now we will describe the algorithm that creates the lowering XSLT script from a PSM schema (Algorithm 7). First we generate the template matching the root *rdf:RDF* element. The XPath expression that finds the *rdf:Description*

Algorithm 7 Lowering XSLT generation

```
1: Generate a template matching the rdf:RDF root element:
2: XPath = "rdf:Description[descendant::rdfs:Class
3:     [@rdf:resource=$oE(I(C_r))$]]
4:     and not(@rdf:about=//@rdf:resource)]/@rdf:about"
5: <xsl:template match="/rdf:RDF"/>
6:   <xsl:call-template name="$TNCT(C_r)$">
7:     <xsl:with-param name="id" select="XPath"/>
8:   </xsl:call-template>
9: </xsl:template>
10: for all PSM classes $C$ do
11:   Generate a template $T$ matching an rdf:Description element corresponding
      to an instance of $C$:
12:   XPath = "//rdf:Description[@rdf:about=$id]"
13:   <xsl:template name="$TNCT(C)$">
14:     <xsl:param name="id"/>
15:     <xsl:for-each select="XPath">
16:       <$xml'(C)$/> {$E$ = this element}
17:     </xsl:for-each>
18:   </xsl:template>
19:   for all PSM attributes $A$ of $C$ do
20:     To $E$ add:
21:     <xsl:attribute name="$xml'(A)$">
22:       <xsl:value-of select="$oE(I(A))$"/>
23:     </xsl:attribute>
24:   end for
25:   for all Children $H$ of $C$ do
26:     $R$ = PSM association connecting $C$ and $H$
27:     To $E$ add:
28:     <xsl:for-each select="$IDtoXML(oE(I(R)))$">
29:       <xsl:call-template name="$TNCT(H)$">
30:         </xsl:with-param
                  name="id" select="@rdf:resource"/>
31:       </xsl:call-template>
32:     </xsl:for-each>
33:   end for
34: end for
```

element representing the root (lines 2 - 4) is a bit complicated. We want an ID contained within the *rdf:about* attribute of the correct *rdf:Description*, which is to be transformed to the root XML element. That is the first and last part of the XPath expression. The middle part says that we want *rdf:Description* corresponding to *ontologyEquivalent*$(I(C_r))$ where C_r is the root PSM class and $I(C_r)$ is the PIM class represented by C_r in the PSM schema. Once we have the query that gets us the ID, we call the appropriate template with the ID as a parameter (lines 6 - 8).

We process each PSM class and create corresponding named templates (lines 10 - 34). For each PSM class C we create a template accepting an ID parameter and creating an element named $xml'(C)$ (lines 11 - 18). Next we process PSM attributes A_i of C, adding an XML attribute named $xml'(A_i)$ with a value set to a content of corresponding XML element in the current *rdf:Description* (lines 19 - 24). Finally, we recursively process the child PSM classes and their *rdf:Description* elements (lines 25 - 33) by calling the template corresponding to a child PSM class in the *xsl:for-each* construct.

13.4.3 SAWSDL extension to XML schema

With the lowering and lifting stylesheets created, there is one last thing to do for the SAWSDL standard compatibility. We add a reference to the description of the root XML element in a form of three attributes: *sawsdl:modelReference* contains an ID of the OWL class corresponding to the root XML element, *sawsdl:loweringSchemaMapping* and *sawsdl:liftingSchemaMapping* indicate the URLs of the lowering and lifiting XSLT stylesheets, respectively. These attributes can be added to the XML schema that can be generated from a PSM schema automatically.

13.5 Implementation

The proposed approach has been implemented in our tool called XCase [54]. It implements all the algorithms from this chapter, the mapping of a PIM schema to an ontology, export of a PIM schema to an ontology (Section 13.3.2), import of an OWL Lite ontology into a PIM schema (Section 13.3.1) and also the generation of lifting and lowering XSLT scripts from PSM schemas (Sections 13.4.1 and 13.4.2). This is in addition to its already implemented and used features like modeling XML data using the conceptual model, exporting PSM schemas to the XML Schema language an more. The implementation is on an experimental level, nevertheless, it is clear that the methods proposed in this chapter are useful, as they make the process of creating the lifting and lowering XSLT quick and easy in contrast to manual creation. This is provided that the user uses XCase and our conceptual model for the management of XML schemas. XCase can be found at `http://xcase.codeplex.com`.

13.6 Related Work

In our previous work we have discussed the possibility of automatic generation of lifting and lowering XSLT schema mappings using a conceptual model. In this chapter we show in detail the actual implementation of the idea, incorporating it to our tool XCase. In [64], a survey of possible approaches to data grounding is given. There it is stated that the creation of lifting and lowering XSLT scripts can be difficult. Our approach automates the process by generating the scripts from our conceptual model, which should be used anyway for the management of XML schemas. In [65], the same author surveys SAWSDL details and applications in a very well-arranged way. In [109], there is a comparison between SAWSDL and OWL-S, which is another language for semantic annotations for WSDL.

In [79], an approach similar to ours is proposed and others are surveyed. The authors suggest a mapping between an XML schema and a WSMO Ontology to be created and expressed in WSML. From there, they suggest that XML data should be translated to WSML instances. This is a similar idea as ours, however, they lack the strong background of a working framework to which this approach could be integrated. In addition, their mappings are between an ontology and XML schema, which can be viewed as our PSM schema. Our advantage is that we map our PIM schema to the ontology and therefore we can create one mapping for use in all our XML schemas, instead of mapping N schemas to one ontology. Also, we present an actual implementation that can generate the mappings.

13.7 Conclusions

In this chapter we have briefly summarized our conceptual model for XML and extended it slightly to support mapping to ontologies. We proposed a method for automatic generation of *lifting* and *lowering* XSLT stylesheets using this model. The method is to be used to ease the management of semantic web services and the experimental implementation is available in our tool, XCase [54].

The need for our conceptual model being present for our approach to be automatic is not so limiting, because we suggest our conceptual model should be used for XML schema management anyway and in this case the generation of the stylesheets is easy. We did not tackle the problem of inheritance in this chapter as it would make the description of our conceptual model more confusing and it is not necessary for our contribution.

14. Using Schematron as Schema Language in Conceptual Modeling for XML

In our previous work, we introduced a conceptual model for XML, which utilizes modeling, evolution and maintenance of a set of XML schemas and allows exporting modeled formats into grammar-based XML schema languages like DTD and XML Schema. However, there is another type of XML schema languages called rule-based languages with Schematron as their representative. Expressing XML schemas in Schematron has advantages over grammar-based languages and in this paper, we identify the advantages and we propose a method for easier creation and maintenance of Schematron schemas using our conceptual model. Also, we discuss the possibilities and limitations of translation from our grammar-based conceptual model to the rule-based Schematron.

This chapter is based on a supervised master thesis by Soběslav Benda.

Contents of this chapter was published as a conference paper called *Using Schematron as Schema Language in Conceptual Modeling for XML* [15] in The Ninth Asia-Pacific Conference on Conceptual Modelling (APCCM 2013) that was awarded the *Best Student Paper Award*.

14.1 Introduction

The standardized schema languages are Document Type Definition (DTD), W3C XML Schema and REgular LAnguage for XML Next Generation (Relax NG). These languages have differences in some features (e.g. expressive power, syntax complexity, object-oriented design, etc). A common feature of these languages is their formal background which is a Regular Tree Grammar (RTG) [81]. RTG determines the maximum expressive power and gives instructions for the construction of validators. Commonly, we call these languages *grammar-based schema languages* or *grammars* for short.

However, it is possible to express XML schemas in other languages that are not based on RTG. An example of such language is an also standardized Schematron [49]. Briefly, Schematron allows describing schemas using XPath conditions, that are evaluated over a given XML document during validation. This brings interesting possibilities for the validation of XML documents.

Motivation In our previous work [96, 89], we developed a methodology for modeling, evolution and maintenance of XML schemas using a multilevel conceptual model based on Model Driven Architecture (MDA) [78] and we introduced many extensions [95, 61] and described several use cases [88] of our approach. So far, we have only supported grammar–based XML schema languages, because of their popularity due to understandable declarations and efficient validation. While it is true that for relatively simple schemas DTD will do and for more complex structures XML Schema will provide the necessary constructs, there are also drawbacks to these widely used languages. For example, when we validate documents using DTD or XML Schema, we usually get a simple *valid/invalid* statement as a result. In the more interesting case of invalidity, the validators usually return a built-in error message, which is often incomprehensible, misleading and does not provide means for quality diagnostics [84]. In addition, it is often not possible (or user-friendly) to pass them directly to the user interface. Regarding this diagnostic problem, Schematron schema can help. Schematron is often described as a language for description of integrity constraints [81], but it is more than that. Using Schematron, it is possible to describe most constraints that can be expressed by grammars. Moreover, it is possible to describe many additional details and even structural constraints that we can not express using grammar-like languages like XML Schema. In [50], the authors identify the demand for Schematron-based solutions for XML schema management, which is another motivation for adding support for Schematron to our conceptual model. Finally, when combined with the approach to express integrity constraints in the conceptual model [72], Schematron becomes a unified schema language for description of the structure and integrity constraints of XML documents and a framework for detailed diagnostics and error reporting. These advantages of using Schematron outweigh its main disadvantage, which is its verbosity and complexity, because it can be eased by the usage of our conceptual model for schema management. In this chapter, we consider our conceptual model for XML as introduced in Chapter 6.

Outline The chapter is organized as follows: In Section 14.2, we introduce the Schematron language. Section 14.3 contains the main contribution of this paper, the translation from the conceptual model to Schematron schemas. In Section 14.4 we discuss related work, in Section 14.5 we evaluate our approach and we conclude in Section 14.6.

14.2 Schematron

Schematron is a declarative language which is a representant of the *rule-based XML schema languages*. These languages are not based on construction of a grammatical infrastructure. Instead, they use rules resembling if-then-else statements to describe constraints. These languages offer the finest granularity of control over the format of the documents [130]. We can even view constructs of other schema languages as a syntactical sugar used instead of sets of rule-based conditions. Schematron was designed in 1999 by Rick Jelliffe. The language was standardized in 2005 as ISO Schematron [49].

Example 14.1 *Consider a specification of a complex element using the following DTD declaration `<!ELEMENT purchase (item+,customer?)>`. In Schematron, it is possible to describe the same semantics and cover valid instances using multiple intuitive conditions, for example: If `purchase` element exists, the element can only have an `item` and a `customer` elements as children. The `item` element has at least one occurrence and the `customer` element has zero or one occurrence. If the `customer` element exists as a child element of a `purchase` element, then the `customer` element has no following sibling elements.*

Schematron is not a standalone language. It is a general framework which allows schema designers to organize conditions which are evaluated over the given documents. These conditions are described using an *underlying XML query language* such as the default XPath. A result of a validation is a report which contains information about evaluation of these conditions. Schematron is an XML-based language and uses only few elements and attributes for schema description.

14.2.1 Core constructs

Now we describe core Schematron constructs. The root element of every schema is a `schema` element introducing the required XML namespace[1]. A `pattern` element is a basic building stone for expressing an ordered collection of Schematron conditions which are ordered in XML document order. A *rule* is a Schematron condition which allows a designer to specify a selection of nodes from a given document and evaluation of predicates in the context of these nodes. The `rule` element has a required `context` attribute used for an expression in the underlying query language. The value of the `context` attribute is commonly called a *path*. Predicates are specified using a collection of assertions. An assertion is a predicate which can be positive or negative. An assertion is represented using the `assert` and `report` elements. Both elements have a required `test` attribute for specification of a predicate using the underlying query language. Both elements also have a text content called *natural-assertion*. Natural-assertion is a message in a natural language, which a validator can return in the validation report. A positive predicate is represented using an `assert` element and if it is evaluated as false, we say that the assert is violated and the document is invalid. A negative predicate is represented using a `report` element and if it is evaluated as true, we say that the report is active and a natural-assertion will be reported. Schematron is not only a validation language. It is a more general *XML reporting language* [103] where one type of report is an error message.

Example 14.2 *A pattern in Figure 14.1 selects all* ***`triangle`*** *elements from a document. In a context of every* ***`triangle`*** *element a positive predicate specified with expression* ***`count(vertex)=3`*** *is evaluated. If the given* ***`triangle`*** *has for*

```
<pattern>
  <rule context="triangle">
    <assert test="count(vertex)=3">
      The element ’triangle’ should
      have 3 ’vertex’ elements.
    </assert>
  </rule>
</pattern>
```

Figure 14.1: Schematron pattern

example four child ***`vertex`*** *elements, then the predicate will be false and the following message will be reported:* The element ’triangle’ should have 3 ’vertex’ elements.

[1]`http://purl.oclc.org/dsdl/schematron`

14.2.2 Additional constructs

In addition to the core Schematron constructs we describe other constructs used in practical applications. Schematron allows using metadata for introduced constructs. *Identifiers* allow identification of a pattern inside a schema, a rule inside the pattern, etc. It is represented using an `id` attribute. A `role` attribute can be used to assign special semantics to Schematron constructs. A *diagnostic* is a natural-language message giving details about a failed assertion, such as found versus expected values and repair hints. It is represented using a `diagnostic` element with required `id` attribute and text content with a message. Diagnostics are referenced by assertions using a `diagnostics` attribute. We can use *substitutions* in natural-assertions for clearer result in validation reports. An element `name` is substituted by the name of a context node. An element `value-of` is substituted by a value found or calculated using an expression in the required `select` attribute. *Abstract rules* provide a mechanism for reducing schema size. An abstract rule can be invoked by other rules belonging to the same pattern. Variables are substituted in assertion tests and other expressions before the expression is evaluated. *Phases* allow to organize patterns into identified parts. Every Schematron schema has one default phase which includes all patterns. Before validation, it can be determined which phase is used and which patterns are activated. This selected phase is called an *active-phase*. A phase is represented using a `phase` element with an `id` attribute. One phase can have multiple `active` elements which refer to patterns using a `pattern` attribute. A *variable* is represented using a `let` element. If the variable is a child of a `rule`, the variable is calculated in scope of the current rule and context. Otherwise, the variable is calculated within the context of the instance document root. *Abstract patterns* allow a common definition mechanism for structures which use different names and paths, but which are the same otherwise.

14.2.3 Schematron implementations

An implementation of Schematron validation is very simple in general, because it is based on already implemented XML technologies. There are two kinds of Schematron validation.

XSLT validation

For this approach, we only need an XSLT processor and a predefined XSLT script[2]. The script translates the given Schematron schema to another XSLT script which is used for the actual XML document validation. During the validation, a given XML document is transformed into another XML document. This document is the result of the validation and may be formatted using standard Schematron Validation Report Language (SVRL) [49], which provides rich information about the validation process, e.g. XPaths for elements which violated assertions.

[2]`http://www.schematron.com/tmp/iso-schematron-xslt1.zip`

Special libraries

Another approach is to use a special (platform-dependent) library. Some libraries[3] only wrap the described XSLT validation. However, there are other implementations not based on XSLT. These libraries are based on the evaluation of XPath expressions. This allows a programmer to adapt the validation for special requirements or possibilities of a target platform. For example, we have implemented a C# validator called *SchemaTron* [16] providing excellent performance for XML content-based message routing inside an intermediate service.

14.2.4 Schematron properties

In this section, we describe selected properties of Schematron schemas in the context of conceptual modeling of XML and compare them with grammar-based schemas. We mostly consider XML Schema 1.0 as their representative because we already support it in our conceptual model for XML.

Platform independence

Schematron is based on standard XML technologies, which are commonly implemented in many software environments, e.g. XSLT processor is natively implemented in standard web browsers. For these reasons, we can see Schematron as a platform independent XML schema language, because we do not need a specific validator.

Expressive power

Presently, there is no formal framework [66] which could capture the broad set of possibilities of Schematron conditions.

The authors of [67] provide some basic expressive possibilities of Schematron, compare it with other schema languages and show by example that Schematron has an excellent expressive power and in this regard it is (e.g. with XSD) a first class XML schema language. The authors describe, that we can specify for example: parent-child relationships, sequences, choices among elements and attributes, unordered sets, min and max occurences, etc. Moreover, we can specify many XML formats, which can not be expressed using grammars, for example conditional definitions (e.g. a presence of elements or attributes is dependent on a value of another element or attribute) or detailed integrity constraints, etc.

However, there is not any precise generalization of Schematron rules which would provide a clever mapping of regular tree grammar into Schematron rules (and vice versa), but experiments show [50] that it is possible to describe many instances of grammars in Schematron, even in diverse ways.

14.3 PSM to Schematron translation

A PSM schema models a grammar-based XML format specification and its concepts are interpreted against PIM concepts. There are several theoretical and

[3] `http://www.probatron.org/`

practical problems that we must consider when we want to describe the translation of a PSM schema to a Schematron schema. In particular, we need to identify groups of Schematron rules that impose equivalent constraints on the documents as constructs of grammar-based languages would.

14.3.1 Overall view of the translation

The translation algorithm (see Algorithm 8) is fully automatic. It has a PSM schema on the input and it gradually builds a Schematron schema on the output. The generated schema covers grammatical structural constraints normally expressed in, for example, XSD.

Algorithm 8 Overall view of the translation algorithm

```
1: <schema xmlns="http://purl.oclc.org /dsdl/schematron">
2:   Generate allowed root element names (Section 14.3.2);
3:   Generate allowed names (Section 14.3.3);
4:   Generate allowed contexts (Section 14.3.4);
5:   Generate required structural constraints (Section 14.3.5);
6:   Generate required sibling relationships (Section 14.3.6);
7:   Generate required text restrictions (Section 14.3.7);
8: </schema>
```

In the first step (line 2), we generate Schematron patterns for XML elements, which are allowed inside valid XML documents as root elements. Similarly, we generate patterns for allowed names of XML elements and XML attributes in the next step on lines 3. On line 4, we produce patterns for allowed contexts, i.e. paths where certain names of elements (and attributes) may occur. We call the generated patterns *absorbing patterns* and we describe them later. The patterns for validation of required complex element structures are produced in the steps on lines 5 and 6. These patterns are more complex, because we must generate an equivalent of regular expressions to obtain the semantics of regular grammars. We call these patterns *conditional patterns* and we describe them later in this section. In the last step (line 7), the patterns for text restrictions, i.e. validation of attribute values and simple element contents, are produced.

14.3.2 Allowed root element names

We need a tool for reporting names of elements which are not allowed in the schema, but are present in the document.

Definition 14.1 An absorbing pattern *is a Schematron pattern for an ordered set of paths P, where the last rule is called* global *and it contains the $*$ wildcard somewhere in its path and no previous rules use wildcards.*

In this definition we defined a special kind of a Schematron pattern which we call an *absorbing* pattern and which allows Schematron to absorb elements (or attributes) specified by paths. It checks for all the allowed elements or attributes in the path and if none of them is found, it matches whatever is found in the path using a wildcard (absorbs it), so that the validation can continue. If the element or attribute is absorbed by the wildcard, it is a violation of the expected format and the element or attribute absorbed by the wildcard rule is reported. In the first step on line 2 in the overall Algorithm 8, we generate an absorbing pattern for checking allowed root elements an the intuitive way described here.

Example 14.3 *As an example, consider the set of paths P, which contains paths for all allowed root elements `/request` and `/response`. We generate the absorbing pattern that is in Figure 14.2. When the validated document has `request` or*

```
<pattern id="allowed-root-elements"
         role="absorbing-pattern">
  <rule context="/request">
    <assert test="true()"/>
  </rule>
  <rule context="/response">
    <assert test="true()"/>
  </rule>
  <rule context="/*">
    <assert test="false()">
      The element '<name/>' is
      not declared in the schema
      as a root element.
    </assert>
  </rule>
</pattern>
```

Figure 14.2: Absorbing pattern example

`response` element as a root, the element is absorbed and the validation continues. When the document has for example an `x` root element, it is absorbed by the wildcard part of the pattern, which means that the document is invalid and the following message is reported: The element 'x' is not declared in the schema as a root element. *However, the validation still continues, which is in contrast to XSD validation, which would end at this point.*

14.3.3 Allowed names

The approach is very similar to generation of allowed root element names, because we also generate absorbing patterns. Patterns for allowed attributes are also similar, so we skip them in this paper. Production of patterns for checking allowed XML elements inside validated documents follows this algorithm: We produce the set P of all paths for allowed element names. For complex elements, we get them from the names of named associations which have classes as children in the PSM schema. For simple elements, we get the names from PSM attributes A' with XML form set to $xform(A') = \texttt{e}$. From the paths and names, an absorbing pattern is generated.

14.3.4 Allowed contexts

Now we introduce stricter patterns for checking allowed contexts, i.e. paths inside documents. We also generate absorbing patterns, but we need more sophisticated paths, because we absorb only element and attribute names in the declared contexts, so the other names (contexts) break validity.

Paths overview A path is described using an XPath expression to select some nodes from the validated XML document. When nodes are selected, we can evaluate assertions, i.e. certain XPath predicates in the context of these nodes. In general, we have two approaches to how we can describe paths, i.e. *absolute paths*, for example `/book/author/name` or *relative paths* for example `name`. If we want to design schemas more powerful than DTD, i.e. local regular tree grammars [81], we need absolute paths to select nodes from documents. However, relative paths are also important for example to design recursive declarations. There is also a possibility to use predicates in paths. We do not deal with predicates for nodes selection, because we aim to design a Schematron schema as simple as possible. For this purpose, we impose a *SORE precondition* on our PSM schemas in Definition 14.2. Every SORE is deterministic as required by the XML specification and more than 99% of the regular expressions in practical schemas are SOREs [19], so the precondition does not limit us much and at the same time simplifies the translation a lot. For instance, `((a|b),c 0..*,d 0..1) 0..3` is SORE while `a(a|b) 0..*` is not as `a` occurs twice.

Definition 14.2 *Let S' be a PSM schema. We will call* SORE precondition *an assumption on S', that every complex element has content described using Single Occurrence Regular Expression, i.e. every element (or attribute) name can occur at most once in this regular expression.*

Paths construction Here we describe the construction of paths for a PSM schema. The main idea is as follows. For each XML element and XML attribute declaration present in a PSM schema, we produce all possible paths (contexts) where they can occur. Every created path is associated with a PSM component, i.e. a complex element, a simple element or an attribute declaration and the pairs are placed into the global set of paths G_p. In the next step, we perform sorting of G_p members. The resulting ordered set $G_p = \{(X', p);\ X' \in (S'_r \cup S'_a)$ and p is path$\}$ is used for generation of Schematron rules in the order of this set in the rest of the translation. We sort members of G_p using the following ordering: (1) The absolute paths without recursions go first (2) The absolute paths with recursions follow, the longest path is the first one (3) The relative paths go last, again, the longest path is the first one to go.

Firstly, we need to create all paths for a given XML element or XML attribute declaration. Let us mark the declaration – the given PSM component $X' \in (S'_a \cup S'_r)$. We build an ancestor tree for X' which represents all achievable ancestor PSM components of X' in the PSM schema. Then we can translate all its paths from leaf nodes to root node into Schematron paths, i.e. XPath expressions. For each $(X', p) \in G_p$ must hold that p is unique, which corresponds to the SORE precondition in Definition 14.2.

Pattern for allowed element contexts Now we can produce patterns for allowed contexts. We go through all members of the ordered set G_p and produce set of paths P only for complex element names and simple element names (PSM attributes with XML form set to `element`). In the last step we produce an absorbing pattern for P with `*`. Similarly, we produce a pattern for allowed attribute contexts.

14.3.5 Required structural constraints

Now we have absorbing patterns for weak validation of XML documents generated. These patterns say what is allowed inside the documents. Now we deal with restrictions which say what the given document must satisfy.

Conditional pattern First of all, we specify another Schematron pattern, which we call conditional pattern (see Definition 14.3).

Definition 14.3 A conditional pattern *is a Schematron pattern for a set of pairs* $E = \{(p, A)$*; where* p *is a path and* A *is a set of predicates*$\}$*. It consists of several rules and the document passes validation by this pattern only if all the rules are satisfied.*

Example 14.4 *Consider a PSM schema where the root element* `customer` *must have a* `ship-to` *child element which must have a* `street` *child element and the* `street` *must not have any child elements. We generate a conditional pattern, which is in Figure 14.3. The pattern resembles a collection of if-then conditions,*

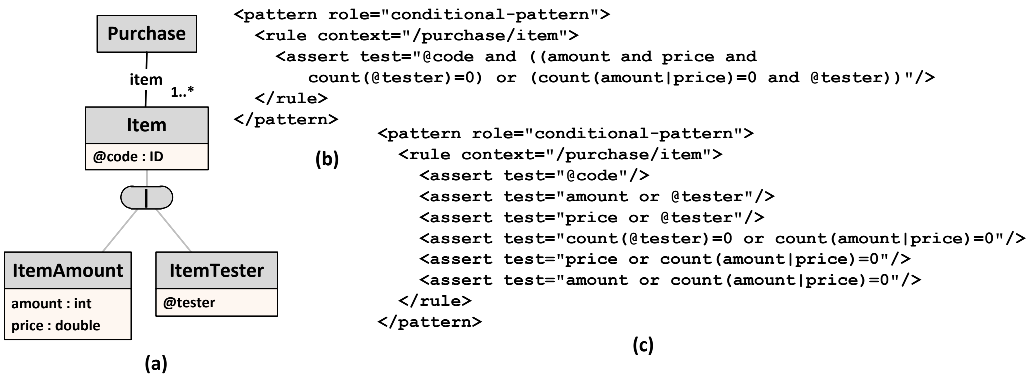
```
<pattern role="conditional-pattern">
  <rule context="/customer">
    <assert test="count(ship-to)=1"/>
  </rule>
  <rule context="/customer/ship-to">
    <assert test="count(street)=1"/>
  </rule>
  <rule
   context="/customer/ship-to/street">
    <assert test="count(*)=0"/>
  </rule>
</pattern>
```

Figure 14.3: Conditional pattern example

because it says: If the `customer` *element exists in the document as a root, it must hold that it has a* `ship-to` *child element. If the* `ship-to` *element exists in the document as a child of the* `customer` *element, it must hold that it has a* `street` *child element, etc. The example demonstrates, that we can create such a pattern with chained rules. If we wanted to describe this pattern using XML Schema, it would be:*

```
<schema>
 <element name="customer">
  <complexType>
   <sequence>
    <element name="ship-to">
```

```
     <complexType>
      <sequence>
       <element name="street"/>
      </sequence>
     </complexType>
    </element>
   </sequence>
  </complexType>
 </element>
</schema>
```



```
<pattern role="conditional-pattern">
  <rule context="/purchase/item">
    <assert test="@code and ((amount and price and
        count(@tester)=0) or (count(amount|price)=0 and @tester))"/>
  </rule>
</pattern>
```

```
<pattern role="conditional-pattern">
  <rule context="/purchase/item">
    <assert test="@code"/>
    <assert test="amount or @tester"/>
    <assert test="price or @tester"/>
    <assert test="count(@tester)=0 or count(amount|price)=0"/>
    <assert test="price or count(amount|price)=0"/>
    <assert test="amount or count(amount|price)=0"/>
  </rule>
</pattern>
```

Figure 14.4: Example of element `item`

For the production of conditional patterns, we need to analyze specifications of complex element contents. The complex element declared in a PSM schema is precisely specified using a regular expression, so we need to analyze such regular expressions and translate them into Schematron predicates. The main idea is as follows. We translate the regular expression into several conditional patterns. These patterns cover the same semantics as the regular expression, when they are evaluated together.

We generate two conditional patterns for checking structural constraints as a part of Algorithm 8 in the step on line 5. One of the generated patterns checks required parent-child relationships, the other pattern checks required parent-attribute relationships and other relationships of attributes and elements (predicates for choices among attributes and elements). There can be also other distributions of conditions into patterns. For example, everything may be inside one pattern, but we believe that our distribution provides flexible solution, because we can check required elements only, required attributes only, etc. In the algorithm, we use translations of regular expressions into boolean expressions and then normalization of boolean expressions into conjunctive normal form (CNF). Due to lack of space and because these are quite standard transformations, we do not describe them here in detail.

Algorithm 9 Generate patterns for structural constrains

```
1: Let $E_1$ be an empty set of pairs $(p, A_e)$;
2: Let $E_2$ be an empty set of pairs $(p, A_a)$;
3: for all $(X', p) \in G_p$ do
4:    if $X' \in S'_r$ then
5:       Let $A_e$ be an empty set of predicates;
6:       Let $A_a$ be an empty set of predicates;
7:       for all $Y \in cnf(be'(X'))$ do
8:          if $Y$ has only elements in its literals then
9:             Add $Y$ into $A_e$;
10:         else
11:            Add $Y$ into $A_a$;
12:         end if
13:      end for
14:      if $A_e$ is not empty then
15:         Add $(p, A_e)$ into $E_1$;
16:      end if
17:      if $A_a$ is not empty then
18:         Add $(p, A_a)$ into $E_2$;
19:      end if
20:   end if
21: end for
22: Generate conditional pattern for $E_1$;
23: Generate conditional pattern for $E_2$;
```

Let us now take a look at Algorithm 9 for production of patterns for structural constraints based on boolean expressions. Firstly, we initialize two empty sets of pairs (p, A_e) (line 1) and (p, A_a) (line 2), where p is a path and A_e is a set of associated predicates for elements, A_a is a set of associated predicates for attributes and relations with elements. Then, we go through pairs $(X', p) \in G_p$ and when X' is a complex element declaration, we initialize new sets A_e and A_a (lines 5 and 6) and translate X' into boolean expression and the boolean expression into conjunctive normal form (line 7). Then, we go through obtained predicates. When a predicate (marked as Y) has only elements in its literals we add it into A_e, else we add it into A_a. Then, if A_e or A_a is not empty, we add a pair (p, A_e) or (p, A_a) into E_1 (line 15), or E_2 (line 18), respectively. In the last step (lines 22 and 23), we generate conditional patterns for E_1, E_2. We have created two patterns, which cover certain structural constrains of modeled complex contents.

Example 14.5 *As an example, consider a regular expression which specifies the complex element* `item` *in Figure 14.4(a):* `(@code,(amount,price)|@tester)`*. We translate it as* `@code and ((amount and price and count(@tester)=0) or (count(amount|price)=0 and @tester))`*, an XPath predicate that we use in Schematron assertion in Figure 14.4(b). This representation is quite straightforward and corresponds well with grammar-based languages like XML Schema. However, it also comes with disadvantages in the form of poor diagnostics. As with XML Schema validation, when we would validate a document using the Schematron rule from Figure 14.4(b), we would only get a valid or invalid statement* *without further details. For this purpose, it is more advantageous to go into more detail and write the same rule as multiple simpler rules. We transform the regular expression, which is a logic formula, to a conjunctive normal form as seen in Figure 14.4(c). Now, we can create user-friendly diagnostics for each of these rules. Note that this is also an example of choice between attributes, which is not possible in XML Schema but can be done using Schematron.*

14.3.6 Required sibling relationships

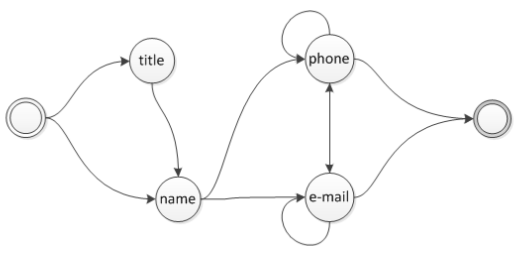
In the previous section we generated structural constraints using boolean expressions, which allow to validate parent-child relationships. So far we did not deal with the order of child elements inside a parent element. Here we describe our approach based on the theory of regular expressions. The main idea is as follows. We build a finite state automaton for a given regular expression. We deal only with SORE so we can build the deterministic SORE automaton, where every name of XML element is assigned to at most one inner state and it has one initial and one final state. Then we translate information obtained from this structure into Schematron conditions. We represent the transition function of the automaton using conditional patterns and we cover for example *the order of XML elements* (sequences, choices among elements) and also cardinalities *zero or one* (0..1, or ?), *just one* (1..1), *zero or more* (0..*, or Kleene star *), *one or more* (1..*, or Kleene cross +). We can also provide clear natural-assertions and diagnostics.

There are also some problems and exceptions. Firstly, we can not cover arbitrary numeric intervals of regular expressions using this approach (it is possible to create an automaton with numeric intervals, but it is not possible to represent it in Schematron). We need another approach for numeric constraints in general, which is not part of this paper. Secondly, a PSM content model SET complicates construction of the algorithm. The restriction (Definition 14.4) for content model SET is similar to restriction of XSD construct ALL.

Definition 14.4 *Let S' be a PSM schema. We introduce* SET precondition, *which is an assumption on S', that for each content model SET it must hold that it has named associations with classes as children in its content and the content model is descendant of associations $R' \in S'_r$, $(name'(R') = \lambda \vee child'(R') \notin S'_c)$ in the complex content, where $card'(R') = 0..1$ or $card'(R') = 1..1$.*

Now we can presume that we can build the automaton for each complex element declaration, i.e. named association with class as a child. We also need to translate obtained information into Schematron rules. For each complex element and for elements in its content we produce a set of predicates. These predicates are composed from `following-sibling` XPath axes. For each of the obtained predicates we generate a conditional pattern in the step on line 6 in Algorithm 8.

Example 14.6 *Consider content `(title?,name, (phone|e-mail)+)`. We can represent this regular expression using the SORE automaton in Figure 14.5(a). Then we generate Schematron rules (see Figure 14.5(b)) which represent the automaton in Schematron. Note that we use `F := following-sibling` substitution for code size reduction in this example. The first rule represents the initial*



(a)

```
<rule context="context">
      <assert test="*[1][self::title or
self::name]"/>
</rule>
<rule context="context/title">
      <assert test="F::*[1][self::name]"/>
</rule>
<rule context="context/name">
      <assert test="F::*[1][self::phone or
self::e-mail]"/>
</rule>
<rule context="context/phone">
      <assert test="F::*[1][self::phone or
self::e-mail] or not(F::*)"/>
</rule>
<rule context="context/e-mail">
      <assert test="F::*[1][self::phone or
self::e-mail] or not(F::*)"/>
</rule>
```

(b)

Figure 14.5: Example of an automaton in Schematron

state of the automaton and says what elements can be at the first position in the content. Other rules represent if-then conditions, i.e. if `title` *element exists, it has a* `name` *follower. If* `name` *element exists, it has a* `phone` *or an* `e-mail` *followers. If* `phone` *element exists, it has the* `phone` *element or the* `e-mail` *element followers or no following-sibling elements.*

14.3.7 Required text restrictions

In this section we show the final patterns of our proposed translation from a PSM schema of our conceptual model for XML to Schematron. They deal with validation of data types for simple element contents and attribute values. The supported set of data types for PSM attributes is implementation dependent, as we need the datatypes to be defined by the designer in Schematron. Our implementation, eXolutio, provides XSD built-in simple data types and we created the rules for their definition in Schematron, because Schematron does not provide built-in data types as default. We can, however, create many data types specifications using XPath expressions placed into abstract rules or patterns. Schematron over XPath 1.0, which we describe here, is worse than XSD in this practical aspect and we need to help ourselves by depending on the designer to define the used datatypes. Examples of these definitions are in Figures 14.6 - 14.10.

Definition 14.5 *Let $\mathcal{S}'$ be a PSM schema. We will call* data types precondition *an assumption on $\mathcal{S}'$, that each data type used in $\mathcal{S}'$ has corresponding declaration in Schematron provided by the designer or by the eXolutio tool.*

In the step on line 7 of Algorithm 8 we generate patterns for data types validation as extension rules of our predefined data type rules using the `<extend/>` element in the `<rule/>` element.

14.3.8 Translation summary

In this section, we introduced the problem of automatic construction of Schematron schemas from PSM schemas. The translation is not simple, because we have

```
<rule id="emptyString"
      abstract="true">
  <assert test="string-length
  (normalize-space(text()))=0"/>
</rule>
```

Figure 14.6: Empty string data type

```
<rule id="string" abstract="true">
  <let name="str"
       value="string(text())"/>
  <assert test="$str"/>
</rule>
```

Figure 14.7: String data type

```
<rule id="normalizedString"
      abstract="true">
  <extends rule="string"/>
  <let name="nstr"
       value="normalize-space($str)"/>
  <assert
       test="string-length($str)
       =string-length($nstr)"/>
</rule>
```

Figure 14.8: Normalized string data type

```
<rule id="boolean" abstract="true">
  <extends rule="string"/>
  <assert test="$str='true'
              or $str='false'"/>
</rule>
```

Figure 14.9: Boolean data type

different models - grammar-based PSM schema (and XML Schema, DTD, etc.) and the rule-based Schematron. However, we showed that Schematron is a very powerful language and it can express many grammatical structural constrains from the grammar-based languages and more.

We started with production of absorbing patterns, which allow to validate allowed occurrences of XML elements and XML attributes inside validated XML documents. Then we produced conditional patterns for validation of required grammatical structural constraints. We analyzed the most used parts of regular expressions which can be represented in Schematron. Then we generated patterns for validation of data types for simple element contents and attribute values.

There are some limitations to our approach that, however, do not seem critical at the moment. The most visible one is the lack of support for arbitrary numeric intervals in cardinalities. We only support the usual `0..*, 0..1, 1..*, 1..1`. This is because the support for arbitrary intervals would necessarily lead to Schematron code explosions which would only complicate and slow down the

```
<rule id="double" abstract="true">
  <let name="num" value="number
     (normalize-space(text()))"/>
  <assert test="$num"/>
</rule>
```

Figure 14.10: Double data type

validation process.

14.4 Related work

In parallel to the research of translation of PSM schemas to Schematron, other PSM schema improvements are also being researched. In particular the support for Object Constraint Language (OCL) [106] and its translation to Schematron for the specification of integrity constraints, where Schematron is used as a complement of grammar-based schemas. These patterns for integrity constraints generated from OCL may be potentially merged with our Schematron schemas. To our best knowledge, little work has been done in the area of translations between Schematron and other XML schema languages. There are sources not based on academic research which provide some basic ideas and techniques for translation of grammar-based schemas to Schematron schemas and vice versa. Most work in this area has been done by Rick Jelliffe and his company Topologi[4]. They have implemented an *XSD to Schematron converter*[5], because their customers preferred Schematron diagnostics over XSD validation. The generated schemas are called *Schematron-ish grammars*. In [96], we provide formal description of mutual translation between PSM schemas and regular tree grammars.

14.5 Evaluation and implementation

With our proposed method, we have generated several Schematron schemas in various data domains using our conceptual model. The schemas are verbose and cannot be shown here whole due to space limitations. Their structure is, however, shown in our examples throughout the paper. During our experiments, we found the Schematron based validation as easy to use from a domain expert's perspective as a validation using XML Schema would be given that both can be generated from our conceptual model for XML. The downside of Schematron mentioned in our motivation, which is its verbosity, is not a problem in the end because the user does not need to read the actual generated Schematron. He only needs to give it as an input to a Schematron validator. From the validation performance point of view, rule-based validation (e.g. Schematron) is computationally more expensive than the linear validation using grammar-based languages (such as XML Schema) [84]. This could be a problem in an environment that requires high performance validations, such as routing of XML messages. Nevertheless, when

[4]http://www.topologi.com/

[5]http://www.schematron.com/resource/XSD2SCH-2010-03-11.zip

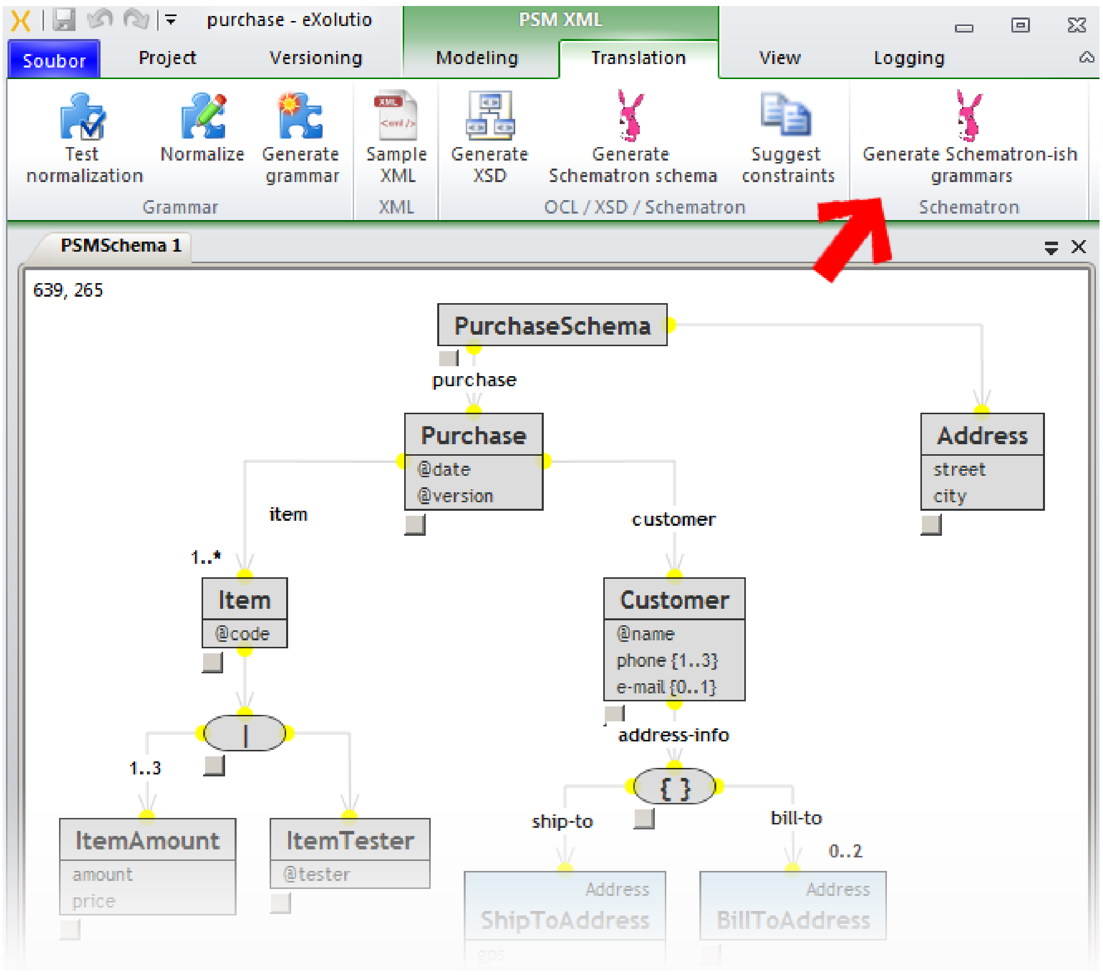
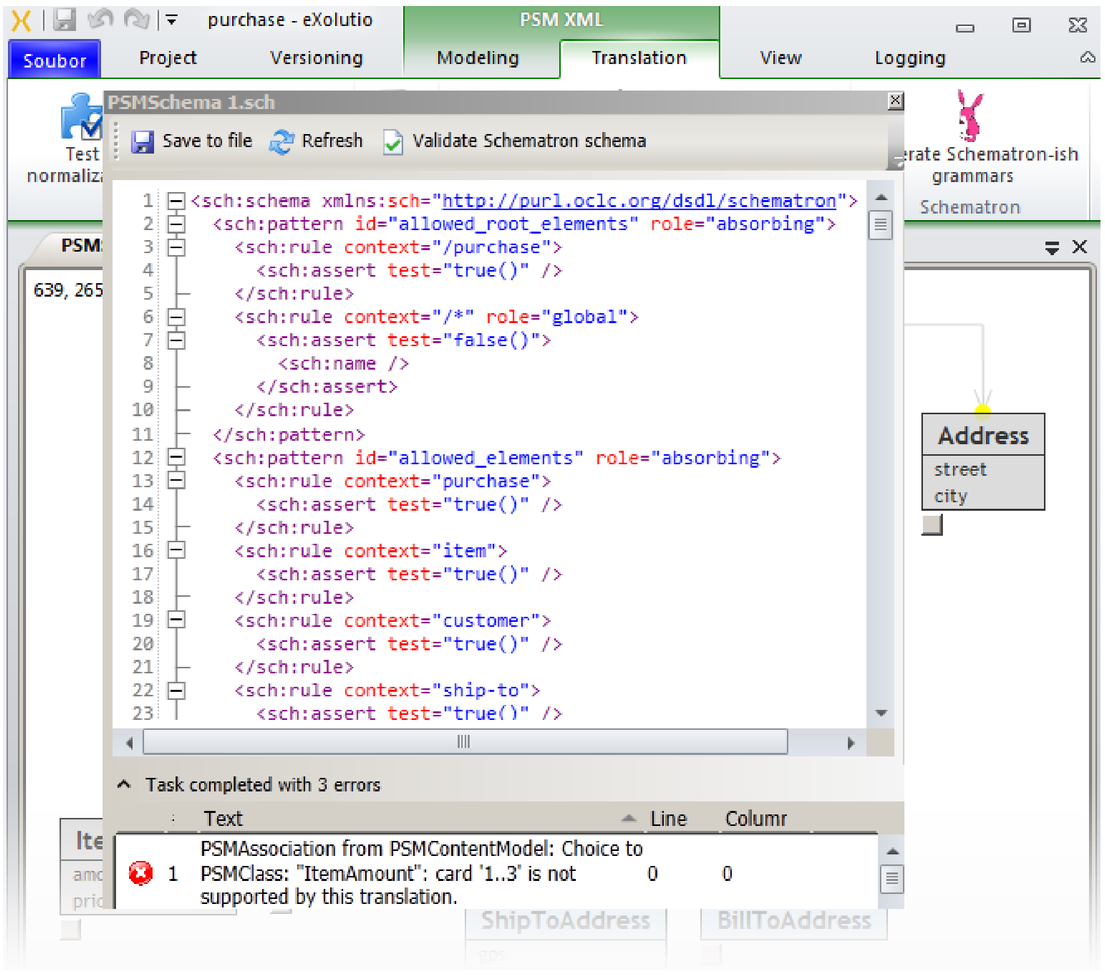


Figure 14.11: PSM schema in eXolutio

performance is not an issue or when validating against complex XML formats, the benefits in form of better diagnostics are more important.

The reward for using our approach is much easier diagnostics of a possible problem in the validated XML document because as we described in this paper, Schematron supports user-friendly and descriptive error messages. Also, its expressive power is greater than that of XML Schema, which can be seen in Figure 14.4, where we use a choice between attributes, which is not possible to express in XML Schema. Our experiments were done using our implementation of the conceptual model for XML, eXolutio.

eXolutio is an application developed in our research group. Its base is the formalism for our conceptual model for XML described in [96] and a complex system of operations and their propagation between the levels of abstraction described in [89]. In addition, it is a platform where novel extensions to XML schema modeling and evolution are implemented. One of them is the approach described in this paper. In Figure 14.11 there is a PSM schema modeled in eXolutio and in Figure 14.12 there is the generated Schematron schema.



```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:pattern id="allowed_root_elements" role="absorbing">
    <sch:rule context="/purchase">
      <sch:assert test="true()" />
    </sch:rule>
    <sch:rule context="/*" role="global">
      <sch:assert test="false()">
        <sch:name />
      </sch:assert>
    </sch:rule>
  </sch:pattern>
  <sch:pattern id="allowed_elements" role="absorbing">
    <sch:rule context="purchase">
      <sch:assert test="true()" />
    </sch:rule>
    <sch:rule context="item">
      <sch:assert test="true()" />
    </sch:rule>
    <sch:rule context="customer">
      <sch:assert test="true()" />
    </sch:rule>
    <sch:rule context="ship-to">
      <sch:assert test="true()" />
```

Task completed with 3 errors

	Text	Line	Columr
1	PSMAssociation from PSMContentModel: Choice to PSMClass: "ItemAmount": card '1..3' is not supported by this translation.	0	0

Figure 14.12: Schematron schema in eXolutio

14.6 Conclusions

In this chapter we introduced Schematron, a rule-based language that can be used for XML schema description, and its constructs. Next, we described how a schema from our conceptual model can be translated to Schematron and described the advantages over grammar-based languages such as XML Schema. We have evaluated our approach and described its implementation in our tool, eXolutio.

Conclusion and Open Problems

In this thesis we presented our contributions in the area of conceptual modeling for XML in the past years. We started with an overview published at a Ph.D. workshop of a major database conference EDBT/ICDT (Chapter 1). Then we presented our work in the area of XML schema integration (Chapter 2 - Chapter 5), where we focused on reverse-engineering of existing XML schemas to our conceptual model. These approaches are useful for users who do not model their problem domain using the conceptual model yet. Then we redefined the conceptual model for XML itself (Chapter 6) as we gained experience from implementing and extending the original one. Based on this formally redefined conceptual model, we also formally defined the operations necessary to keep the multi-level model consistent. We described how the simple, formally defined operations can be combined to create user-friendly operations that propagate correctly between the levels of the conceptual model (Chapter 7 - Chapter 10). We also showed how the conceptual model can be used to improve desing qualities of families of XML schemas (Chapter 11). We showed that our research contributions are also implemented in our tools XCase and eXolutio (Chapter 12), which correspond to the two main versions of the conceptual model for XML as they evolved during the past years. In addition, we showed how the conceptual model can be used to easily semantically enhance (lift) XML data to become RDF data and vice versa (Chapter 13). Finally, we showed how Schematron - a rule based language can be used in combination with the conceptual model for XML (Chapter 14).

Open Problems Since the area of XML evolution is relatively new, the number of current approaches and consequently solved issues is not high; there is a significant amount of open problems and future directions we want to focus on. The key areas involve:

- *Specifics of XML Schema Languages:* In [96], we show that the introduced PSM is equivalent to the formalism of regular tree grammars [81] which are considered as a basic formalism of XML schema languages. However, practical XML schema languages such as [128], introduce various other concepts (e.g. namespaces, user-defined simple data types, etc.) which we did not consider in this thesis.
- *Other Conceptual Modeling Constructs:* It is common in practice to use other modeling constructs (e.g. n-ary relationships, etc.).
- *Operational and Extensional Level of the Framework:* As we have described in Section 8.3, in this thesis we focused on a subpart of the proposed framework – data representation. So, naturally, there is the need to focus on those parts which were omitted, especially extensional and operational level. The extensional level is crucial for the applicability of our solution during system run-time. The operational level is also very important and has been mostly omitted in the current literature.
- *Modeling of Storage Strategies:* Similar to the previous point, there is need to focus on other parts of the framework which were omitted. An important

aspect that has so far not been much considered is the relation of change propagation and XML storage strategies. Currently there are approaches that deal with evolution of database schemas [32, 12], but in our case we have to consider a set of applications that form the system, the fact that the XML views of the data can and will overlap and exploitation of the relations between components of the framework. At the same time, we want to preserve the optimal storage strategy for a given application.

- *Modeling of Business Processes:* As we have mentioned, in this text we considered only the modeling of XML data processed and exchanged within an XML system. However, not only the data structures, but also the respective business processes need to be designed, maintained and updated within the evolution process. There is neeed to extend and combine the conceptual models of XML data with the respective business processes, to preserve mutual relations and exploit them during the evolution process [82].

- *Relation to Ontologies:* An up-to-date and important aspect of data management is establishing and exploiting their semantics. Undoubtedly, the most popular tool for this purpose are currently ontologies. Since an ontology can be viewed as a particular type of schema which has strong relationship to a given XML schema, a natural open issue is developing and maintenance of such relationship under application evolution [138, 7]. However, since the ontologies bear a special type of information, their treatment requires specific approaches [99]. We provided a step in this direction in Chapter 13, but further research is needed.

- *Advanced Operations with an XML System:* In this thesis we described two types of operations that can occur in an XML system – atomic and composite. However, these are not the only operations that can occur within the system. If we consider the area of integration, we need to deal with the problem of a new incoming application and its integration with the current ones [98], or even integration of a whole XML system. This wide area involves issues such as reverse-engineering of conceptual models and schema matching [134].

Bibliography

[1] C. ai Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello. Modeling and Managing the Variability of Web Service-Based Systems. *J. of Systems and Software*, 83(3):502 – 516, 2010.

[2] L. Al-Jadir and F. El-Moukaddem. Once Upon a Time a DTD Evolved into Another DTD... In *Object-Oriented Information Systems*, pages 3–17, Berlin, Heidelberg, 2003. Springer.

[3] R. Al-Kamha, D. W. Embley, and S. W. Liddle. Representing Generalization/Specialization in XML Schema. In *EMISA'05*, pages 250–263, 2005.

[4] R. Al-Kamha, D. W. Embley, and S. W. Liddle. Augmenting Traditional Conceptual Models to Accommodate XML Structural Constructs. In *Proceedings of 26th International Conference on Conceptual Modeling*, pages 518–533, Auckland, New Zealand, Nov. 2007. Springer.

[5] A. Algergawy, R. Nayak, and G. Saake. XML Schema Element Similarity Measures: A Schema Matching Context. In *OTM '09*, pages 1246–1253. Springer, 2009.

[6] Altova Inc. XML Spy 2009. `http://www.altova.com`.

[7] Y. An, A. Borgidaa, and J. Mylopoulos. Discovering and Maintaining Semantic Mappings between XML Schemas and Ontologies. *J. of Computing Science and Engineering*, 2(1):44–73, 2008.

[8] Y. An, X. Hu, and I.-Y. Song. Round-Trip Engineering for Maintaining Conceptual-Relational Mappings. In *CAiSE '08: Proc. of the 20th Int. Conf. on Advanced Information Systems Engineering*, pages 296–311, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou. Managing the Evolution of Service Specifications. In *CAiSE '08: Proc. of the 20th Int. Conf. on Advanced Information Systems Engineering*, pages 359–374, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] L. Aversano, M. Bruno, M. D. Penta, A. Falanga, and R. Scognamiglio. Visualizing the Evolution of Web Services using Formal Concept Analysis. In *IWPSE '05: 8th Int. Workshop on Principles of Software Evolution*, pages 57–60, 2005.

[11] A. Badia. Conceptual Modeling for Semistructured Data. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering Workshops*, pages 170–177, Singapore, Dec. 2002. IEEE Computer Society.

[12] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD Rec.*, 16(3):311–322, 1987.

[13] I. Bedini, G. Gardarin, and B. Nguyen. Deriving Ontologies from XML Schema. *CoRR*, abs/1001.4901, 2010.

[14] Z. Bellahsene, A. Bonifati, and E. Rahm. *Schema Matching and Mapping.* Data-Centric Systems and Applications. Springer Berlin Heidelberg, 2011.

[15] S. Benda, J. Klímek, and M. Nečaský. Using Schematron as Schema Language in Conceptual Modeling for XML. In *Conceptual Modelling 2013*, volume 143 of *Conferences in Research and Practice in Information Technology (CRPIT).* Australian Computer Society, Inc., 2013.

[16] S. Benda, B. Zámečník, M. Cicko, P. Sobotka, T. Kroupa, and M. Nečaský. SchemaTron - Native C# validator of ISO Schematron language, September 2011.

[17] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Rec.*, 28(1):54–59, 1999.

[18] M. Bernauer, G. Kappel, and G. Kramler. Representing XML Schema in UML - A Comparison of Approaches. In N. Koch, P. Fraternali, and M. Wirsing, editors, *Web Engineering*, volume 3140 of *Lecture Notes in Computer Science*, pages 767–769. Springer, 2004.

[19] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. *Inference of Concise DTDs from XML Data.* ACM, 2006.

[20] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language.* W3C, January 2007. `http://www.w3.org/TR/xquery/`.

[21] A. Boronat, J. A. Carsí, and I. Ramos. Algebraic Specification of a Model Transformation Engine. In *FASE '06: Proc. of the 9th Int. Conf. Fundamental Approaches to Software Engineering, Vienna, Austria*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006.

[22] B. Bouchou, D. Duarte, M. H. F. Alves, D. Laurent, and M. A. Musicante. Schema Evolution for XML: A Consistency-Preserving Approach. In *Mathematical Foundations of Computer Science*, pages 876–888, Prague, Czech Republic, 2004. Springer-Verlag.

[23] R. Bourret. XML and Databases, September 2005. `http://www.rpbourret.com/xml/XMLAndDatabases.htm`.

[24] F. Cavalieri. EXup: an Engine for the Evolution of XML Schemas and Associated Documents. In *EDBT '10: Proc. of the 2010 EDBT/ICDT Workshops*, pages 1–10, New York, NY, USA, 2010. ACM.

[25] D. Chamberlin, D. Florescu, J. Melton, J. Robie, and J. Siméon. *XQuery Update Facility 1.0.* W3C, 2007.

[26] P. Chen. The Entity-Relationship Model–Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, Mar. 1976.

[27] P. Chen. Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned. In *Software Pioneers: Contributions to Software Engineering*, pages 296–310, New York, NY, USA, 2002. Springer.

[28] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Managing Dependent Changes in Coupled Evolution. In *Proc. of the 2nd Int. Conf. on Model Transformations, ICMT 2009, Zurich, Switzerland*, volume 5563 of *LNCS*, pages 35–51. Springer, 2009.

[29] J. Clark and M. Makoto. *RELAX NG Specification.* Oasis, December 2001. `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[30] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks. In *IJCAI '03 Workshop on Information Integration*, pages 73–78. AAAI, 2003.

[31] S. V. Coox. Axiomatization of the Evolution of XML Database Schema. *Program. Comput. Softw.*, 29(3):140–146, 2003.

[32] C. Curino, H. J. Moon, and C. Zaniolo. Automating Database Schema Evolution in Information System Upgrades. In *HotSWUp '09: Proc. of the 2nd Int. Workshop on Hot Topics in Software Upgrades*, pages 1–5, New York, NY, USA, 2009. ACM.

[33] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench. *Proc. VLDB Endow.*, 1(1):761–772, 2008.

[34] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[35] D. Booth, C. K. Liu. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer.* W3C, June 2007. `http://www.w3.org/TR/wsdl20-primer/`.

[36] H. H. Do and E. Rahm. COMA – A System for Flexible Combination of Schema Matching Approaches. In *VLDB '02*, pages 610–621. Morgan Kaufmann, 2002.

[37] G. Dobbie, W. Xiaoying, T. Ling, and M. Lee. Designing Semistructured Databases Using ORA-SS Model. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering*, pages 171–182, Kyoto, Japan, Dec. 2001.

[38] E. Domínguez, J. Lloret, B. Pérez, A. Rodríguez, A. Rubio, and M. Zapata. A Survey of UML Models to XML Schemas Transformations. In B. Benatallah, F. Casati, D. Georgakopoulos, C. Bartolini, W. Sadiq, and C. Godart, editors, *Web Information Systems Engineering - WISE 2007*, volume 4831 of *Lecture Notes in Computer Science*, pages 184–195. Springer Berlin / Heidelberg, 2007.

[39] E. Domínguez, J. Lloret, A. L. Rubio, and M. A. Zapata. Evolving XML Schemas and Documents Using UML Class Diagrams. In K. V. Andersen, J. K. Debenham, and R. Wagner, editors, *DEXA*, volume 3588 of *Lecture Notes in Computer Science*, pages 343–352. Springer, 2005.

[40] R. dos Santos Mello and C. A. Heuser. A Bottom-Up Approach for Integration of XML Sources. In *Workshop on Information Integration on the Web*, pages 118–124, 2001.

[41] J. Euzenat and P. Shvaiko. *Ontology matching.* Springer-Verlag, Heidelberg (DE), 2007.

[42] J. Fong, S. K. Cheung, and H. Shiu. The XML Tree Model - toward an XML conceptual schema reversed from XML Schema Definition. *Data Knowl. Eng.*, 64(3):624–661, 2008.

[43] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[44] P. Geneves, N. Layaida, and V. Quint. Identifying Query Incompatibilities with Evolving XML Schemas. In *ICFP '09: Proc. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 221–230, New York, NY, USA, 2009. ACM.

[45] H. Hai, Do. *Schema Matching and Mapping-based Data Integration: Architecture, Approaches and Evaluation.* VDM Verlag, Saarbrücken, Germany, Germany, 2007.

[46] T. Halpin and T. Morgan. *Information Modeling and Relational Databases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[47] M. Hartung, J. Terwilliger, and E. Rahm. Recent advances in schema and ontology evolution. In *Schema Matching and Mapping*, Data-Centric Systems and Applications, pages 149–190. Springer Berlin Heidelberg, 2011.

[48] Int. Organization for Standardization. *ISO/IEC 9075-14:2003 Part 14: XML-Related Specifications (SQL/XML).* Int. Organization for Standardization, 2006.

[49] ISO. *Information Technology Document Schema Definition Languages (DSDL) Part 3: Rule-based Validation Schematron. ISO/IEC 19757-3*, feb 2005.

[50] R. Jelliffe. Converting XML Schemas to Schematron, 2007.

[51] M. R. Jensen, T. H. Møller, and T. B. Pedersen. Converting XML Data to UML Diagrams For Conceptual Data Integration. In *In Proceedings of DIWeb'01*, 2001.

[52] M. Kay. *XSL Transformations (XSLT) Version 2.0.* W3C, January 2007. `http://www.w3.org/TR/xslt20/`.

[53] M. Klettke. Conceptual XML Schema Evolution - The CoDEX Approach for Design and Redesign. In M. Jarke, T. Seidl, C. Quix, D. Kensche, S. Conrad, E. Rahm, R. Klamma, H. Kosch, M. Granitzer, S. Apel, M. Rosenmüller, G. Saake, and O. Spinczyk, editors, *Workshop Proceedings Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, pages 53–63, Aachen, Germany, March 2007.

[54] J. Klímek, L. Kopenec, P. Loupal, and J. Malý. XCase - A Tool for Conceptual XML Data Modeling. In *Advances in Databases and Information Systems*, volume 5968/2010 of *Lecture Notes in Computer Science*, pages 96–103. Springer Berlin / Heidelberg, March 2010. `http://www.springerlink.com/content/v45198r1v783xu13`.

[55] J. Klímek, J. Malý, I. Mlýnková, and M. Nečaský. *eXolutio*: Tool for XML Schema and Data Management. In J. Pokorný, V. Snásel, and K. Richta, editors, *DATESO*, volume 837 of *CEUR Workshop Proceedings*, pages 69–80. CEUR-WS.org, 2012.

[56] J. Klímek, J. Malý, and M. Nečaský. XML Schema Integration with Reusable Schema Parts. In V. Snásel, J. Pokorný, and K. Richta, editors, *DATESO*, volume 706 of *CEUR Workshop Proceedings*, pages 13–24. CEUR-WS.org, 2011.

[57] J. Klímek, I. Mlýnková, and M. Nečaský. A Framework for XML Schema Integration via Conceptual Model. In *Advances in Web, Intelligent, Cloud, and Mobile Systems Engineering - WISE 2010 Symposium and Workshops.* Springer, 2011.

[58] J. Klímek and M. Nečaský. Integration and Evolution of XML Data via Common Data Model. In *Proceedings of the 2010 EDBT/ICDT Workshops, Lausanne, Switzerland, March 22-26, 2010*, New York, NY, USA, 2010. ACM.

[59] J. Klímek and M. Nečaský. Reverse-engineering of XML Schemas: A Survey. In J. Pokorný, V. Snásel, and K. Richta, editors, *DATESO*, volume 567 of *CEUR Workshop Proceedings*, pages 96–107. CEUR-WS.org, 2010.

[60] J. Klímek and M. Nečaský. Semi-automatic Integration of Web Service Interfaces. In *IEEE International Conference on Web Services (ICWS 2010)*, pages 307–314. IEEE Computer Society, 2010.

[61] J. Klímek and M. Nečaský. Generating Lowering and Lifting Schema Mappings for Semantic Web Services. In *25th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2010, Biopolis, Singapore, 22-25 March 2011.* IEEE Computer Society, 2011.

[62] J. Klímek and M. Nečaský. Formal Evolution of XML Schemas with Inheritance. In C. A. Goble, P. P. Chen, and J. Zhang, editors, *ICWS*, pages 496–503. IEEE, 2012.

[63] J. Klímek and M. Nečaský. On Inheritance in Conceptual Modeling for XML. In *The 3rd International Conference on Ambient Systems, Networks and Technologies*, volume 10, pages 54–61, 2012.

[64] J. Kopecký, D. Roman, M. Moran, and D. Fensel. Semantic Web Services Grounding. In *In Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, 2006.

[65] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11:60–67, 2007.

[66] A. Kwong and M. Gertz. On Tree Pattern Constraints for XML Documents, 2006.

[67] D. Lee and W. W. Chu. *Comparative Analysis of Six XML Schema Languages.* ACM, 2000.

[68] B. S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. *ACM Trans. Database Syst.*, 25(1):83–127, 2000.

[69] B. Loscio, A. Salgado, and L. Galvao. Conceptual Modeling of XML Schemas. In *Proceedings of the Fifth ACM CIKM International Workshop on Web Information and Data Management*, pages 102–105, New Orleans, USA, Nov. 2003.

[70] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *In Proceedings of VLDB'01*, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[71] J. Malý, I. Mlýnková, and M. Nečaský. XML Data Transformations as Schema Evolves. In *15th International Conference on Advances in Databases and Information Systems (ADBIS 2011)*, LNCS, Berlin, Heidelberg, 2011. Springer-Verlag.

[72] J. Malý and M. Nečaský. Utilizing new capabilities of XML languages to verify integrity constraints. In *Proceedings of Balisage: The Markup Conference 2012*, volume 8, 2012.

[73] M. Mani. Erex: A conceptual model for xml. In *Proceedings of the Second International XML Database Symposium*, pages 128–142, Toronto, Canada, Aug. 2004.

[74] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[75] Microsoft. Silverlight. `http://www.microsoft.com/silverlight/`.

[76] Microsoft. Windows Presentation Foundation (WPF), December 2010. `http://msdn.microsoft.com/en-us/library/ms754130.aspx`.

[77] G. A. Miller. WordNet: a lexical database for English. *Commun. ACM*, 38(11):39–41, November 1995.

[78] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1.* Object Management Group, 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.

[79] M. Moran. Towards Translating between XML and WSML based on mappings between XML Schema and an equivalent WSMO Ontology. In *In Proceedings of the WIW 2005 Workshop on WSMO Implementations*, page 134, 2005.

[80] M. M. Moro, S. Malaika, and L. Lim. Preserving XML Queries During Schema Evolution. In *WWW '07: Proc. of the 16th Int. Conf. on World Wide Web*, pages 1341–1342, New York, NY, USA, 2007. ACM.

[81] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML Schema Languages using Formal Language Theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[82] M. Murzek, G. Kramler, and E. Michlmayr. Structural Patterns for the Transformation of Business Process Models. In *EDOCW '06: Proc. of the 10th IEEE on Int. Enterprise Distributed Object Computing Conf. Workshops*, page 18, Washington, DC, USA, 2006. IEEE Computer Society.

[83] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: representing knowledge about information systems. *ACM Trans. Inf. Syst.*, 8(4):325–362, 1990.

[84] P. Nálevka. Grammar vs. Rules, May 2010.

[85] M. Nečaský. Conceptual Modeling for XML: A Survey. In *Dateso '08*, pages 40–53. CEUR-WS, Vol. 176, April 2006.

[86] M. Nečaský. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems Series.* IOS Press/AKA Verlag, January 2009.

[87] M. Nečaský. Reverse Engineering of XML Schemas to Conceptual Diagrams. In *Proceedings of The Sixth Asia-Pacific Conference on Conceptual Modelling*, pages 117–128, Wellington, New Zealand, January 2009. Australian Computer Society.

[88] M. Nečaský, J. Klímek, and J. Malý. When theory meets practice: A case report on conceptual modeling for XML. In *ICDIM*, pages 242–251. IEEE, 2011.

[89] M. Nečaský, J. Klímek, J. Malý, and I. Mlýnková. Evolution and Change Management of XML-based Systems. *Journal of Systems and Software*, 85(3):683 – 707, 2012.

[90] M. Nečaský, J. Malý, J. Klímek, and I. Mlýnková. Evolution and Change Management of XML Applications. Technical Report May 2011, XML and Web Engineering Research Group, Charles University in Prague, 2011.

[91] M. Nečaský and I. Mlýnková. On Different Perspectives of XML Data Evolution. In A. M. Tjoa and R. Wagner, editors, *DEXA Workshops*, pages 422–426. IEEE Computer Society, 2009.

[92] M. Nečaský and I. Mlýnková. When Conceptual Model Meets Grammar: A Formal Approach to Semi-structured Data Modeling. In *WISE'10*, volume 6488 of *LNCS*, pages 279–293. Springer Berlin / Heidelberg, 2010.

[93] M. Nečaský and I. Mlýnková. Five-Level Multi-Application Schema Evolution. In *DATESO '09: Proc. of the Databases, Texts, Specifications, and Objects*, pages 213–217. MatfyzPress, April 2009.

[94] M. Nečaský and I. Mlýnková. A Framework for Efficient Design, Maintaining, and Evolution of a System of XML Applications. In *DATESO '10: Proc. of the Databases, Texts, Specifications, and Objects*, pages 38 – 49. MatfyzPress, April 2010.

[95] M. Nečaský, I. Mlýnková, and J. Klímek. Model-Driven Approach to XML Schema Evolution. In *OTM Workshops*, pages 514–523, 2011.

[96] M. Nečaský, I. Mlýnková, J. Klímek, and J. Malý. When conceptual model meets grammar: A dual approach to XML data modeling. *Data & Knowledge Engineering*, 72(0):1 – 30, 2012.

[97] M. Nečaský. Conceptual Modeling for XML: A Survey. In V. Snásel, K. Richta, and J. Pokorný, editors, *DATESO*, volume 176 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[98] H.-Q. Nguyen, D. Taniar, J. W. Rahayu, and K. Nguyen. Double-Layered Schema Integration of Heterogeneous XML Sources. *J. of Systems and Software*, 84(1):63 – 76, 2011.

[99] N. F. Noy and M. Klein. Ontology Evolution: Not the Same as Schema Evolution. *Knowl. Inf. Syst.*, 6(4):428–440, 2004.

[100] OASIS. *Web Services Business Process Execution Language (WSBPEL) TC.* OASIS, 2007.

[101] Object Management Group. *UML Infrastructure Specification 2.1.2*, nov 2007. `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/`.

[102] Object Management Group. *UML Superstructure Specification 2.1.2*, nov 2007. `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/`.

[103] U. Ogbuji. *A hands-on introduction to Schematron.* IBM, 2004.

[104] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0.* Object Modeling Group, April 2008. `http://www.omg.org/spec/QVT/1.0/`.

[105] OMG. *MOF 2.0 / XMI Mapping Specification, v2.1.1.* OMG, 2009.

[106] OMG. *Object Constraint Language Specification, version 2.0.* OMG, 2009.

[107] OpenTravel. Opentravel xml schemas, version 2009a. `http://opentravel.org/Specifications/SchemaIndex.aspx?FolderName=2009A`.

[108] L. Palopoli, G. Terracina, and D. Ursino. DIKE: a system supporting the semi-automatic construction of cooperative information systems from heterogeneous databases. *Softw. Pract. Exper.*, 33(9):847–884, 2003.

[109] M. Paolucci, M. Wagner, and D. Martin. Grounding OWL-S in SAWSDL. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 416–421, Berlin, Heidelberg, 2007. Springer-Verlag.

[110] C.-S. Park and S. Park. Efficient Execution of Composite Web Services Exchanging Intensional Data. *Information Sciences*, 178(2):317 – 339, 2008.

[111] K. Passi, D. Morgan, and S. Madria. Maintaining Integrated XML Schema. In *IDEAS '09: Proc. of the 2009 Int. Database Engineering, Applications Symp.*, pages 267–274, New York, NY, USA, 2009. ACM.

[112] G. D. Penna, A. D. Marco, B. Intrigila, I. Melatti, and A. Pierantonio. Interoperability mapping from xml schemas to er diagrams. *Data & Knowledge Engineering*, 59(1):166 – 188, 2006.

[113] G. Psaila. ERX: A Conceptual Model for XML Documents. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 898–903, Como, Italy, Mar. 2000. ACM.

[114] R. Ravichandar, N. C. Narendra, K. Ponnalagu, and D. Gangopadhyay. Morpheus: Semantics-based Incremental Change Propagation in SOA-based Solutions. *Services Computing, IEEE Int. Conf. on*, 1:193–201, 2008.

[115] C. Reynaud, J.-P. Sirot, and D. Vodislav. Semantic integration of xml heterogeneous data sources. In *In Proceedings of IDEAS '01*, pages 199–208, Washington, DC, USA, 2001. IEEE Computer Society.

[116] P. Rodríguez-Gianolli and J. Mylopoulos. A Semantic Approach to XML-based Data Integration. In *ER '01: Proceedings of the 20th International Conference on Conceptual Modeling*, pages 117–132, London, UK, 2001. Springer-Verlag.

[117] N. Routledge, L. Bird, and A. Goodchild. UML and XML Schema. In *Proceedings of 13th Australasian Database Conference (ADC 2002)*. ACS, 2002.

[118] S. H. Ryu, F. Casati, H. Skogsrud, B. Benatallah, and R. Saint-Paul. Supporting the Dynamic Evolution of Web Service Protocols in Service-Oriented Architectures. *ACM Trans. Web*, 2(2):1–46, 2008.

[119] A. Sengupta, S. Mohan, and R. Doshi. XER - Extensible Entity Relationship Modeling. In *Proceedings of the XML 2003 Conference*, pages 140–154, Philadelphia, USA, Dec. 2003.

[120] F. T. Sheldon, K. Jerath, and H. Chung. Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance*, 14(3):147–160, May 2002.

[121] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, 4:146–171, 2005.

[122] A. A. Simanovsky. Data Schema Evolution Support in XML-Relational Database Systems. *Program. Comput. Softw.*, 34(1):16–26, 2008.

[123] R. Sindhgatta and B. Sengupta. An Extensible Framework for Tracing Model Evolution in SOA Solution Design. In *OOPSLA '09: Proc. of the 24th ACM SIGPLAN Conf. Companion on Object Oriented Programming Systems Languages and Applications*, pages 647–658, New York, NY, USA, 2009. ACM.

[124] S. Sorrentino, S. Bergamaschi, M. Gawinecki, and L. Po. Schema Normalization for Improving Schema Matching. In *ER '09*, pages 280–293. Springer, 2009.

[125] P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and System Modeling*, 9(1):7–20, 2010.

[126] T. Bray and J. Paoli and C. M. Sperberg-McQueen and E. Maler and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition).* W3C, September 2006. `http://www.w3.org/TR/REC-xml/`.

[127] M. Tan and A. Goh. Keeping Pace with Evolving XML-Based Specifications. In *EDBT'04 Workshops*, pages 280–288, Berlin, Heidelberg, 2005. Springer.

[128] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition).* W3C, October 2004. `http://www.w3.org/TR/xmlschema-1/`.

[129] P. T. T. Thuy, Y.-K. Lee, and S. Lee. DTD2OWL: Automatic Transforming XML Documents into OWL Ontology. In *In Proceedings of ICIS '09*, pages 125–131, New York, NY, USA, 2009. ACM.

[130] E. Vlist. *XML Schema The W3C's Object-Oriented Descriptions for XML.* O'Reilly Media, June 2002.

[131] W3C. Semantic Annotations for WSDL and XML Schema, August 2007.

[132] W3C OWL Working Group. *OWL 2 Web Ontology Language.* W3C, October 2009. `http://www.w3.org/TR/owl2-overview/`.

[133] Y. Weidong, G. Ning, and S. Baile. Reverse Engineering XML. *Computer and Computational Sciences, International Multi-Symposiums on*, 2:447–454, 2006.

[134] A. Wojnar, I. Mlýnková, and J. Dokulil. Structural and Semantic Aspects of Similarity of Document Type Definitions and XML Schemas. *Information Sciences*, 180(10):1817–1836, 2010. Special Issue on Intelligent Distributed Information Systems.

[135] L. Xiao, L. Zhang, G. Huang, and B. Shi. Automatic Mapping from XML Documents to Ontologies. In *CIT '04: Proceedings of the The Fourth International Conference on Computer and Information Technology*, pages 321–325, Washington, DC, USA, 2004. IEEE Computer Society.

[136] W. Yang, N. Gu, and B. Shi. Reverse Engineering XML. In J. Ni and J. Dongarra, editors, *IMSCCS (2)*, pages 447–454. IEEE Computer Society, 2006.

[137] A. Yu. An Overview of Research on Reverse Engineering XML Schemas into UML Diagrams. In *ICITA'05 Volume 2 - Volume 02*, ICITA '05, pages 772–777, Washington, DC, USA, 2005. IEEE Computer Society.

[138] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings when Schemas Evolve. In *VLDB '05*, pages 1006–1017. VLDB Endowment, 2005.

List of Tables

List of Figures