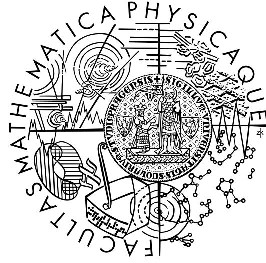

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Marek Polák

Evolution and Adaptability of Complex Applications

Department of Software Engineering

Thesis supervisor: doc. RNDr. Irena Holubová, Ph.D.
Study programme: Computer Science
Specialization: Software Systems

Prague 2017

First of all I would to sincerely thank my supervisor Irena Holubová for her support and guidance throughout my doctoral studies, especially to her helpful comments and prompt responses, whenever I needed. Next, my thanks belong to Eva Mládková for her dedication and sometimes difficult administrative support, as well as to members, colleagues and fellow students from the XML and Web Engineering Research Group. However, achieving results of my research would not be possible even without the effort of all the anonymous reviewers. Nevertheless, my greatest gratitude no doubt deserve my my parents, all the other family members and friends for their support, patience as well as encouragement. Last but not least, I would like to thank for the financial support provided by the following institutions, grants and projects: Charles University Grant Agency (SVV-2013-267312, SVV-2014-260100, SVV-2015-260222, SVV-2017-260451, GAUK-1416213) and Czech Science Foundation (P202/10/0573).

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

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

In Prague on June 16, 2017 Marek Polák

Název práce: Evoluce a adaptabilita komplexních aplikací
Autor: Marek Polák
Katedra: Katedra softwarového inženýrství
Vedoucí disertační práce: doc. RNDr. Irena Holubová, Ph.D.

Abstrakt: V současné době se aplikace stávají stále složitějšími, což přináší problémy během jejich vývoje. Změna v jedné části aplikace může netriviálně ovlivnit jiné části aplikace. Dalším aspektem mohou být systémy, které s aplikací komunikují. Ty musí být upraveny, aby se zajistila správná funkcionalita. Tyto problémy se mohou týkat různých domém – UML diagramů, XML schémat, relačních schémat, API, atd. V této práci jsme se na zmíněný problém zaměřili z perspektivy MDA, která pro obecný náhled na problém využívá platformově nezávislého modelu (PIM) a pro konkrétní domény využívá platformově specifické modely (PSM). Tyto modely mohou být navíc propojeny a vzájemně závislé. Náš návrh obsahuje nové definice modelů z různých, široce využívaných domén, operace nad těmito modely a algoritmy pro transformace modelů. Díky principu MDA je možné představené modely kombinovat a modelovat tak komplexní aplikace. Veškeré prezentované modely a algoritmy byly experimentálně implementovány ve frameworku DaemonX a testovány na reálných datech, aby byla ověřena jejich správnost.

Klíčová slova: MDA, MDD, Management evoluce, Transformace

Title: Evolution and Adaptability of Complex Applications
Author: Marek Polák
Department: Department of Software Engineering
Supervisor: doc. RNDr. Irena Holubová, Ph.D.

Abstract: In these days the applications become more complex that causes maintenance issues while evolving these applications. A change in one part of the application can significantly affect other parts of the application. The next aspect can be related systems which communicate with this application. They must be updated to satisfy their correct functionality. These issues can concern multiple domains, e.g., UML diagrams, XML schema diagrams, relational schemas, APIs, etc. We focus on this problem from the perspective of the MDA, which uses the platform independent model (PIM) for a general view of the problem and the platform specific model (PSM) for particular domains. Moreover, these models can be interconnected and related to each other. We propose novel PSM models from various widely used domains, operations over these models and algorithms for model transformations. Thanks to the MDA principle, it is possible to combine presented models and model a complex application. All models and related algorithms we present were experimentally implemented and tested in the DaemonX framework on real-world data for their verification.

Keywords: MDA, MDD, Evolution Management, Transformation

Contents

1 Introduction **5**

2 Preliminaries **17**
2.1 Models 17
2.2 Operations 18
2.3 Model Relations 18
2.4 Evolution Process 19

3 DaemonX **21**
3.1 Introduction 21
3.2 Related Works 22
3.2.1 eXolutio 22
3.2.2 Enterprise Architect 22
3.2.3 Power Designer 22
3.2.4 Eclipse 23
3.2.5 Visual Studio 23
3.2.6 Comparison of the Related Works 23
3.3 Architecture 25
3.4 Plug-in Support 25
3.5 Evolution Process Management 26
3.6 Undo/Redo Management 28
3.7 Additional Framework Extensions 28
3.8 Conclusion 28
3.8.1 Future Work 30

4 XML Query Evolution **33**
4.1 Introduction 33
4.2 Related Works 34
4.2.1 Preserving XML Queries During Schema Evolution 34
4.2.2 Identifying Query Inconsistencies with Evolving XML Schemas 36
4.2.3 Transformation of structure-shy programs with application to XPath queries and strategic functions 38
4.2.4 Evolution of XML-Based Mediation Queries in a Data Integration System 40
4.2.5 Comparison of the Related Works 41
4.3 Models for XML Schema and XPath 42
4.4 Evolution Algorithm 44
4.4.1 Operations for the XSEM Model 44
4.4.2 Operations for the XPath Model 45
4.5 Analysis of Propagation of Operations 47
4.5.1 Adding 47
4.5.2 Removing 49
4.5.3 Renaming 50
4.5.4 Reordering 51
4.5.5 Reconnection 53

4.6 Implementation and Experiments . . . 55
4.7 Conclusion . . . 58
4.7.1 Future Work . . . 58

5 Relational Schema and SQL Queries Evolution **59**
5.1 Introduction . . . 59
5.2 Related Works . . . 60
5.2.1 Database Schema Integration Process . . . 60
5.2.2 QuickMig . . . 61
5.2.3 The PRISM Workbench . . . 63
5.2.4 Automating the Database Schema Evolution Process . . . 66
5.2.5 Adaptive Query Formulation . . . 67
5.2.6 Synchronization of Queries and Views Upon Schema Evolutions: A Survey . . . 69
5.2.7 Comparison of the Related Works . . . 70
5.3 Database Model . . . 71
5.4 Query Model . . . 73
5.5 SQL Query Visualization Model . . . 74
5.5.1 Visualization Model Components . . . 74
5.5.2 Mapping to the Database Model . . . 76
5.5.3 Mapping of Operations . . . 77
5.6 Change Propagation in the Graph . . . 78
5.6.1 Query Graph Operations . . . 78
5.7 Implementation and Experiments . . . 79
5.8 Conclusion . . . 83
5.8.1 Future Work . . . 83

6 Service Interfaces and Business Processes Evolution **85**
6.1 Introduction . . . 85
6.2 Related Works . . . 87
6.2.1 UML and XML Schema . . . 87
6.2.2 XSEM – A Conceptual Model for XML . . . 88
6.2.3 An Extension of Business Process Model for XML Schema Modeling . . . 89
6.2.4 Comparison of the Related Works . . . 90
6.3 Business Processes and Conceptual Modeling . . . 91
6.4 Proposed Approach . . . 92
6.4.1 Derivation Part . . . 95
6.4.2 Evolution Part . . . 105
6.5 Implementation and Experiments . . . 107
6.6 Conclusion . . . 107
6.6.1 Future Work . . . 109

7 REST API Management and Evolution **113**
7.1 Introduction . . . 113
7.2 Related Works . . . 114
7.2.1 API Versioning Best Practises . . . 114
7.2.2 API Documentation Generation . . . 114
7.2.3 Comparison of the Related Works . . . 115

7.3 Resource Model 115
7.4 Mapping and Evolution 117
7.4.1 Model Mapping 117
7.5 Atomic PIM Model Operations 118
7.6 Atomic Resource Model Operations 121
7.7 Operation Propagation Policies 124
7.8 Propagation Algorithms 125
7.8.1 Cardinalities 136
7.8.2 API Versioning 137
7.8.3 View Model 137
7.8.4 Model Nesting 137
7.8.5 Resource Parameters Evolution 138
7.8.6 Applying the Solution on Existing Clients 138
7.9 Implementation and Experiments 139
7.9.1 Experimental Data 139
7.9.2 Experiments 141
7.10 Conclusion 146
7.10.1 Future Work 147

8 Schema Mapping **149**
8.1 Introduction 149
8.2 Related Works 150
8.2.1 COMA 150
8.2.2 Similarity Flooding 151
8.2.3 Decision Tree 151
8.2.4 XML Schema Clustering with Semantic and Hierarchical Similarity Measures 152
8.2.5 Minimizing User Effort in XML Grammar Matching 153
8.2.6 XML Matchers: Approaches and Challenges 155
8.2.7 Comparison of the Related Works 157
8.3 Schema Matching 157
8.3.1 Applications of Schema Matching 158
8.4 Proposed Solution 159
8.4.1 Original Decision Tree Construction 159
8.4.2 Decision Tree Training via C5.0 161
8.5 Implementation and Experiments 166
8.6 Conclusion 173
8.6.1 Future Work 174

9 Experiments **177**
9.1 Description of Experiments 177
9.1.1 Experimental Data 177
9.1.2 Experimental Evaluation 178
9.1.3 Database Model 179
9.2 Particular Experiments 179
9.2.1 Experimental Results 184
9.3 Conclusion 185

10 Conclusion **187**

1. Introduction

The most common application of the today's information-technology (IT) world are *information systems* (IS). They can be characterized as a network of software and hardware components that enable people and companies to create, collect, distribute and process data. There exist various types of ISs, such as, e.g., decision-support systems, database-management systems, office-information systems, customer relationship management systems, etc.

Currently a very popular kind of IS in these days are distributed ISs and the micro-service architecture that are typically base on the *Service Oriented Architecture* (SOA) [54] and its most common implementation – *Web Services* (WS) [131], *Representational State Transfer* (REST) [39] and/or *Web Sockets* [133]. This architecture brings many advantages especially to system scalability, performance, and resource management. On the other hand, separation of the system to micro-services brings drawbacks, such as more demanding change management and version compatibility. For example, a change of the message structure in one part of the system can influence all related services, integration tests are more complex and must cover more edge cases than in case of monolithic systems, etc. A mechanism that can analyze the changes, propagate them, and/or at least inform the developer about possible inconsistency can reduce time needed for updates and unnecessary troubleshooting.

An IS often involves a huge set of *data resources* (known as *data intensive ISs* [102]) and applications which process them. A typical current IS also usually consists of a set of sub-applications, each being responsible for a particular logical execution part (termed a *system of applications*), e.g., database management, business logic, route management, balancing management, etc. An example of a typical IS is depicted in Figure 1.1.

A data resource consists of the particular pieces of information (i.e., *data instances*), *integrity constraints* (ICs) over the data instances, and selected *interpretations* of the data. For instance, an IS for storing and analysis of scientific publications of academic institutions involves records of publications of a particular university, i.e., articles, books, SW prototypes, etc. Over the records we can identify various ICs, such as, e.g., that "each book must be assigned with a unique ISBN" or that "a publication cannot have two authors with the same name". Finally, for the purpose of presentation of the results at web sites of the institutions, the data may be exported in XML [129], JSON [26], HTML [112] or RDF [14], described in WSDL [128], exchanged using SOAP [127] messages or REST API, etc. [95].

The life-cycle of a complex system of applications is similar to the life-cycle of a single application; however, the complexity is much higher. First of all we need to design numerous data structures, i.e., schemas, which are usually mutually related or overlayed. In other words, each application of the system utilizes several *views* of a common problem domain. Hence, they cannot be designed separately. In addition, sooner or later the user requirements for the applications change and, hence, the data structures they process must be modified respectively – this is known as the problem of *evolution*. Due to the relations and overlays, such a modification can influence multiple parts of the system and we need to

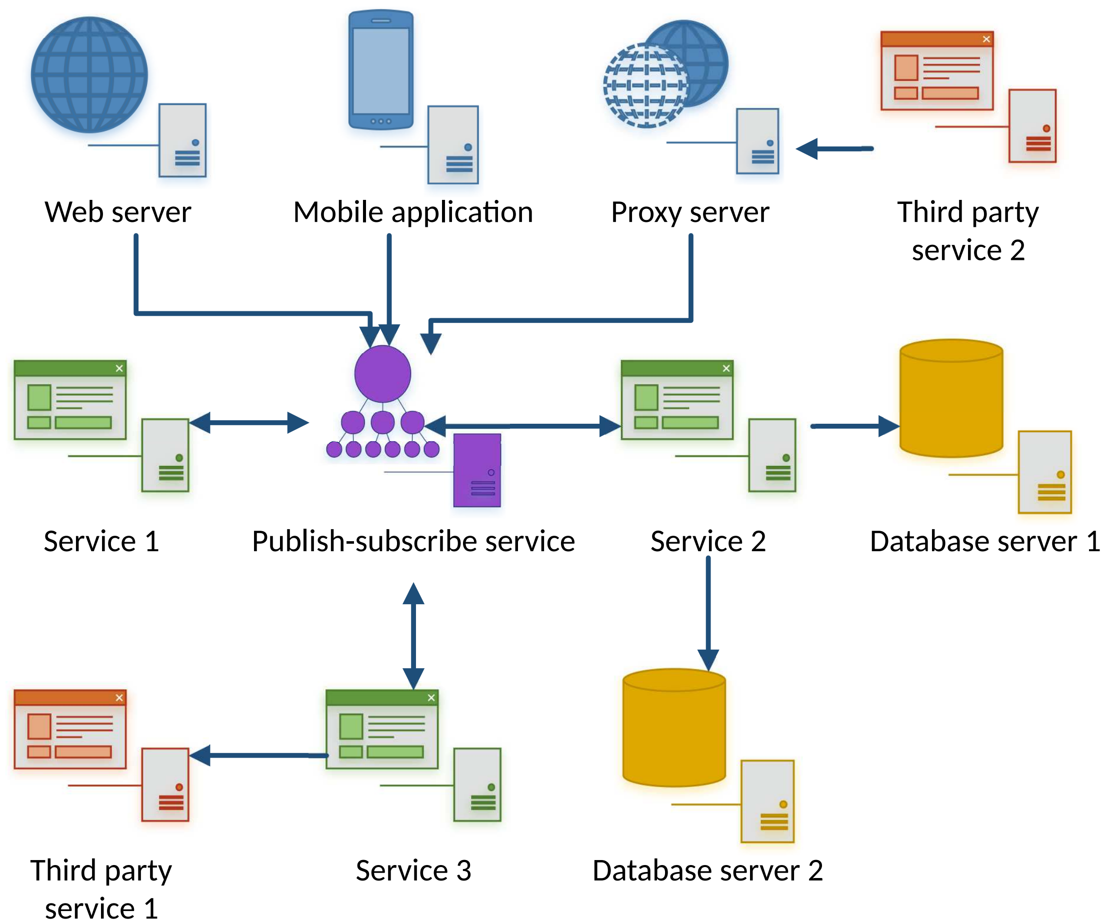

Figure 1.1: An example of simple IS

maintain them during the whole life-cycle to be able to propagate the changes to all affected parts correctly and efficiently. The ability of an IS to adapt to the changes is called *adaptability*.

Even though such ability is a rather natural feature, currently it is being solved mostly manually by an IT expert who knows the IS well and is able to identify all its components that need to adapt. In some situations there exist special mechanisms that can help with necessary changes, e.g., automatic database migration in most *Object Relational Mapping* (ORM) tools [8]. But, in a complex IS it is not possible for a single person to consider and cover all the components and aspects, which can lead to faults in the system.

Another approach, utilized mainly by standardization entities is preservation of *backward compatibility*. For example, an open XML standard *OpenTravel.org*[1] currently offers 329 XML schemas which standardize communication in the traveling community. *OpenTravel.org* is changed twice a year (A and B revisions) and the changes are published in a form of a new version of the XML schemas and a documentation of the changes in a form of human-readable document. This solution, however, requires tremendous manual effort from designers to adapt their transformation scripts and potentially their database, their own XML formats and their program code as well. Thus, *OpenTravel.org* tries to preserve the backward compatibility as much as possible. The amount of backward inconsistent changes is very small – 7,5% on average. However, this approach results in

[1] `http://www.opentravel.org`

artificial data structures with plenty of optional items, obsolete data structures, etc.

Definition 1. *(Forward Compatibility). Let S be an original schema and S' a new version of S. S' is called* forward-compatible *when all documents valid against S are also valid against S'.*

Definition 2. *(Backward Compatibility). Let S be an original schema and S' a new version of S. S is called* backward-compatible *when all documents valid against S' are also valid against S.*

Model Driven Architecture

MDA (Model Driven Architecture) [47] is an approach to software development which enables to define parts of systems via various models for specific purposes. It is *model-driven*, because it provides means for using models to understanding, design, construction, deployment, operation, maintenance and modification. One of the main aims is to separate a design from an architecture. It shifts the development process from a code-centric to model-centric approach. Instead of writing lines of a code, architects model systems according to business processes. MDA deals with the idea of separating the specification of the operation of a system from details of the way the system uses the capabilities of its platform. There are defined three different types of models (layers of abstraction) for specific purposes:

- A *Computation Independent Model (CIM)* is a view of a system from the computation independent point of view. It focuses on the environment of the system, and the requirements for the system. It has no detailed specification of the structure of the system. (Note that we omit this part in our work because it is not important from our point of view.)
- A *Platform Independent Model (PIM)* is a view of a system from the platform independent point of view. It focuses on the operation of a system and hides details necessary for a particular platform. A common language used for modeling PIM diagrams is UML [50].
- A *Platform Specific Model (PSM)* is a model representing specific platform. It uses PIM and adds a view specific for a particular model.

Over all models there can be defined various operations for manipulating with them, e.g., creating of model parts, updating of properties, removal of model parts, etc.

The next important aspect of MDA is a definition of relations (interconnections or mappings) between the models. Mapping marks can be represented as (directed) lines between PIM and PSM items. A general example of a simple system is depicted in Figure 1.2. This aspect, together with operations, enables to define transformations based on model operations propagated to other model(s), called *Model Transformation*. Thanks to these abilities it is possible to design a complex system which contains various models at various levels of abstraction. And, additionally, when one model is changed, these changes can be propagated to other related models.

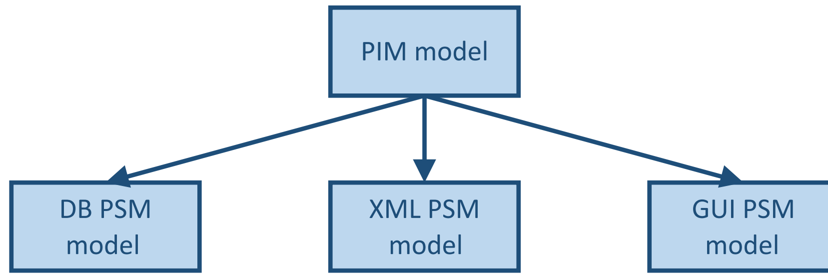

Figure 1.2: An example of the PIM-to-PSM mapping from one PIM model to multiple PSM models

MDA provides an approach for the following actions:

- Specifying a system independently of the platform.
- Specifying platforms.
- Choosing a particular platform for the system.
- Transforming the system specification into the one selected for a particular platform.

In the context of example in Figure 1.2, if we make changes in PIM model, a model transformation is executed and changes are propagated to related PSM models.

Evolution Management

To be able to manage models, operations over them and respective transformations, there must be defined a process capable of performing it.

As we have mentioned, a natural and real-world solution of the evolution problem is to rely on an IT expert who knows the information system well and is able to denote the part of the system which is modified and has to be evolved. This task is related to several nontrivial problems (P1-P5), namely being able:

(P1) To make the required change semantically correctly

All changes which can be done over any of the models must be defined semantically correctly. Before and after the change, the system must be in a consistent state.

(P2) To identify all affected parts of the system

It means that before the change, all related parts must be identified and checked if it is needed to change them too.

(P3) To make the respective changes of the affected parts semantically correctly

If the related parts will not be changed, the whole system can become inconsistent which is not required.

(P4) To express the changes also syntactically correctly regarding the selected format

In a complex information system involving hundreds of schemas it is impossible for a single person to consider and cover all the components and aspects. Especially if the system may involve multiple formats.

(P5) To integrate new schemas and discover relations to the current ones

It is common that the system may naturally grow, e.g., new models or schemas may appear or be required and, hence, the possibility of necessary system change must be analyzed, e.g., to guarantee backward compatibility.

Solution Based on System Partitioning

In papers [92, 94] the authors described the idea of a *five-level evolution management framework*. Using several levels of abstraction it enables to model all parts of the system regardless technical details of a selected format. Preserved relations between the levels enable to propagate the changes correctly among multiple related and overlapping schemas.

An example of a complex system representing an evolution management framework is depicted in Figure 1.3. If we consider the vertical partitioning, we can identify multiple views of the system. 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 a *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, e.g., using BPEL scripts [98].

If we consider the horizontal partitioning, we can identify five levels, each representing a different view of the system and its evolution. The lowest 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 [11] or SQL/XML [61] 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) [62].

Even these three levels indicate problems related to 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, in practice there are usually multiple XML schemas (or schemas in other formats) applied in a single system, i.e., multiple views of the

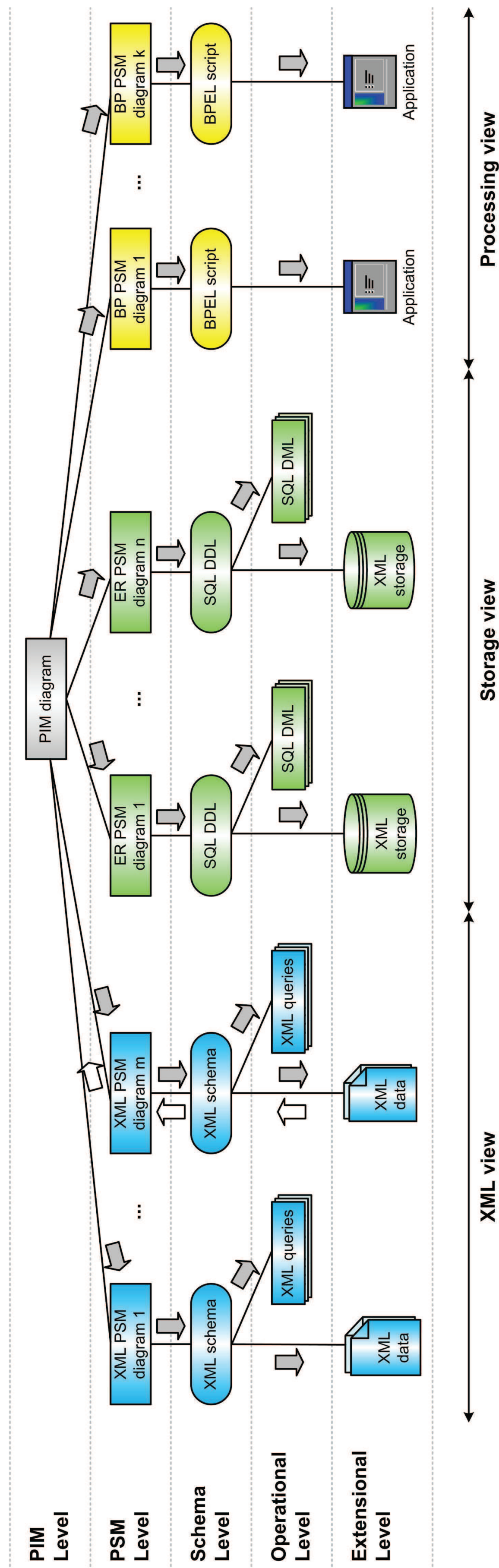

Figure 1.3: Five-level evolution management framework

common problem domain. In general, a change at one level can trigger a cascade of changes at other levels.

Considering only the three levels leads to evolution of each affected schema separately. However, this is a highly time-consuming and error-prone solution. Therefore, we introduce two additional levels, which follow the MDA principle, i.e., modeling of a problem domain at different levels of abstraction. The topmost one is the PIM 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 [91] schemas which model hierarchical data structures implemented using a selected XML schema language or ER [17] schemas which are typically implemented using relational schemas. Each PSM schema can be then automatically translated into a particular language used at the schema level (e.g., XML Schema [123] or SQL DDL) and vice versa.

Now, having a hierarchy of models which interconnect all the applications and views of the data domain using the common PIM level, change propagation 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. Hence, the change propagation is realized using this common point. For instance, if a change occurs in a selected XML schema, or even (although not usually [12]) in a selected XML document, it is first propagated to the respective XML schema, PSM schema and, finally, PIM schema. It is known as *upwards propagation*, in Figure 1.3 represented by the white arrows. It enables one to identify the part of the problem domain that was affected. Then, we can invoke the *downwards propagation* and propagate the change of the problem domain to all the related parts of the system. In Figure 1.3 it is denoted by the grey arrows.

In paper [93] the authors focused on one specific area called *XML view* [93], i.e., evolution of a set of XML schemas based on evolution of underlaying PSM. In this thesis we extend the idea towards other possible formats, i.e., we consider the system in its full generality – to be able to evolve every defined model or schema on different levels.

XML Data Similarity

In these days, over the internet, there are millions of various documents. Documents from different sources, but similar sectors can highly resemble in their structures. If we want to use them, we have to manage every single structure. Documents can differ in their naming conventions, dialect, etc. An example can be a service that aggregates (AG) other similar services, extracts data from them and recalls them, e.g., a service for reserving flights or accommodations. The problem is that there is no general structure of the data that these third party services accept – AG has to store and maintain specific external structures for each third party service. Additionally, AG has own internal data structures and they are mapped to external structures. And, in addition, a very common situa-

tion is that AG designers want to add a new third party service. So they have to analyze data structures that the service provides and map them to internal data structures.

If we look at this problem from the MDA perspective, there can be created a common PIM (internal data structure) and specific PSM for each third party service (external data structures). The problem is that rewriting all these similar structures into particular PSM, their analysis and mapping them into PIM is time-consuming and error-prone process. This problem can be solved by the following steps:

- PSMs of third party services are generated from particular XML schemas.
- These PSMs are analyzed by algorithms for similarity measurement.
- Results of the similarity analysis are used for mutual mapping of common PIM to particular PSMs. This step can be done (semi)automatically, depending on the analysis results.

The described functionality, in combination with an evolution framework, brings a possibility to extend existing systems with new data structures in a more simple way.

Contributions

In order to provide a quick navigation, the references to the author's original contributions are marked with a star (see on the left). We result from the mentioned architecture designs, frameworks and ideas and define various platform-specific models based on these ideas. These models cover currently used technologies and languages, such as XML Schema, XPath [132], SQL, REST, or BPMN [49]. All these models are described in the subsequent chapters. When we combine all the presented models, algorithms, experimental implementations and experiments together, we obtain a complex solution with the following abilities:

Definition of new models We provide a set of various platform-specific models of technologies and languages widely used in these days, namely:

- The model representing XPath query language described in Chapter 4, denoted with number *1* in Figure 1.4.
- The model representing SQL query model described in Chapter 5, denoted with number *2* in Figure 1.4.
- The model representing BPMN described in Chapter 6, denoted with number *3* in Figure 1.4.
- The model representing REST API described in Chapter 7, denoted with number *4* in Figure 1.4.

Definition of model operations A model itself is not sufficient – it just describes a structure or a view. Hence, we provide a set of operations defined over these models for model design and management. These operations are the most important part and essential condition to be able to use and work with models. Thanks to them we can change/update models by well-defined conditions. Operations are also needed for subsequent definition of transformation algorithms. They must be analyzed and they form the output of the transformation algorithm as well.

Transformation algorithms We present algorithms for performing transformations and evolution process between particular models. This is the natural next step after definition of models and respective operations. Transformation algorithms use operations defined over particular models – they accept operations, make an analysis of the operations and generate operations for other model(s) as an output. That means that if one model is changed, these changes can be propagated to related models based on the performed operations and transformations. This concept is described more precisely in Chapter 2.

Experimental implementation and experiments We have experimentally implemented all the presented models, their operations and transformation algorithms and used real-world data in complex tests to verify their correctness. Except for a proof of correctness, an experimental implementation is important from the user perspective too. Although an algorithm of transformation can be correctly defined, it is possible that it is not correct or does not make sense from the user point of view. To be able to validate this situation, real-world implementation is an important task.

DaemonX framework The experimental implementation was carried out as multiple extensions (plug-ins) of the *DaemonX* framework. It is described in Chapter 3. The framework was designed to be a general base for modeling that contains a support for model transformations. Thanks to this solution we have a common base for all implemented models and defined transformation algorithms.

XML data similarity We have also focused on the XML data similarity topic to be able to map multiple similar PSM schemas to the common PIM schema. In this case we do not define any new model or transformation algorithms. We approach from different point of view – we have multiple PSM schemas that have similar structure. If we can map multiple PSM schemas to one PIM schema, the respective changes can be propagated to all affected schemas in a simple way. Specifically, we have improved existing algorithms to obtain more accurate results of the document similarity.

Advanced undo/redo management in complex environments We have also dealt with the issue of providing advanced undo/redo management in complex environments (IDEs). As we have mentioned, almost every GUI application, e.g., development IDEs, document or graphical editors, supports undoing and redoing of user actions. Although they can support separate undo/redo for

multiple tabs, it can be not sufficient for an application where we expect multiple models which can be interconnected/related to each other. Additionally, with the support for model transformation, a change in one model can change other related models. And a subsequent switch to a different model and application of an undo operation must be processed correctly. And, finally, there is another important fact that is user experience – the behavior must be understandable for users. We tried to analyze requirements and define a new manner how to solve this problem from both technical and user viewpoint.

Due to space limitation, the chapter describing this topic was omitted from this thesis. Even the chapter discuss important subject of the complex applications, it has the least relation to *evolution* and *adaptability* which is the theme of this work.

Publication of the research All the mentioned research results were published and presented at multiple international conferences or in journals, namely:

- Paper *XML Query Adaptation as Schema Evolves* was presented at ISD'2013, Prato, Italy [107].
- Paper *Evolution of a Relational Schema and its Impact on SQL Queries* was presented at IDC'2013, Prague, Czech Republic [20].
- Paper *DaemonX: Design, Adaptation, Evolution, and Management of Native XML (and More Other) Formats* was presented on iiWAS'2013, Vienna, Austria [108].
- Paper *Data and Query Adaptation using DaemonX* was published in Computing and Informatics Journal'2014 [104]
- Paper *Adapting Service Interfaces when Business Processes Evolve* was presented at IEEE RCIS'2014, Marrakesh, Morocco [75].
- Paper *Undo/Redo Operations in Complex Environments* was presented at ANT'2014, Hasselt, Belgium [64]. Due to a space limitation, this paper is not presented in this thesis.
- Paper *Adaptive Similarity of XML Data* was presented at ODBASE'2014, Amantea, Italy [66].
- Paper *REST API Management and Evolution Using MDA* was presented at C3S2E'2015, Yokohama, Japan [106].
- Paper *Advanced REST API Management and Evolution Using MDA* was presented at DChanges'2015, Lausanne, Switzerland [105].
- Paper *Information System Evolution Management – A Complex Evaluation* was accepted for ECBS'2017, Larnaca, Cyprus [83].

Thesis Outline

In Chapter 2 we present basic models and important formal definitions used in this thesis.

In Chapter 3 we describe *DaemonX* – a general framework used for experimental implementation and experiments of all presented models and transformation algorithms.

Chapters 4, 5, 6, and 7 describe multiple models belonging to different layers in the five-level evolution management framework.

In particular, the purpose of Chapter 4 is to introduce evolution management of XPath queries based on changes done in a related XML schema. We describe a model for XPath queries, its operations and transformation algorithms from PIM to XPath PSM.

In Chapter 5 we present a solution for evolution of SQL queries when the underlaying database schema changes. As in the previous chapter, it contains description of the model, operations and transformation algorithms.

Chapter 6 describes an approach for adapting and optimizing service interfaces when related business process changes. It presents the model representing business processes, operations and algorithms for transformation and a strategy for optimization of underlaying XML schemas.

In Chapter 7 we introduce a model representing a REST API resource and algorithms for its transformation based on changes in the underlaying model.

The purpose of Chapter 8 is to present an approach dealing with similarity mapping of XML schemas to a common PIM schema. It can be integrated with the models, e.g., with the model from Chapter 4.

In Chapter 9 we present a complex example of evolution process of an IS based on real-world data. We model complex situations that, starting from a single point, influence the whole system, e.g., change the database schema, change the producer resource or producer data structures.

To conclude, in Chapter 10 we review all the defined models and algorithms and recall their novel features and contributions.

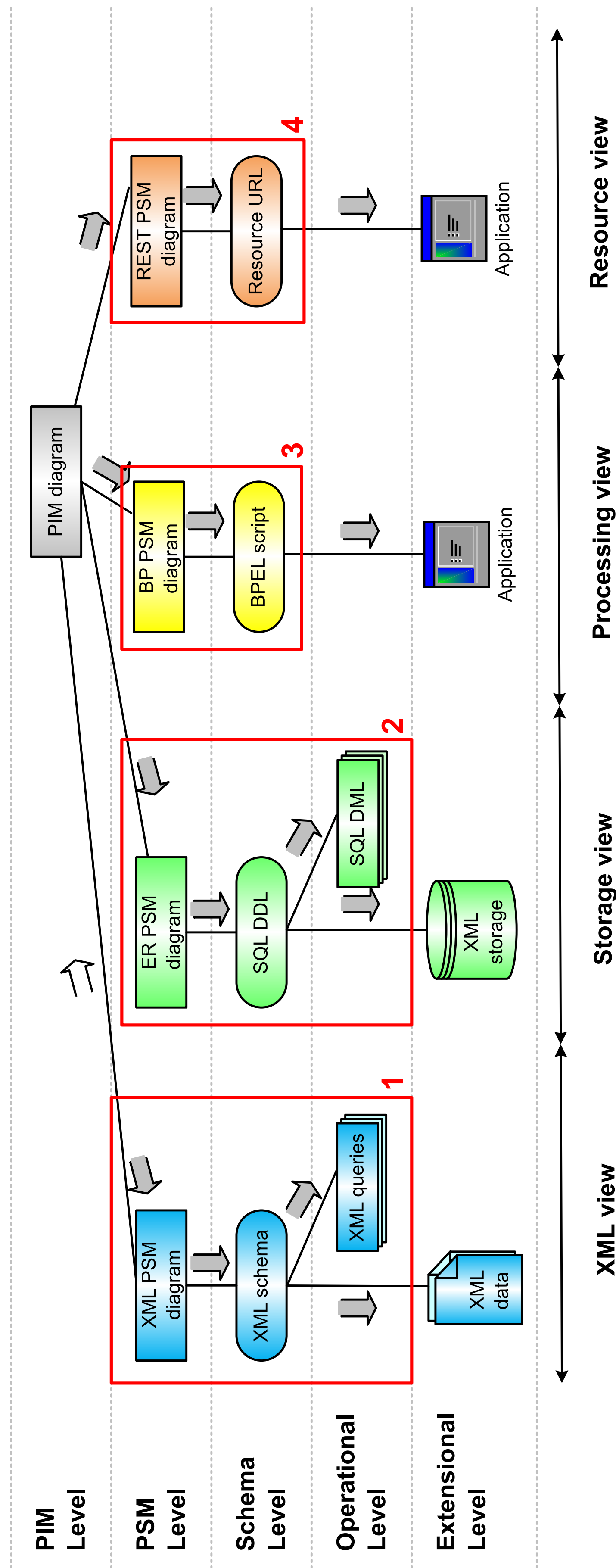

Figure 1.4: Five-level evolution management framework with denoted chapters

2. Preliminaries

The purpose of this section is to provide common definitions, notations and theoretical background used in this thesis. First we define basics of the MDA and its parts. Next we describe terms related to model evolution.

2.1 Models

Definition 3. *(PIM model). Let $M_S = (C_S, I_S)$ be a* PIM *model. C_S is a set of classes, I_S is a set of connections, where $I_k(C_l, C_m)$, $k \in [1, n]$ is a connection between classes C_l and C_m and it has a name I_{k_N}. Each class $C_i \in C_S$, $i \in [1, v]$ has a name C_{i_N}, a set of attributes C_{i_P}, and a set of functions C_{i_F}. Every attribute P_j, $j \in [0, s]$ has a name P_{j_N} and a type P_{j_T}. Every function F_o, $o \in [0, t]$ has a name F_{o_N}, a return type F_{o_T}, and a set of parameters F_{o_R}. Each function parameter R_q, $q \in [0, u]$ has a name R_{q_N} and a type R_{q_T}.*

An example of a PIM model representing structures used in GitHub API [45] is depicted in Figure 2.1.

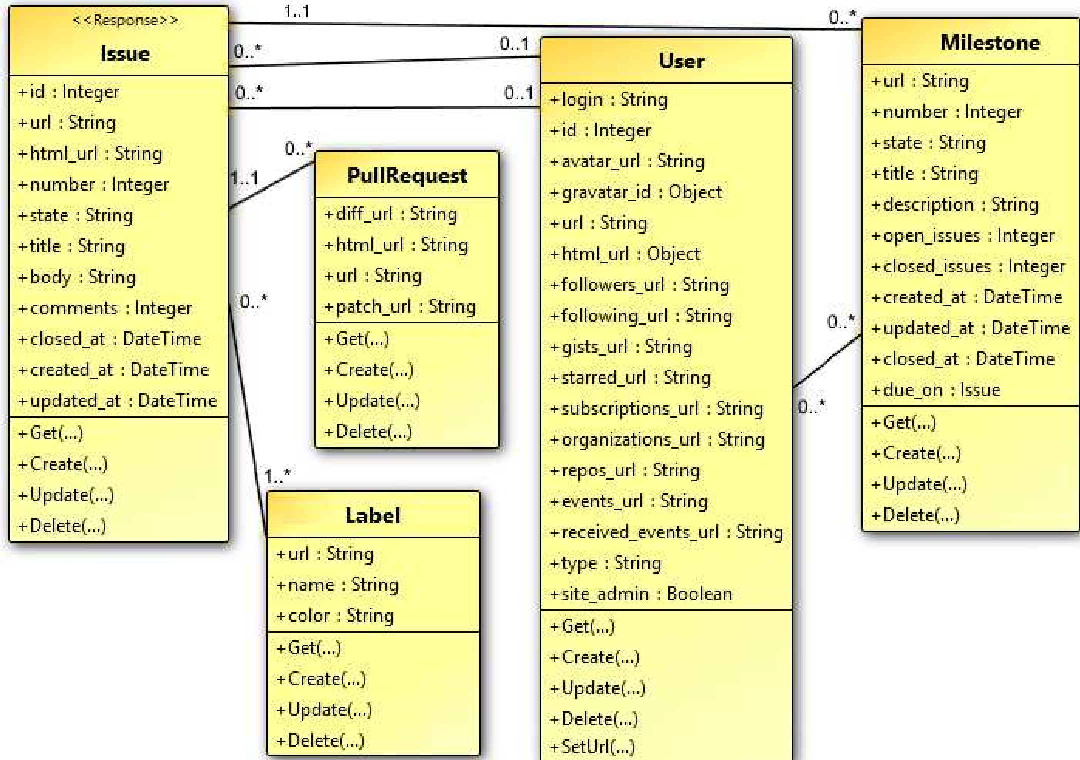

Figure 2.1: PIM model example

Since there exist different PSM models for specific domains, there is no common definition for PSM model. Particular models will be defined separately in the respective chapters.

2.2 Operations

Operations are needed for changing and updating models. There exist multiple types of operations.

Definition 4. *(Atomic operation). An* atomic operation *is the minimum indivisible operation which can be applied on a model.*

Definition 5. *(Composite operation). A* composite operations *consists of one or more atomic operations or other composite operations.*

Atomic operations form a collection. When a composite operation is executed, its atomic operations are executed sequentially. Naturally, composite operations can consist also of other composite operations (eventually combined with atomic operations). Then we use the idea of the collection of operations recursively.

Example 1. *Suppose we have defined the following atomic operations over PIM from Definition 3: operation CreateAttribute(class, name) which creates a new attribute of* name *in* class *and operation DeleteAttribute(class, attributeId) which removes attribute of* attributeId *from* class*. But in the model we want to have the ability to move an attribute from one class to another class. One possibility is to define a new atomic operation MoveAttribute(classFrom, classTo, attributeId). Or we can compose such operation in the following way:*

- *Call operation CreateAttribute(targetClass, nameOfTheAttribute) in the target class to create copy of the attribute in the target class.*
- *Call operation DeleteAttribute(sourceClass, attributeId) in the source class to remove original attribute.*

2.3 Model Relations

To be able to propagate operations (changes) between models, there must be defined relevant relations (mappings) between these models.

Definition 6. *(Relation). Let O_S be a set of items of model S (source model) and O_T be a set of items of model T (target model).* Relation $R(s,t)$ *is a tuple, where $s \in O_S$ and $t \in O_T$.*

Additionally, there can be two types of relations: *single-directional relation* and *bi-directional relation*. In case of single-directional relation, the source model is not related to the target model. Only the target is related to the other member of the relation and can react on actions performed in the source model. On the other hand, when models are in a bi-directional relation, there are two single-directional relations – one from the source model to the target model and second one from the target model to the source model.

Next, there can be specified restrictions if particular parts of different models can be in relation or not. This situation depends on the author of the model or the particular relation. Suppose we want to define relations between two PIM models. Relations which make sense are *Class – Class*, *Attribute – Attribute*, *Method – Method*. But the relation *Class – Attribute* does not.

An example of relations between PIM and PSM (XSEM, defined in Chapter 4) is depicted in Figure 2.2. Using red lines relations between PIM and PSM classes are depicted and using blue lines respective relations between attributes are depicted.

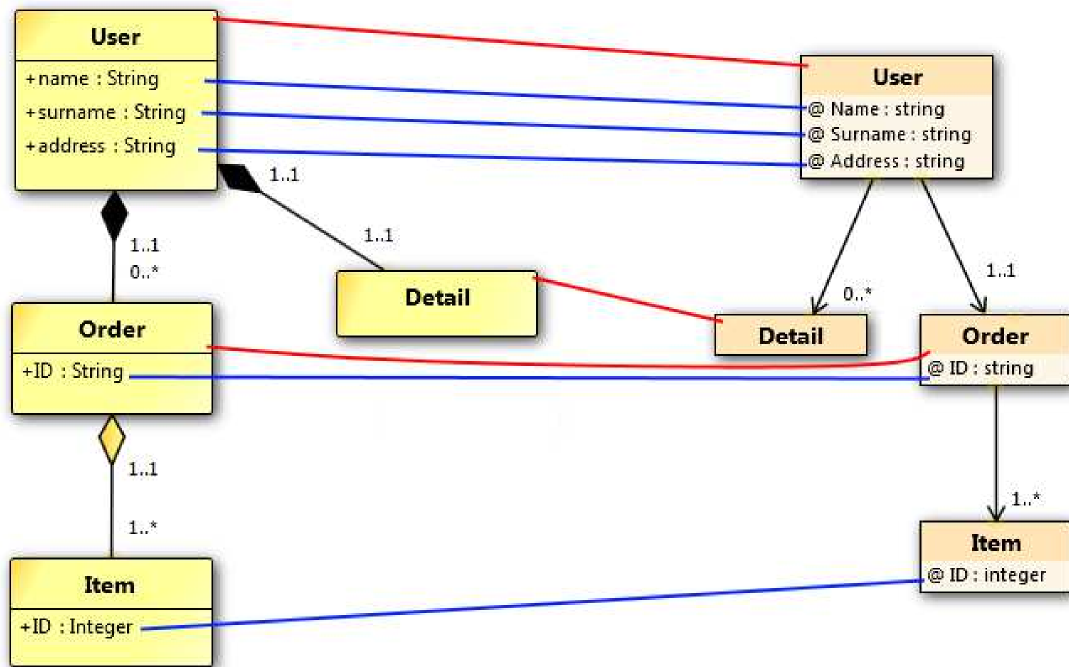

Figure 2.2: An example of relations between PIM and PSM models

2.4 Evolution Process

The *evolution process* is based on analysis of operations done in the source model. The result of the analysis is a collection of operations which must be applied on the target model. It is possible to set these generated operations as source operations for transitive propagation to other models. A simple process diagram is depicted in Figure 2.3.

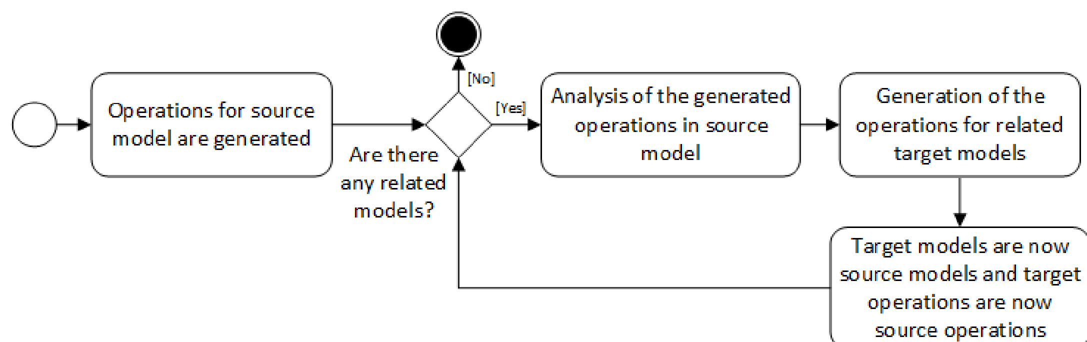

Figure 2.3: Evolution process diagram

Based on the design of the evolution process, models and relations between models, there can exist further restrictions. E.g., to prevent infinite processing, operations can be generated for one model only once in the process. On the other

hand, it is not needed to restrict relations between models. They can form a general graph including cycles.

Example 2. *In Figure 2.4 there is a simple diagram of related models. As we can see, there are relations* Model 1 → Model 3 *and* Model 3 → Model 1 *which form a cycle. If the initial source model is* Model 1*, operations will be propagated to* Model 2*, to* Model 3 *and from* Model 3 *to* Model 4*. To prevent infinite propagation, propagation from* Model 3 *to* Model 1 *is not possible. This situation is depicted in Figure 2.5.*

Another situation will be if the initial model is Model 3*. In this case, operations will be propagated to* Model 1 *and to* Model 4*. Next from* Model 1 *to* Model 2*. Again, to prevent infinite propagation, propagation from* Model 1 *to* Model 3 *is not possible. This situation is depicted in Figure 2.6.*

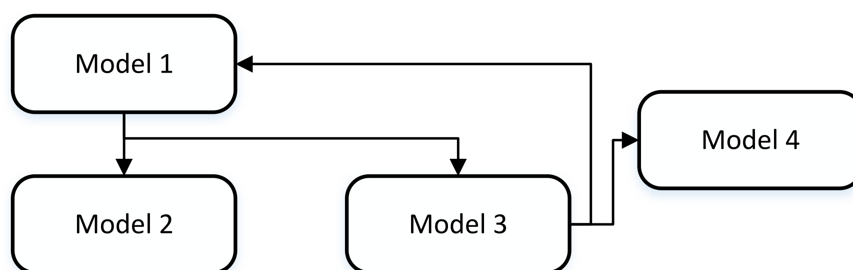

Figure 2.4: An example of related models

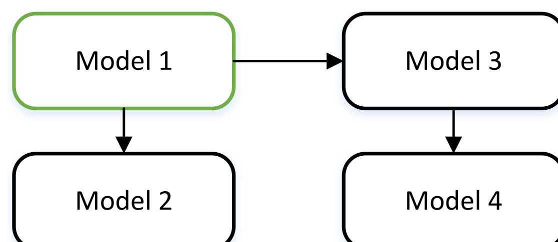

Figure 2.5: An example of propagation from Model 1

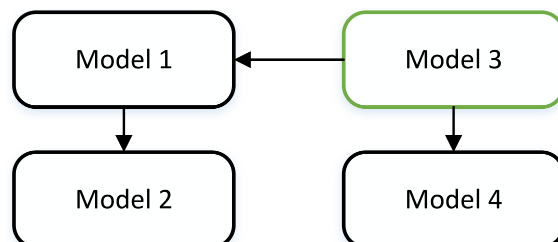

Figure 2.6: An example of propagation from Model 3

3. DaemonX

In this chapter we introduce DaemonX – an evolution management framework, which enables to manage evolution of complex applications efficiently and correctly. Using the idea of plug-ins, it enables to model almost any kind of a data format (currently XML, UML, ER, BPMN and REST). Since it preserves relationships among the modeled constructs, it naturally supports propagation of changes to all related affected parts. We describe the general proposal of the framework and, then, its architecture and implementation. All presented approaches except for the one described in Chapter 8 were implemented in DaemonX as an experimental implementation and proof of the concept. DeamonX was described in detail in papers [108, 104].

3.1 Introduction

The *DaemonX* project is a plug-in-able framework tool for data and/or process modeling. It was developed by the *DaemonX* team at the Faculty of Mathematics and Physics of the Charles University in Prague as a student software project and served as an environment for various experimental implementations. The application was designed to support user-defined plug-ins which define particular functionality needed by the author of the plug-in. These plug-ins are then managed by the application core to provide inter-operability and evolution process between defined models. If we consider Figure 1.3, *DaemonX* covers all five levels of the evolution management framework.

The key abilities of *DaemonX* are as follows:

- Support for user defined plug-ins – for model and evolution process.
- Support for core functionalities for plug-in inter-operability.
- Evolution process management of the models which are defined in user-defined plug-ins.
- Multiple views (diagrams) of the model, i.e., when one model can be displayed in different views for particular purposes.
- Multi-diagram undo/redo management.

DaemonX is currently available with the support for modeling of:

- UML Class diagrams for general data structures.
- PIM model for base models.
- Relational database model for database schema.
- SQL model for SQL database query.
- BPMN models for business processes.
- XSEM model for XML data.
- XPath model for XPath query.
- REST model for REST resource.

3.2 Related Works

In this section the best known and popular Computer-Aided Software Engineering (CASE) tools and Integrated development Environments (IDEs) and their key functionalities are discussed. At the end, a comparison of these tools and *DaemonX* is presented.

3.2.1 eXolutio

eXolutio [71] is a tool based on the MDA approach and models XML schemas at two levels – PIM and PSM. It is a predecessor of *DaemonX* developed as a SW project at the Charles University in Prague. It allows to model in PIM and PSM, interconnect these models and propagate changes between them in both directions. Its key features are as follows:

- Modeling of PIM and PSM of XML schema.
- Schema mapping.
- Model transformation.

3.2.2 Enterprise Architect

Enterprise Architect 13 [118] (EA) is one of the most commonly used commercial UML CASE tool developed by Sparx Systems[1]. It enables to build and model complex diagrams of various types. Next, a particular code can be imported and/or exported to/from these models. E.g., generation of an SQL schema by EA and import of the schema to the EA is possible. It is also possible to extend the tool with multiple plug-ins. The key features of the tool are mentioned as follows:

- Requirements management.
- Business modeling and analysis.
- Simulation.
- System development.
- Test management.
- Visual execution analysis.
- System engineering.
- Data modeling.
- Project management.
- Change management.

3.2.3 Power Designer

Power Designer [115] 16 is another robust enterprise CASE tool for business modeling developed by SAP [2]. As EA, it provides a huge set of functionalities such as UML modeling, code generation, ETL transformation (based on Visual Basic scripts), etc. In the following list the key features of the tool are mentioned:

- Business process modeling.
- Code generation.
- Data modeling.
- Data warehouse modeling.
- Eclipse plug-ins.
- Object modeling.

[1] https://www.sparxsystems.com/

[2] https://www.sap.com/

- Report generation.
- Requirements analysis.
- Model repository.

3.2.4 Eclipse

Eclipse Neon [41] by the Eclipse Foundation [3] is one of the widely used development IDEs with modeling support by default or via plug-ins. It involves:

- Business process modeling.
- Object modeling.
- MDD support.
- Plugin support.

3.2.5 Visual Studio

As a second very popular development IDE with multiple abilities and possible extensions we mention Visual Studio 2015 [24] by Microsoft [4]. It involves:

- Business process modeling.
- Object modeling.
- MDD support.
- Plugin support.

3.2.6 Comparison of the Related Works

In this section we discuss the key features that we defined and demanded from an IDE to fulfil our requirements.

Extensibility by Plug-ins Extensibility is an important aspect of IDEs. Even if it is a robust and mature tool, there can be missing some specific functionality required by the user, but its implementation is not an important feature for the tool provider. An ability to be able to implement this feature itself can solve this problem.

Modeling Abilities An ability to model parts of an application is the main feature of CASE tools we were looking for. Some IDEs support this feature by default or via plug-ins.

Model Transformation Model transformation is also one of the most important features. It enables to transform a model to another model or (source) code. An example can be transformation of the UML class diagram to the *C#* code.

Evolution Support Thanks to this ability it is possible to transform one model based on changes done in another model via defined rules and relations.

[3] `https://eclipse.org/org/foundation/`
[4] `https://www.microsoft.com/`

Open Source An ability to get a source code of an application allows to make more complicated changes that cannot be provided by a plug-in due to some design limitations. An example can be an update of the undo/redo functionality with more advanced features which can impact the core of the tool. But a second aspect must be considered in this situation – the spent time and the complexity of the change versus implementation of a new tool with required functionalities covered in the application design.

Even the presented IDEs are mature and widely used, no of them fulfil all requirements we defined. The aim of *DaemonX* was to define and develop a robust and extensible evolution framework that enables to concurrently model all related parts of the system. Next important requirement was to be able to implement plug-ins for various models and evolution algorithms in a simple way without need of deep knowledge of the tool internals.

Schema Mapping Schema mapping is an important feature during integration of various data sources or systems. Mapping a schema manually is a tedious, error-prone and expensive work. Therefore, automatic schema mapping ability brings significant savings of manual effort and resources.

In Table 3.1 there are displayed particular tools compared by selected criteria.

	Enterprise Architect	Power Designer	Eclipse	Visual Studio	eXolutio	DaemonX
Extensibility by plug-ins	✓	✓	✓	✓	x	✓
Modeling abilities	✓	✓	✓	✓	✓	✓
Model transformation	✓	✓	✓	✓	✓	✓
Open source	x	x	✓	x	✓	✓
Evolution support	x	x	x	x	✓	✓
Schema mapping	x	x	x	x	✓	✓

Table 3.1: Comparison of the related tools

As we can see, abilities such as plug-in extensibility (except eXolutio), modeling ability and model transformation are provided by all mentioned tools. Open source property is not so common, especially for commercial tools. Finally, evolution support and schema mapping are supported by *DaemonX* only. The main

	Enterprise Architect	Power Designer	Eclipse	Visual Studio	eXolutio	DaemonX
PIM (UML)	✓	✓	✓	✓	✓	✓
DB	✓	✓	✓	✓	×	✓
SQL	×	×	×	×	×	✓
XML	✓	✓	✓	✓	✓	✓
XML Query	×	×	×	×	×	✓
REST	×	×	×	×	×	✓
BPMN	✓	✓	✓	✓	×	✓

Table 3.2: Comparison of the related tools models

reasons why *DaemonX* was developed as an environment for various experimental implementations instead of use and/or modification of existing solutions.

Next, in Table 3.2 we compare the mentioned tools according to the available models. As we can see that all tools support PIM (or UML class) model, a model representing database schema (DB), an XML schema model (XML) and a BPMN model. The rest of them is available in *DaemonX* only. Some of the tools allow export from these models, e.g., export of source code in a particular language from PIM (UML class) model or generation of an XSD or DTD from the XML schema model.

3.3 Architecture

The core of *DaemonX* is based on the *Meta-Object Facility* (MOF) approach [52], a standard for model-driven engineering. This architecture pattern gives the ability to define a model at different layers of abstraction. *DaemonX* itself defines the meta-meta-model (*M2 layer* in MOF). This M2 layer model is used by *DaemonX* core as an abstraction over all model plug-ins which application controls. Next, the design of the application uses the *Model-View-Controller* pattern (MVC) [136]. This approach enables the design of a model (e.g., UML) in logically separated parts. The model consists of application data, the view (or multiple views) represents the output of the data for the user, and the controller mediates input and actions and manages the model and the view.

Both these approaches – MOF and MVC – are loosely coupled in *DaemonX*.

M2 Layer

In the M2 layer there are defined *M2 Construct*, *M2 Controller* and other needed structures of this layer like *M2 Property*, which represents a property of the M2 Construct and *M2 Relation*, which is a special structure representing connection between two M2 Constructs (for the full list see the developer documentations [119]). The authors of the particular model plug-in have to inherit from these structures in their model design and give specific behaviors to their model.

M1 Layer

Specific models form the next MOF layer, called *M1 layer*. This layer represents a specific model, e.g., the UML class diagram.

M0 Layer

Finally, an instance of the specific UML class in the diagram is called *M0 layer*. A user of the framework works with instances of this class. The described idea with particular framework elements and an example of UML class diagram model is depicted in Figure 3.1.

3.4 Plug-in Support

As we have mentioned, the strength of *DaemonX* is based on the idea of plug-ins. Basically, the application provides two types plug-ins:

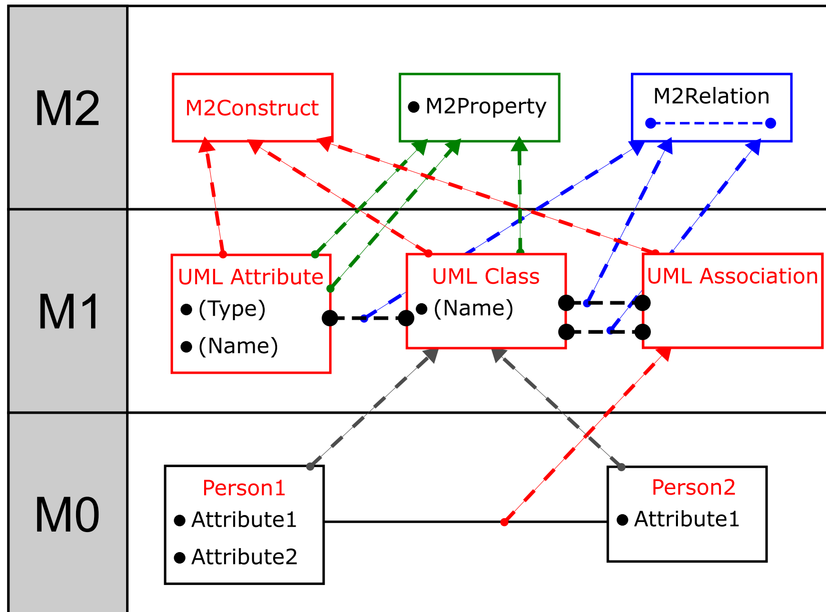

Figure 3.1: Schema of MOF layers and a UML example

- Modeling plug-in.
- Evolution plug-in.

Modeling Plug-in
This type of plug-in defines a specific model which will be used by the designer (for example the UML class diagram model). In the plug-in there are defined all behaviors and operations of the model. There are given no limitations of the plug-in abilities except for *DaemonX* interfaces which the plug-in has to implement.

Evolution Plug-in
The main purpose of this type of plug-in is to support single-directional evolution process between two particular modeling plug-ins. As in the modeling plug-in, there are no limitations of the functionality except for implementation of the required interfaces. The next restriction is that the plug-in is naturally related to two modeling plug-ins, called *source* and *target* modeling plug-ins.

3.5 Evolution Process Management

The next important and novel part of *DaemonX* is the evolution process (described in Section 2.4) which is ensured by the *evolution manager*. Evolution manager controls the evolution plug-ins defining single-directional description of propagation changes from a particular source model to a specific target model. This means that the plug-in knows about all public operations of both model plug-ins.

Next important part defined by the evolution plug-in are so-called *evolution references* (relations) which define how elements from one model can be related to elements from another model (for example that a UML class can be related only to a UML class, but not to a UML class attribute). Hence, there can be defined specific behavior of the evolution process between the models.

The evolution process is based on analysis of the operations done in source modeling plug-ins. The result of the operation analysis is a collection of operations generated by the evolution plug-in which must be processed in the target models. And these changes can be subsequently set as an input of another evolution plug-in. A simple process diagram was depicted in Figure 2.3.

The framework core and the evolution manager are loosely coupled, so its implementation can be easily modified. The current implementation of the manager supports transitive propagation in a tree graph which satisfies that one diagram can be changed by evolution process only once and the process can not get into an infinite loop.

The first release of *DaemonX* contained the basic core framework with experimental implementation of the following model plug-ins:

- *PIM Model* for modeling of the problem domain,
- *XSEM PSM Model* for modeling of XML data,
- *UML Class Model* for modeling general data structures,
- *Relational database schema model* for modeling relational data, and
- *BPMN Model* for modeling business processes.

Also the following evolution plug-ins between existing model plug-ins were involved:

- *UML* $\rightarrow$ *UML*,
- *PIM* $\rightarrow$ *XSEM PSM*,
- *XSEM PSM* $\rightarrow$ *PIM*, and
- *PIM* $\rightarrow$ *relational database schema model*.

In Figure 3.2 and 3.3 there are depicted screenshots of the application and diagrams created in the implemented modeling plug-ins. In Figure 3.2 we can see simple PIM and XSEM PSM models. In Figure 3.3 we can see simple BPMN model, UML class diagram model, and relational model.

Example 3. *In Figure 3.4 and 3.5 there is depicted an example of the evolution process between two related models. In Figure 3.4 on the left we can see a PIM model and on the right its related XSEM PSM model. Parts of both models are connected with references defined by the user. Figure 3.5 presents evolution management between these models. In particular, we moved attribute address from class User to class Details in the PIM model. This change is automatically propagated to the related PSM model, as we can see in Figure 3.5 on the right. Relations between these two models are depicted in Figure 3.6 in a special window called Evolution Manager Window, where relations between models can be viewed and managed.*

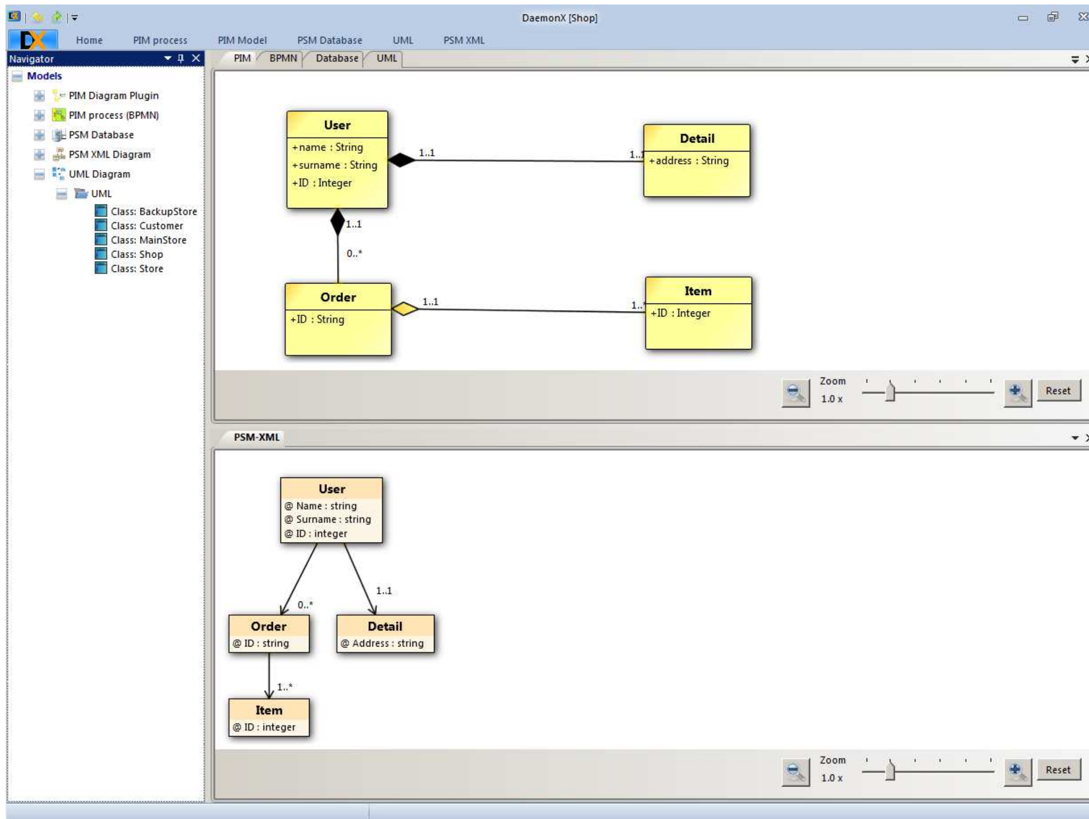

Figure 3.2: A screenshot of *DaemonX* with PIM and XSEM PSM models

3.6 Undo/Redo Management

Another ability of *DaemonX* is that it is fully command-based. All operations which are done by user with the model or view must be defined as commands in the model plug-in. This ensures the ability to provide full undo/redo support by framework part called *undo/redo manager*. As evolution manager, it is also loosely coupled with *DaemonX*.

In Figure 3.7 we can see a screenshot of *DaemonX*. A *command stack* for undo/redo management is situated on the right-hand side. Commands which can be undone are marked with blue color and commands which can be redone are marked with green color.

3.7 Additional Framework Extensions

Since its first release *DaemonX* was significantly extended within multiple papers which enrich the first release application core and/or new add modeling and evolution plug-ins. First, it was practically verified that framework can be used for various models and, second, that introduced algorithms are defined correctly. We will describe selected extensions in more detail in the following chapters.

3.8 Conclusion

The aim of *DaemonX* was to create a robust and extensible evolution framework that enables to concurrently model all related parts of the system (i.e., data

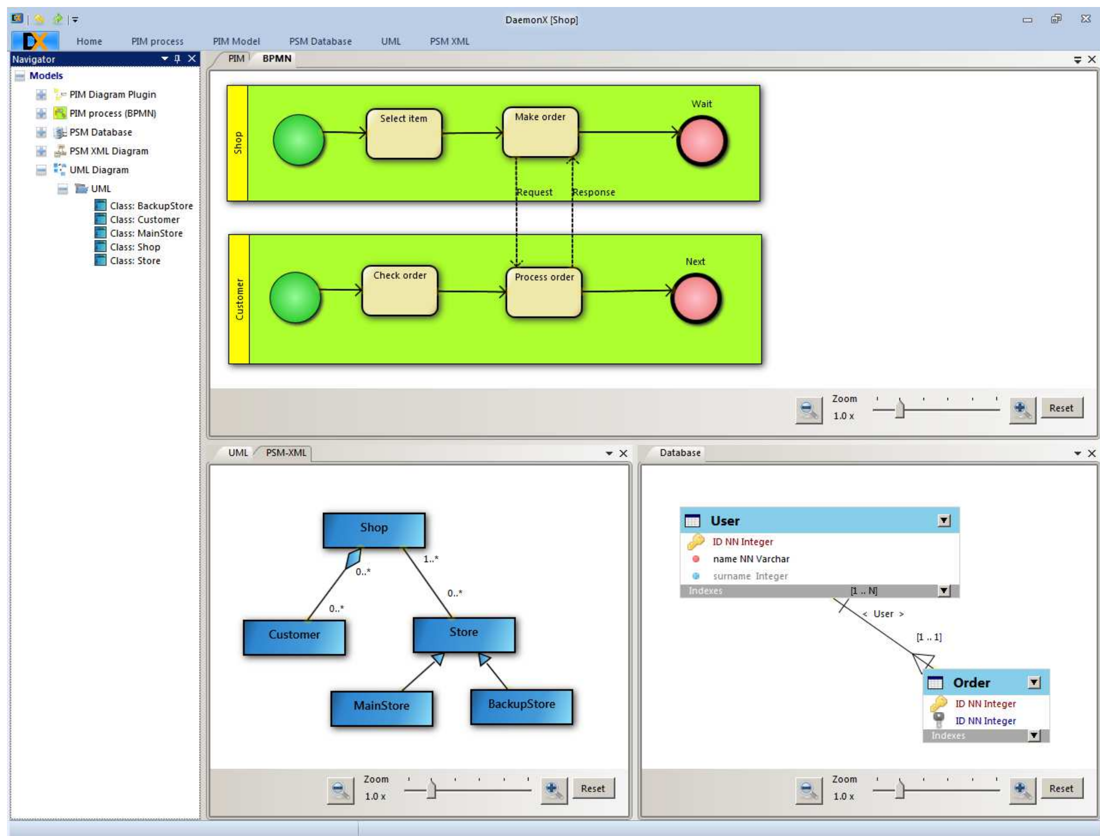

Figure 3.3: A screenshot of *DaemonX* with BPMN, UML class and relational models

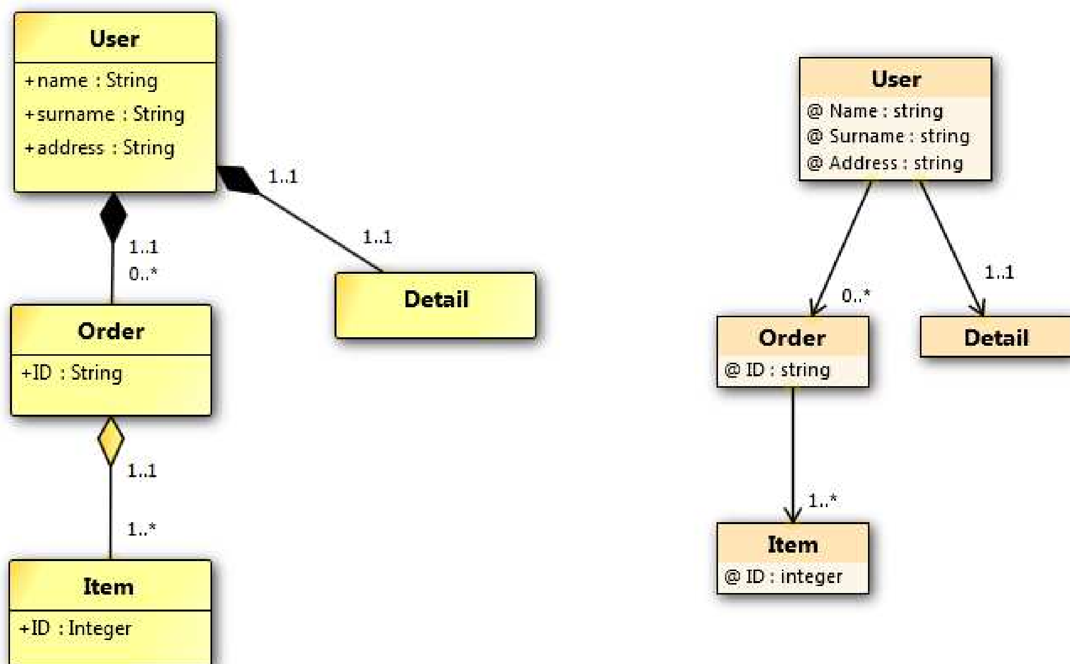

Figure 3.4: An example of evolution process (PIM and XSEM PSM model) – initial state

structures, ICs and operations) as precisely as possible (i.e., with a rich set of constructs), to preserve the relations between system components, and to enable

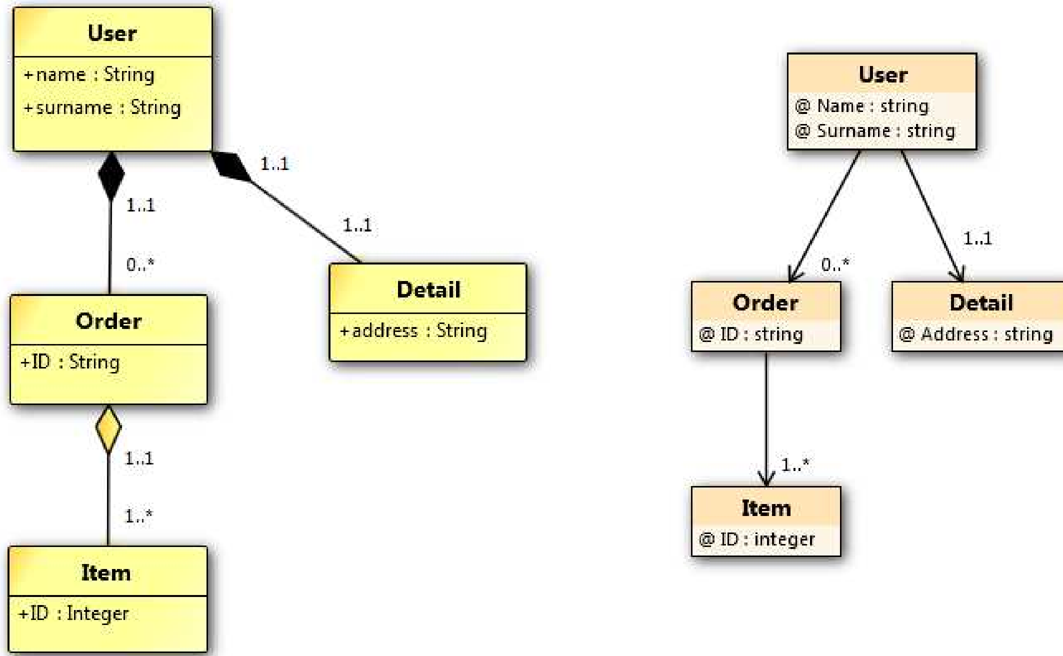

Figure 3.5: An example of evolution process (PIM and XSEM PSM model) – propagation

respective change management (i.e., correct propagation of changes to all the affected parts).

The key contributions of the approach can be summed up as follows:

- In all the cases we bring new ideas and contributions that have not been considered yet in the literature, or have not been integrated in such a complex system.
- Consequently, we provide a robust and open source tool that covers various possible use cases of change management.
- At the same time, using plug-ins the framework is extensible and, hence, can be further adapted for new use cases.

3.8.1 Future Work

- *Support for more complex constructs:* For the purpose of demonstration of the proposed strategies we used only a subset of respective standards (i.e., XML Schema, SQL, XPath, etc.). Hence, a natural following step is to extend the particular plug-ins towards more complex constructs of the standards and possibly their full support. While some of the constructs require only straightforward implementation effort, others require specific algorithm approaches that need to proposed first.
- *Integrity constraints*: In almost all data models we specify not only the structures, but also various integrity constraints, i.e., conditions which further specify the data in terms of logical predicates. Currently the most common language for this purpose is OCL [51]. This topic is partly covered in work [101] which extends *DaemonX* towards integrity constraints evolution management.

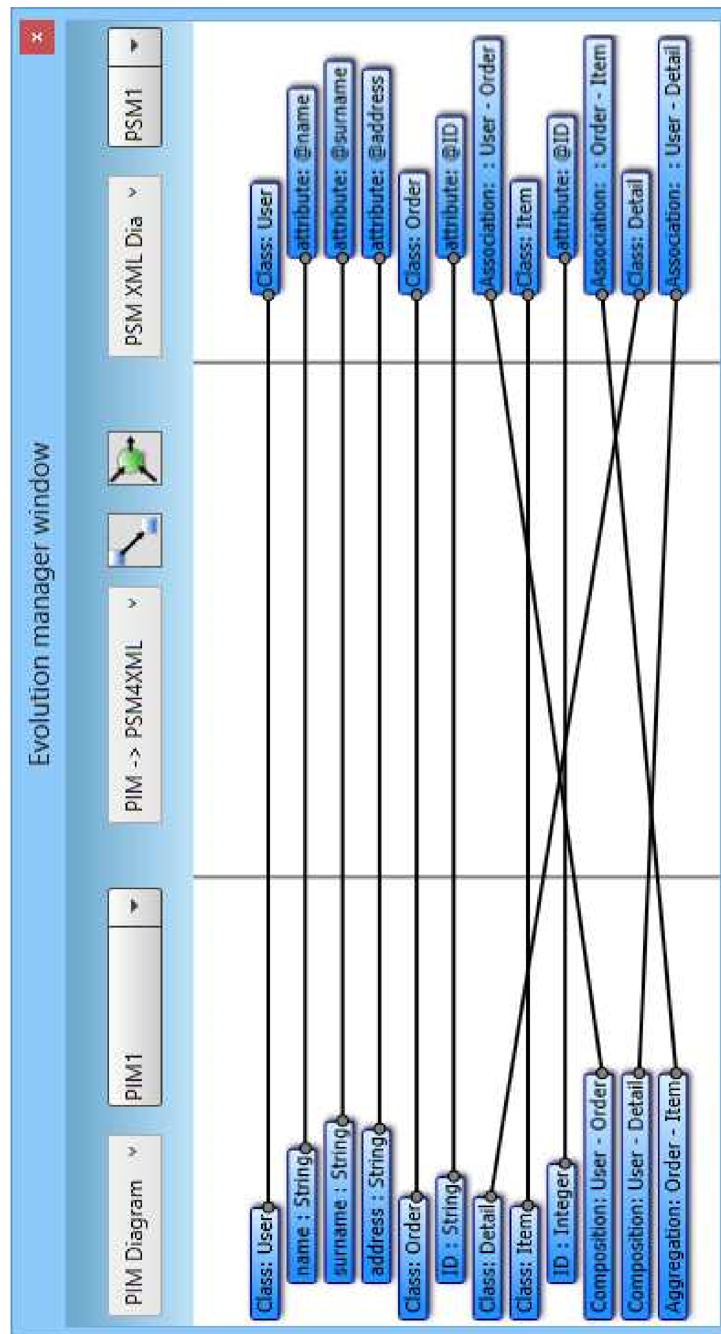

Figure 3.6: A screenshot of *DaemonX* – evolution manager window

- *Storage strategies*: So far we have considered only the data models and respective operations. Even in the storage view we have considered the relational model and the model of SQL queries. However, the modeled data (of any kind) should be somehow persistently stored. And considering this aspect, the change of the data structures may influence also the storage strategies and, consequently, also the efficiency of respective operations (e.g., query evaluation). Hence, we not only need to deal with query consistency, but also query effectiveness.

- *Reverse Engineering*: The five-level framework can be built in two directions – either in the top-down manner starting from the most abstract level PIM or in the bottom-up manner starting from the most concrete extensional

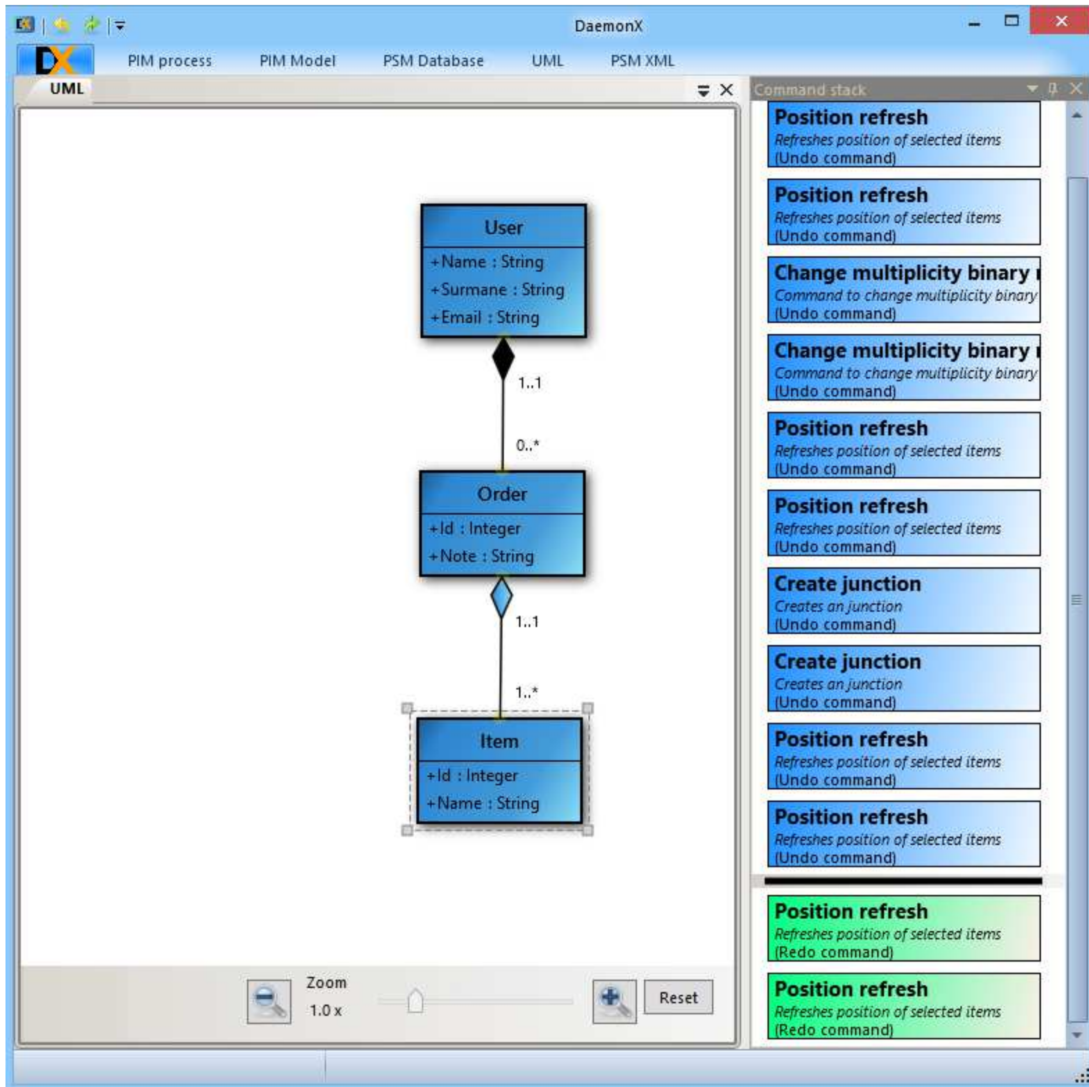

Figure 3.7: A screenshot of *DaemonX* with command stack for undo/redo management

level. Such example is mapping of XML schema presented in Chapter 8. However, in case of other views (data formats) the related problems will probably be quite different and need to be explored and studied with respect to the different context.

4. XML Query Evolution

In this chapter we study the impact of XML schema evolution on related XML queries when the evolution can affect the result of those queries. We provide a novel approach and present preliminary solution to the problem. In our approach, we define changes in the schema and propagate these changes to the queries. We focus on a subset of XPath queries and show how particular changes in data structure can be propagated to them, either automatically or with user interference. The model presented in this chapter is depicted in Figure 4.1 in the context of the five-level evolution management framework. The implementation of our approach is incorporated into DaemonX framework and it enabled us to provide a proof of the concept. The approach was presented in [107].

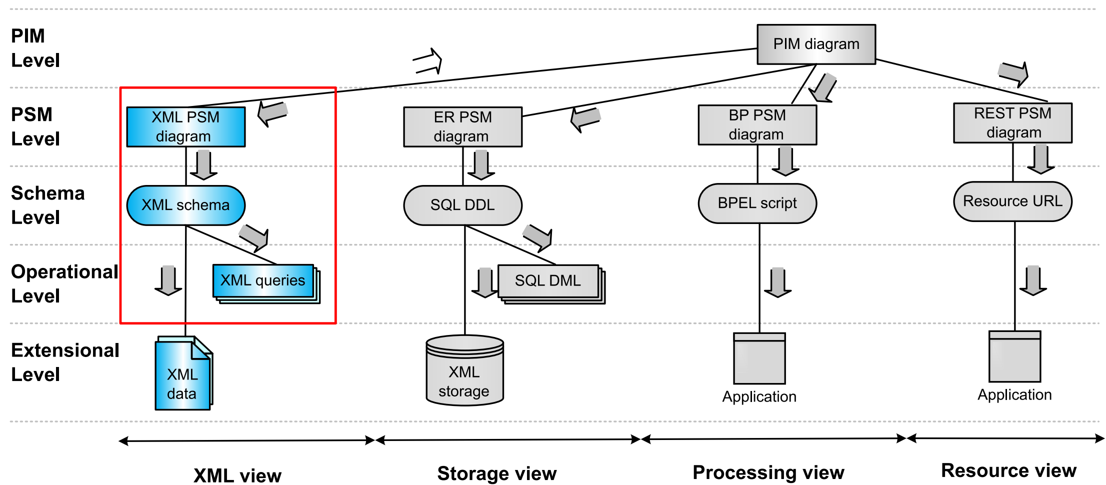

Figure 4.1: Location of the XPath model and change propagation in the context of the five-level evolution management framework

4.1 Introduction

As XML has become a de-facto standard for data representation, there exists a large number of XML-based applications. Since most applications are also dynamic, sooner or later the structure of the data in these applications needs to be changed and other related aspects have to be changed also to preserve consistency. This fact has raised the issue of *evolution* and *adaptability* of XML-based applications. One particular task that has received much attention is the evolution of XML schemas.

Current approaches involve techniques to propagate changes from selected XML schema such as DTD [3] or XSD [87] to the respective data, and vice versa [12]. Other approaches consider changes in an abstraction of a schema through a graphical representation [70, 34], and the propagation of the changes to the data. However we can observe that schema change can cause not only *data inconsistency*, but also *query inconsistency*. The XML queries over the original schema and over the evolved schema may return different results. The result can also be an error caused by a non existing element, as one example. These possibilities are not desirable and should be recognized and corrected.

4.2 Related Works

There exist several approaches focused on query compatibility or analyzing possible changes of the document schema and including recommendations how the schemas should be written to reduce potential changes of the related queries [88]. But we have found no paper discussing how the evolution of the queries should be done (semi)automatically with a minimal contribution of the designer. This section contains an analysis of five most related works. The first one deals with preserving valid queries during schema evolution. The second paper presents a framework for analyzing compatibilities between different versions of schemas. The last one presents a framework for processing evolution of XML schemas.

4.2.1 Preserving XML Queries During Schema Evolution

Paper [88] discusses XML schemas, their evolution and transformation in time and the problems which it brings to administrators. The main aims of the paper are:

- To present taxonomy of possible changes in XML schemas.
- To overview their impact on schema structure.
- To introduce guidelines for managing schema by controlling its changes and writing queries across schema versions.

Taxonomy of XML Schema Changes

The authors examined and divided the changes into two basic groups which are shown in Table 4.1 and described below.

Basic Changes	Complex Changes
Refinement (adds optional or required elements to the schema)	Element composition (groups related elements under a new element)
Removal (deletes elements from the schema)	Element decomposition (ungroups subelements into individual elements)
Extension (adds new constructs to the schema)	Renaming (updates the name of an element or attribute)
Reinterpretation (changes the semantics of an element without changing its structure)	Optionality (changes the participation semantics of an element from optional to required)
Redefinition (updates the schema without changing the document instances format)	Renumbering (changes the element cardinality)
	Retyping (modifies the data type of an element)
	Namespaces (changes the namespace)
	Default values (changes the element)
	Reordering (changes the order of the elements inside a complex type)

Table 4.1: Defined changes

Impact on Queries

Changes of document schema have variable influence on the queries. In this paper possible impacts on queries are divided into the following three groups of schema changes.

1. **General queries**: These are all queries which do not use any element affected by the evolution process. The queries work in both schema versions – original and new one in the same way.
2. **Basic changes**: From the basic changes defined in Table 4.1, only *Removal* and *Refinement* have an impact on queries which can cause inconsistency.
3. **Complex changes**: From complex changes all possibilities can cause an inconsistency of evaluation of queries on new and original schemas.
   (a) **Element composition**: There are two major problems. First, the name of the newly created element can be different from the original element. Second, the query may refer to a specific path in the document tree. So, although there are the data in the document, evaluation of query returns no data.
   (b) **Element decomposition**: As in the previous case if the query refers to a specific path, evaluation of the query can return no data.
   (c) **Renaming**: The change of an element name causes that the result of the query is different or it returns no data.
   (d) **Optionality**: A change from *required* to *optional* is not an issue, but a change in the opposite direction causes that the original query can return more data in the evolved schema version.
   (e) **Renumbering**: Changing a cardinality can cause issues in both directions. A change from a singleton to multiple elements evokes that there can be more possible results. A change from multiple elements to a singleton only reduces the result.
   (f) **Retyping**: In case of retyping it needs checking of the value type and, if needed, the value must be cast to the needed type.
   (g) **Default value**: If the element value is changed, different values can be returned in the evolved schema.
   (h) **Namespaces**: This change can be a problem if the query uses a specific namespace. A different namespace can give different results.
   (i) **Reordering**: Usage of positions in queries can cause that different data is returned if the position of elements changes.

Compatibility of Queries Across Schema Versions

The last section of the paper gives advices or patterns how to write queries to prevent changes in queries while schema evolves and ensure that queries will return correct results.

1. Required elements (or attributes) should not be set in the middle of a sequence of elements. For example all required elements in a sequence should be set at the beginning, followed by optional ones. If necessary, queries should have *ancestor//descendant* axis.

2. Do not delete required elements from the middle of the sequence of elements. A query can contains *exists*() function, which can cause a problem in the evolved schema if position predicate is used in the query.

3. If queries depend on the order of elements, do not change it if not necessary.

4. If queries are strongly typed, do not change the atomic type of the used values.

5. Do not change the name of elements used in queries. If it is necessary, use dictionary of synonyms to map them.

6. If queries are sensitive to namespaces which are changed, a query can return an empty set for a new document which has a different schema. If the requested behavior is to return the same results, the query should be written with $'*'$ for namespace.

7. Functions like *exist* should be used carefully. If the required element is removed, the query will always return *false* in the new XML schema.

Discussion

The paper presents a set of possible changes which can be done through evolution of XML schemas. It classifies them into categories according to their complexity and shows an impact on related queries. As a conclusion the authors give advices how the queries should be written to ensure minimal additional changes in queries while XML schema is being changed. But, it does not give an optimal solution how to provide XML schema evolution without inspection of related queries in all cases.

4.2.2 Identifying Query Inconsistencies with Evolving XML Schemas

Paper [43] discusses a system for monitoring the effect of schema evolution on a set of admissible documents and on the results of queries. An implementation of a framework for automatic verification of properties related to XML schema and query evolution is presented. As a query language the framework considers XPath.

The system is based on a set of predicates which allow for an analysis of a wide range of forward and backward compatibility issues. On the other hand, the system can produce counter examples to prove inconsistency of schemas and queries. The framework was tested with realistic use cases on the real-world data.

Internal Representation

As an internal representation *regular tree type expressions* [43] are used. This representation can capture and convert a schema expressed, e.g., in DTD, XML Schema and Relax NG [21]. The defined tree type expressions are shown in Table 4.2.

τ::=	tree type expression
$\emptyset$	empty set
()	empty sequence
τ — τ	disjunction
τ , τ	conjunction
$l(a)[\tau]$	element definition
x	variable
$let\ \overline{x = \tau}\ in\ \tau$	binder

Table 4.2: Tree type expressions

Logical Formulas

The core of the framework are *logical formulas.* They operate on binary trees with attributes. The framework translates all *unranked trees* [43], which represent XML documents, into *binary trees.*

The semantics of formulas corresponds to $\mu - calculus$ [44] interpreted over finite trees. All XPath expressions defined by the authors can be translated into these logical formulas. A translated XPath expression operates with a binary tree and uses only *forward axes.* Conversion to $\mu - calculus$ is done, because only with this modification the program can solve both XPath emptiness and other decision problems such as *containment.* For this purpose the framework implements a compiler which takes any XPath expression and makes its logical translation.

Definition 7. *(Containment Problem). The* containment problem *takes as an input XPath expressions E and E', asking whether the output of E is contained in the output of E' on any source document at any node.*

Query Representation

The framework is focused on the XPath language which is used in many cases and other standards like XQuery or XSLT [135]. The used semantics of the XPath language is described in [132].

Analysis Predicates

Special predicates and a compiler for them are defined to solve decision problems at a higher level of abstraction. Users can use the predicates to do basic verification like backward or forward compatibility. Within predefined predicates it is possible to create own custom predicates on the basis of defaults by combining them.

An example of a predicate is $backward_inconsistent(\tau, \tau')$ which takes two type expressions as parameters and assumes that τ' is an altered version of τ. This predicate is unsatisfiable if all instances of τ' are also valid against τ.

Framework Evaluation Process

The process of evaluation which is done by the presented framework is as follows:

1. A predicate is provided to the framework. It contains all information for evaluating and returning a result.

2. The given predicate is parsed. The input schema is converted to regular tree expressions and the input XPath query is converted to a logical formula.

3. A satisfiability test which returns either of two possible results is carried out. If both schema versions are compatible, information about this fact is returned. Otherwise a message with a counter example of the inconsistency is returned.

Framework Real-World Use Case Tests

The paper also gives examples of real usage of the presented framework based on checking backward inconsistency between XHTML 1.0 and XHTML 1.1 schema versions. An internal *backward_inconsistent* predicate is used in the test. It takes DTDs of XHTML 1.0 and XHTML 1.1 as parameters. As a result it returns a counter example of an HTML document, which is permitted in XHTML 1.1 schema definition, but prohibited in XHTML 1.0.

Discussion

The presented solution can be used by XML designers to recognize if the query needs to be evolved due to schema evolution. The tool expects a predicate which should be verified and both versions of the schemas. It returns a result of the predicate, or it can return a counter example, which can help designers with facilitating the queries. The solution covers most of the frequently used XML schema languages such as DTD, XML Schema and Relax NG, whose definitions are converted into a common representation of regular-expression tree. The XPath language is translated into logical formulas which allow for validation for the given expression tree.

4.2.3 Transformation of structure-shy programs with application to XPath queries and strategic functions

In paper [27] the authors present an algebraic approach to transformation of declarative structure-shy programs, in particular for strategic functions and XML queries. A structure-shy program specifies type-specific behaviour for a selected set of data constructors only. For the remaining structure, generic behaviour is provided. It allows to program process various data from various sources without knowing its structure and/or need to update program when a new document structure is published. It enables to focus on algorithms, reduces development time and improves understandability, e.g., using XPath for querying.

The other side of this approach is a potentially worse space and time behavior, e.g., a query //*title* should traverse the whole document to find all title elements in document.

The presented approach builds on the pioneering work of Backus [7] and the ensuing tradition of algebraic transformation. Authors introduce various algebraic and structure-shy approaches, e.g., point-free functional programming (a variable-free style of functional programming, on the basis of the ease of formulating and reasoning with algebraic laws over such programs).

XPath Optimizer

The application of these technics is in optimizing of compilations of XPath queries and in query migration in the context of coupled transformation of schemas, documents, queries, and constraints. The authors incorporated the rewrite system into a schema-aware XPath compiler called XPath Optimizer (XPTO) [38]. XPTO receives as input an XML schema and an XPath query. As output it produces an executable file that can be used to run the query against multiple XML documents conforming to the schema. The compilation process has two phases:

1. The schema and the query are parsed into the respective type-safe representations. The query is then optimized. The resulting point-free expression is written to an intermediate Haskell file, together with datatype declarations to represent all XML elements. The main function of this file parses an XML document using the HaXml library, converts it to the respective datatype, applies the optimized query and pretty-prints the results.

2. The intermediate Haskell file generated in the first phase is then compiled using GHC library in order to obtain the desired executable.

Two-level Transformations

The authors extend their previous works for coupled transformations and generalize rewrite systems for point-free program transformations to structure-shy programs. As an immediate consequence, their approach to coupled transformations now also encompasses migration and mapping of structure-shy queries and constraints. In particular, we can use the rewrite systems for structure-shy programs to:

1. Determine whether query q on the original schema A can be re-used as is on the transformed schema A' with meaning.

2. Migrate a structure-shy query q on schema A to a new structure-shy query q' on an evolved schema A'.

The authors conclude with a statement that the presented core fragments of strategic programming and XPath can be used for more complicated languages and rules.

Discussion

The authors present an algebraic approach to transform declarative structure-shy programs. They formulate sets of algebraic equivalences for strategic programs and for the conversion between strategic and point-free programs. They model the core of the XPath language in terms of strategic program combinators,

augmented with universal node type and associated operations. The model relies on generalized algebraic datatypes, rather than type classes. They formulate sets of algebraic equivalences for XPath queries, and for their conversion into strategic and point-free programs. These equivalences allow derivation of static types for dynamically typed queries. Finally, the authors offer a unified framework for point-free, strategic, and XPath transformations, where structure-sensitive, point-free programs are used as the solution space for transformation of structure-shy programs.

4.2.4 Evolution of XML-Based Mediation Queries in a Data Integration System

The authors of paper [77] present an approach to the problem of mediation queries maintenance for data integration systems which adopt the relational model as the common data model. They deal both with the evolution of the user needs and the evolution of the data source schemas. The mediation queries evolution process was developed as part of a data integration system [78], which adopts the global-as-view (GAV) approach. Due to its flexibility to represent both structured and semi-structured information, XML is used by the system as the common language to data exchange and integration.

In the approach, the mediation schema and the data source schemas are defined in the XML Schema language. The XML schema is used to validate the local data returned by the data sources as well as the integrated data returned by the mediator in response of a user query. Although being very useful for these tasks, an XML schema is not suitable for tasks requiring knowledge about the semantics of the represented data. For such tasks, as generation and maintenance of mediation queries, the system needs a high level description. To provide a high-level abstraction for information described in an XML schema they propose a conceptual data model, called X-Entity, that is an extension of the Entity-Relationship (ER) model model. Next, from an X-Entity schema can be generated XML Schema or DTD as described in [78].

To specify modifications performed in the mediation schema or in the local schemas the authors define a set of X-Entity schema change operations.

Mediation Queries Definition

As the mediation schema is represented by an X-Entity schema, the process of mediation queries generation consists of discovering a computing expression for each entity in the mediation schema. At the end of the mediation queries generation process, each mediation entity is associated with a mediation query, which is represented by an operation graph. The operation graph describes all information that is relevant to compute a given integrated view. Another important issue to be considered is that an operation graph can be incrementally created and it can be easily modified.

Data source schemas or user's requirements changes are propagated to the mediation queries through a set of event-condition-action (ECA) rules, which are triggered according to the different schema changes. The propagation process consists of two main tasks: first, the triggering, evaluation and execution of the

rules in order to update the mapping views and the operations among them, and secondly, new mediation queries are generated using the modified operation graphs.

Discussion

The authors present an approach of the process of managing the evolution of XML-based mediation queries. They define a new X-Entity model and schema used for query evolution process where the XML schema is not sufficient. The proposed solution was developed as part of a data integration system which adopts the GAV approach.

4.2.5 Comparison of the Related Works

All presented works study the problem of schema evolution from different points of view and propose solutions how to ensure the compatibility in the system of the schemas and the related data and queries.

The first paper [88] analyzes possible operations which can be used for an evolution of the schema, classifies them according to their complexity and gives advices to designers how to design XPath queries to reduce possibility of query inconsistency while schema evolves.

The second paper [43] presents a complex framework for identifying inconsistencies of the queries and the evolved schemas. They use own internal structures for schemas and predicates for queries. Thanks to this solution they can cover the most frequently used schema languages such as DTD, XML Schema and Relax NG. But no suggestion is given how to evolve related queries automatically.

Paper [70] presents a tool for providing evolution of the schemas and for updating related XML documents by using these schemas. It offers a complex framework which was tested on real-world data similarly to the second paper.

Paper [27] is based on an algebraic approach and transformation of structure-shy programs. The authors formulate sets of algebraic equivalences for XPath queries and present a framework for point-free, strategic, and XPath transformations.

Paper [77] presents a process of managing the evolution of XML-based mediation queries. These queries may be changed due to changes in data sources or in the user's requirements. The approach was developed as a part of data integration system based on the GAV approach.

All the mentioned works present various solutions how to preserve compatibility during schema evolution. This chapter uses some ideas from mentioned papers, put them together and creates a complex solution. The main motivation was to define models representing XPath queries over XML schema and to provide (semi)automatic change management for model evolution analysis and processing. It focuses on the following key topic:

- The definition of a model representing XPath query.
- The relation between an XML schema and an XPath query.
- The analysis of changes done in an XML schema.
- The propagation of changes to preserve compatibility of an XML schema and an XPath query.

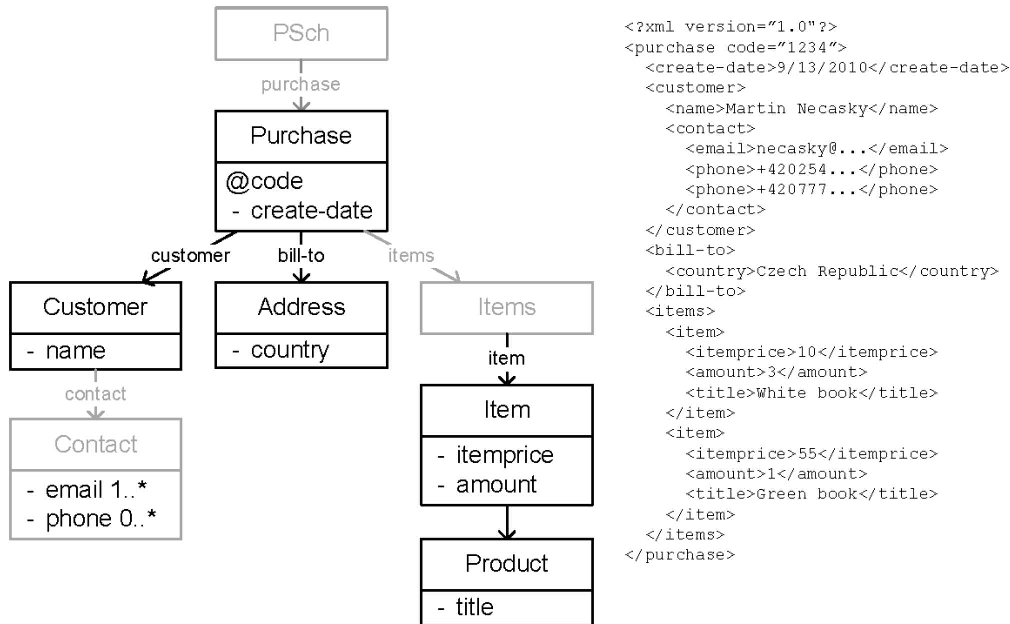

Figure 4.2: Sample PSM schema

4.3 Models for XML Schema and XPath

The PSM of the platform-specific level enables us to specify how a part of the reality modeled at the platform-independent level is represented in a particular XML schema. In addition, the designer works in a UML-style way which is more user-friendly than editing the XML schema. The model we use is called XSEM.

Definition 8. *(XSEM PSM Schema). An* XSEM PSM schema *is an* 8*-tuple* $\mathcal{S} = (\mathcal{S}_C, \mathcal{S}_A, \mathcal{S}_R, \mathcal{S}_E, C_\mathcal{S}, content, class, participant)$*.* $\mathcal{S}_C$*,* $\mathcal{S}_A$*, and* $\mathcal{S}_R$ *are sets of* classes, attributes*, and* associations*, respectively.* $\mathcal{S}_E$ *is a set of* association ends*. An association is an ordered pair* $R = (E_1, E_2)$*, where* E_1*,* E_2 *are different association ends. Any two associations are disjoint.* $C_\mathcal{S} \in \mathcal{S}_C$ *is a* schema class *of* $\mathcal{S}$*. Function content assigns a class* C *with an ordered sequence of all associations with* C *as the parent.*

An XSEM PSM schema is displayed as a UML class diagram in an ordered tree layout which reflects the hierarchical structure of XML data. Note that we omit names, types and cardinalities from the definition for simplicity. We do not cover all the schema constructs – they are covered in the full definition of the model [91]. An actual XML schema can be automatically generated from our PSM schema and vice versa. A sample self-explanatory XSEM PSM schema is depicted in Figure 4.2.

For the purpose of evolution of XPath queries related to XML schemas, there must exist a mapping between an XML schema and an XPath query. Since the full XPath syntax is too extensive, we use its subset based on the *Positive Core XPath* [58] with some modifications. Our syntax at the current stage does not consider predicates and we add the operator except to the definition.

Definition 9. *(Updated Positive Core XPath). Let* $\boldsymbol{X}$ *denote a* location path *and* $\boldsymbol{A}$ *represent an* axis*. Then* updated positive core XPath *is defined as:*

$\boldsymbol{X} \equiv \boldsymbol{X}|\boldsymbol{X} \parallel /\boldsymbol{X} \parallel \boldsymbol{X}/\boldsymbol{X} \parallel (\boldsymbol{X}) \parallel \boldsymbol{X}\ except\ \boldsymbol{X} \parallel \boldsymbol{A} :: \boldsymbol{L}$
$\boldsymbol{A} \equiv self \parallel child \parallel parent \parallel descendant \parallel ancestor \parallel preceding \parallel following \parallel descendant-or-self \parallel ancestor-or-self \parallel preceding-sibling \parallel following-sibling$
where $\boldsymbol{X}$ *denotes* location path *and* $\boldsymbol{A}$ *represents an* axis.

As we can see, the only one node test is possible – *name test*, denoted **L**. The original Positive Core XPath definition contains predicates, but it can only be used to test for element/attribute occurrence. A query using predicates can be rewritten to a query without them and still returns the same *result set* [16]. This solution has only one problem – the query is transformed to a complex form not transparent for the designer at the first sight. In the defined syntax, it is also possible to use all classical XPath abbreviations for axes, such as "∗" for all child elements, " " for the child axis, "." for the self axis, ".." for the parent axis and "//" for $/descendant - or - self :: node()/$.

To be able to map an XSEM PSM diagram to an XPath query, an *XPath model* must be defined. We propose a model that follows ordered tree structure of the XPath query, it results from the presented syntax, and it visualizes its textual representation. The components of the model can be divided into two parts – nodes which represent nodes in the location path and edges that represent axes. An edge and a node together comprise a *location step* of the XPath query. The model contains the following components:

- **Node** (E) representing node test, or name test if name is specified
- **Axes** child (L_{ch}), descendant (L_d), descendant-or-self (L_{dos}), parent (L_{pa}), ancestor (L_a), ancestor-or-self (L_{aos}), following (L_f), following-sibling (L_{fs}), preceding (L_{pr}), preceding-sibling (L_{prs}) and self (L_s)
- **Expression node** (E_{ex}) representing *disjunction* (denoted by '|') and *except* operators. Its first output edge represents the first part of the expression, the second output edge represents the second part of the expression. The third edge represents the following part of the query in the sense of: $(first_expression\ operator\ second_expression)/third_expression$

Definition 10. *(XPath Model). An* XPath model P *is a directed graph* $P = (P_E, P_A)$*, where* P_E *is a set of nodes or expression nodes and* P_A *is a set of axes. For the XPath model it must hold that it is a tree. Every node* E_i *has a name* E_{i_N}*. An axis is an ordered pair* $A = (E_i, E_j)$*, where* E_i*,* E_j *are different nodes.*

Nodes and axes are visualized in Figure 4.3; an expression node is visualized in Figure 4.4. In particular, we use the notation we have proposed in [103] and implemented in [119].

Since the XSEM PSM schema has a tree structure and the XPath query follows a tree structure, it is straightforward and natural to map XSEM PSM to a location path. An example is shown in Figure 4.5; its formal definition is provided in [103]. As we can see, an axis can intervene not only a single node in the schema tree, but also a part of a tree. We say that the part of the tree is *hit* by the location step. When the schema evolves, the query is gradually evaluated and the hit parts are compared with the previous version. If a difference is discovered, the evolution algorithm is executed.

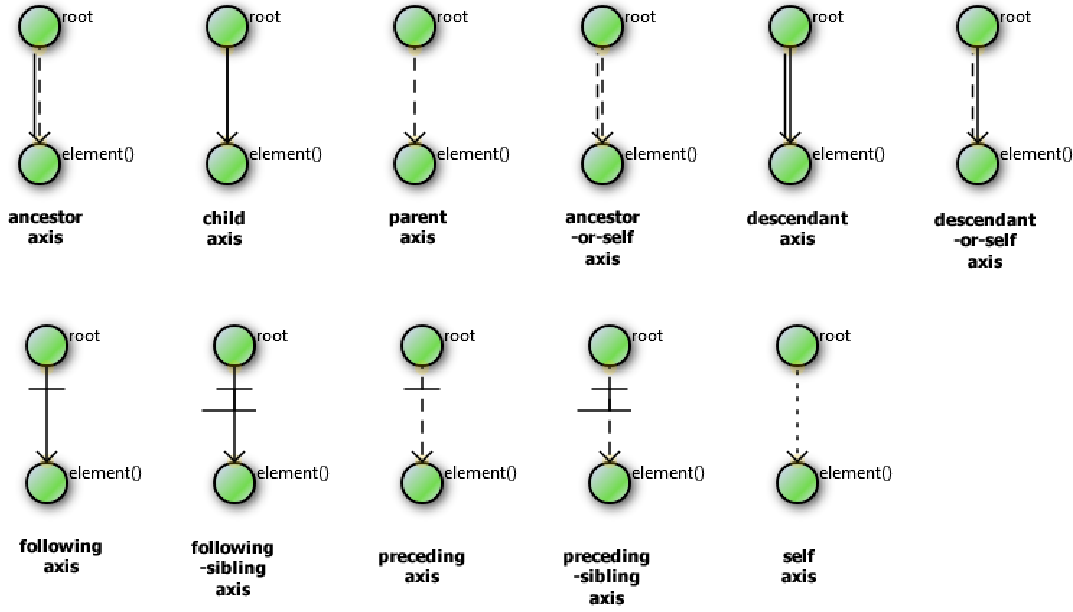

Figure 4.3: XPath axes and nodes

4.4 Evolution Algorithm

Every change in the source XSEM PSM can cause changes in multiple location steps of the XPath model. Furthermore, there can be done multiple changes in the PSM model step by step. All operations which change the source XSEM PSM schema are atomic. We use the subset of operations for query adaptation identified in [72] (see Section 4.4.1). Naturally, from the atomic operations any composite and more user-friendly operations can be created.

4.4.1 Operations for the XSEM Model

First, we need to define operations for the XSEM model:

- **Root Class Adding** ($\alpha_C : (S, C_i) \rightarrow S'$): The operation adds root class C_i as a root in model S. It returns model S', where $S'_C = S_C \cup \{C_i\}$.

  Precondition: Class name C_{i_N} must be set (eventually to a default value).

- **Class Adding** ($\alpha_C : (S, C_i, C_j) \rightarrow S'$): The operation adds class C_i as a child of class C_j in model S. It returns model S', where $S'_C = S_C \cup \{C_i\}$. This operation creates an association A_k between classes C_j and C_i – $S'_A = S_A \cup \{A_k\}$.

  Precondition: Class name C_{i_N} must be set (eventually to a default value).

- **Class Removing** ($\delta_C : (S, C_i) \rightarrow S'$): The operation removes class $C_i \in S_c$ from model S. It returns model S', where $S'_C = S_c \backslash \{C_i\}$. If there exists an association $A_k(C_j, C_i) \in S_a$ for a $C_j \in S_C$, it is first removed. So, then the operation returns model S', where $S'_C = S_C \backslash \{C_i\}$ and $S'_A = S_A \backslash \{A_k\}$.

  Precondition: Class C_i must exist in model S. There must not exist an association $A_k(C_j, C_i) \in S$ for any $C_j \in S_A$, i.e., C_i must be a *leaf* node.

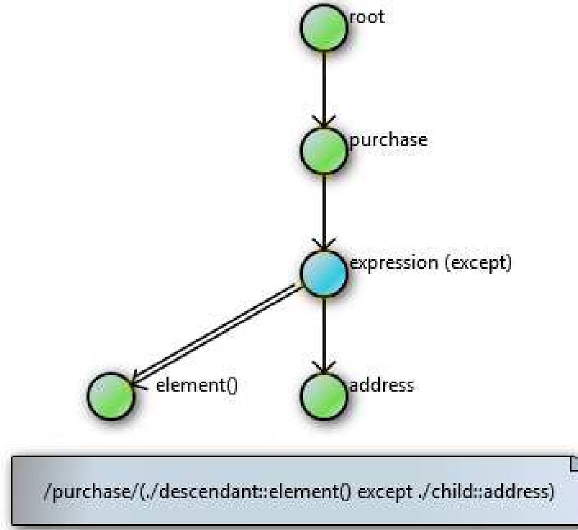

Figure 4.4: XPath expression node

- **Association Removing** ($\delta_A : (S, A_i) \rightarrow S'$): The operation removes association A_i from model S. It returns model S', where $S'_A = S_A \backslash \{A_i\}$.

  Precondition: Association $A_i(C_i, C_j)$ must exist in model S. Either classes C_i or C_j must be a *leaf* node, i.e., there must not exist an edge outgoing from C_i or C_j.

- **Class Renaming** ($\gamma_C : (C_i, m) \rightarrow C'_i$): The operation returns class C'_i, where $C'_{i_N} = m$.

  Precondition: Class C_i must exist in model S.

- **Class Moving** ($\psi_C : (C_i, d)$): The operation moves class C_i to the left or to the right in the sequence of its siblings.

- **Class Reconnection** ($\mu_C : (C_i, C_j)$): The operation reconnects class C_i as a last child of class C_j.

4.4.2 Operations for the XPath Model

Subsequently we established similar set of operations for the XPath model:

- **Root Node Adding** ($\alpha_E : (X, E_i) \rightarrow X'$): The operation adds root node E_i as a root in model X. It returns model X', where $X'_E = X_E \cup \{E_i\}$.

  Precondition: Node name E_{i_N} must be set (eventually to a default value).

- **Node Adding** ($\alpha_E : (X, E_i, E_j) \rightarrow X'$): The operation adds node E_i as a child of node E_j in model X. It returns model X', where $X'_E = X_E \cup \{E_i\}$.

  Precondition: Node name E_{i_N} must be set (eventually to a default value).

- **Node Removing** ($\delta_X : (X, E_i) \rightarrow X'$): The operation removes node $E_i \in X_E$ from model X. It returns model X', where $X'_E = X_E \backslash \{E_i\}$. If there exists an axis $A_k(E_j, E_i) \in X_A$ for a $E_j \in X_E$, it is first removed. So, then the operation returns model X', where $X'_E = X_E \backslash \{E_i\}$ and $X'_A = X_A \backslash \{A_k\}$.

  Precondition: Node E_i must exist in model X.

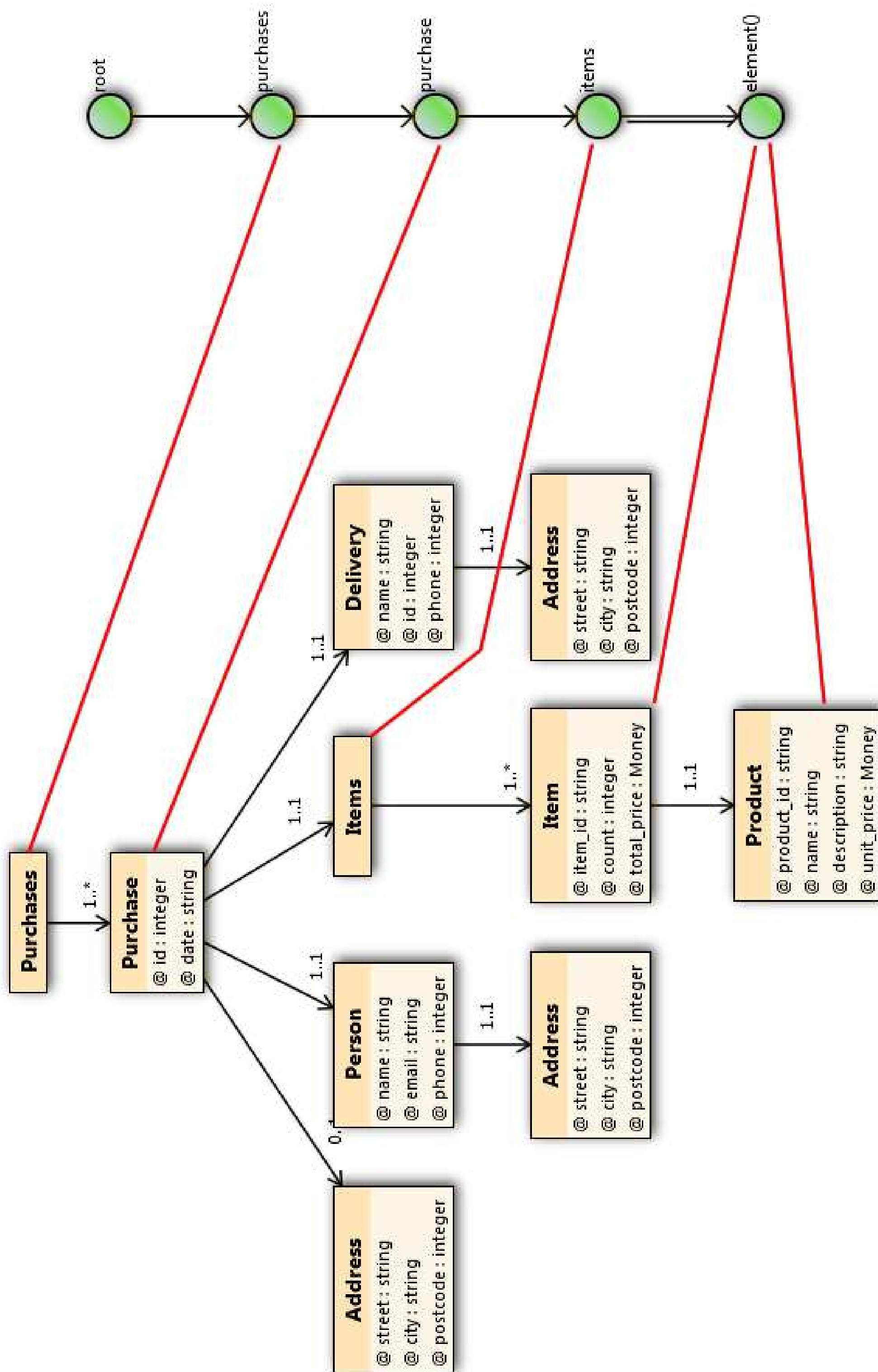

Figure 4.5: Mapping between XSEM and XPath models

- **Axis Edge Creating** ($\alpha_E : (X, A_i, E_j, E_k) \to X'$): The operation creates axis edge $A_i(E_j, E_k)$ between nodes E_j and E_k in model X. It returns model X', where $X'_A = X_A \cup \{A_i\}$.

  Precondition: Nodes E_j and E_k must exist in model X.

- **Axis Edge Removing** ($\delta_A : (X, A_i) \to X'$): The operation removes axis edge A_i from model X. It returns model X', where $X'_A = X_A \backslash \{A_i\}$.

  Precondition: Axis edge $A_i(E_i, E_j)$ must exist in model X.

- **Node Renaming** ($\gamma_E : (E_i, m) \to E'_i$): The operation returns node E'_i, where $E'_{i_N} = m$.

  Precondition: Node E_i must exist in model X.

Formally, let Q be the original query over the original schema S, Q' be the adapted query over the evolved schema S', $R = Q(S)$ be the result set of Q over S and $R' = Q'(S')$ be the result set of Q' over S'. Let AO_{XSEM} be an atomic operation done in an XSEM PSM schema (from list in Section 4.4.1) which should be propagated and let OS_{XPath} be a sequence of atomic operations in an XPath model which was generated from AO_{XSEM} to preserve the same results of the queries with the original and the new XSEM PSM schema. Then $R = R' = Q(S) = Q'(S') = OS_{XPath}(Q)(AO_{XSEM}(S))$ if there exists an appropriate propagation algorithm which generates OS_{XPath}.

In the following cases, we will use some simplifications. (For the full description see [103].) We will consider changes with a single class corresponding to element x (to be added, deleted, etc.). Also, if not specified otherwise, all considered elements are in a *sequence* element. In all presented situations we suppose that there exist no two sibling elements of the same name in S and in S'. In the description we will use functions with self-explanatory names, such as *parent()*, *descendant()*, *absolute_path_to()*, *absolute_path_to_previous_sibling()* etc.

In the following text we will consider cases when query consistency is violated (i.e., $R \neq R'$) and Q' needs to be adapted accordingly. Since each query can be divided into separate location steps, we can consider only one location step of the query Q.

4.5 Analysis of Propagation of Operations

In this chapter we analyze operations done in the XSEM PSM model and their propagation to the XPath model.

4.5.1 Adding

This operation adds element x as a child of an existing element in S. In the current location step we consider context element $p \in S$.

- **Ancestor, Ancestor-or-self, Parent axes**: Since x can be added only as a child element (see Section 4.4.1), adding x as ancestor/self/parent of p to the root will be solved in another location step.

- **Child axis**: If $Q = p/child :: *$, then $Q' = p/child :: *\ except$ $absolute_path_to(x)$ (see Example 4).
- **Descendant, Descendant-or-self axes**: If $Q = p/descendant :: *$, then $Q' = p/descendant/ :: *\ except\ absolute_path_to(x)$.
  Note: This modification is possible only if there exists no sibling element $q \in S'$, s.t. $name(q) = name(x)$. Otherwise, we should use function $position$ which in combination with different values of $minOccurs$ disallows precise selection of x. Therefore, we assume no sibling elements in S and S' with the same name.
- **Following, Following-sibling axes**: If $Q = p/following :: *$, then $Q' = p/following :: *\ except\ absolute_path_to(x)$.
- **Preceding, Preceding-sibling**: If $Q = p/preceding :: *$, then $Q' = p/preceding :: *\ except\ absolute_path_to(x)$.
- **Self**: New added element cannot cause a change in a result of the self axis. If the adding operation cause a change, it will be detected by another axis.

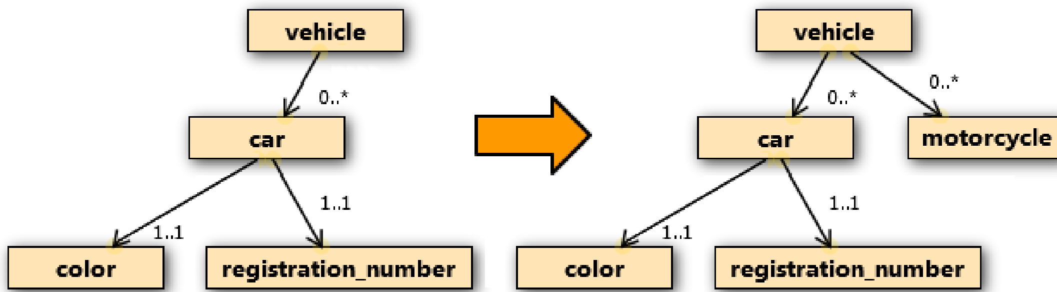

Figure 4.6: Schema example for adding

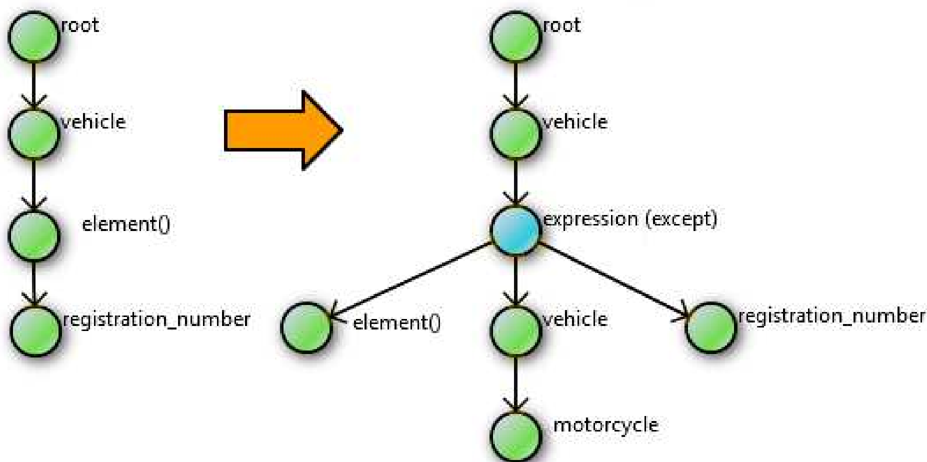

Figure 4.7: Query example for adding

Example 4. *Consider S in Figure 4.6 on the left and XPath model of $Q = /vehicle/child :: */registration_number$ in Figure 4.7 on the left. If element motorcycle is added as a child of element vehicle (see Figure 4.6 on the right),*

sub-query $/vehicle/child :: *$ *will return all elements including motorcycle. Hence, the location step is updated from* $child :: *$ *to* $child :: *\ except\ /vehicle/motorcycle$ *and* $Q' = /vehicle/(child :: *\ except\ /vehicle/motorcycle)\ /registration_number$. *The model of* Q' *is shown in Figure 4.7 on the right.*

4.5.2 Removing

This operation removes element x from S. According to operation definition, the removed element must be a leaf of the schema tree. If we want to remove a whole sub-tree, it can be done by its iteration. Let $p = parent(x)$, the particular cases are discussed in following list.

- **Ancestor axis**: If $Q = x/ancestor :: *$, then $Q' = absolute_path_to(p)/ ancestor - or - self :: * \mid x/ancestor :: *$.
  If the removed element is not x, it is solved in another location step.
- **Ancestor-or-self axis**: If $Q = x/ancestor - or - self :: *$, there is no possibility to preserve query compatibility, because $R' \subset R$. In other cases, no update is needed.
- **Child, Descendant axes**: If $Q = x/child :: *$ and x is a leaf node, then $R = \emptyset$.

  If $Q = p/child :: *$, it is not possible to preserve query compatibility, because the removed element x cannot be hit.
- **Following axis**: If there are more occurrences of element x (i.e., elements with the same name) in S and x is hit, then:
  If x is not the only child of p and not the last one, then $Q = x/following :: *$ is updated to $Q' = x/following :: * \mid absolute_path_to_next_sibling(x)/ (descendant - or - self :: * \mid following :: *)$.

  If x is the only child of p or the last one with this name in S, then $Q = x/following :: *$ is updated to $Q' = x/following :: * \mid absolute_path_to_next_element_right(x)/\ (descendant - or - self :: * \mid following :: *)$.
  Note: Next element right is the first element returned by the following axis.

- **Following-sibling axis**: If x is not the only child of p, then $Q = x/following-sibling :: *$ is updated to $Q' = absolute_path_to_next_sibling(x)/\ (self :: * \mid following - sibling :: *) \mid x/following - sibling :: *$.

  If x is the only child of p, then $R = \emptyset$.

  If the removed element is not x, but is hit by the following-sibling axis, it is not possible to preserve query consistency.
- **Preceding axis**: A situation symmetric to the following axis.
- **Preceding-sibling axis**: A situation symmetric to the following-sibling axis.

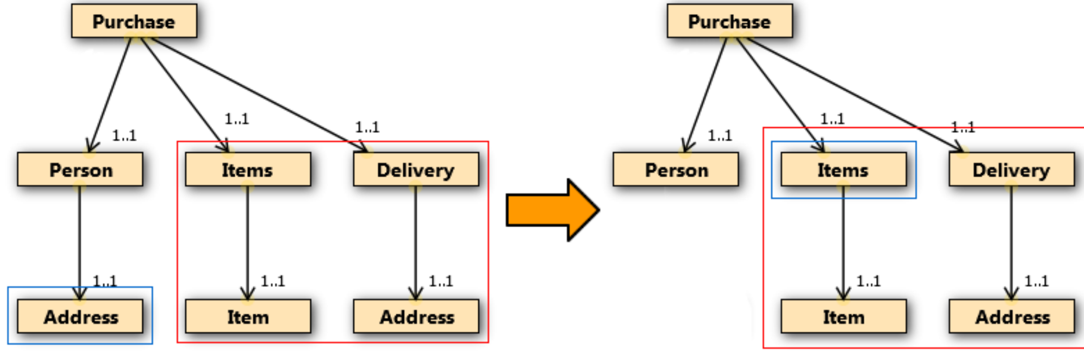

Figure 4.8: Schema example for removing

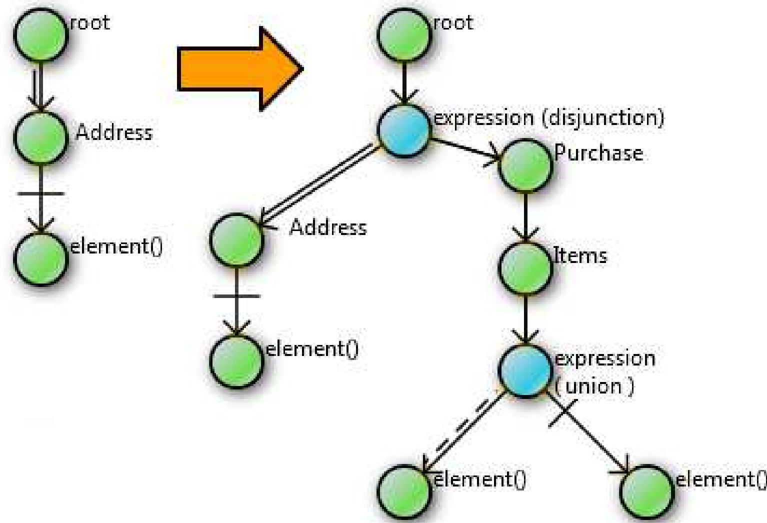

Figure 4.9: Query example for removing

- **Parent axis**: $Q = x/parent :: *$ is updated to $Q' = absolute_path_to(p)/self :: * \mid x/parent :: *$.
- **Self axis**: $Q = x/self :: *$ cannot be updated, since x does not exist any more.

Example 5. *Consider S in Figure 4.8 on the left and $Q = //Address/following ::$ $*$ in Figure 4.9 on the left. If element $/Purchase/Person/Address$ is removed (Figure 4.8 on the right), Q is updated as depicted in Figure 4.9 on the right.*

4.5.3 Renaming

Renaming operation changes the name of a selected element x. A change of the name can cause a change of the result set R. Possible situations are similar for all axes, so we do not provide the respective list. Update of Q must be done only if x is hit by the name test. If the location step uses name test with $*$ or $element()$, no change is needed. There are two cases how the change of the name can affect the result of the query. Let the new name of x be y, then:

- If more elements are in the result set ($R \subset R'$), the location step must be extended with *except* $absolute_path_to_element(y)$

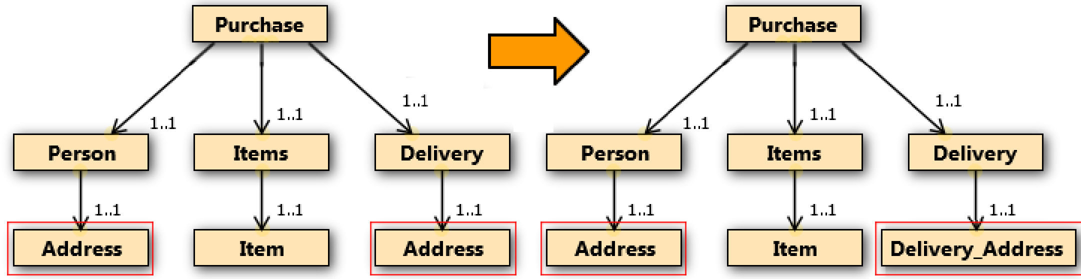

Figure 4.10: Schema example for renaming

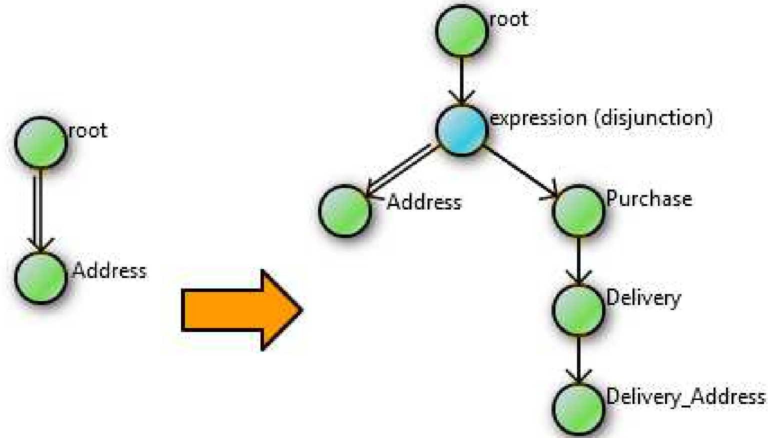

Figure 4.11: XPath example for renaming

- If less elements are in the result set ($R' \subset R$), the location step must be extended with $|$ $absolute_path_to_element(y)$ (see Example 6)

Example 6. *Consider S in Figure 4.10 on the left. In the red rectangles there are classes returned by $Q = //Address$ shown in Figure 4.11 on the left. If the name of class $/Purchase/Delivery/Address$ is changed to $Delivery_Address$, then $Q' = //Address \mid /Purchase/Delivery/Delivery_Address$ depicted in Figure 4.11 on the right.*

4.5.4 Reordering

Now we suppose that an element x is in a *sequence* where the order of elements is significant and that element x has at least one sibling. It can be moved only one position left or right in one step (see Section 4.4.1). Again, its iteration can provide various reordering of the whole sequence. Let $pos(x)$ be the position of x in S within its siblings and $pos'(x)$ be the position of x in S' within its siblings. Consider elements x, y, z, p such that $parent(x) = parent(y) = parent(z) = p$, $pos(y) = pos(x) + 1$ and $pos(z) = pos(x) - 1$.

- **Self, Child, Parent, Ancestor, Ancestor-or-self, Descendant, Descendant-or-self axes**: Change of the position of element x has no impact on the result.

- **Following axis**: If the position of x is changed to $pos'(x) = pos(x) + 1$, then $Q = x/following :: *$ is updated to$Q' = x/following :: * \mid absolute_path_to(y)/descendant - or - self :: *$ (see Example 7).

  If the position of x is changed to $pos'(x) = pos(x) - 1$, then $Q = x/following :: *$ is updated to $Q' = x/following :: *\ except\ absolute_path_to(z)/descendant - or - self :: *$.

- **Following-sibling axis**: If the position of x is changed to $pos'(x) = pos(x) + 1$, then $Q = x/following - sibling :: *$ is updated to $Q' = x/following - sibling :: * \mid absolute_path_to(y)/self :: *$.
  If the position of x is changed to $pos'(x) = pos(x) - 1$, then $Q = x/following - sibling :: *$ is updated to $Q' = x/following - sibling :: *\ except\ absolute_path_to(z)/self :: *$.

- **Preceding**: A situation symmetric to the following axis.

- **Preceding-sibling**: A situation symmetric to the following-sibling axis.

Example 7. *In Figure 4.12 on the left there is schema S before reordering. In the blue rectangle is x. In the green rectangle is marked result R returned by the following axis. On the right there is schema S' after moving element Item to the left. Element Address in the red rectangle is now additional to R and must be eliminated from the query.*

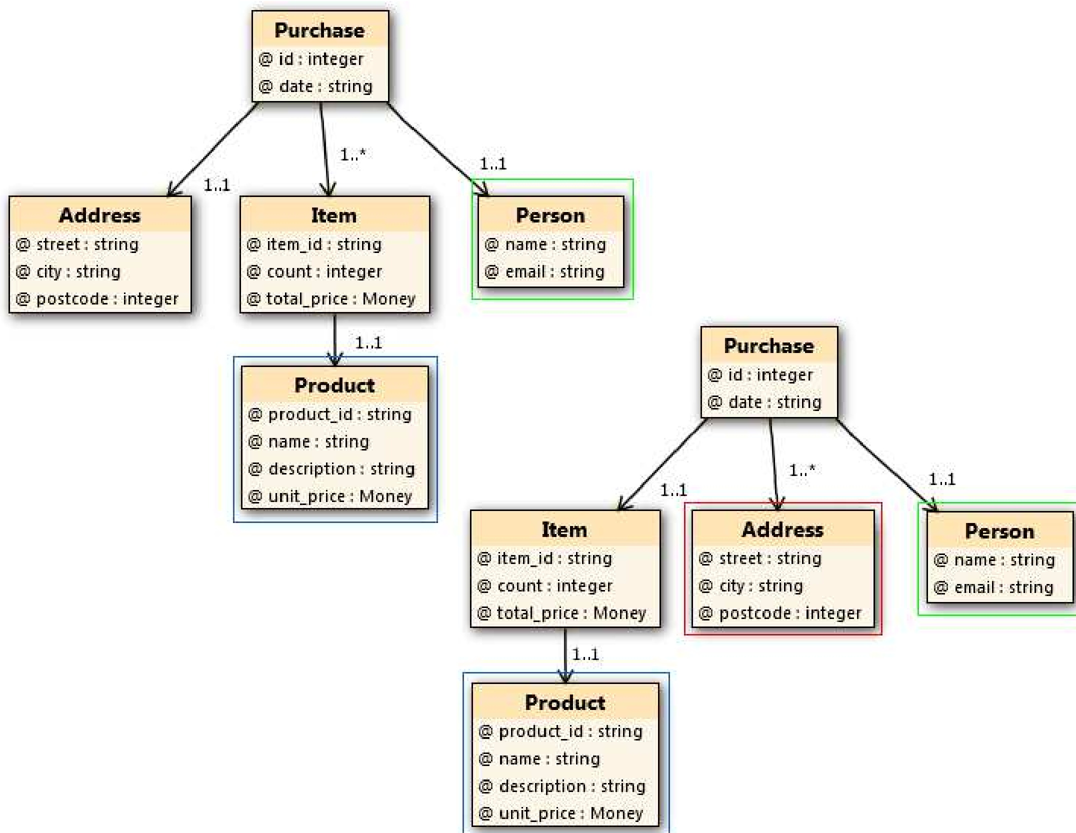

Figure 4.12: Reordering and following axis

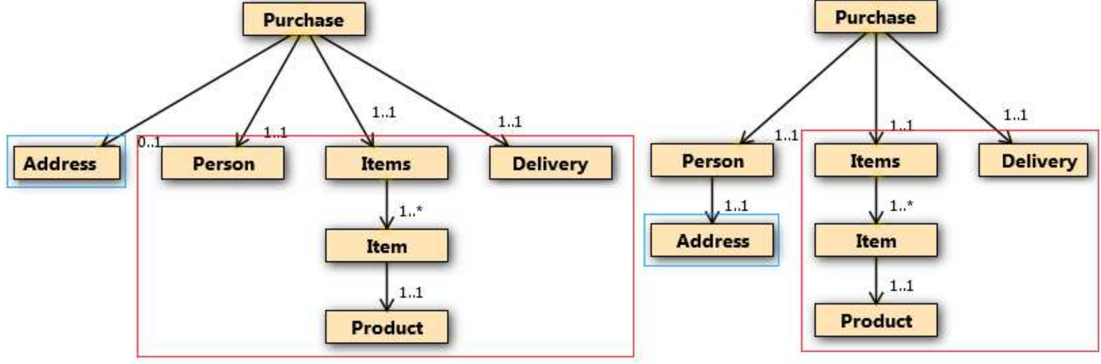

Figure 4.13: Example of reconnection problem – following axis

4.5.5 Reconnection

For simplicity, we consider that element x can be reconnected only as a child of one of its siblings or as a sibling of its parent. Multiple iterations of the operation enable moving of the element to any place in the schema. Let $l(x)$ denote the level of x and $l(root) = 0$. Then x can move to $l(x) + 1$ or $l(x) - 1$. Next, we again assume that it is not possible to move an element as a child of an element where its child has the same name.

Note that an evolution of the query after a reconnection in the schema can be done with all axes, only if the location step is the last one in the query. If not, it is not possible to ensure query consistency in a simple way.

In general, reconnection of an element implies that an axis applied on this element can return different result than if they were applied on the element in the original position (see Figure 4.13 for an example with following axis when element *Address* is moved). The particular location step has to be evolved with disjunction of $R \setminus R'$ and exclusion of $R' \setminus R$.

Suppose that element x is moved to another position in the schema tree, i.e., we change parent element p of x to element p'. Let q be a parent of element p and r be a sibling of x. The particular situations are discussed in following list.

- **Ancestor axis**: If x is moved up, s.t. $parent(x) = q$, then $Q = x/ancestor :: *$ is updated to $Q' = x/ancestor :: * \mid absolute_path_to_element(q)$.

  If x is moved down, s.t. $parent(x) = r$, then $Q = x/ancestor :: *$ is updated to $Q' = x/ancestor :: *\ except\ absolute_path_to_element(r)$.
- **Ancestor-or-self axis**: If x is moved up, s.t. $parent(x) = q$, then $Q = x/ancestor - or - self :: *$ is updated to $Q' = x/ancestor - or - self :: * \mid absolute_path_to_element(q)$.

  If x is moved down, s.t. $parent(x) = r$, then $Q = x/ancestor-or-self :: *$ is updated to $Q' = x/ancestor - or - self :: *\ except\ absolute_path_to_element(r)$.
- **Child axis**: If x is moved up, s.t. $parent(x) = q$, then $Q = p/child :: *$ is updated to $Q' = p/child :: * \mid absolute_path_to_element(x)$.

  If x is moved down, s.t. $parent(x) = r$, then $Q = p/child :: *$ is updated to $Q' = p/child :: * \mid absolute_path_to_element(x)$.

If x is moved up, s.t. $parent(x) = q$, then $Q = q/child :: *$ is updated to $Q' = q/child :: * \mid absolute_path_to_element(x)$.

If x is moved down, s.t. $parent(x) = r$, then $Q = r/child :: *$ is updated to $Q' = r/child :: *\ except\ absolute_path_to_element(x)$.

- **Descendant, Descendant-or-self axis**: If x is moved up, s.t. $parent(x) = q$, then $Q = p/descendant :: *$ is updated to $Q' = p/descendant :: * \mid absolute_path_to_element(x)/\ descendant - or - self :: *$.

  If x is moved down, s.t. $parent(x) = r$, then $Q = p/descendant :: *$ does not need any update.

  If x is moved down, s.t. $parent(x) = r$, then $Q = r/descendant :: *$ is updated to $Q' = r/descendant :: *\ except\ absolute_path_to_element(x)/ descendant - or - self :: *$.

- **Following axis**: If $Q = x/following :: *$, the reconnected element is not x and the reconnection is done in the part of the tree hit by the following axis, no update is needed.

  If $Q = x/following :: *$ and the reconnection of an element y causes that it is added into a part of tree hit by the axis ($R \subset R'$), then $Q = x/following :: *$ is updated to
  $Q' = x/following :: *\ except\ absolute_path_to_element(y)/descendant - or - self :: *$.

  If an element y is moved out from the hit part of the tree ($R' \subset R$), then $Q = x/following :: *$ is updated to
  $Q' = x/following :: * \mid absolute_path_to_element(y)/descendant - or - self :: *$.

  If the reconnected element is x, the revalidation depends on its position of x among siblings. Location paths to the missing elements and location paths to exclude redundant elements must be added.

- **Following-sibling axis**: If y is a sibling of x, where $pos(x) < pos(y)$, and if y is moved up as a sibling of p or down as a child of one of its siblings, then $Qx/following - sibling :: *$ is updated to $Q' = x/following - sibling :: * \mid absolute_path_to_element(y)$.

  If y is a sibling of p or a child of one of siblings of x and if it is reconnected as sibling of x, where $pos(x) < pos(y)$, then $Q = x/following{-}sibling :: *$ is updated to $Q' = x/following{-}sibling :: *\ except\ absolute_path_to_element(y)$.

  When we reconnected element x, the same situation as in the case of following axis occurs.

- **Preceding axis**: A situation symmetric to the following axis.

- **Preceding-sibling axis**: A situation symmetric to the following-sibling axis.
- **Parent axis**: Reconnection of element x in both cases (up or down) causes that $Q = x/parent :: *$ is updated to $Q' = absolute_path_to(p)/\ self :: *exceptabsolute_path_to_element(x)/parent :: * \mid x/parent :: *$.
- **Self axis**: A change of the position of x does not change the result of the query.

4.6 Implementation and Experiments

The full experimental implementation of the proposed approach was integrated into the *DaemonX* framework. Since there are no existing approaches providing similar abilities, it is not possible to compare our solution and results of others. Therefore, queries from the *XPathMark XPath-TF* [42] are used to provide a proof of the concept and for validation. In addition, since this test set is quite simple, we also created our own more complex queries using various axes to test abilities of the solution.

Firstly, from the XPathMark test set we selected tests corresponding with our XPath syntax, i.e., A1 – A11, P1 – P11 (rewritten to queries without predicates) and operators O1, O3, O4. Examples of the queries are as follows:

```
//l/ancestor::* (A5)
//l/following::* (A9)
//l/descendant::* (P5)
//l/preceding-sibling::* (P7)
//q/following::*/parent::* except //g/ancestor::* (O1)
```

All these queries were applied on the respective schema and then all possible edit operations (see Section 4.4.1) were tested.

Secondly, to test the approach on more complex queries, we took a real-world XML schema of an order from the *Amazon AWS* [5] used for communication with customers by Web Services. An XSEM PSM model was created from this schema and a set of XPath queries utilizing all available axes in various combinations was defined. These queries were automatically mapped to the schema by the *DaemonX* framework. Examples of the queries are as follows:

```
//RegionDefinition/parent::ExcludedRegions/parent::*
/Order/ParameterizedUrls/*/*
//AmazonUpsellPreferences/child::*
//ShippingRate/following-sibling::*/descendant::*
//MerchantUpsellItem/Images/preceding-sibling::*
//ShippingMethods/following::*
//RegionDefinition/ancestor::*
//Taxamount/following::Shipping/child::*
//Images/parent::*/ItemCustomDate/ancestor::Cart
```

Next, we made various changes in the schema to simulate a designer. After propagation, the results of both original and new queries were checked.

From the experiments, the following results were observed:

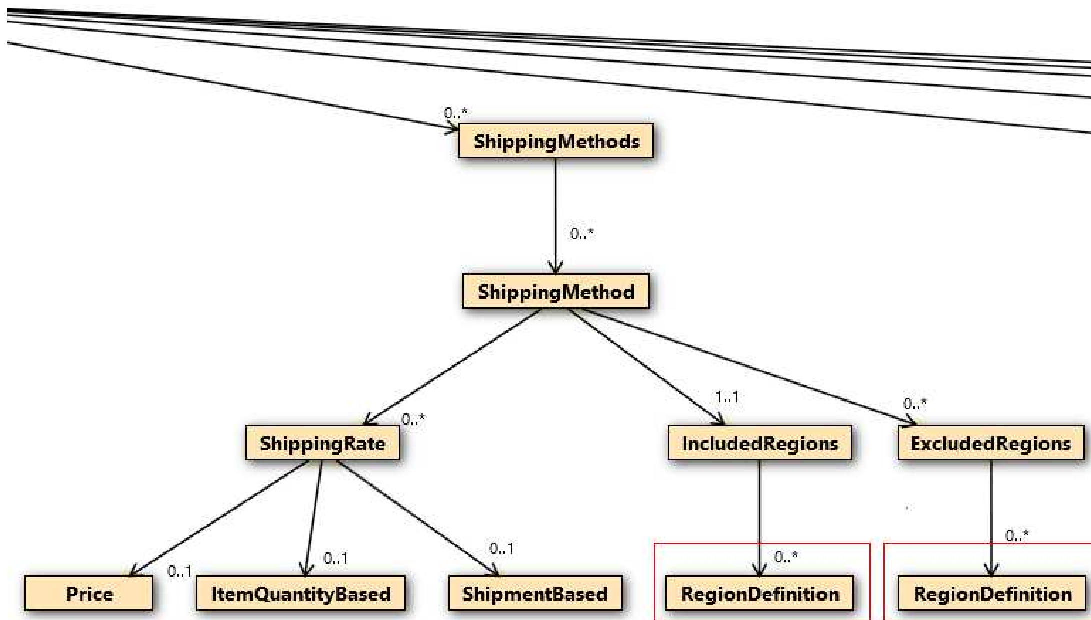

Figure 4.14: Initial schema part of the Amazon example

- Applying of the operations in XSEM PSM model (see Section 4.4.1) over related 40 queries produced 388 changes in the evolved queries. A manual check and evolution of these queries is a big and time consuming task for the designer.
- The resulting query strongly depends on the original query and the update operation. E.g., absolute query $/db/user/address/city$ that returns elements $city$ needs to be updated when element name $address$ is changed to $primary_address$. On the other side a query $//city$ does not need any update if the element name is changed.
- Usage of axes returning a subtree, not only a single node or value, e.g., *following-sibling* axis and subsequent changes in schema model, can evolve the query into a complex form depending on particular schema model.
- Removing of a part of schema can invalidate a related query, e.g., removing of an element that is queried by its name. In this situation a manual update is required.

We present an example of adding a new element $DefaultRegion$ to the Amazon AWS schema and its impact on the query $//ShippingRate/following-sibling :: */descendant :: *$. The initial schema (its affected part) and the initial query is depicted in Figure 4.14 and Figure 4.16 (on the left) respectively. The query result is marked with red rectangles. When a new element $DefaultRegion$ is added as the last child of the element $ShippingMethod$ (see Figure 4.15, the new element is marked with a green rectangle), it must not be returned in the query result. The initial query is evolved to the form $//ShippingRate/(following-sibling :: *exceptDefaultRegion)/descendant :: *$. The final query is depicted in Figure 4.16 on the right.

The implementation and all queries and applied changes of the schema can be found at this resource[1].

[1] http://www.ksi.mff.cuni.cz/~polak/daemonx/

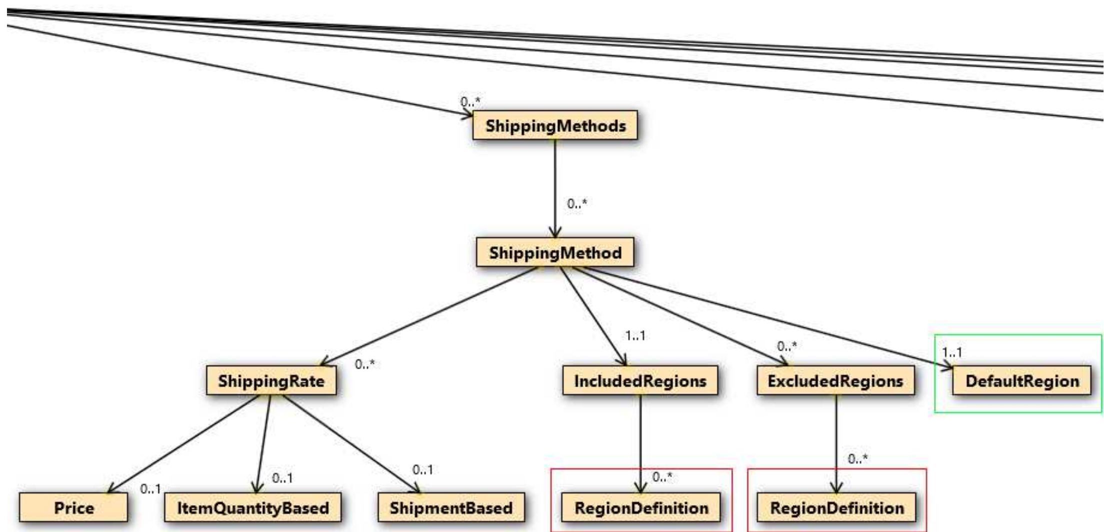

Figure 4.15: Evolved schema part of the Amazon example

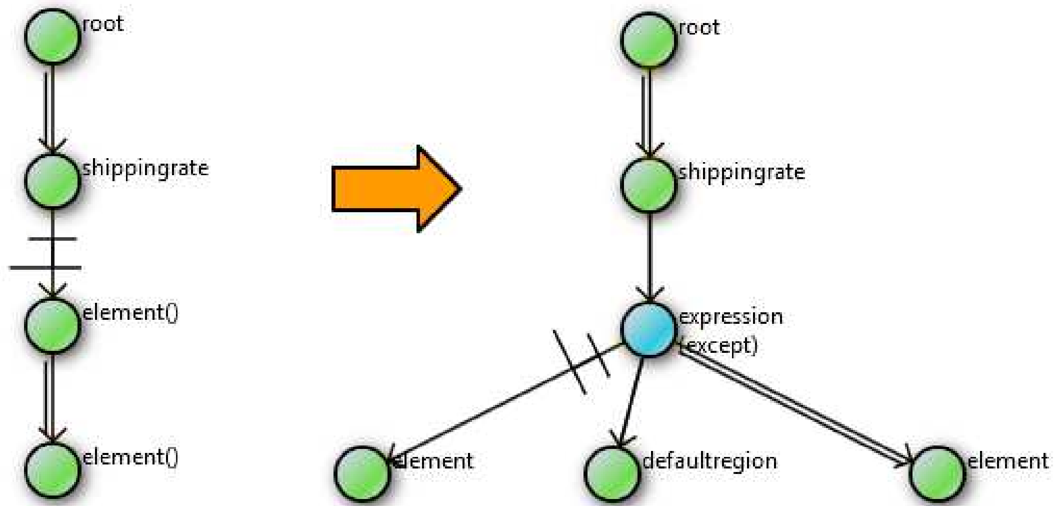

Figure 4.16: Original and evolved query of the Amazon example

4.7 Conclusion

The aim and the main contribution of our approach is the ability to recognize and analyze changes in XML schema and to automatically update related queries respectively based on non-trivial algorithms. If the revalidation of the query is not possible, this situation is reported to the designer with information that a manual update is required. For this purpose we defined a model representing XPath query and operations for model manipulation. Next we defined a mapping between model representing XML schema to be able to analyze changes in schema model and propagate them to query model. Transformation algorithms are defined over model of updated positive core XPath and support all XPath axes, *union* and *except* operators and *name* test.

4.7.1 Future Work

Even though the approach is complex and robust, there exists problems that are not covered:

- *Not decisive cases*: There are cases when the propagation cannot be proceeded and a designer must interfere. A natural extension would be to suggest possible "clues" to simplify the process.
- *Query optimization*: Also, after the changes are made to the queries, it can get into a non-optimized state. Hence, an optimization would be a useful extension.
- *Extension of XPath query syntax*: Used query syntax can be extended to support more constructs, e.g., more tests which enables to transform more complex queries.

5. Relational Schema and SQL Queries Evolution

In this chapter we study another specific part of the evolution and change propagation problem – evolution of a relational database schema and its impact on related SQL queries. The proposed approach shows an ability to model database queries together with a database schema. The feature then provides a solution how to adapt database queries related to the evolved database schema. The model presented in this chapter is depicted in Figure 5.1 in the context of the five-level evolution management framework. The proposal was implemented within the DaemonX framework and various experiments proving the concept were carried out. This approach was presented in [20].

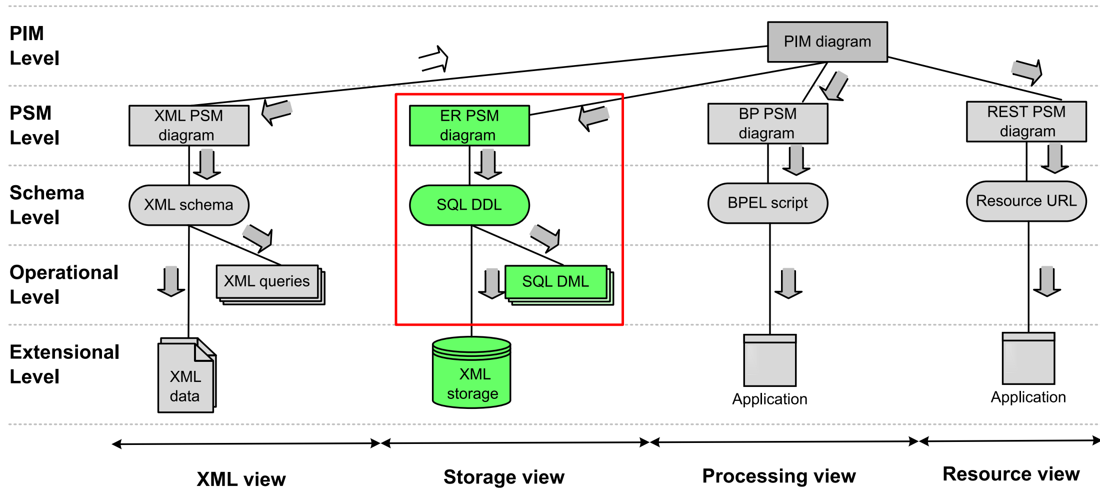

Figure 5.1: Location of the relational database model and related change propagation in the five-level evolution management framework

5.1 Introduction

One of the aspects of the evolution and adaptability problem is an adaptation of the respective storage of the data. The adaptation of the storage covers many related issues, such as *database schema evolution* (i.e., retaining system functionality despite schema changes), *database schema integration* (i.e., cases when more database schemas have to be combined together), *data migration* (i.e., a situation when data have to be moved from one system to another), or *adaptation of respective queries* (i.e., reformulation of queries with regard to changes in data schemas).

A change of the underlying database schema can cause that SQL queries over this schema may become inconsistent with the evolved schema, e.g., a database table has a new name in the evolved schema, a table column has a new name in the evolved schema, a database table does not exist in the evolved schema, a table column does not exist in the new schema, a new table column, which should be used in an SQL query, appeared in the evolved schema, etc. All these cases lead to *incorrect* SQL queries or queries that do not return the original result. So they

should be corrected. For example, suppose an SQL view *PendingOrders*, which returns all information about the pending orders, including all items of the given order. Now, when the name of the column *itemName* is changed (for instance to *productName*), all SQL queries where this column is used have to be checked by a designer and updated respectively.

In this chapter we present an ability to model database queries together with a database schema model. The feature then provides a solution how to adapt database queries related to the evolving database schema.

5.2 Related Works

This section contains an analysis of four works dealing with the database schema adaptation problem. It introduces the given problem, the solution described in the work and possibilities of the solution. The last part of this section provides a comparison of all discussed works and briefly introduces the main issue of this work.

5.2.1 Database Schema Integration Process

Paper [79] describes an approach for integration of complex database schemas. The main aims of the paper are:

- To design a database schema through the approach of a gradual integration of external schemas.
- To suggest a new conceptual modeling design able to be used for conceptual database schema design instead of ER data model.
- To develop a CASE tool which provides complete support for the schema integration process.

Phases of the Database Schema Design Process

Design of a complex database schema is based on a gradual integration of external schemas. An *external schema* is a complex structure that formally defines a user view of a database schema (at the conceptual level). The authors divide the whole process of the database schema design into five phases:

1. Identifying groups of similar end-user business tasks.
2. Conceptual modeling (integration of the external schema into a common conceptual schema).
3. Transformation of the subschema into relational data model.
4. Generating a database schema.
5. Consolidation (detection of constraint collisions and their resolving).

Form Type Concept

Form type concept is an approach for conceptual database schema design. The authors assume that the form type concept may be used for conceptual database schema design instead of ER data model or UML class diagrams. The concept is based on the fact that users communicate with an information system through application forms (see [18]). So the designer's work is to specify screen forms of transaction programs. The main reason for using form type is the fact that the concept is more familiar to end-users' perception of an information system than the concepts of entity and relationship types in ER data model. In addition, form type is a concept that is formal enough to precisely express all the rules significant for structuring future database schema.

IIS*Case

*Integrated Information Systems*Case* (IIS*Case) is the resulting software developed by the authors of the paper, that supports an approach for gradual integration of external schemas. It is based on the form type concept mentioned before. IIS*Case is designed to provide complete support for developing complex database schemas with regard to the number of concepts used, and to give an intelligent support in the course of the whole schema integration process. IIS*Case supports:

- Conceptual modeling of external schemas.
- Automated design of the so-called relational database subschemas in the 3rd normal form [82].
- Automated integration of relational database schema from designed subschemas.
- Detecting and resolving the constraint collisions between a database schema and a set of subschemas.

Discussion

The paper presents an original approach to automatic integration of database schemas. The approach is based on the form type data model. From the designer's point of view, the form type data model offers a simple way for defining the initial set of attributes and constraints. For the designers, the database design of even complex information systems became easier because the process of modeling is shifted to the level of a user without an advanced knowledge of the database design. The paper also suggests methods how to detect and finally resolve collisions between independently designed database schemas. The presented approach is supported by own CASE tool – IIS*Case.

5.2.2 QuickMig

Paper [35] describes a (semi)automatic approach to determining semantic correspondences between schema elements for data migration applications.

The QuickMig Migration Process

The QuickMig approach divides the process of migration into five phases:

1. Answering a questionnaire
2. Injection of sample instances
3. Schema and instance import
4. Matcher execution
5. Review

Answering a Questionnaire The purpose of the first phase is to collect as much information as possible about the source system. This data will be used to automatically reduce the complexity of the target schemas. This also helps with reduction of the complexity of the matching process. The first phase has to be performed manually by a person with some knowledge of the capabilities of the source system.

Injection of Sample Instances In this phase instances existing in the target system are manually created in the source system. These samples are used by the instance-based matching algorithms in order to determine correspondences between the source and the target schemas. Using sample data this knowledge can be exploited by the matching algorithms. By injecting sample data into the source system the matching algorithms not only have arbitrary instances available but also one dedicated instance which maps exactly to a specific instance of the target schema.

Schema and Instance Import In the third phase of migration process the source schemas as well as corresponding sample instances are imported into the QuickMig system.

Matcher Execution In this phase schema matching algorithms are executed. These automatically determine a mapping proposal using different matching algorithms. The resulting mapping proposal includes similarities between elements of the source and target schemas as well as a proposal for the *mapping categories*.

Review In the final phase a developer reviews and corrects the mapping proposal. When the mapping proposal is acceptable, real mapping code is generated and the mapping is stored.

Mapping Categories

Each returned correspondence from the matching phase (phase 4 of the migration process) is associated with some *mapping category*. These mapping categories are used to create parts of the necessary mapping expressions. At least it can be used to provide a mapping expression template that can be easily completed by a developer. QuickMig distinguishes the following eleven mapping categories:

- CreateInstance
- KeyMapping
- InternalId
- LookUp
- Move
- ValueMapping
- Code2Text
- DefaultValue
- Split
- Concatenate
- Complex

The QuickMig Architecture Overview

The QuickMig is based on COMA++ [55]. QuickMig extends COMA++ by implementing three new matching algorithms:

- **Equality Matcher**: It is the simplest matcher. It tries to identify equal instance values in the source and target schema.
- **SplitConcat Matcher**: This matcher checks the instance data for splitting or concatenation relationships.
- **Ontology-Based Matcher**: This matcher exploits background knowledge provided in a domain ontology in addition to instance data in order to identify corresponding schema elements.

Each matching algorithm returns a list of correspondences between the source and the target schema and also the associated mapping category. The results of all algorithms are combined in order to create the resulting mapping.

Discussion

QuickMig is a system for the (semi)automatic creation of schema mappings in data migration projects based on 5-phase migration process. The most important phase is the phase of matching execution, where the mapping proposal is created and each found correspondence is associated with some mapping category.
The proposal approach was experimentally evaluated using real SAP [25] schemas. In these experiments QuickMig achieved very good results. QuickMig is planned to be integrated into SAP data migration tools in future.

5.2.3 The PRISM Workbench

Paper [29] describes an advanced approach to support database schema evolution in traditional information systems. In such projects the frequency of database schema changes has increased while tolerance for down-times has nearly disappeared. The main aim of the paper is to create a tool to manage and automate:

- Predicting and evaluating the effects of the proposed schema changes,
- rewriting queries and applications to use the evolved schema, and
- migrating the database.

SMO Language

The whole evolution process is described by *Schema Modification Operators* (SMO), e.g., table creation, table merge, table split, column creation, etc. Each SMO represents an atomic operation performed on both schema and data. All SMOs together represent the *SMO language*. Basically, PRISM provides the simple SMO language to express schema changes in the process of designing evolution steps.

Evolution Process

The process of evolution is divided into five phases:

1. Evolution design
2. Inverse generation
3. Validation and query support
4. Materialization and performance
5. Deployment

Evolution Design The first phase of evolution process can be further divided into the sequence of six steps:

- The database administrator expresses by the SMO language the desired atomic changes to be applied to the input schema.
- The system virtually applies the SMO sequence to input schema and creates the candidate output schema.
- The system verifies whether the desired evolution is resistant against loss of data during executing the evolution. If so, we call the evolution as *information preserving.*
- Each SMO in sequence is analyzed for redundancy. For instance, *Copy table* SMO generates redundancies. A database administrator has to decide whether such redundancy is intended or not.
- The system translates the desired SMO sequence into a logical mapping between schema versions. This mapping is expressed by so-called Disjunctive Embedded Dependencies (DEDs).
- Query chase engine rewrites the queries expressed over the old schema into equivalent queries expressed over the evolved schema. Alternatively the SMO sequence is translated into SQL Views corresponding to the mapping between versions to support queries over the data stored in the basic tables.

The whole phase can be iterated until the candidate schema is satisfactory and the final schema is obtained.

Inverse Generation In this phase, on the basis of the forward SMO sequence, the system computes the candidate inverse sequence. Then it checks, whether the inverse SMO sequence is information preserving. If both forward and inverse sequences are information preserving, the schema evolution is guaranteed to be completely reversible.

Validation and Query Support In this phase the system translates the inverse SMO sequence into a logical mapping using DEDs. As in the last step of the first phase, queries expressed over the evolved schema are rewritten into equivalent queries expressed over the old schema. Also corresponding SQL views are generated. According to results of this phase, the database administrator can repeat phases one to three to improve the evolution.

Materialization and Performance This phase can be further divided into four steps:

- The system translates both the forward and inverse SMO sequences into SQL data migration scripts.
- On the basis of the previous step the system materializes evolved schema and supporting queries in the old schema by views or query rewriting.
- Rewritten queries are tested against the materialized evolved schema for absolute performance testing.
- Old queries are tested natively against the old schema. The results are compared with the results of the previous step.

The database administrator can improve performance by modifying the schema layout, for instance by modifying indices in evolved schema.

Deployment In the last phase, the old schema is dropped and old queries are supported by SQL views or by query rewriting. The whole evolution process is recorded into an enhanced *information schema*, to allow schema history analysis. It is possible to preform a *late rollback* by generating an inverse data migration script from inverse SMO sequence.

Data Migration And Query Support

The logical mapping between versions is expressed by DEDs. Each SMO in an input SMO sequence is converted into DED, so each DED represents a given mapping rule between the old schema and the evolved schema. Each SMO produces both forward and backward mapping. Forward mapping expresses how to migrate data from the source (old) schema version to the target (new) schema version. Using the generated DEDs, the queries are rewritten by engine using the *chase and backchase* algorithm [30].

SMO to SQL SMOs are tailored to assist data migration tasks, therefore many SMOs combine actions on both schema and data. In phase 4 of evolution process, each SMO is translated into corresponding SQL data migration code. PRISM also supports expressing of the mapping between versions in terms of SQL views. This often happens in the data integration area. Views are usually used to enable what-if analysis (forward views), or to support old schema versions (backward views). Each SMO can be translated into a corresponding set of SQL views.

Discussion

PRISM workbench is a system for the support of database schema evolution. The whole evolution process is based on 5-phase evolution process, including manual design of desired evolution by SMO language and automated work of PRISM tools to perform corresponding schema evolution, generate data migration scripts and rewrite queries into equivalent one expressed on the new database schema. The system was tested on Wikipedia database schema. Its 170+ schema versions provided good testing environment for validating PRISM tools and ability to support legacy query rewriting. PRISM deserves further research, especially in the field of optimization of performance or query rewriting, but it is clear that PRISM takes a big step toward needs of database administrators to have methods and tools to manage and automate the whole process of database schema evolution.

5.2.4 Automating the Database Schema Evolution Process

In paper [28] the authors describe techniques and systems for automating the critical tasks of migrating the database and rewriting the legacy applications. The approach is an extension of the PRISM/PRISM++ framework [29]. Specifically, SMOs are extended with integrity constraints (ICs) modification operators (key, foreign key, and value). The main requirement mentioned by the authors is to reduce time needed for particular migration process, e.g., downtime of the system needed for migration. The paper extends previous results with following extensions:

- Characterization of the impact on integrity constraints of structural schema changes.
- Proposal of representations that enable the rewriting of updates.
- A unified approach for query and update rewriting under constraints.

Impact of an SMO on Integrity Constraints

To be able to handle IC problem, there must be generated a correct set of output ICs for each SMO type and each set of input ICs. Suppose a schema S_1 with IC IC_1:
$S_1 : V(a, b, c)$
$IC_1 : V(a, b, c), V(a, b', c') \Rightarrow b = b', c = c'$.

Next, let the SMO is a decomposition of V into $V1(a, b)$ and $V2(a, c)$ Then, the resulting schema S_2 with output IC_2 will be:
$S2 : V1(a, b), V2(a, c)$
IC_2: $V1(a, b), V1(a, b') \Rightarrow b = b'$
$V2(a, c), V2(a, c') \Rightarrow c = c'$
$V1(a, b) \Rightarrow \exists c V2(a, c)$
$V2(a, c) \Rightarrow \exists a V1(a, b)$

Rewriting Technology

The authors present a new rewriting technology for queries and for SMOs with ICs. To be able to incorporate new features presented in this paper, the rewriting engine had to be completely redesigned.

Automating the extraction of SMO

The next part of the paper introduces a (semi)automatic approach for generation of SMOs from migration scripts written in MySQL dialect of SQL.

Discussion

The paper presents new abilities of the PRISM/PRISM++ framework – handling ICs over SMOs. The authors present this feature as another important aspect of the DBMS migration. Next, they introduce an approach for (semi)automatic generation of SMOs from an SQL migration script. Conclusion of the work compares the presented solution with a number of commercial tools used in IT industry.

5.2.5 Adaptive Query Formulation

Papers [100] and [99] describe a graph-based approach to database schema evolution.

Graph-based Model

The authors propose a graph-based model that uniformly covers relational tables, views, database constraints and SQL queries. Formally, the given database schema is represented as directed graph $G = (V, E)$, where V are the nodes of the graph representing the entities of the model, and E are the edges representing the relationships between these entities. There are the following essential components:

- **Relations**: The relation in the database schema is represented as a directed graph, which includes:
  - *Relation node*, representing relational schema.
  - *Attribute node*, one for each attribute.
  - *Relationship edges* directing from relation node to attribute nodes, indicating belonging of an attribute to the relation.
- **Conditions**: The conditions refer to selection conditions (of queries and views) and constraints (of a database schema). A *condition node* represents a given condition. The node is tagged with appropriate operator and it is connected to the *operand nodes.* Composite conditions are simply constructed by tagging the condition node with a Boolean operator and the appropriate edges to the conditions composing the given composite condition.

- **Queries**: The graph representation of the query includes a *query node* and *attribute nodes* corresponding to the query projection. In order to determine relationships between the query and the relations, the query is divided into these essential parts:

  – **Select part**: This part of the query maps appropriate attributes of the involved relations to the attributes of the query projection through edges of type *map-select* directing from the query attributes towards the relation attributes.

  – **From part**: This part of a query is considered as the relationship between the query and involved relations. The relations involved in this part are combined with the query node through edges of type *from-relationship* directing from the query node towards the relation nodes.

  – **Where and Having parts**: These clauses are assumed to be in *conjunctive normal form* (CNF) [73]. There exist two edge types *where-relationship* and *having-relationship* directing from a query node towards an operator node at the highest level of the conjunction.

  – **Group and Order By parts**: For this part there are two special nodes: a *group-by node* (GB) to capture the set of attributes acting as the aggregators and an *aggregate function node.* There are the following types of edges: *group-by* directing from a query node to a GB node, and from a GB node to each aggregator, and *map-select* directing from each aggregated attribute to an *aggregate function node* and from an *aggregate function node* to an appropriate relation attribute. Order-by clause is performed similarly.

- **Views:** Views are considered either as queries or in case of materialized views as relations.

Evolution Policies

In the context of the proposed graph-based model, changes in the database schema are events which transform specific parts of the graph and eventually affect other dependent graph constructs, which recursively may raise new changes, with an impact on other graph constructs. To handle schema evolution, the graph constructs have to be annotated with policies that allow the designer to specify the behavior of a given construct whenever change events occur. This combination of event and policy triggers the execution of the appropriate action - blocks the event or reshapes the graph respectively. Possible events are defined as the Cartesian product of a set of hypothetical actions (addition, deletion, modification) and set of graph constructs, which are subject of evolution (relations, views, attributes and conditions). There exist three kinds of policies, which can be used with a given event:

- **Propagate** the change, i.e., the graph has to be reshaped to adjust new semantics.

- **Block** the change to retain old semantics and the event has to be blocked.
- **Prompt** the user to interactively decide what will eventually happen.

SQL Extensions

The authors propose an extension of a database system catalog with extra information regarding evolution purposes. Each assertion is considered as a tuple (*event, policy*). These assertions extend SQL syntax both in DDL statements as well as in SQL queries. The general syntax is: $ON < event > THEN < policy >$. An *event* refers to evolution events in the database schema (delete, add, modify, rename) and a construct type (node, relation, query, view, attribute, condition, PK, FK, NNC, UC). The *policy* can take the values mentioned in previous section (propagate, block, prompt).

Discussion

The papers present a graph-based approach to performing database schema evolution. They focus on propagating potential changes of the database system to all the affected parts of the system. There was introduced a complex graph-based model, which covers the whole database system, including such elements like queries or views. Also there was introduced an extension to the SQL language specifically tailored for the management of evolution. The applicability and efficiency of this approach has been tested in real-world evolution scenario extracted from an application of the Greek public sector in a *Hecateus* tool. The main goal of the test was to minimize the human effort required for defining and setting the evolution metadata by using the proposed language extension.

5.2.6 Synchronization of Queries and Views Upon Schema Evolutions: A Survey

Paper [15] presents an overview of various approaches dealing with database schema evolution and query-view synchronization (QVS) in the last decades. The authors divide approaches into three base groups:

1. **Operation based approach**: These approaches define commands for every supported change operation and are based on analysis of these commands.
2. **Mapping based approach**: The idea of mapping based approaches is based on comparing and mapping the original and the evolved schema and subsequent analysis.
3. **Hybrid approach**: Hybrid approaches exploit advantages and strength points of both basic types.

Classification Framework

The authors define a classification framework for evaluation and the characterization each approach in terms of its capabilities and its proper context of use. The proposed framework is structured into two groups of parameters:

1. **Structural issues**: The parameters in this group characterize how the QVS problem is tackled by a single approach.
2. **Semantic issues**: The parameters in this group concern the aspects of the QVS problem of a single approach faced by each approach.

The authors present and analyze particular approaches by the defined characteristics and compare them by their abilities.

Comparison Results

Operation Based Approaches The authors compared seven approaches. They concluded that only two of them are completely implemented. Next, they evaluated that approaches differed in used query languages – some of them used SQL, its extensions or custom non-SQL languages. Concerning the semantic issues, most of the surveyed operation-based approaches provided a complete automated support to QVS.

Mapping Based Approaches The authors compared five approaches when only one approach completely implemented the approach. As for schema change languages, except for two approaches, which used algebraic mapping operators, each of the remaining three approaches used a different mapping language, such as matching formulas and local and external mappings.

Hybrid Approaches The authors evaluated six approaches from this group. As was expected, hybrid approaches mix the characteristics of both operation-based and mapping-based approaches. Hybrid approaches are based on many different models and techniques that have little in common with the other two classes of approaches. Thus, one important conclusion about this survey was that there had been no leading model or technique that had conditioned the development of QVS approaches.

Discussion

The paper presents an extensive set of current approaches of schema evolution. The authors present a framework for comparison these approaches by defined characteristics and present results of these analyzes. The analysis of the existing approaches reveals that a considerable research effort has been devoted to this area, which has led to the proliferation of many different approaches and tools. They claim that although some of the surveyed approaches seem to be perfect from a theoretical and an architectural point of view, they still lack implementation accuracy and completeness. They conclude that although the proposed analysis does not allow them to determine which is the best approach, it enables them to perform both a general comparison based on relevant characteristics and to detect the most useful and suitable approach for a given application context.

5.2.7 Comparison of the Related Works

All presented works study the problem of database schema adaptation, but each of them focuses on a separate part of this complex issue.

The first paper [79] describes an approach to the process of integration of complex database schemas. It is based on the form-type concept, which is used for conceptual database schema design in contrast to mainly used ER data model or UML class diagrams. The second paper [35] describes a (semi)automatic approach to determining semantic correspondences between schema elements for data migration applications. It focuses on the matching process where a mapping proposal is created and each found correspondence is associated with some mapping category. These are then used to create parts of the mapping expressions.

The authors study the same problem in works [29, 28] – database evolution, but each from the different point of view. The work [29] focuses on the support of database schema evolution in traditional information systems. It introduces the SMO language to design the desired evolution changes. Paper [28] is an extension of the previous approach [29]. It adds a support for integrity constrains over SMOs and a (semi)automatic generation of SMOs from SQL migration scripts. To be able to incorporate these new abilities, current engine of the PRISM/PRISM++ tool had to completely redesigned.

Approach [99] is based on graph model of the whole database system, including views, queries or constraints. The graph is annotated by policies to specify behavior of given database system construct whenever change events occur. The result of the work [29] is a sequence of traditional SQL statements, in contrast to the last work [99], where the evolution is performed in place and all affected elements are automatically adjusted into new version.

Paper [15] presents an extensive survey of existing approaches focused on schema evolution. The authors classify the approaches by various issues and provide an analysis that can be used as an overview and start point of subsequent approaches.

In this chapter we focus on the problem of database queries adaptation. We focus especially on the following problems which were not solved in the papers:

- Modeling of database queries in a CASE tool together with the database schema model.
- The mapping between a database schema and database queries.
- The propagation of changes between a database schema and database queries.

The main motivation for this solution was an ability to manage SQL schema model changes, change analysis and (semi)automatic propagation to related SQL query models without need of manual analysis and changes. Subsequently, this solution can be interconnected with another models to form a complex tool for system evolution management.

5.3 Database Model

The database model we consider is based on the relational database model. Such a database model is used as platform-specific according to the MDA approach.

We are using a database model in two roles. Besides the classical PSM for data, it is used as the PIM for a query model described in Section 5.4. The idea

of using the database model this way is simple. From the perspective of an arbitrary query language (not only SQL), the database model creates a basic concept of a database schema. Each query language then creates own platform-specific view of the platform-independent database model. The PSM database model is described as follows:

Database Model Constructs

The model contains the following constructs:

- **Table** represents a given table in a given database system. Each table construct can represent only one class from the PIM model. Every table construct has a name and contains column constructs. Table constructs can be connected by relationships.
- **Column** is a construct that can exists only as a child of a table construct. Each column construct represents at most one PIM attribute. Every column construct has a name, a data type, an indicator if it is used as primary key, nullable or with a unique value.
- **Relationship** construct represents a connection between two table constructs. Every relationship construct has a role, which represents a logical label for such connection, and cardinality of relationship. There can exist more than one relationship between two table constructs, but each such relationship must have a unique role.
  The relationship between two tables is not only logical or virtual. It is based on the concept of *keys* and *foreign keys*. If there exists a column construct C, which is in table construct T_1 used as primary key and in table construct T_2 used as foreign key, there must exist a relationship construct connecting constructs T_1 and T_2.

An example of the model visualization is depicted in Figure 5.2.

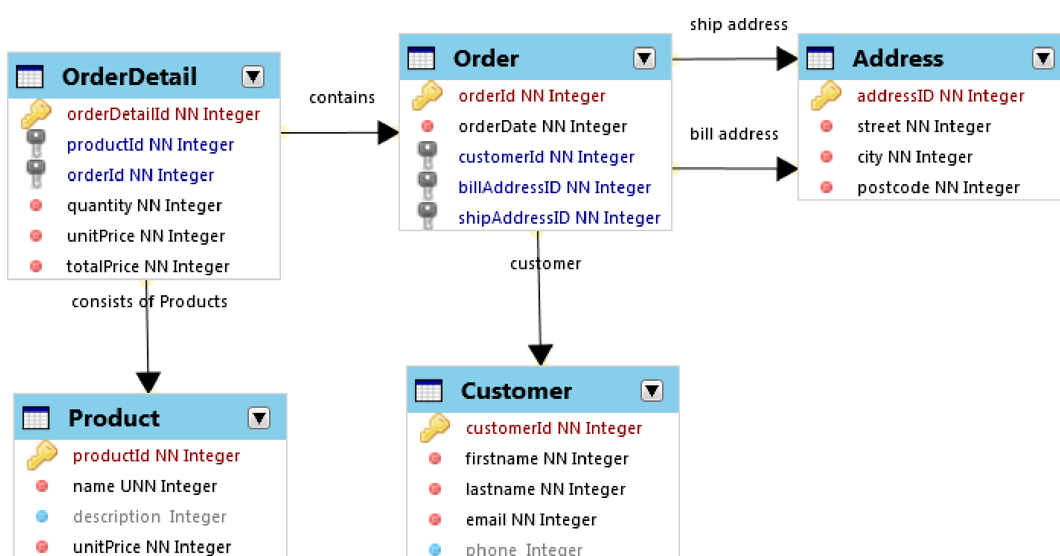

Figure 5.2: An example of PSM database model

5.4 Query Model

For the ability of evolution of SQL queries related to a given database schema there must exist a mapping between an SQL query and a database schema model. This mapping helps to manage the evolution process to evolve the query related to the evolved database schema. In this section we introduce a graph-based SQL query model which is particularly designed for the evolution process. We describe its visualization model, limitations and possibilities. The mapping between the SQL query model and the database schema will be introduced as well. (The algorithm to generate the SQL query from the model can be found in [19].)

The full SQL language syntax was not used in the proposal and for this reason there exist some limitations on the used subset of SQL language:

- Projection operator '$*$' is banned to use. It is always necessary to enumerate all the columns used in the *SELECT* clause.
- In queries it is possible to use only simple column enumeration, other expressions or functions other than aggregate functions are banned to use. This limitation relates to the *CASE* construct as well.
- The model does not support *UNION*, *INTERSECT* and *EXCEPT* constructs.
- Each condition used in the SQL query is assumed to be in the *conjunctive normal form* (CNF) [73].

The idea of the graph-based model results from papers [100] and [99]. However, in this approach we use a graph-based model for query modeling, in contrast to the mentioned papers, where graph-based model is implemented as a part of the *database management system* (DBMS). The idea of a graph-based model from [100] and [99] is adjusted and extended for the purposes of our approach.

First, each SQL query in the model is represented as a directed graph with particular properties.

Definition 11. *(Query Graph). A* query graph *G of the SQL query Q is a directed graph $G_Q = (V, E)$, where V is a set of query vertices and E is a set of query edges.*

Definition 12. *(Query Model). A* query model *M of the SQL query Q is a pentad $M_Q = (G_Q, T_V, T_E, \tau_V, \tau_E)$, where G_Q is a query graph $G_Q = (V, E)$, T_V is a set of vertex types {AggregateFunction, Alias, BooleanOperator, CombineSource, ComparingOperator, ConstantOperand, DataSource, DataSourceItem, From, FromItem, GroupBy, Having, OrderBy, OrderByType, QueryOperator, Select, SelectItem, Where}, T_E is a set of edge types {Alias, Condition, ConditionOperand, DataSource,DataSourceAlias, DataSourceItem, DataSourceParent, FromItem, FromItemParent, FromItemSource, GroupBy, GroupByColumn, Having, MapColumn, MapSource, OrderBy, OrderByColumn, SelectColumn, SelectQuery, SourceTree, Where}, a function $\tau_V : V \rightarrow T_V$ assigns a type to each vertex of the query graph G_Q and a function $\tau_E : E \rightarrow T_E$ assigns a type to each edge of the query graph G_Q.*

A *query vertex* represents a particular part of the SQL query, e.g., a database table, a table column, a comparing operator in condition, a selected column in the *SELECT* clause, etc. A *query edge* connects parts of the SQL query together and gives a particular semantics to this connection. For instance the edge connecting a *From* vertex and a *Where* vertex means that the query contains a *WHERE* clause represented by the *Where* vertex.

Each query graph can be logically divided into smaller subgraphs. These subgraphs are called *essential components*. Each essential component has a visual equivalent in the query visualization model (described in Section 5.5). There exist the following essential components: *DataSource*, *From*, *Select*, *Condition*, *GroupBy*, *OrderBy*. For instance, the simplest SQL queries of the form *'SELECT projection FROM table'* require only *DataSource*, *From* and *Select* components. Figure 5.3 illustrates a simple example of the modeled *GROUP BY* clause. The example is equivalent to the following parts of the SQL query:

```
SELECT
  CustomerId
    as cid,
  COUNT(OrderId)
    as orderCount
  ...
GROUP BY
  CustomerId
```

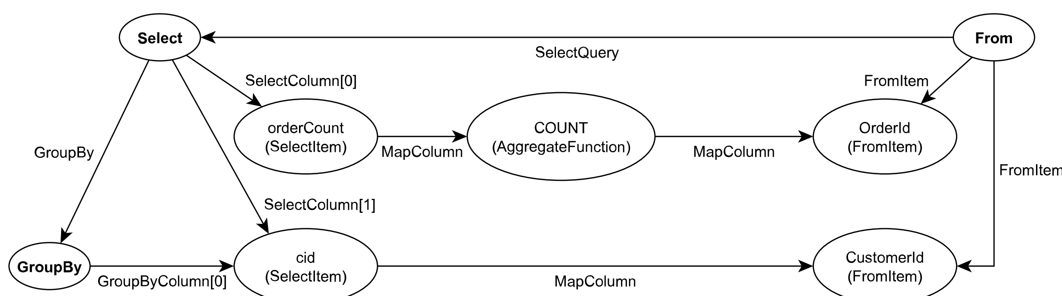

Figure 5.3: An example of a simple model of the *GROUP BY* clause

(All these components contain subcomponents. The full list of the subcomponents with their descriptions can be found in [19].)

5.5 SQL Query Visualization Model

Although the graph-based query model can describe any SQL query, it is relatively complex. Even the query model for a simple SQL query contains a lot of vertices and edges (see Figure 5.3). For this reason we proposed a visualization model, which simplifies an underlaying query model for users and model creation.

5.5.1 Visualization Model Components

The visualization model is divided into so-called *essential visual components*. Each essential component described in Section 5.4 has its visual equivalent by

an essential visual component, so each visual component represents a part of the SQL query graph model. We distinguish the following visual components: *DataSource*, *QueryComponent* and *Component Connection*.

DataSource Visual Component A *DataSource* visual component visualizes a *DataSource* essential component. Each *DataSource* has a name, which clearly identifies the given data source. The content of the *DataSource* component is a list of *DataSourceItem* visual components. The *DataSource* visual component itself corresponds to the *DataSource* vertex. The *DataSourceItem* visual components correspond to the *DataSourceItem* vertices. For example, the rectangle *Customer* in Figure 5.4 shows an example of *DataSource* visual component. The example represents a database table *Customer* with columns: *customerId*, *firstname*, *lastname*, *email* and *phone*.

QueryComponent Visual Component A *QueryComponent* is a universal visual essential component, which represents parts of the SQL query. A basic appearance of all query components looks the same. We distinguish the following types of query components according to the clause of the SQL query they represent: *Select*, *From*, *Where*, *GroupBy*, *Having*, *OrderBy*. For example, the rectangle *Where* in Figure 5.4 illustrates an example of the *QueryComponent* visual component. The example shows visualization of the *WHERE* clause.

Component Connection A *Component Connection* does not correspond directly to any essential component of the query graph. Instead, it covers a connection of two essential components to finish the correct and complete query graph. We distinguish the following types of connections: *DataSource* → *From*, *Select* → *From*, *Where* → *From*, *GroupBy* → *Select*, *Having* → *From*, *OrderBy* → *From*. For example, Figure 5.4 shows a visualization model of a more complex SQL query. The modelled SQL query is the following one:

```sql
SELECT
   c.firstname, c.lastname,
   a.street, a.city,
   a.postcode
FROM
   Customer as c JOIN
   Address as a
     ON c.customerId = a.customerId
WHERE
   (c.firstname = 'John' OR
    c.firstname = 'Jane')
      AND
   (c.lastname = 'Doe')
ORDER BY
   c.customerId ASC,
   c.lastname ASC, a.postcode DESC
```

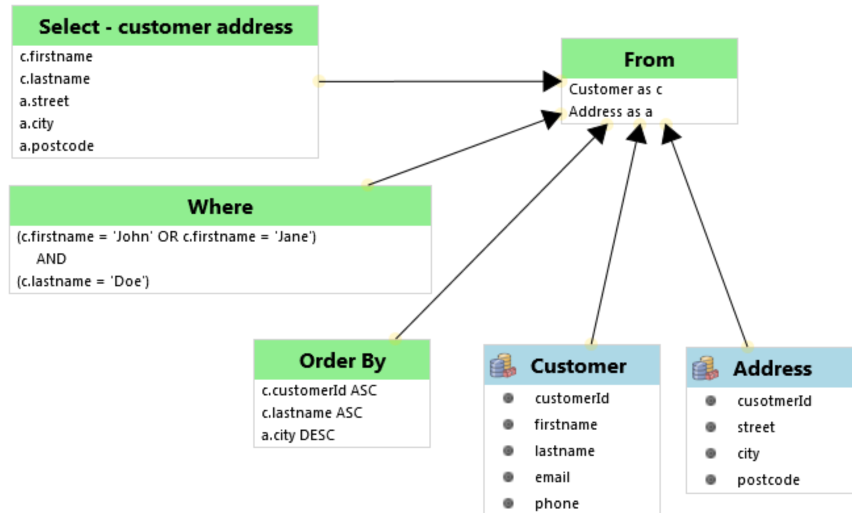

Figure 5.4: An example of a visual model of a more complex SQL query

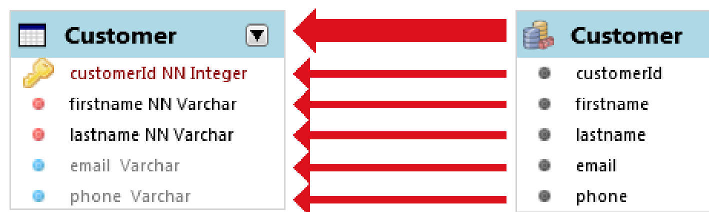

Figure 5.5: An example of a mapping between database model and query model

For comparison, the underlaying query graph model consists of 45 vertices connected by 87 edges.

5.5.2 Mapping to the Database Model

Since the database model consists of tables and its columns which we can interpret as a general source of data, we have a direct mapping from the database model to the query model. We do not consider database relationships between database tables in the database model. For the purpose of the query model they are not important.

The mapping between the database model and the query model is described as follows:

- **Database table → *DataSource*** The database table in the database model is mapped to the *DataSource* visual component which corresponds to the *DataSource* vertex of the underlaying query graph.
- **Table column → *DataSourceItem*** The table column in the database model is mapped to the *DataSourceItem* visual component which corresponds to the *DataSourceItem* vertex of the underlaying query graph.

An example of the mapping is shown in Figure 5.5. The example illustrates the mapping from a database table *Customer* to a *DataSource* visual component called *Customer*.

Note that the mapping does not preserve keys (primary keys, foreign keys) and any other column attributes like *NOT NULL* or *UNIQUE*. For the purpose of the data querying this property is insignificant.

5.5.3 Mapping of Operations

In this section we describe the mapping of the operations of database and query models used in the evolution process.

All changes in the database model are done via atomic operations. All atomic operations in the database model which have an impact on the SQL query model are translated by the evolution process into corresponding atomic operations in the SQL query model.

Database Model Operations

Let us suppose a database model M_D, which is a set of tables T_i, $i \in [1, n]$. Each table $T_i \in M_D$ has a name T_{i_N} and a set T_{i_C}, which is a set of columns c_j, $i \in [1, n_i]$. Each column c_j has a name c_{j_N}.

- **Renaming Database Table** ($\alpha_T : (T_i, m) \to T'_i$): The operation returns table T'_i where $T'_{i_N} = m$ and $T'_{i_C} = T_{i_C}$.
- **Removing Database Table** ($\beta_T : (M_D, T_i) \to M'_D$): The operation removes database table $T_i \in M_D$ from the database model M_D. It returns database model M'_D where $M'_D = M_D \backslash \{T_i\}$.
- **Creating Table Column** ($\gamma_C : (T_i, c_j) \to T'_i$): The operation adds the column c_j into table T_i. It returns table T'_i where $T'_{i_N} = T_{i_N}$ and $T'_{i_C} = T_{i_C} \cup \{c_j\}$.
- **Renaming Table Column** ($\alpha_C : (c_j, m) \to c'_j$): The operation returns column c'_j where $c'_{j_N} = m$.
- **Removing Table Column** ($\beta_C : (T_i, c_j) \to T'_i$): The operation removes column $c_j \in T_i$ from the table T_i. It returns table T'_i where $T'_{i_N} = T_{i_N}$ and $T'_{i_C} = T_{i_C} \backslash \{c_j\}$.

SQL Query Model Operations

Let us suppose a query model M_Q, whose query graph G_Q consists of a set of *DataSources* D_i, $i \in [1, k]$ and other components, which are not important for our purpose. Each *DataSource* $D_i \in M_Q$ has a name D_{i_N} and a set D_{i_I}, which is a set of *DataSourceItems* d_j, $j \in [1, k_i]$. Each *DataSourceItem* d_j has a name d_{j_N}.

- **Renaming *DataSource*** ($\alpha_D : (D_i, m) \to D'_i$): The operation returns *DataSource* D'_i where $D'_{i_N} = m$ and $D'_{i_I} = D_{i_I}$.
- **Removing *DataSource*** ($\beta_D : (M_Q, D_i) \to M'_Q$): The operation removes *DataSource* $D_i \in M_Q$ from the query model M_Q. It returns query model M'_Q where $M'_Q = M_Q \backslash \{D_i\}$.

- **Creating *DataSourceItem*** ($\gamma_I : (D_i, d_j) \rightarrow D'_i$): The operation adds *DataSourceItem* d_j into *DataSource* D_i. It returns the *DataSource* D'_i where $D'_{i_N} = D_{i_N}$ and $D'_{i_I} = D_{i_I} \cup \{d_j\}$.
- **Renaming *DataSourceItem*** ($\alpha_I : (d_j, m) \rightarrow d'_j$): The operation returns *DataSourceItem* d'_j where $d'_{j_N} = m$.
- **Removing *DataSourceItem*** ($\beta_I : (D_i, d_j) \rightarrow D'_i$): The operation removes *DataSourceItem* $d_j \in D_i$ from the *DataSource* D_i. It returns *DataSource* D'_i where $D'_{i_N} = D_{i_N}$ and $D'_{i_I} = D_{i_I} \backslash \{d_j\}$.

Complex Operations

More complex operations like *Split*, *Merge*, *Move* done in the database model can be propagated to the SQL query model as well. In the SQL query model these operations are simply composed of the mentioned atomic operations.

5.6 Change Propagation in the Graph

The database model operations described in Section 5.5.3 have an impact on the queries in the SQL query model. During the evolution process, changes in the database model have to be propagated to the SQL query model, where the modeled queries are adapted to the current database model. However, sometimes a simple direct propagation is not correct. For instance, if a new column is added to the database table, we may not want to add a new column to the *SELECT* clause of the query. For this reason we propose so-called *propagation policies*, which influence the behavior of the propagation. The policies are defined for the vertices of the query graph, which participate in the change distribution process.

We distinguish the following propagation policies:

- **Propagate**: This policy allows to perform the change directly. Subsequently, the propagation is passed on the following vertices in the change process.
- **Block**: This policy does not allow to perform the change. The subsequent propagation is stopped and the following vertices in the change process are not visited.
- **Prompt**: The system asks a user which of the two above policies should be used to continue.

5.6.1 Query Graph Operations

As mentioned before, the SQL query model operations described in Section 5.5.3 are atomic operations, i.e., they cannot be divided into smaller operations. In fact, these atomic operations consist of many smaller steps called *graph operations*, which modify the query graph of the SQL query model. In the following definitions G_Q represents a query graph $G_Q = (V, E)$. We distinguish the following graph operations:

- **CreateVertex** ($\gamma_v : (G_Q, v) \rightarrow G'_Q$): The operation returns graph $G'_Q = (V \cup \{v\}, E)$.

- **CreateEdge** ($\gamma_e : (G_Q, v_{source}, v_{target}, e_{type}) \rightarrow G'_Q$): The operation creates edge $e = (v_{source}, v_{target})$ such that $EdgeType(e) = e_{type}$ and returns graph $G'_Q = (V, E \cup \{e\})$.
- **RemoveVertex** ($\beta_v : (G_Q, v) \rightarrow G'_Q$): The operation returns graph $G'_Q = (V\backslash\{v\}, E \bigcap \binom{V\backslash\{v\}}{2})$.
- **RemoveEdge** ($\beta_e : (G_Q, e) \rightarrow G'_Q$): The operation returns graph $G'_Q = (V, E\backslash\{e\})$.
- **ChangeLabel** ($\lambda : (v, l) \rightarrow v'$): The operation returns query vertex v', where vertex type $v'_{type} = v_{type}$ and label $v'_L = l$. (For instance, it returns the *DataSourceItem* vertex with a new name.)
- **ChangeConnectionType** ($\eta : (C, t) \rightarrow C'$): This operation returns *CombineSource* vertex C' with connection type $C'_T = t$.
- **ResetContent** ($\rho(G_Q, C)$): Since the visualization model visualizes the query graph, they have to be synchronized. This operation is used to signal the parent visual component C that a change in the query graph G_Q has been done and the content of the visual component has to be updated.

These atomic operations are combined in the set of composite operations which are called in the evolution process. All these functions with their algorithms are described in [19].

Example 8. *First, Algorithm 1 creates a new* FromItem *vertex in the corresponding* From *vertex and connects it with appropriate vertices. Subsequently it traverses to the* Select *vertex, where it creates corresponding vertices and edges using algorithm* DistributeCreatingDatasourceItemSelect *(see [19]). Finally, it traverses to the* OrderBy *vertex, where it creates corresponding vertices and edges using algorithm* DistributeCreatingDatasourceItemOrderBy. *Figure 5.6 depicts adding of a new* DataSourceItem *OrderDate to the* DataSource *component using Algorithm* DistributeCreatingDatasourceItem *and to the* From *component using Algorithm 1. In Figure 5.6 the original elements are black and the new elements of the query graph are highlighted with a red color.*

Algorithm *DistributeCreatingDatasourceItemSelect* creates a new *SelectItem* vertex in the *SELECT* clause. Then it checks, whether the *GroupBy* vertex exists. If it does, it connects the *GroupBy* vertex with the new *SelectItem* vertex. Finally, it traverses to all Alias vertices of dependant queries and applies already mentioned Algorithm 1.

5.7 Implementation and Experiments

The full experimental implementation of the presented approach was incorporated into the *DaemonX* framework. (In fact, Figures 5.2, 5.4 and 5.5 are screenshots of the application.) Our approach adds two new plug-ins into the framework. The first plug-in is a plug-in for SQL query modeling described in Section 5.5 (a screenshot from this plug-in is depicted in Figure 5.4). The second plug-in is

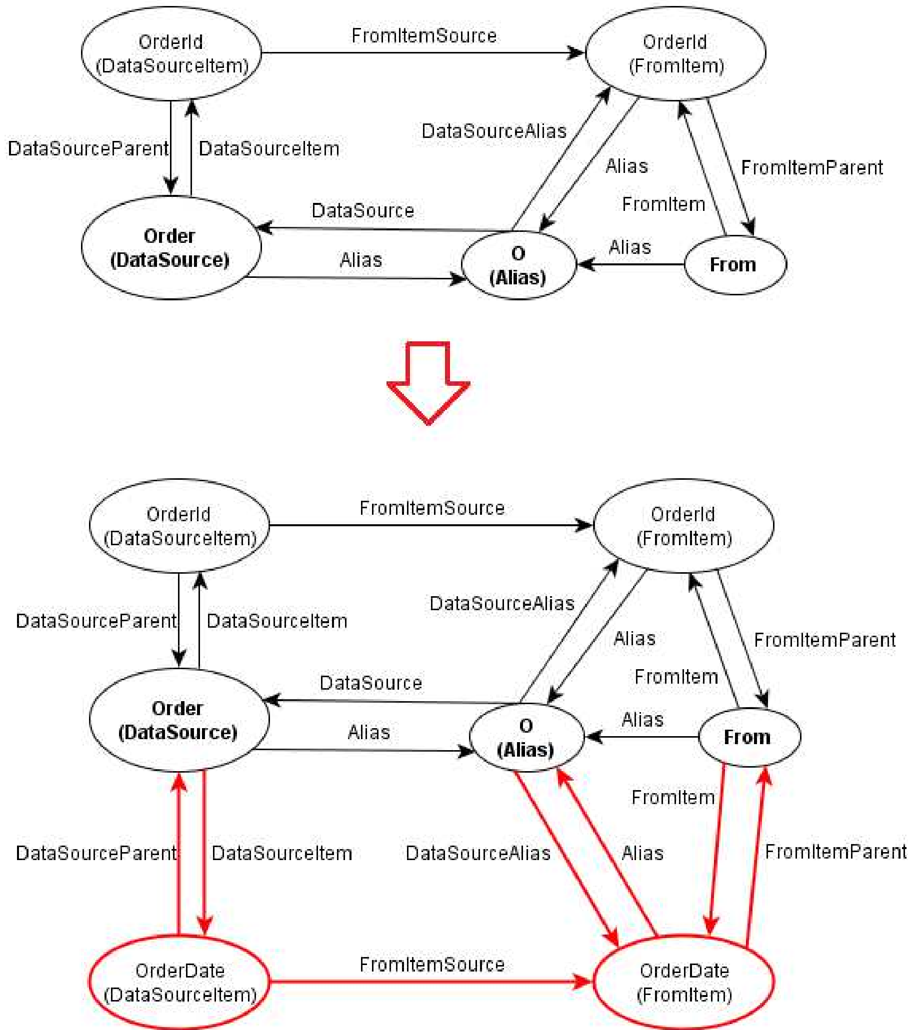

Figure 5.6: An example of adding new *DataSourceItem* to the *DataSource* and to the *From* components

used for evolution propagation from the PSM database model to the SQL query model (as described in Section 5.6).

Since there are no existing applications which provide similar functionality, to be able to compare our approach in a meaningful way we used an existing database project *Adventure Works* [1] to evaluate validity of the algorithms. Next, we measure and compare the effort that must be done by manual update of the queries against (semi)automatic changes done by the algorithms. An inspiration was taken from [99]. We modeled in the database model a set of tables of a given database schema (the list of them can be found in [19]). From this database model (74 tables) we derived to the query model tables as *DataSources*, which we used to model queries and views. Next, we applied various operations over the database model to simulate propagation to the query model. After the propagation the new queries were inspected if they correspond to the expected results. Next, the number of required manual changes to update related queries was evaluated.

To demonstrate a part of the experiments we present an example of adding a new column to the table and its propagation to the related *GroupBy* query. Suppose the following SQL query which model is depicted in Figure 5.7:

Algorithm 1 DistributeCreatingDatasourceItemOnAlias

Require: *Alias* vertex A, change context C
Ensure: Graph operations to create new *DataSourceItem*.

1: $fromVertex \leftarrow A.GetNeighbour(DataSourceAlias)$
2: $newItem \leftarrow$ new $FromItem(C.Name)$
3: $C.Plan \leftarrow CreateVertex(C.G_Q, newItem)$
4: $C.Plan \leftarrow CreateEdge(C.G_Q, C.Originator, newItem, FromItemSource)$
5: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, A, Alias)$
6: $C.Plan \leftarrow CreateEdge(C.G_Q, fromVertex, newItem, FromItem)$
7: $C.Plan \leftarrow CreateEdge(C.G_Q, newItem, fromVertex, FromItemParent)$
8: $C.Originator \leftarrow newItem$
9: $selectVertex \leftarrow fromVertex.GetNeighbour(SelectQuery)$
10: **if** $selectVertex$!= $null$ **then**
11: $\quad DistributeCreatingDatasourceItemOnSelect(selectVertex, C)$
12: **end if**
13: $orderByVertex \leftarrow fromVertex.GetNeighbour(OrderBy)$
14: **if** $orderByVertex$!= $null$ **then**
15: $\quad DistributeCreatingDatasourceItemOnOrderBy(orderByVertex, C)$
16: **end if**
17: $C.Plan \leftarrow ResetContent(C.G_Q, fromVertex)$

```sql
    SELECT
      d.GroupName,
      COUNT(d.Name)
        as NumberOfDepartments
    FROM
      HumanResources.Department as d
    GROUP BY
      d.GroupName
    HAVING
      COUNT(d.Name) > 2
ORDER BY NumberOfDepartments DESC
```

A new column *GroupID* was added to the table *HumanResources.Department*. This change was propagated to the complex *GroupBy* query. The new column was added to all its components. A manual update of the query requires three updates (to *SELECT*, *GROUP BY* and *ORDER BY*). The original query was transformed by algorithm *DistributeCreatingDatasourceItem* (described in [19]) to the new query (its model is depicted in Figure 5.8):

```sql
    SELECT
      d.GroupName,
      COUNT(d.Name)
        as NumberOfDepartments,
      d.GroupID
    FROM
      HumanResources.Department as d
    GROUP BY
      d.GroupName, d.GroupID
    HAVING COUNT(d.Name) > 2
```

```
ORDER BY
    NumberOfDepartments DESC,
    d.GroupID ASC
```

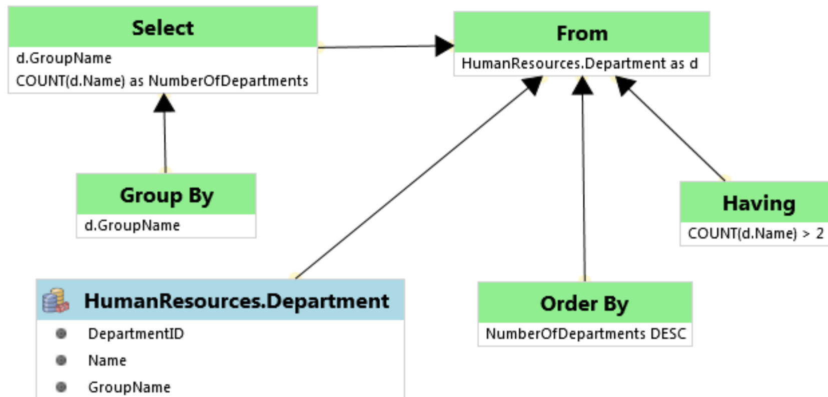

Figure 5.7: Model of the complex usage of complex *GroupBy* query

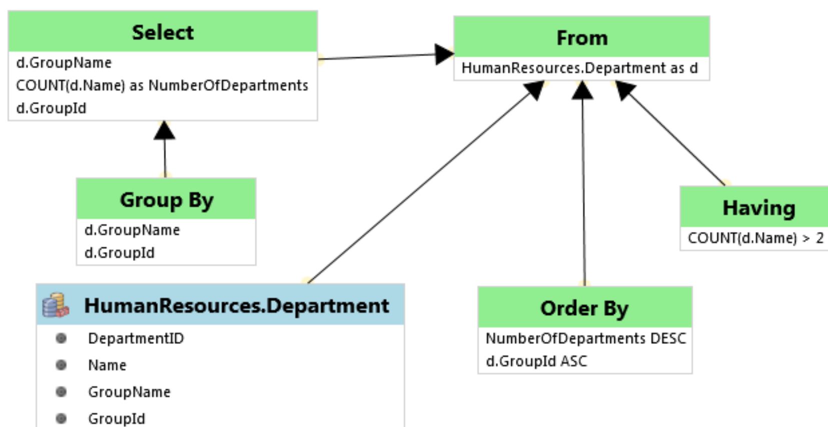

Figure 5.8: Updated model of the complex *GroupBy* query

In general, the full experiment uses all 74 tables with correspondent relations. As SQL queries we used 20 views included in the DB scripts plus 10 additional, more complex queries (involving use of *GROUP BY*, *HAVING*, multiple *JOIN*s, etc.). We updated the SQL table model by adding and removing and renaming various tables and columns, totally by applying 20 changes. Updated queries were verified to return expected results. From the experiment we can observe the following results:

- The number of changes is naturally related to occurrences of tables and columns of database model contained (related) in query models.
- During the experiments 476 query updates were done automatically.
- All changes except adding a new table have an impact on the SQL queries if the constructs are presented in the queries.

- There were found no invalid queries except some edge cases, e.g., removing of the column that was used alone in the *SELECT* clause which leads to an invalid SQL query. The same holds for table removing. This situation should be reported to the user and needs a manual update.

The experimental implementation can be found at this resource[1].

5.8 Conclusion

In this chapter we have focused on a specific part of the problem of change management of applications – evolution of a relational schema and its propagation to respective SQL queries. For this purpose we defined a model representing SQL queries. The main contribution of our approach is the ability to model SQL queries concurrently with the respective schema, to interconnect schema and query models and to analyze changes performed in the database model and to update the query model to preserve their compatibility and correctness. The approach enables the user to perform changes in the database model without an inspection of all the related views / queries by looking for an inconsistency of the queries and the new database schema. Changes in the database schema model are propagated immediately to the respective SQL query model using mutual mapping algorithms.

5.8.1 Future Work

Although the approach presents a new SQL query model and multiple algorithms of model operations and evolution, there exists a number of open problems:

- *More complex constructs*: A natural first extension is towards more complex SQL constructs we have omitted, e.g., use of asterisk in *SELECT* clause.
- *Support for other languages*: we can consider also other query languages, such as XQuery for XML data or SPARQL [130] for RDF data. And, last but not least, there is an important aspect of semantics of the data and queries which may highly influence the propagation mechanism.

[1] `http://www.ksi.mff.cuni.cz/~polak/daemonx/`

6. Service Interfaces and Business Processes Evolution

In the previous two chapters we focused on evolution of XML and relational schemas and propagation of the changes to related queries. This idea can however be extended to a distinct area of business processes. If there exists a model describing a business process, it is possible to analyze changes and define algorithms providing evolution of a business process – i.e., not only data or query model. In this chapter we present a method which derives and adapts optimal communication XML schemas for a given conceptual schema of a business process, complemented with a conceptual schema of the exchanged data. The model presented in this chapter in the context of the five-level evolution management framework is depicted in Figure 6.1. The approach was implemented within the DaemonX framework including experiments proving the concept. The approach was presented in [75].

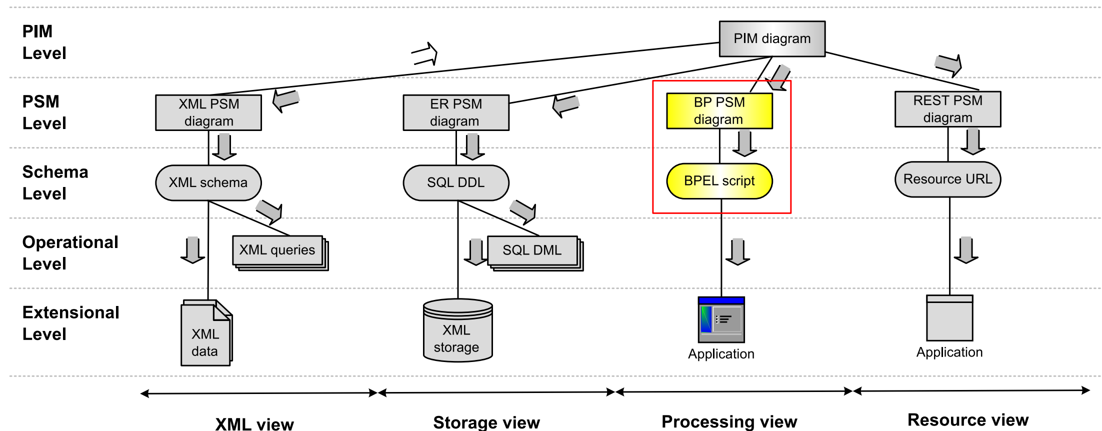

Figure 6.1: Location of the business process model and respective change propagation in the five-level evolution management framework

6.1 Introduction

Nowadays, there are various popular notations of business process modeling, such as, e.g., *BPMN*. Modeling of business processes at the conceptual level allows domain experts to cooperate in the analysis and to design the software at a higher level of perspective. Business processes are modeled as *tasks* and *communications* among them. One communication is usually represented by one *data object* transferred from one task to another. Hence, business process modeling notations are usually extended with a conceptual data modeling language, e.g., UML, to support modeling of the data objects. It is also common that related *business rules* are expressed in special languages, e.g., in OCL, to represent restrictions on the data objects.

The business process model can be then translated to a set of *web services* and executable BPEL scripts which orchestrate all related parts together. Besides this, it is necessary to define the structure of each data object in the business model. Since web services usually communicate by exchanging XML data, a

software architect also has to define XML schemas of the data, e.g., using the XML Schema language. In addition, there exist common requirements for the structure of the XML schema used in web services – the structure should be readable, it should satisfy the conditions of business rules and it should not contain redundant data.

Situations such as changes of BPEL scripts or XML schemas can raise several problems. In particular, adaptation of business processes covers many related issues. The two key ones are the *evolution of a software* and *integration with other softwares*.

- A change of the conceptual model of a data object can cause a malfunction of the whole system. It is necessary to change an XML schema of the data object.
- A change of business rules of a data object can cause an inadequate behavior and response of the system. A previous XML schema of the data object was designed for previous business rules.
- Adding or removing of tasks can cause a malfunction of the whole system. It is necessary to change an XML schema of the data object reconnected to the new or different tasks.

In particular in this chapter we focus on the sub-problem of changes in conceptual models and their consequences on the data objects. For example, assume that for one business process sequence flow *Order* which transfers data about user's order a data object was designed. This data object represents a part of the conceptual data model of the whole system. The business of the company was later extended and it is necessary to have more information about the user. Therefore, the conceptual model of the system was extended with two more information about the user, e.g., *phone* and *email*. This information has to be transferred everywhere where the user's information occurs. That is why the sequence flow *Order* and all others, which contain the user's information, have to be updated. In particular, the task is:

1. *to identify all XML schemas working with the user's information*

   and

2. *to define a change for each found XML schema* from the first phase.

We present a method which derives (and adapts) optimal XML schemas for a given conceptual schema of a business process, complemented with a conceptual schema of the exchanged data. We derive several XML schemas from the given inputs and choose the one which has minimal values for three proposed metrics which measure the quality of an XML schema document with respect to the conceptual model of the exchanged data and business rules applied on them. Hence, we also discuss the problem of change adaptation of the derived XML data. In other words, an *optimal communication XML schema* is a schema which has the best result given by the several quality metrics with respect to the business process conceptual schema defined and applied on the schema. The quality metrics measure redundancies in exchanged XML documents and complexity of evaluating business rules specified by the business process conceptual schema.

6.2 Related Works

In this section, three papers related to generation of an XML schema from an other model are discussed. The last part of this section provides a summarizing comparison of all discussed papers and briefly introduces the main issue of this work.

6.2.1 UML and XML Schema

Paper [96] describes an approach for mapping between the UML class diagram and the XML Schema using a three-level design approach. The main aims of the paper are:

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

Three-Level Design Approach

The paper introduces the three-level design approach consisting of the following parts:

- Conceptual level
- Logical level
- Physical level

The conceptual level is the UML class diagram with several non-standard annotations, such as a primary identification of a class. The logical level uses a defined UML profile for an XML Schema (i.e., an extension of the UML). It uses mainly stereotypes of classes to define XML Schema concepts in the UML profile. A detailed description of this UML profile can be found in Section 3 of paper [96]. The physical level defines data structures by using the implementation language – in this case the XML Schema. It is a particular XML Schema generated from the logical level model. There is a direct bidirectional mapping from the logical to the physical level.

Conceptual to Logical Level Mapping

An XML schema is hierarchical structure from its nature. Hence, generating of a logical level model from a conceptual one requires to choose one or more conceptual classes to root the tree-hierarchy. The approach proposed by the authors of paper [96] aims to minimize redundancy and to maximize connectivity of XML data structures (i.e., make the structure compact without redundant elements) and it is directly based on paper [9]. There are however differences in the algorithm because of differences between *Object Role Modeling* (ORM) [57] and UML. One of the main differences is that the concept of *Anchors* used in paper [9] to identify the direction of the nesting was not used in this approach. Instead, the authors use the navigation and cardinalities of a relation.

The mapping consists of the following four steps:

- **Generate Type Definitions**: It creates type definitions for each attribute and class in the conceptual diagram. It means, creation of complex types and primitive types with restrictions.
- **Determine Class Groupings**: It determines how to group and nest conceptual classes based on relations between them. It uses simple rules based on the navigation of relations and their cardinalities. These rules try to reduce redundancy of the created XML schema and to maximize the connectivity.
- **Build the Complex Type Nestings**: After identifying of nesting directions from the previous step, it is necessary to perform complex type nesting. This is done in this step which creates the hierarchical structure. This is done by an analysis of uniqueness key and mandatory constrains of elements. An example can be elements *Lecturer* and *Subject*. If a *Subject* can be headed by more than one *Lecturer*, nesting *Subject* inside *Lecturer* causes redundancy. Next, if a *Subject* does not need to be headed, then nesting a *Subject* inside a *Lecturer* makes impossible to represent a *Subject* without a *Lecturer*.
- **Create a Root Element**: At the end, it is necessary to define the root element of the created XML schema. A new element representing a root of the schema is created and nested elements created from the previous step are added as subelements.

Discussion

The paper presents an approach to automatic generation of an XML schema from a UML model. It uses a three-level design which is similar to the MDA. From the user's point of view, this method is useful for reducing redundancy and maximizing connectivity in the created XML schema. Beside advantages of this paper, it has some limitations. It needs additional constructs in UML. In the solution presented in this paper, it is also impossible to define a mixed content.

6.2.2 XSEM – A Conceptual Model for XML

Paper [90] describes an approach for conceptual modeling of XML data. It presents a two layer design. The first layer, *XSEM-ER*, represents an overall conceptual layer. It uses an extension of the ER. The second layer, *XSEM-H*, represents hierarchical organization of structures from the first layer. It introduces the transformation from XSEM-ER to XSEM-H by *Operators*. It is possible to create more than one XSEM-H model from one XSEM-ER model. It is also possible to use only a part of XSEM-ER in the transformed XSEM-H model.

During the transformation of XSEM-ER to XSEM-H, an interconnection between two models is created. Thanks to this, it is possible to support evolution operations, which were described in [93]. The evolution is based on a propagation of changes from the XSEM-ER model to the connected XSEM-H model. Changes on XSEM-ER are made by atomic operations, which have their counterparts on XSEM-H.

The visualization of the XSEM-H model is the most important part of this paper for the purpose of our approach. This model and the visualization with

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

XSEM-H

XSEM-H is an extension of the UML class diagram. It takes a part of class diagram components and adds new components. Next, there is defined a bidirectional mapping between XSEM-H and the XML schema.

The new components of the XSEM-H are as follows:

- **Class**: It represents a class from the XSEM-ER model. Each class has a name, a label and it contains attributes. It can be connected with other classes by associations.
- **Attribute**: It represents an attribute from the XSEM-ER model. It can exist only as a part of the class. It can represent zero or one attribute from the XSEM-ER model. Each attribute has a name, a data type, a cardinality and a position.
- **Association**: It represents a connection between two XSEM-H classes in the relation parent-child. It can represent a whole set of XSEM-ER associations.
- **Content Choice**: It represents *choice element* of the XML schema.
- **Content Sequence**: It represents *sequence element* of the XML schema.
- **Content Set**: It represents *all element* of the XML schema.

This functionality of the mapping was implemented in tool XCase [121] and visualization of XSEM-H was implemented in DaemonX framework as a modeling plugin.

Discussion

The paper presents a novel and detailed approach for conceptual modeling of XML data that uses two interconnected layers of abstractions. In related paper [93], the evolution process between these two layers is described. Presented approach was experimentally implemented in [121] and [119].

6.2.3 An Extension of Business Process Model for XML Schema Modeling

Paper [80] describes an approach for deriving optimal communication XML formats for a given conceptual schema of a business process with a conceptual schema of exchanged data. The authors use the approach of MDA. They introduce PIM and PSM models for a conceptual model of exchanged data and they further extend it with OCL constraints over the PIM model.

Metrics

An optimal XML format is defined by two metrics computed over the PSM model. To choose the optimal PSM model, it is necessary to create all possible PSM models from the given PIM model. Hence, this approach has an exponential complexity in all cases.

This paper is however important because of the presented metrics:

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

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

Discussion

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

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

6.2.4 Comparison of the Related Works

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

Paper [96] describes an approach to three-level conceptual modeling and to generation of a logical level using UML and UML profile. It uses a set of rules to define nesting and hierarchical structure.

Paper [90] describes an approach to conceptual modeling and to generation of the PSM model. It defines an extended ER model for conceptual modeling. It uses own PSM model XSEM-H. It introduces a transformation, which allows multiple PSM models for one PIM model. The authors continue with the work and introduce methods for propagation of evolution operations.

Paper [80] describes an approach to conceptual modeling and to generation of the PSM model by defining own PIM and PSM model. It uses metrics to choose an optimal PSM model from the set of all possible PSM models.

In this chapter, we focus on the problem of derivation an optimal XML schema for business process models and on the influence of changes in the business process

model on the generated XML schemas. The main motivation of this solution was to evolve business model and to generate related optimal communication schemas in a (semi)automatic way to limit manual effort of user. This solution brings benefits in situations of complex business models and communication schemas where the analysis of the process change and subsequent update of the schema is a non-trivial task. We focus especially on the following problems:

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

6.3 Business Processes and Conceptual Modeling

First of all we will provide a brief overview of modeling tools related to business processes which we exploit in the rest of the chapter. We refer the interested reader to the respective specifications for further details.

The *Business Process Model and Notation* (BPMN) describes graphical notation that business process analysts and users can use to model and to define business processes. It is also used as a support for process interactions and a documentation of a system. A business process diagram consists of five primary elements: (1) *flow objects* divided into *activities*, *events* and *gateways*, (2) *connection objects* which connect flow objects (we distinguish *sequence flows* and *message flows*), (3) *swimlanes* used to organize processes and responsibilities in the diagram, (4) *artifacts* extending basic BPMN with *associations*, *groups* and *text annotations*, and (5) *data objects*.

The UML specification contains a lot of different types of diagrams. The class diagram is the most important one for the purpose of our solution. It divides the modeled structures of a system into *classes* with different *attributes* representing their properties and *relationships* representing relations to other classes, such as *association*, *aggregation*, *composition* or *generalization*. In the area of business processes UML is used to describe the structure of data objects.

The OCL is a formal language used to describe expressions over UML models, especially class diagrams. These expressions usually define invariants and conditions over objects described in the model. In the area of business processes we use OCL to create business rules connected to data objects. Each OCL expression has a *context*, i.e., an instance of a class from the class diagram. Expressions consist of *paths* navigating through the class diagram and *functions* from the standard OCL library.

6.4 Proposed Approach

Primarily we had to extend *DaemonX* for the purposes of business process modeling indicated in Section 6.3. Thanks to its plug-in-ability it was not a difficult task. First, we used the common conceptual model of the whole problem domain of the system (the general PIM). Second, since the data objects are usually associated with only a part of this problem domain, we use a new model called *PIM-View* for this purpose. It is a general UML class diagram extended with a new value. Each class contains a property *count*, where *count* $\in N_0 \cup \{*\}$ is the maximum number of instances of the class assigned by a domain expert (we will show its usage later). Third, the resulting optimal XML format is stored as a conceptual model XSEM PSM. Fourth, for business process modeling, we have implemented a new BPMN model called *PIM process* which is a classical BPMN model extended with one new artifact named *data artefact*. It represents the data exchanged in the business process model. Fifth, to express business rules over a conceptual model of the data objects we use OCL. This functionality was introduced to *DaemonX* within Master thesis [101]. Figure 6.2 depicts the MDA architecture of the models. Note that the figure shows also the schema and physical level which we omit for space limitations. For the formal definitions of the models see [74].

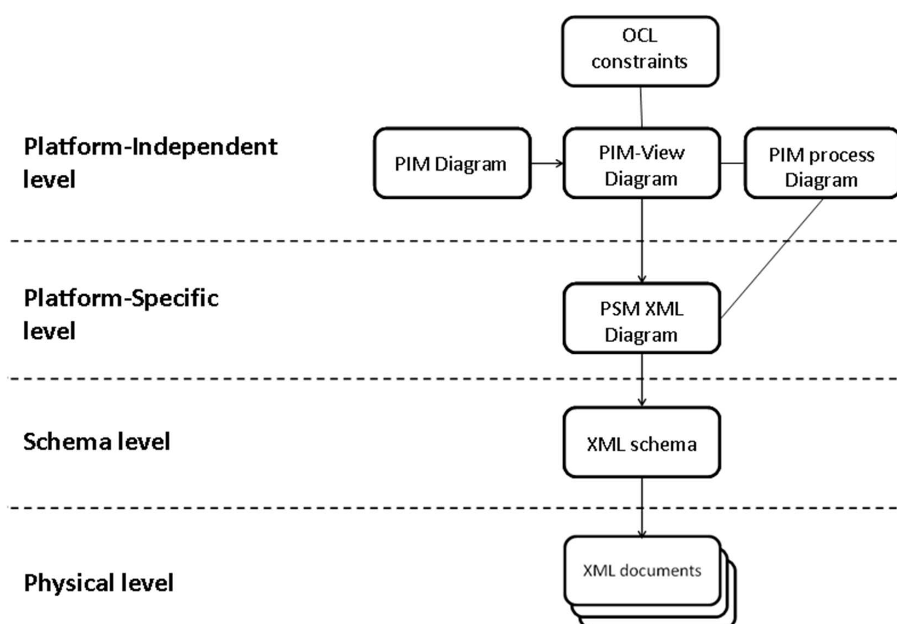

Figure 6.2: MDA and business process modeling in *DaemonX*

For an illustrative example consider Figure 6.3. The PIM diagram contains the conceptual model of the whole problem domain. The PIM-View diagram is based on the PIM diagram. Mapping between elements of diagrams are depicted by blue lines. Both diagrams have to be created by a domain expert. OCL constraints can be created too. They have to be connected to elements from the PIMs. This can be done by choosing correct values in drop-down lists. The mappings are depicted by yellow lines. The PIM process diagram has to be created by the domain expert too. Its data artifacts have to be connected to the created PIM-View diagram and, consequently, the XSEM PSM. This mapping is depicted by green lines.

The final optimal XML schema (format) is stored as a conceptual model presented in paper [90]. However, we had to extend this XML conceptual model with specialization, keys and key references to support all functionalities used in our approach. This extension is straightforward and based on XML Schema constructs inheritance, key and key-ref. Therefore, we omit its explanation. The business process model is modeled by BPMN and business rules are described in OCL.

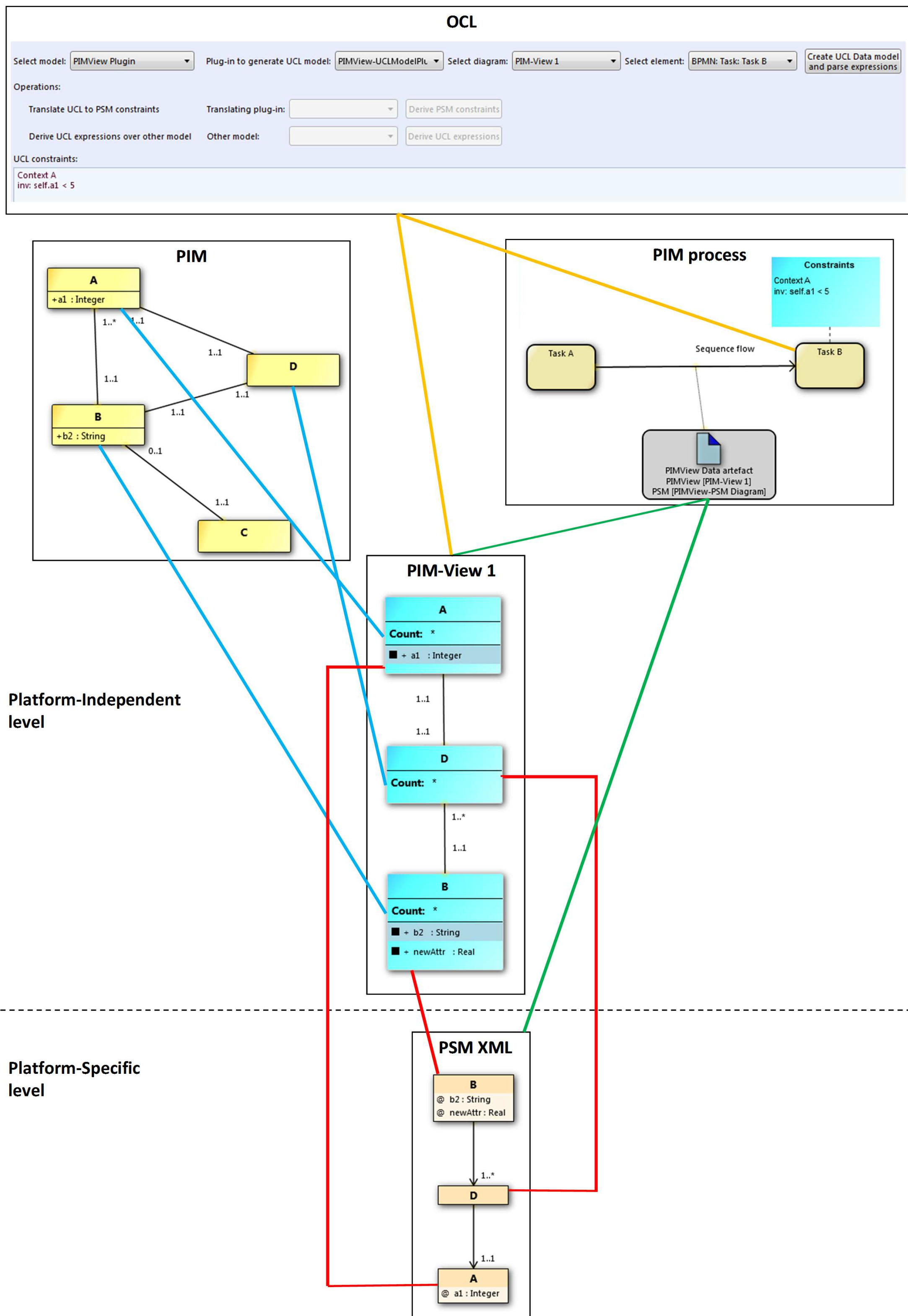

Figure 6.3: PIM and PSM example of business process modeling

As we have mentioned, we also extended BPMN with one new artifact called *Data Artefact*. It can be connected only with sequence flows, whose source is an activity. It represents the data exchanged in the business process model. During creation of this element, it is necessary to define a PIM-View diagram, which represents a conceptual model of this data. After derivation of an optimal XML schema, it stores the connection to the created XSEM PSM diagram. Figure 6.4

depicts the whole architecture of our models.

Our approach can be divided into two parts. The *derivation part* (Section 6.4.1) involves derivation of several XML schema documents for given inputs, computing their quality and selecting the optimal one. The *evolution part* (Section 6.4.2) involves the analysis of user's changes and their influence on derived XML schema document.

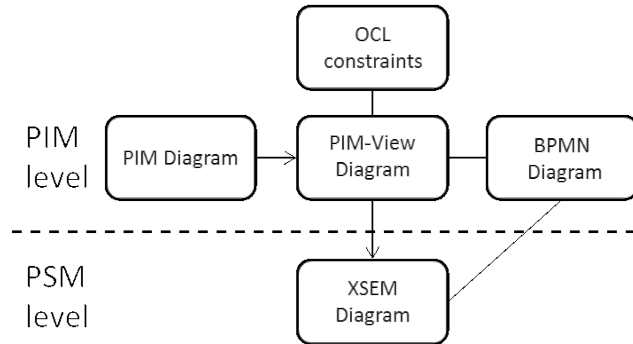

Figure 6.4: Architecture of *DaemonX* models related to business process modeling

6.4.1 Derivation Part

We re-use and extend the approach of paper [80] to define features of an optimal communication XML schema for a given conceptual schema of a business process, complemented with a conceptual schema of the exchanged data. Especially, we re-use two metrics and we add one new. In particular, paper [80] ignores cardinalities of relations which leads to loss of information, specifically for cardinalities with the bound value equal to zero. Hence, we use the idea presented in [96] and we study navigation and cardinalities of relations in the conceptual model of the exchanged data. First, we generate candidate XML schemas, then we apply the given metrics and choose the optimal one.

In the whole derivation process, we work with a forest of XML schemas. The root of an XML schema is an artificial class, which is not important for derivation process. We call it *schema root class* and its children we call *root classes*.

Relation Types and Cardinalities

The first step in deriving an optimal XML schema is to derive XSEM PSM schemas, i.e., to create hierarchical tree structures from general PIM graphs. First, let us discuss types of relations. In general, relation types of a PIM-View model are equivalent in semantics to relation types of a UML class diagram. *Association* is a general relationship between two classes. Hence, we cannot use this information to specify the nesting of classes. *Aggregation* is a specialization of the association. It specifies a whole-part relationship. In basic aggregation relationships, the lifetime of a part class is independent of the whole class's lifetime. Therefore, we also cannot use this information. *Composition* is a stronger form of the aggregation, where the whole and parts have coincident lifetimes. Thus, we can use only this type of a relation to specify the nesting of classes. If we do

not nest the part class into the whole class, we have to use keys to refer to this part class. We always nest classes according to the direction of the composition.

On the other hand, cardinalities are the main indicator for nesting of classes. Note that a domain expert can change cardinalities in a PIM-View diagram, i.e., they do not have to be the same, as in the PIM diagram. For us only few types of cardinalities are important: $\langle 0,1\rangle$, $\langle 0,m\rangle$, $\langle 0,*\rangle$, $\langle 1,1\rangle$, $\langle 1,m\rangle$, $\langle 1,*\rangle$, $\langle m,n\rangle$, $\langle m,*\rangle$, where $m,n \in N_0$ and $m,n > 1$ and $m,n \neq *$. We will work with two features of the XML schema:

- *Data redundancy* means occurrence of the same data more than once.
- *Compactness* (or *connectivity*) means storing related data together in the hierarchical structure of the XML schema.

Having a relation between classes A and B, we will discuss only the most important cardinality combinations (denoted $card(A)$ and $card(B)$):

1. $card(A) = \langle 1,1\rangle$ and $card(B) = \langle x,y\rangle$ where x and y are ≥ 0 (see Figure 6.5 (a)): If we nest class B into class A, it leads to data redundancy and to creation of a separate root class for class B in the XML schema. Therefore, we nest class A into class B.

2. $card(A) = \langle x,y\rangle$, where $A.x \geq 1$ and $A.y$ is arbitrary and $card(B) = \langle 0,y\rangle$, where $B.y \in \{1,m,*\}$ (see Figure 6.5 (b)): If we nest class B into class A, we get the same problem as in the previous case. If we nest class A into class B, we create data redundancy, but there is no need to create a separate global class. Therefore, we nest class A into class B.

3. $card(A)$ and $card(B)$ are of type $\langle x,y\rangle$, where $x \geq 1$ and $y > 1$ (see Figure 6.5 (c)): In this case, we try to reduce data redundancy by nesting the class with the lower upper cardinality into a class with the higher upper cardinality. If $A.y < B.y$, we nest class A into class B. If $A.y \geq B.y$, we nest class B into class A. If $A.y = B.y = *$, we can use only the direction of the relation, if it is defined, i.e., if the direction is from class A to class B, we will nest class A into class B. If the direction is not defined, we will process the relation like in the last case.

4. $card(A) = card(B) = \langle 1,1\rangle$ (see Figure 6.5 (d)): In this case, we do not use cardinalities or directions to define nesting. We process these relations after all the others. First, we compute the depth of subtrees of class A and class B. If one of the depths is smaller, we nest the subtree with smaller depth into the other one. If the depths are equal, we check if class A or class B is nested in some other class. If, e.g., class A is nested and class B not, we nest class B into class A. In other cases we nest class A into class B.

5. Other relations (see Figure 6.5 (e)): In all other cases we can not use cardinality information for nesting. Therefore, we create two keys[1]. One becomes the child of class A and it points to class B by the relation's orientation. The other one vice versa.

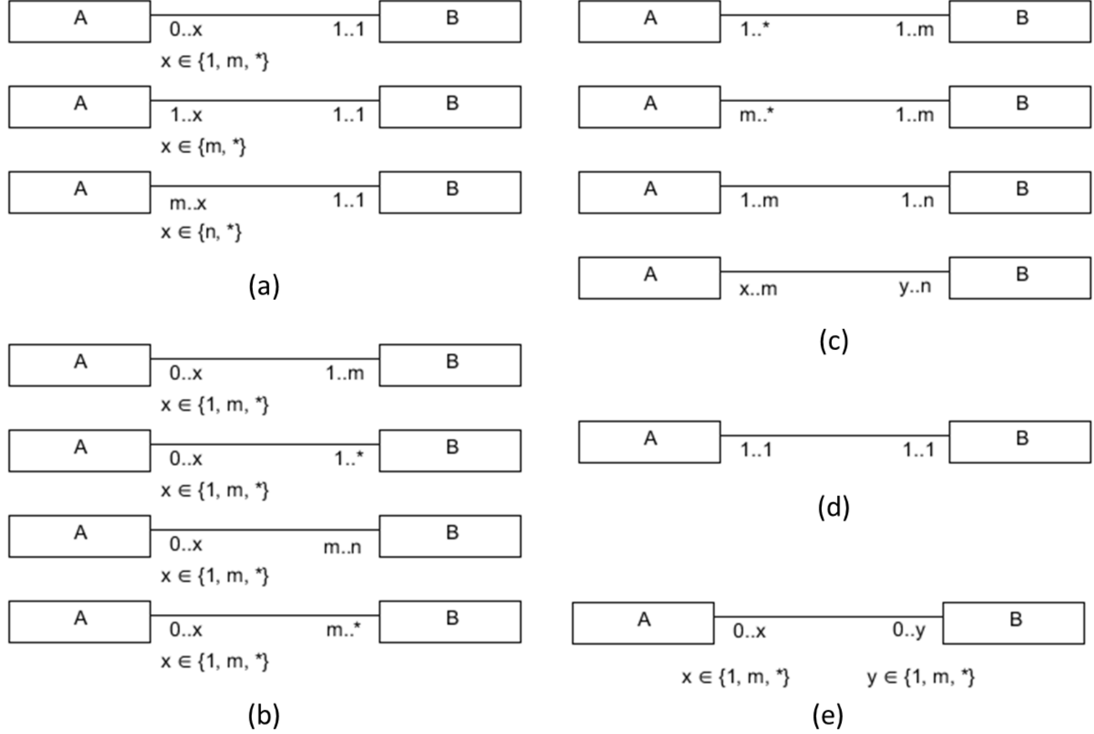

Figure 6.5: Cardinality examples

Note that there can be also generalizations in a PIM-View. Hence, we use a special operation to *hide* generalizations in the first part of the derivation process (we need to re-create them later). This operation is necessary to simplify creation of the hierarchical structure. It replaces each generalization tree in the PIM-View diagram with one new class. Each relation connected to classes in the generalization tree is reconnected to this new class. This new class stores information about the hidden classes, generalizations between them and relations connected to them.

The inverse operation then creates the whole generalization tree in XSEM PSM diagram (here it is specialization) and reconnects XML schema relations (associations, specializations) according to the hidden relations from PIM-View diagram. After this operation, there can be relations in XSEM PSM diagram violating conditions of XML schema structure. There can be XML schema relations violating the condition of rooted directed tree. Therefore, a *corrective operation* is required. It removes the XML schema relation and replaces it with two new keys like in the discussion on relation cardinalities (see Section 6.4.1).

Derivation of the First XML Schema

The derivation algorithm[2] starts with derivation of the first XML schema which is used in the next part to derive other XML schemas. For the sake of simplicity we assume that the PIM-View diagram is an acyclic graph (with hidden generalizations). This creates a copy of the PIM-View diagram with a modified structure.

[1] We use classical XML Schema `key` and `keyref` constructs.

[2] For formal pseudo codes of all the mentioned algorithms see [74].

First, we find nested classes according to our discussion about relation types and cardinalities (see Section 6.4.1). The nested classes define the hierarchical structure of our XML schema. However, after this first definition there might be some duplicities in nesting – e.g., one class is nested into two different classes. We begin to build the XML schema tree. At first, we find *root classes*, i.e., classes which are not nested in any other class. For each root class R, we recursively search classes, which nest into R and we construct the respective XML schema tree. During this construction phase, we store information about duplicated nesting of classes. After processing the whole graph, we take classes that were used more than once and we replace them with keys like in the last case of relation cardinalities (see Section 6.4.1, point e)). At the end, we re-create the hidden generalizations. This operation creates the first XML schema, which is derived from the given PIM-View diagram.

Derivation of Other XML Schemas

The first derived schema contains lots of often unrealistic XML Schema keys and key references which can be replaced with XML schema parent-child relations instead. This replacement leads to a more redundant XML schema, but it is more compact and for some metrics it can have better results. Figure 6.6 depicts examples of possible reducing of keys.

In the process of replacing, it is necessary to switch the parent, the child and the cardinalities of selected XML schema relations in particular cases. The main part of the operation is to reverse some relations in the tree, s.t. one particular class is a new root class of the tree. However, in some cases it is not possible:

Specializations We cannot switch the parent and the child of a specialization. This replacement changes the stored information derived from PIM-View diagram generalizations. An example is depicted in Figure 6.7 (a).

Cardinalities In this case there is a problem with cardinalities which have the lower bound equal to zero. Replacement of these cardinalities does not solve the problem of keys. Since we do not want to lose any information, we need to create new keys to preserve the information. Examples are depicted in Figure 6.7 (b).

Hence, to be able to replace the keys, we define two conditions when it is possible:

Condition 1. *(Key Relation) Let R'_k be an XML schema relation, where the ends of the relation are (c', k') for a defined key k' and c' being a parent class of k'. R'_k was derived from a PIM-View relation R. Let c_1 be a PIM-View class, which c' was created from and let c_2 be a PIM-View class, which k' was created from. Then, conditions (I) or (II) has to be satisfied for R:*

1. *both cardinalities of R have lower bounds > 0,*
2. *$card(c_1) = \langle 0, x\rangle \wedge card(c_2) = \langle z, x\rangle$, where $x \in (N \cup \{*\})$, $z \geq 1$.*

Condition 2. *(Reverse Path) Let c'_r be a class referenced by a defined key k'. Then we have the XML root-path (R'_1, ..., R'_n), where child of R'_n is $C'_n = c'_r$, C'_0 is the root class and $\forall i \in \{1..n\}$: parent of $R'_i = C'_{i-1} \wedge$ child $R'_i = C'_i$. It is the*

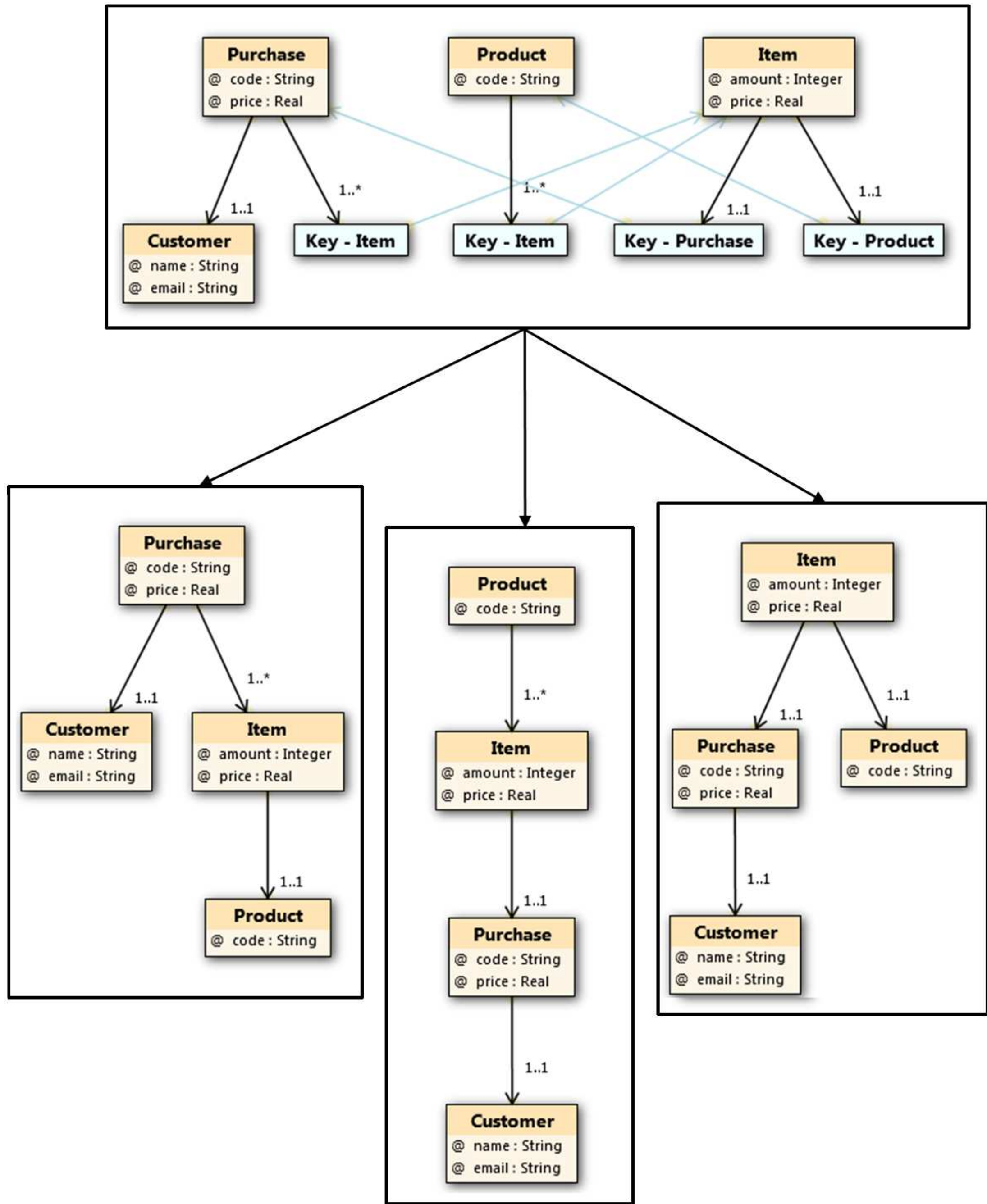

Figure 6.6: Examples of reducing keys in a PSM schema

shortest path from the given class to the root of the tree. Then, conditions (a) and (b) must be satisfied for each R'_i*, where* $i \in \{1..n\}$*:*

1. R'_i *is not a specialization,*

2. R'_i *is derived from PIM-View relation* R_{pv} *and both cardinalities of* R_{pv} *have lower values* > 0.

Condition 1 checks cardinalities of the PIM-View relation from which the XML schema relation was derived. This XML schema relation is depicted in Figure 6.8 by the green color. The blue XML schema relation is derived from the same PIM-View relation but its parent and child are switched. Condition 2 checks

relations and specializations in the whole XML root-path. This XML root-path is depicted in Figure 6.8 by the red color. Both conditions check cardinalities, but with different strictness. The first condition checks the relation which is not going to be reversed. Therefore, it allows 0 for the lower bound of the parent cardinality. The second condition checks relations and specializations which are going to be reversed. Therefore, it does not allow any 0 for lower bounds of both cardinalities and it also does not allow any specializations.

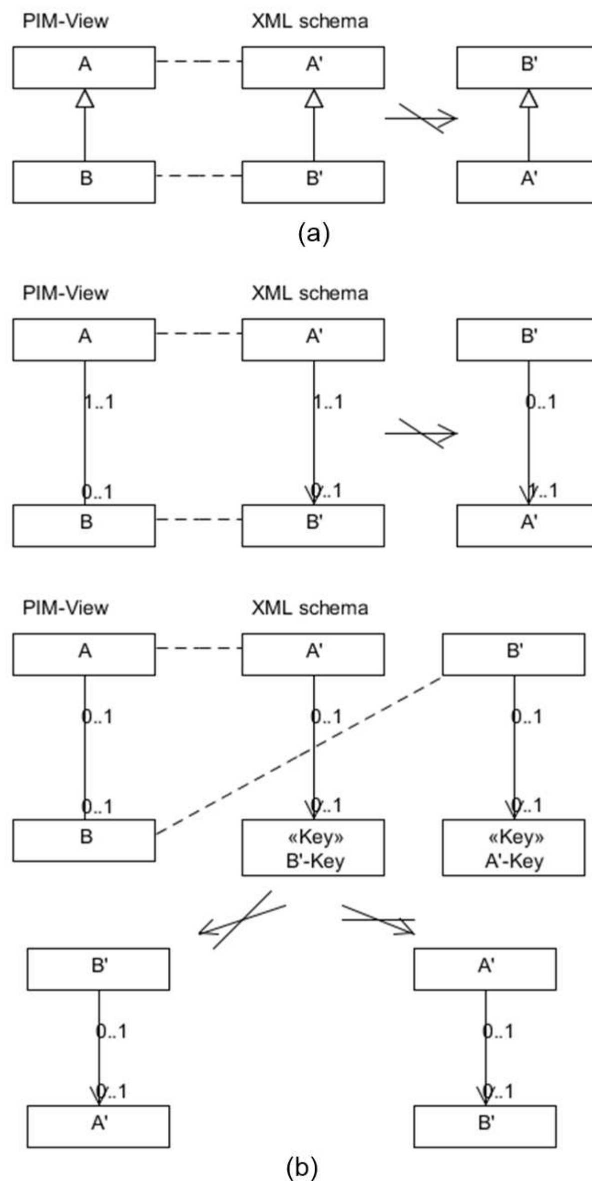

Figure 6.7: Replacement problems for specializations (a) and cardinalities (b)

The algorithm of this part of the derivation process uses only one operation which *replaces* the key with a referenced tree. The key can be replaced with the referenced tree only if it satisfies both the conditions. This part of the algorithm is, hence, a single recursive function, which iterates through all unprocessed keys. If a key satisfies the conditions, the algorithm applies the replace operation and starts a new recursion. The result of this algorithm is a set of XML forests, which represent a set of XML schemas. Figure 6.8 depicts an example of the replace operation.

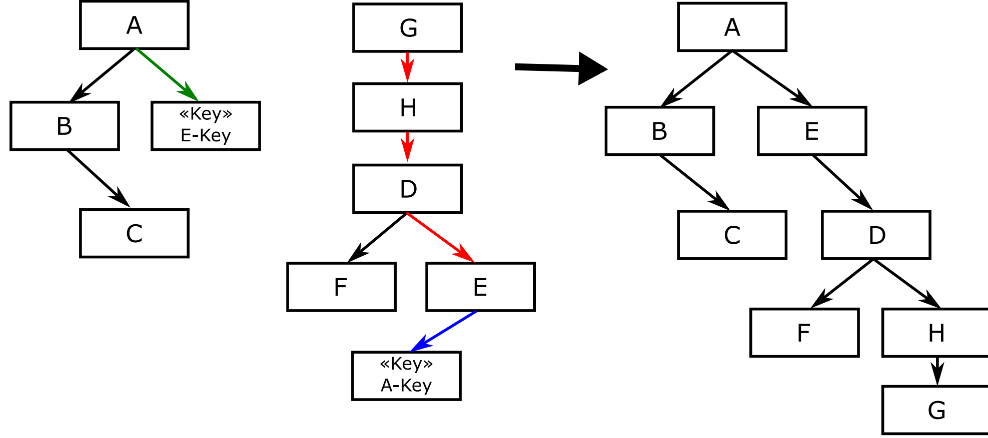

Figure 6.8: Examples of *replace key* operation

Metrics

The described algorithms create several XML schemas. Now, we need to use appropriate metrics to choose the optimal one for the modeled case. As we have mentioned, we use three metrics, two inspired by paper [80], one brand new. (Note that character $*$ which is used in function *count* and in cardinalities represents ∞.)

Redundancy Metric The redundancy metric calculates the measure of redundancy in a given XML schema. This metric is based on [80].

Definition 13. *(XML Class Redundancy). An* XML class redundancy red_{psm} *is a total function, which assigns a positive integer (including $*$) to each XML schema class C' and key K'. For a given XML schema class or key D', it is defined as follows:*

- *If D' is a root class and it is derived from PIM-View class C, then $red_{psm}(D') = count(C)$.*
- *If D' is not a root class, then let us have XML schema relation R' and let C'_{rp} be a parent in R', D' is a child in R'.*
  - *If R' is a relation, then $red_{psm}(D') = red_{psm}(C'_{rp}) * u$, where u is the upper value of $card(C'_{rp})$.*
  - *If R' is a specialization, then $red_{psm}(D') = red_{psm}(C'_{rp})$.*

Note that we work with specializations in this definition. The specialization is not translated into an XML document directly. It just specifies semantics between types of elements in an XML schema. Therefore, the value of XML class redundancy of the child in a specialization is taken from the parent.

Definition 14. *(Redundancy Infliction). Let D' be an XML schema class or key and let R' be an XML schema relation or specialization, s.t. child of $R' = D'$. We say that an XML schema class or key* inflicts redundancy, *if $red_{psm}(D') > 1$ and cardinality of D' in R' is > 1.*

The XML class redundancy reflects redundancy of one XML schema class. Therefore, if $red_{psm} > 1$, it means that the class can represent redundant data in some XML document. However, this is not sufficient, because the parent can be redundant, but its subtree does not have to contain any redundant data. Therefore, we also check child participation in the relation.

Definition 15. *(Redundancy Metric). Let S'_c be a set of XML schema classes and keys for XML schema $\mathcal{S}'$. A* redundancy metric Ω *is a function, which assigns a positive number or zero to XML schema $\mathcal{S}'$ as follows:*

$$\Omega(\mathcal{S}') = \sum_{C' \in S'_c} \begin{cases} 0 & \text{if } C' \text{ does not inflict redundancy} \\ size(C') & \text{if } C' \text{ inflicts redundancy} \end{cases}$$

where $size(C')$ denotes the number of XML schema classes and keys in the subtree of C' including C'. Similarly for the XML schema key.

In general, it represents how redundant can be one PSM schema class. We count the size of the subtree to represent it. It can be computed by the depth-first search traversal of $\mathcal{S}'$.

Context Metric The context metric calculates the position of context classes from business rules. These context classes are important, because they are the beginning of all expressions used in business rules. Therefore, we try to put these classes close to the top of the hierarchical structure of an XML schema. In the following definition we use function $class_{context}$ which is an auxiliary function used to get a context class from a business rule. In other words, it is a function which maps a business rule to a PIM-View class.

Definition 16. *(Context Metric). Let S'_{br} be a set of business rules for XML schema $\mathcal{S}'$. A* context metric Φ *is a total function, which assigns a positive number to an XML schema S' as follows:*

$$\Phi(S') = \sum_{r \in S'_{br}} depth(class_{context}(r))$$

where depth denotes the number of XML schema relations and specializations in the XML root-path for $C' = class_{context}(r)$.

The context metric finds all elements of the BPMN schema, which can have an influence on the connected PIM-View schema. It takes all business rules for found elements and for each it gets a depth of a context class. The context metric is a sum of these depths. It can be computed by the depth-first search traversal of $\mathcal{S}'$.

Path Metric This metric calculates paths in business rules. Business rules contain paths different from a PIM-View. The metric measures the position of the paths in an XML schema. It is similar to the context metric, i.e., we try to place paths from business rules closer to root classes.

Definition 17. *(PIM Path) A* PIM path *P is a sequence $(R_1, \ldots R_n)$ of relations or generalizations from $\mathcal{R} \cup \mathcal{R}_\mathcal{G}$, where $(\forall i \in \{1, n\})(ends(R_i) = (C_{i-1}, C_i))$. C_0 and C_n are called the start and the end of P. Functions start and end return for P the start and end of P, respectively. $\mathcal{P}$ denotes the set of all PIM paths in $\mathcal{S}$. This definition can be applied to the PIM-View schema similarly with restricted sets and functions.*

Definition 18. *(PIM Subpath). Let $P = (R_1, \ldots, R_k)$ be a PIM path in $\mathcal{S}$. A* PSM subpath *of P in $\mathcal{S}'$ is each PSM path $P' = (R'_1, \ldots, R'_n)$, where*

- $R'_i \in \mathcal{R}' \cup \mathcal{R}'_\mathcal{K} \cup \mathcal{R}'_\mathcal{S}$
- $1 \leq k \leq n$
- $(1 \leq i \leq n-1)(child'(R'_i) = parent'(R'_{i+1}) \vee parent'(R'_i) = child'(R'_{i+1}))$
- $(\forall i \in \{1 \ldots n\})(I_{pim_v}(R'_i) \in \{R_1, \ldots, R_k\})$
- $start(P) = I_{pim_v}(start'(P')) \wedge end(P) = I_{pim_v}(end'(P'))$
- *P' is maximal, i.e., adding any PSM relation to P' violates previous conditions*

Functions $start'$ and end' are equivalents to functions start and end from the definition of PIM path, respectively. We use $psmsubpaths(P)$ to denote the set of all PSM subpaths of P in $\mathcal{S}'$.

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

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

The function *paths* is an auxiliary function used to get all PIM paths from business rules $rules(E)$, where $E \in \mathcal{T} \cup \mathcal{O}$. These expressions can contain more PIM paths for one expression, because they can contain functions working with collections. The function always returns a pair (PIM-View path, PSM path).

Definition 19. *(Path Metric). Let S'_{br} be a set of business rules for XML schema $\mathcal{S}'$. A* path metric *Ψ is a total function, which assigns a positive number to an XML schema S' as follows:*

$$\Psi(S') = \sum_{r \in S'_{br}} \{ \sum_{(P,P') \in paths(r)} \frac{length'(P')}{length(P)} * depth_{path}(P') \}$$

where $length(P)$ and $length'(P')$ denote the number of PIM-View relations and generalization and XML schema relations and specializations in P and P', respectively.

Note that we count XML schema relations with key twice to reflect referencing using keys. We also use $depth_{path}(P')$ to denote minimal depth of the first or the last XML schema class of the given path P'. The single path metric computes a positive number for one PIM path. This number represents a position of a given

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

Final Metric Formula To use the metrics together, we need three weights reflecting their importance depending on the nature of a business process. Therefore, we need a project analyst to define the weight of each metric.

Definition 20. *(Final Metric). The* final metric *consists of metric functions and three user-defined positive numbers* (α, β, γ)*. It is a total function* Δ*, which assigns a positive number to* $\mathcal{S}'$ *as follows:*

$$\Delta(\mathcal{S}', \alpha, \beta, \gamma) = \alpha * \Omega(\mathcal{S}') + \beta * \Phi(S') + \gamma * \Psi(S')$$

Example 9. *Finally, we will explain the metrics on an example. In particular, we will use the PIM-View schema depicted in Figure 6.9 and two PSM schemas depicted in Figure 6.10. (Note that, child cardinalities are depicted by the red color.) And we will consider the following business rule:*

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

For the first business rule, there is no PIM path. The expression uses attributes of the context class. For this business rule there is only one PIM path $(Purchase, Item)$*. The lengths of the PIM path and the PSM path are the same, both 1. Therefore, we only need to work with the beginning or the end of the path.*

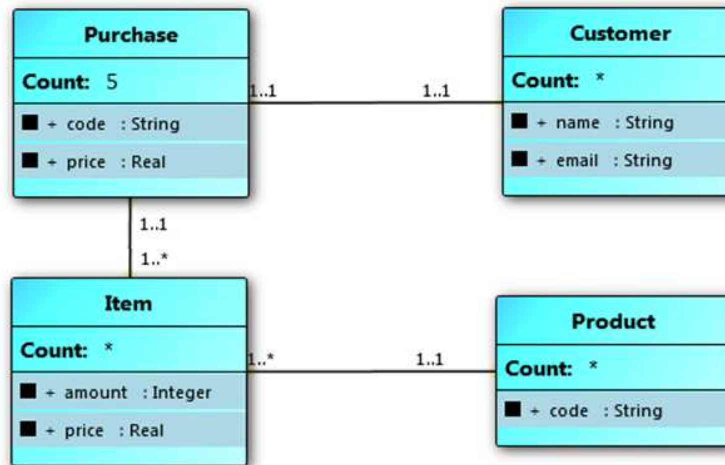

Figure 6.9: An example of a PIM-View schema for metrics evaluation example

In the PSM schema a) only the class Purchase *can inflict redundancy. All other child classes participate in the relation with the upper value of the cardinality equal to 1. The class redundancy metric of its parent* Item *is* $*$*, because the upper value of the* Product*'s cardinality is* $*$*. The size of the subtree is 2. Therefore, the value of* $\Omega(schema\ a)) = 2$*. Context classes are in depths 2 and 1. Therefore,*

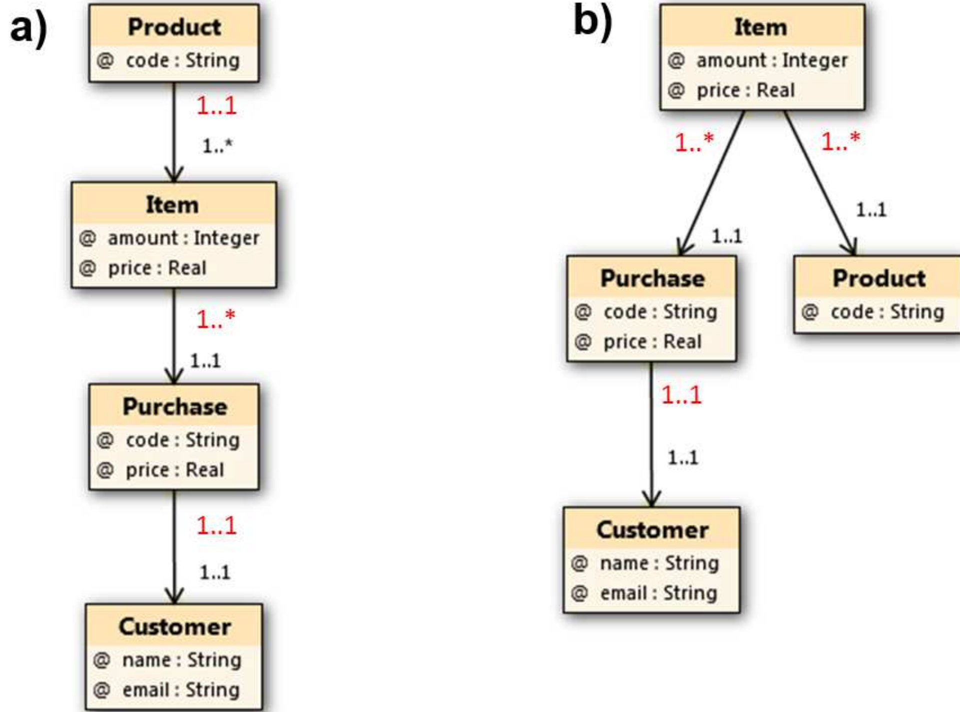

Figure 6.10: Examples of a PSM schemas for metrics evaluation example

$\Phi(schema\ a)) = 3$. The end of the path, class Item, is in a lower depth than the beginning, hence $\Psi(schema\ a)) = 1$.

In the PSM schema b), only classes Purchase *and* Product *can inflict redundancy for the same reason. The class redundancy metric of their parent* Item *is $*$. Sizes of the subtrees are 2 and 1. Therefore, $\Omega(schema\ b)) = 3$. Context classes are in depths 0 and 1. Therefore, $\Phi(schema\ b)) = 1$. The end of the path, class Item, is in a lower depth than the beginning. $\Psi(schema\ b)) = 0$.*

Having weights $\alpha = 0.33$, $\beta = 0.34$, $\gamma = 0.33$, we get $\Delta(schema\ a)) = 2$ and $\Delta(schema\ a)) = 1.35$. Therefore, schema b) is optimal for the given models.

Since there is no existing real-world project that provides a similar functionality, it is not possible to compare our solution and results with others. Therefore, we created a complex example to test abilities of the solution. Thesis [74] presents this example having 10 activities, 5 conceptual models of the exchanged data and 9 sets of business rules. We used different types of cardinalities and relations. Our derivation process created 7 XML schemas for the most complex conceptual model of the exchanged data. The first derived XML schema has 10 keys. The optimal XML schema does not have any key and its depth is 5.

In this complex example we prove useability of the presented approach on bigger conceptual and business process model. We also tested our metrics and algorithms on different models and business rules.

6.4.2 Evolution Part

In this section, we analyze the influence of user-specified changes on the derived optimal communication XML schema. Since we use the data artefact to connect the conceptual model of the exchanged data with the business process model, we

can focus on changes with an influence on the data artefact and on parts used to derive the XML schema.

Conceptual Model of the Exchanged Data

Any change made in the conceptual model of the exchanged data has a direct influence on the derived XML schema. For example, a change of the name of a class has to be propagated into the derived XML schema. This example is quite simple, but these changes can be more complex, e.g., adding a new relation or a new class. This change can lead to changes of the hierarchical structure of the derived XML schema. During derivation, we create connections between the derived XML schema and the PIM-View diagram. It serves for propagation of changes from PIM-View to PSM. This problem is already discussed in papers [90] and [93]. Therefore, for these changes, we can use the described approach directly.

Business Rules

Any change made in business rules used in the derivation process has also a direct influence on the derived XML schema. For example, adding a new business rule can rapidly change metrics used in the derivation process and some other hierarchical structure can be more optimal than the one which is already used. During the derivation, we use business rules only in the metric part. Hence, we cannot make a partial propagation of changes made in business rules.

Business Process Model

The data artefact is connected to a sequence flow in a business process model and business rules are connected to elements of this business process model. Any change in the business process model can have an influence on the derived XML schema:

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

Changes of Business Rules

According to the previous discussion about changes of business rules, we propose an approach for updating the derived XML schema. We apply the derivation process described in the previous section on a data artefact and we compute metrics for the XML schema connected to the data artefact. Then, we display statistic information about a new XML schema and the connected XML schema to the user. After that, the user has to choose one of these XML schemas. The chosen XML schema is reconnected to the data artefact.

6.5 Implementation and Experiments

The prototype of the proposed approach is implemented as an extension of *DaemonX* framework. This prototype uses existing PIM, XSEM PSM and PIM process modeling plug-ins and an extension which was created in [101]. Note that Figures 6.3, 6.6, 6.9, 6.10, 6.12, 6.11, 6.13 and 6.14 are screenshots of the application. The implementation adds 4 new plug-ins:

- The first plug-in is a modeling plug-in for the PIM-View model.
- The second plug-in enables creation of mapping between the PIM model and the PIM-View model.
- The third plug-in enables creation of mapping between the PIM-View model and the XSEM PSM model.
- The fourth plug-in is used to express OCL expressions over the PIM-View model.

Since there is no existing real-world project that provides similar abilities, it is not possible to compare our solution and results with others. Therefore, we created own complex example to test abilities and advantages of the solution. In particular, our example contains 10 activities, 5 conceptual models of exchanged data provided with 9 sets of business rules. The business process model is depicted in Figure 6.11. It models booking of rooms in a hotel and paying by a credit card. The conceptual model of the whole problem domain is depicted in Figure 6.12. It is not the whole model of the problem domain. It contains only a part necessary for this example. As we can see, we use different types of cardinalities and relations.

Our derivation process created 7 different XML schemas for the most complex conceptual model of the exchanged data. The first derived XML schema had 10 keys. The optimal XML schema had no key and its depth was 5. Next we tested changes of business rules. In these tests we focused on context metric. To depict the result, we will show for example the PIM-View schema named *PIMView CustomersInfo*. It is connected to the sequence flow, which starts in the task *Get Customers details*. This PIM-View schema is depicted in Figure 6.13.

The PSM schema derived from PIMView CustomersInfo is depicted in Figure 6.14. It is a single tree, because we can use class *Book* as the root class. Note that in this case, there is no relation which violates conditions described in Section 6.4.1. Therefore, it is possible to create a PSM schema with a single tree.

Using this complex example we proved useability of the presented approach on a bigger conceptual and business process model. We also tested our metrics and algorithms on different models and business rules and showed that the resulting XML schemas have a realistic structure. Note that all schemas related to the example can be found in [74]. The implementation can be found at this resource[3].

6.6 Conclusion

The aim of the approach was to present a novel approach to generation of service interfaces in business process models and to analysis of the influence of user's

[3] `http://www.ksi.mff.cuni.cz/~polak/daemonx/`

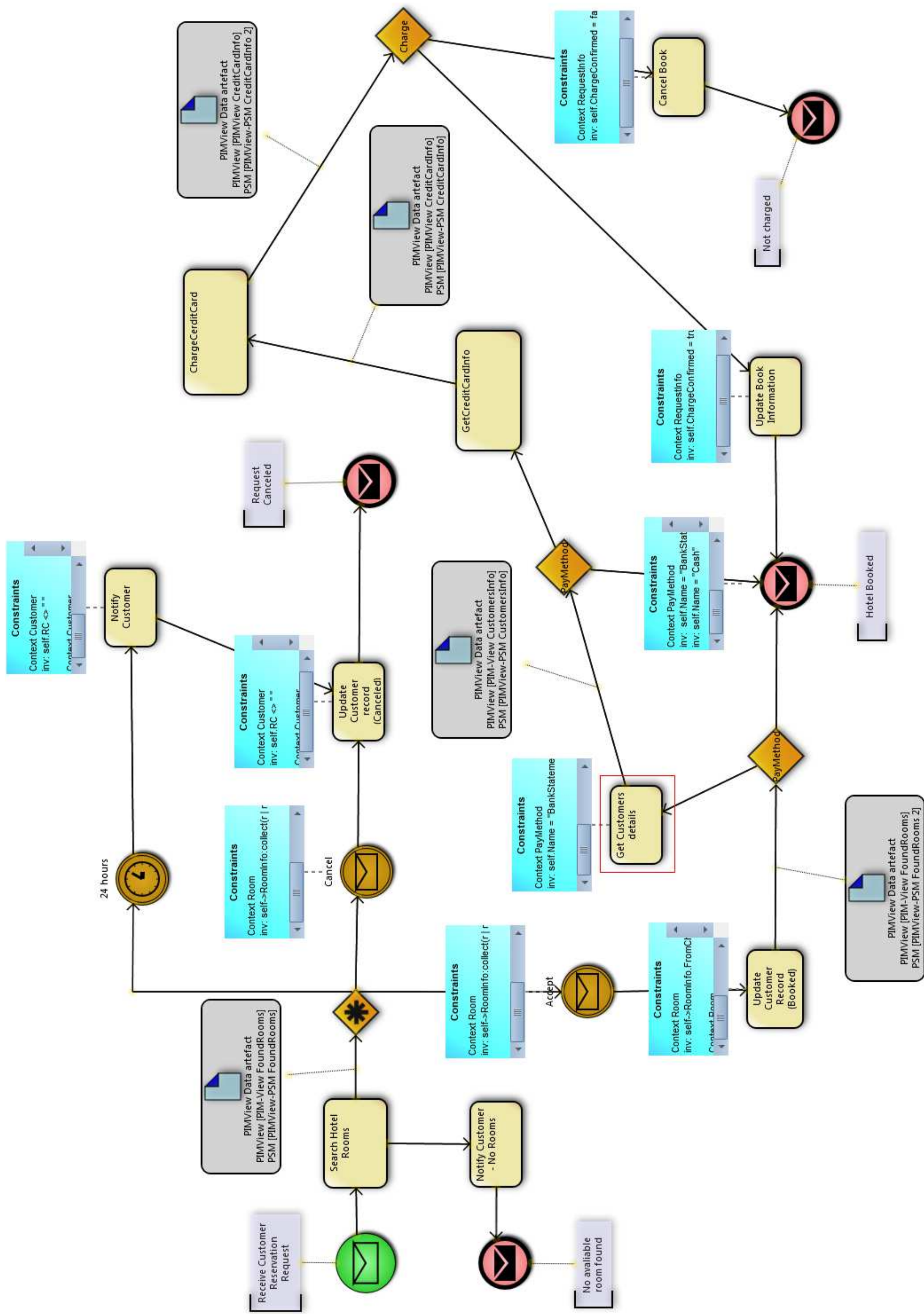

Figure 6.11: Business process model of the experiment

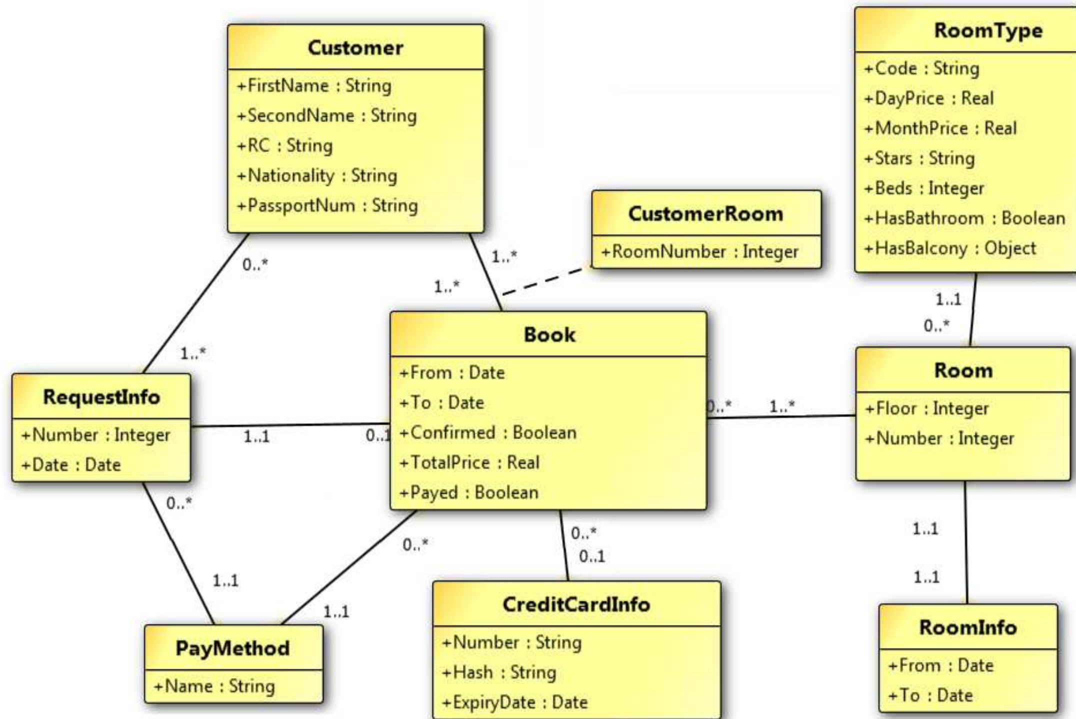

Figure 6.12: PIM schema of the experiment

changes on the derived XML schema. The main contribution of our approach is the ability to create a conceptual model of the exchanged data as a partial view of the whole problem domain and automatic derivation of an optimal communication XML schema based on the defined metrics. It also involves a (semi)automatic algorithm for updating already derived communication XML schema according to user-specified changes of business rules which reduces possible errors during manual schema update. Finally, the presented solution was implemented and tested on real-world examples to provide its correctness and usability.

6.6.1 Future Work

Even though the proposed approach can be applied successfully in real-world situations, there are still several open problems and issues:

- *Changes in the Business Process Model*: In our work, we focus on derivation of an optimal communication XML schema and on changes of inputs of the derivation process. The data artefact is connected to a sequence flow in a given business process model and business rules are connected to elements of this business process model. However, any change in the business process model can have an influence on the derived XML schema:
  - Reconnection of the sequence flow can change business rules, which are used in the derivation process,
  - adding of a new gateway or event can change business rules, which are used in the derivation process,
  - removing of a task can lead to a change in a conceptual model of the exchanged data, which are used in the derivation process.

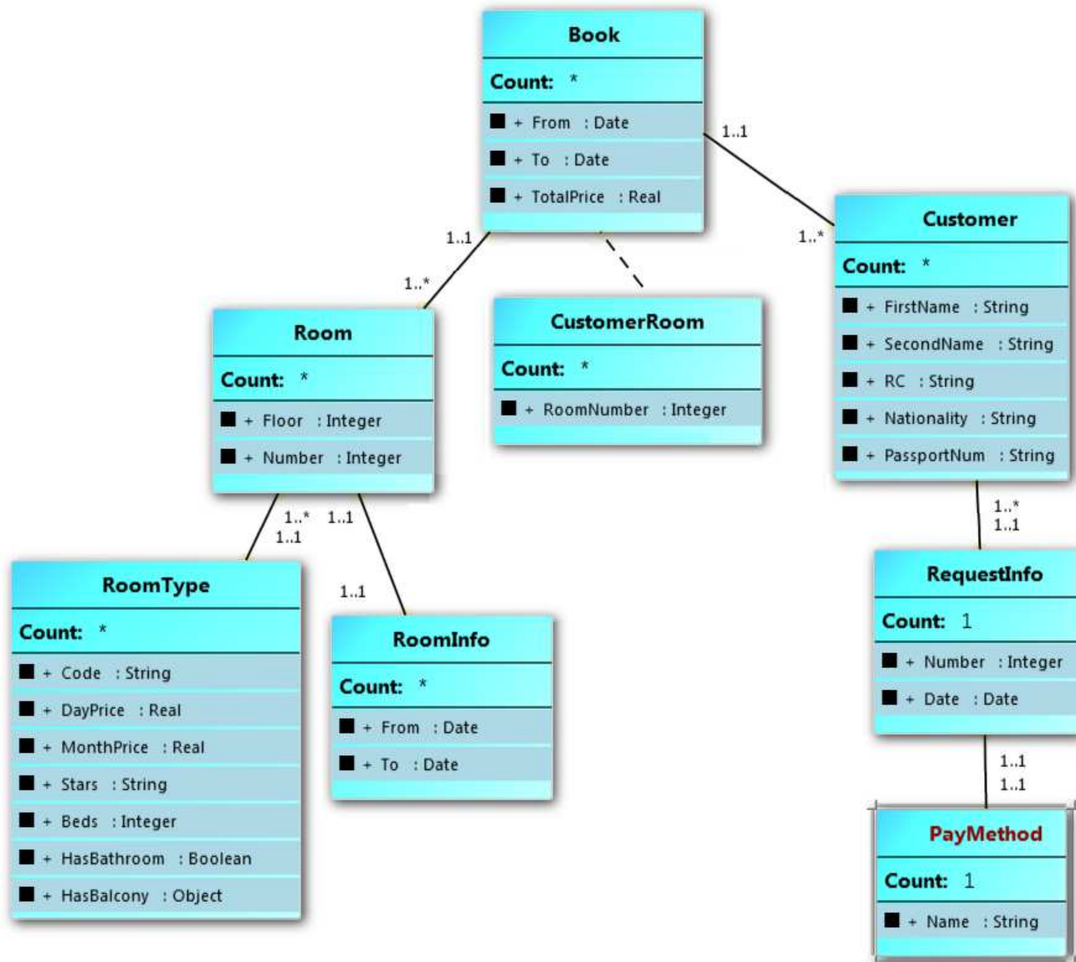

Figure 6.13: PIM-View schema PIMView CustomersInfo

- *Richer Derivation Process*: The main limitation which is most binding for a domain expert is the acyclic conceptual model of exchanged data. This can be in some cases a difficult task.
- *Attribute Type Derivation*: Our work does not cover mechanisms for deriving data types of attributes and their constraints. More complex data types must be defined using classes and relations.
- *Order of Children in the Generated XML Schema*: An XML schema has a hierarchical structure and ordering of children is important in this hierarchy. It can carry a particular information. As we derive an XML schema from the conceptual model which does not have hierarchical structure, it can have a position information stored only in its elements, e.g., a class or an attribute. Therefore, positions of children are not important in our proposed algorithm and we do not work with it.
- *Optimization of Derivation Process*: The derivation process presented in this approach has in some cases better time complexity in some cases than the derivation process presented in paper [80]. However, it can still have an exponential time complexity in cases, where there are relations with both lower values of cardinalities equal to $*$. To improve the time complexity, we would need additional information, such as semantic information about the conceptual model of exchanged data.

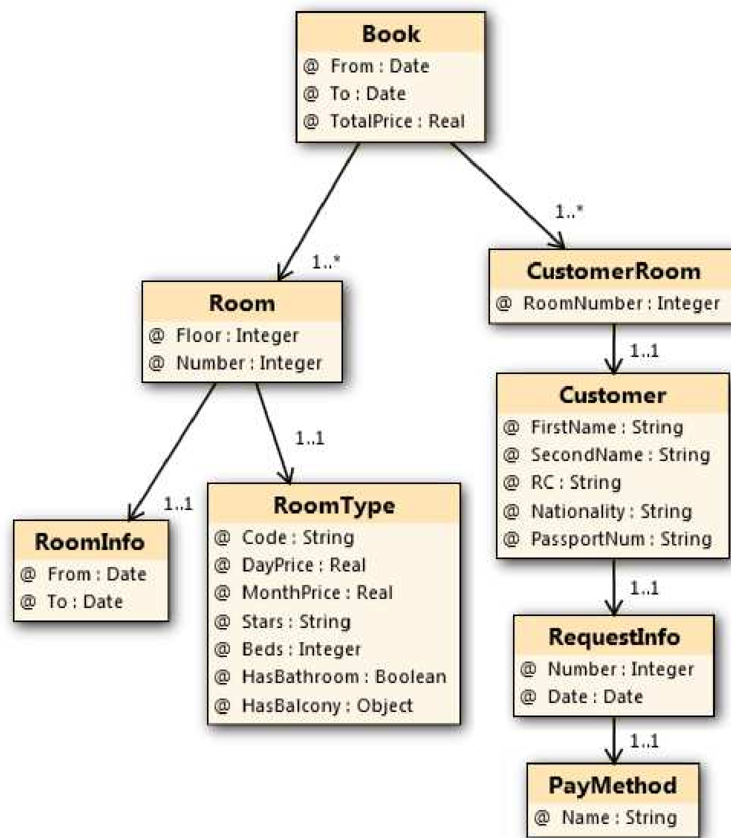

Figure 6.14: PSM schema derived from PIMView CustomersInfo

7. REST API Management and Evolution

The REST has become a popular and preferred way of communication on the Web. In this chapter we focus on managing and generating REST resources based on the MDA principle which enables design and maintenance of complex projects. We introduce a way how to describe the REST API in MDA and how to provide automatic evolution management between subsequent API versions derived from the original version. The proposed solution describes a novel model which represents REST resources and algorithms for providing evolution of this model based on changes done in the PIM of MDA. The model for REST and related change management presented in this chapter in the context of the five-level evolution management framework is depicted in Figure 7.1. The approach was presented in [106, 105].

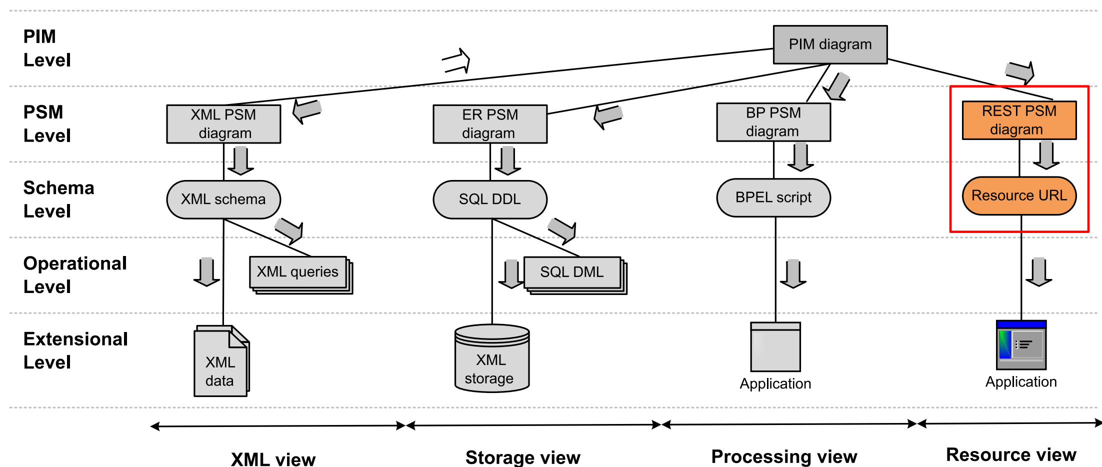

Figure 7.1: Location of the REST model and related change management in the context of the five-level evolution management framework

7.1 Introduction

Nowadays, the REST is becoming the most commonly used way how to create, publish, and consume Web Services. The most popular data interchange format is the JSON. However, although these technologies are used widely over the Internet, there are no official standards for REST (like, e.g., the WSDL for Web Services) and for JSON (like, e.g., the XML Schema for XML documents). There only exist several projects partially covering these topics for specific purposes, such as, e.g., RAML [137], Swagger [120], or HAL [68], which enable generating human/machine description (documentation) of the API, its routing, JSON content structure, etc. But a more important and difficult related issue is versioning of the APIs. The mentioned solutions, surprisingly, do not solve this problem at all. They do not provide any relations between two subsequent versions except for API designer/developer comments. This brings the need to manually check every new version by the API consumer and subsequent updates of the consumer's code to ensure compatibility. With the growing size of an application this task

becomes highly difficult and error-prone.

Hence, our aim is to define a way how to describe the REST API as a PSM, called *Resource Model*, based on the MDA and how to provide (semi)automatic change management between subsequent API versions derived from an original version. The proposed solution describes a novel model which represents REST requests and algorithms for ensuring evolution management of this model based on changes done in the PIM of the MDA. Thanks to MDA it is possible to incorporate our approach with other PSM models to have a complex solution for distinct parts of the application.

7.2 Related Works

In this section we describe and compare related solutions. Currently there exist papers discussing best practises how to define and design API to prevent complex future changes. Next, we discuss papers that present various methods how to generate documentation of API from a source code.

7.2.1 API Versioning Best Practises

Papers dealing with API versioning [22, 97] propose creation of a new API version and suggest a list of the best practises for their maintenance such as:

- **Keep compatible name conventions**: Beside the original resource `http://api.example.com/customer` a new one such as, e.g., `http://api.example.com/v2/customer`, is created.
- **Avoid new major versions**, because multiple major versions need more support and maintenance.
- **Make changes backwards compatible** to reduce needed changes.
- **Provide documentation of the new version changes** to have as much information as possible needed by API consumers.

Discussion

The mentioned papers do not provide full solution how to manage API versioning, but they give instructions how to reduce and prevent possible problems during the development process.

7.2.2 API Documentation Generation

There are various solutions for the REST services management (documentation) like the API Blueprint [10], RAML, or Swagger. All these frameworks involve own proposal how to describe the REST API to be provided to consumers in a comprehensive way. They offer, e.g., own description language, editors for API creation, documentation, stub (code) generation, etc. But none of them provides a way how to manage changes between API versions efficiently, e.g., recording of the changes or generation of the differences between particular versions. Next, there exist specific solutions for various frameworks like ASP.NET WebApi [23] or

Flask [114] which provide generation of REST resource documentation based on the implementation of a project (source code). But this documentation is specific for the particular project version (release) and has no relations to the previous versions. It just describes the resources (methods, parameters, response codes, response structure, etc.).

Discussion

Papers dealing with API documentation generation provide a way how automatically create a description of the API without manual rewriting. They focus on various topics and abilities such as simpler testing or team collaboration. But there is no solution to the problem how to handle changes and relations between particular API versions.

7.2.3 Comparison of the Related Works

The presented related works do not give a full solution to the problem of API change management. Papers [22, 97] propose a list of best practises for API versioning, whereas frameworks [137, 120, 10] provide an API documentation, but without any (semi)automatic generation of changes between particular versions.

In general, change management is not critical for a small project or during the first phases of development. But it becomes a big problem in larger projects and later during the maintenance phases of a software. In this situation, every change must be done precisely, correctly, and completely to prevent errors. A (semi)automatic mechanism which helps to identify the affected parts for the developer or even performs the change automatically is then very important. This chapter focuses on the following problems which were not solved in the mentioned papers:

- The the definition of a resource model.
- The relation between a PIM model and a resource model.
- The analysis of changes done in a source PIM model.
- The propagation of changes to preserve compatibility of a PIM and resource model.

7.3 Resource Model

To be able to solve this problem and to be able to handle the propagation there must be defined a model representing platform-specific view of the resource addresses, called the *Resource Model*. Then, we can model and visualize the resource structure. Over the model there can also be defined algorithms for, e.g., generation of the base code structure for particular platforms such as, e.g., AJAX [134] calls or resource API documentation.

Definition 21. *(Resource Model). A* resource model *G of a resource R is a directed graph $G_R = (V, E)$, where V is a set of resource vertices and E is a set of resource edges. For the resource model it must hold that it is a tree.*

Let D be a set of parameters, whereas each $d \in D$ has a name and a data type. Let D' be a set of results, each having a data type. A resource function *$f : 2^{(D)} \rightarrow D'$ provides a particular result $d' \in D'$ for a particular subset of parameters from D. Every resource vertex $v \in V$ of a resource model $G_R = (V, E)$ has a set of resource functions F_v.*

From the MDA perspective, *Resource Model Vertex* is an extended UML Class and *Resource Model Edge* is an extended UML Association. There are defined no profiles over the model.

An example (and our resulting application screenshot) of the Resource Model representing a simple e-shop diagram is depicted in Figure 7.2. The green rectangles represent vertices, the black arrows represent edges of the Resource Model. The red arrows represent a mapping to the sample REST resource. This model represents the following REST resource: Every word represents a vertex in the Resource Model or a vertex function (displayed as green rectangle). A word in the curly brackets represents a parameter. This parameter is applied as a function parameter or as a selector of an item from the collection. Particular usage and implementation depends on the author. A slash denotes an edge having the direction from the left to the right.

Note that the same resources (e.g., `shop/user`) can be combined with different HTTP methods (e.g., *GET*, *POST*, *PUT* or *DELETE*) for different purposes. A list for the example from Figure 7.2 is as follows:

From this model we can generate REST resources, such as, e.g.:

- `shop` combined with *GET*, *PUT*, *DELETE* methods.
- `shop/user` combined with *POST* method.
- `shop/user/{:uid}` combined with *GET*, *PUT*, *DELETE* methods.
- `shop/user/{:uid}/address` combined with *POST* method.
- `shop/user/{:uid}/address/{:aid}` combined with *GET*, *PUT*, *DELETE* methods.
- `shop/user/{:uid}/order` combined with *POST* method.
- `shop/user/{:uid}/order/{:oid}` combined with *GET*, *PUT*, *DELETE* methods.
- `shop/user/{:uid}/order/{:oid}/item/` combined with *POST* method.
- `shop/user/{:uid}/order/{:oid}/item/{:iid}` combined with *GET*, *PUT*, *DELETE* method.
- `shop/user/{:uid}/order/{:oid}/item/{:iid}`
  `/setcount/{count}` combined with *POST* method.

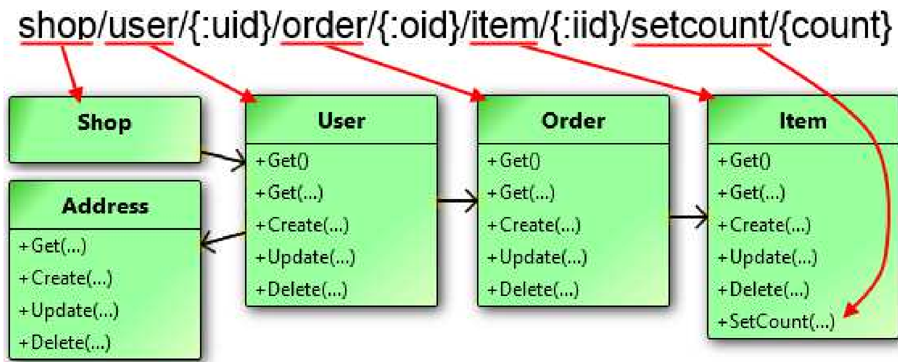

Figure 7.2: An example of the Resource Model of a simple e-shop

The full list of the REST resources generated from this simple example has 20 items (for every vertex there is a resource plus operations over it) and thus any future update of the model (e.g., renaming of class *User* to *Customer*) can trigger many necessary changes which is a difficult and error-prone task. For this purpose, an ability to (semi)automatically propagate changes from the PIM to the corresponding Resource Model is essential. In the following section we will show how this problem can be solved fully, correctly, and efficiently.

7.4 Mapping and Evolution

First, we will describe the relatively straightforward mapping of the models. Next, we will focus on the operations over the models and their propagation which is a natural consequence of the mapping.

7.4.1 Model Mapping

Thanks to the definition of the Resource Model, the mapping between the PIM and the Resource Model is relatively straightforward:

- **PIM Class → Resource Model Vertex**: A PIM Class is mapped directly to a Resource Model Vertex.
- **PIM Class Function → Resource Model Vertex Function**: A PIM Class Function is mapped to a Resource Model Vertex Function. Resource Model Vertex Function represents an operation over the PIM Class instance.
- **PIM Association → Resource Model Edge**: A PIM Association is mapped to a Resource Model Edge. As in PIM Association in PIM, Resource Model Edge represents a relation between Resource Model Vertices.
- **PIM Attribute → Resource Model Vertex Attribute**: A PIM Attribute is mapped to a Resource Model Vertex Attribute.

An example of a mapping between a PIM and a Resource Model is depicted in Figure 7.3. All classes of the PIM are mapped to the corresponding Resource Model vertices (the mapping is represented with black dashed lines). The only PIM class function is mapped to the corresponding Resource Model vertex function (marked with a green dashed line). PIM associations are mapped to edges in Resource Model (marked with blue dashed lines).

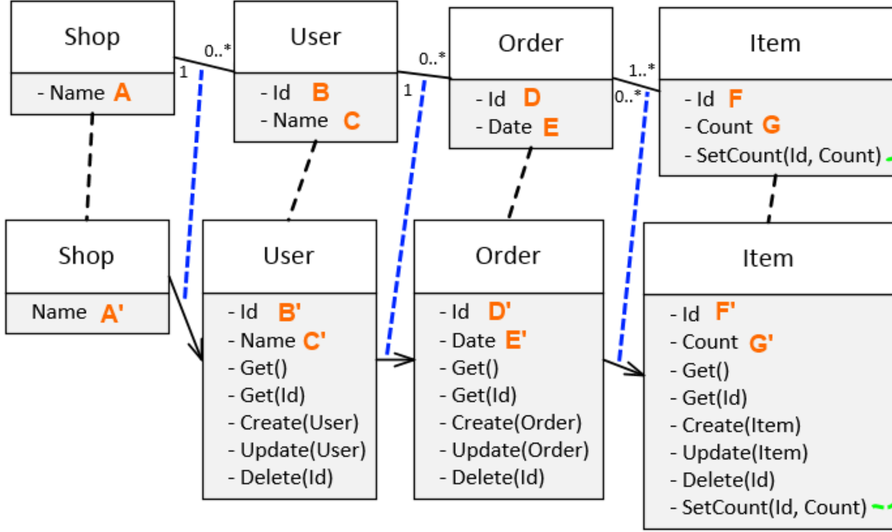

Figure 7.3: An example of the PIM-Resource Model mapping

For both the models we first need to define atomic operations. Atomic or composite operations in the PIM (source) which have an impact on the related Resource Model (target) are then translated to the corresponding atomic or composite operations in the Resource Model such as, e.g., *PIM Class renaming* $\rightarrow$ *Resource Model Vertex renaming*. This translation is called *change propagation*.

7.5 Atomic PIM Model Operations

Let $M_S = (C_S, I_S)$ be a PIM. C_S is a set of classes, I_S is a set of connections, where $I_k(C_l, C_m)$, $k \in [1, n]$ is a connection between classes C_l and C_m and has name I_{k_N}. Each class $C_i \in C_S$, $i \in [1, v]$ has a name C_{i_N}, a set of attributes C_{i_P} and a set of functions C_{i_F}. Every attribute P_j, $j \in [0, s]$ has a name P_{j_N} and a type P_{j_T}. Every function F_o, $o \in [0, t]$ has a name F_{o_N}, a return type F_{o_T}, and a set of parameters F_{o_R}. Each function parameter R_q, $q \in [0, u]$ has a name R_{q_N} and a type R_{q_T}.

PIM atomic operations of attributes are defined as follows:

- **Class Creating** ($\gamma_C : (M_S, C_i) \rightarrow M'_S$): The operation creates class C_i in model $M_S = (C_S, I_S)$. It returns model $M'_S = (C'_S, I_S)$, where $C'_S = C_S \cup \{C_i\}$.

  Precondition: Class name C_{i_N} must be set (it can be a default value). (Note that there is no restriction for unique class names – the operation depends on the instances, not names.)

- **Class Renaming** ($\alpha_C : (C_i, m) \rightarrow C'_i$): The operation returns class C'_i, where $C'_{i_N} = m$, $C'_{i_P} = C_{i_P}$, and $C'_{i_F} = C_{i_F}$.

  Precondition: Class C_i must exist in model $M_S = (C_S, I_S)$.

- **Class Removing** ($\delta_C : (M_S, C_i) \rightarrow M'_S$): The operation removes class $C_i \in C_S$ from model $M_S = (C_S, I_S)$. It returns model $M'_S = (C'_S, I_S)$, where $C'_S = C_S \backslash \{C_i\}$.

Precondition: Class C_i must exist in model $M_S = (C_S, I_S)$ and there must not exist connections to C_i.

- **Property Creating** ($\gamma_P : (C_i, P_j) \to C'_i$): The operation adds property P_j to class C_i. It returns class C'_i, where $C'_{i_N} = C_{i_N}$, $C'_{i_F} = C_{i_F}$, and $C'_{i_P} = C_{i_P} \cup \{P_j\}$.

  Precondition: Property P_j must have name P_{j_N} and type P_{j_T}. Class C_i must exist in model $M_S = (C_S, I_S)$. The name of the property P_{j_N} must not be present among the names of properties in C_{i_P}.

- **Property Renaming** ($\alpha_P : (P_j, m) \to P'_j$): The operation returns property P'_j, where $P'_{j_N} = m$, $P'_{j_T} = P_{j_T}$.

  Precondition: Property P_j must exist in some set C_{i_P} of class C_i in model $M_S = (C_S, I_S)$. Name m must not be present among the names of properties in C_{i_P}.

- **Property Type Changing** ($\beta_P : (P_j, t) \to P'_j$): The operation returns property P'_j, where $P'_{j_T} = t$ and $P'_{j_N} = P_{j_N}$.

  Precondition: Property P_j must exist in some set C_{i_P} of class C_i in model $M_S = (C_S, I_S)$.

- **Property Removing** ($\delta_P : (C_i, P_j) \to C'_i$): The operation removes property $P_j \in C_{i_P}$ from class C_i. It returns class C'_i, where $C'_{i_N} = C_{i_N}$, $C'_{i_F} = C_{i_F}$ and $C'_{i_P} = C_{i_P} \backslash \{P_j\}$.

  Precondition: Class C_i with property P_j must exist in model $M_S = (C_S, I_S)$.

- **Connection Creating** ($\gamma_I : (M_S, I_k) \to M'_S$): The operation creates connection $I_k(C_i, C_j)$ between classes $C_i, C_j \in C_S$ in model $M_S = (C_S, I_S)$. It returns model $M'_S = (C_S, I'_S)$, where $I'_S = I_S \cup \{I_k\}$.

  Precondition: Classes C_i and C_j must exist in model $M_S = (C_S, I_S)$. (Note that connection name I_{k_N} does not have to be set – a default value can be applied – and the name does not have to be unique.)

- **Connection Renaming** ($\alpha_I : (I_k, m) \to I'_k$): The operation returns connection I'_k, where $I'_{k_N} = m$.

  Precondition: Connection I_k must exist in model $M_S = (C_S, I_S)$.

- **Connection Removing** ($\delta_I : (M_S, I_k) \to M'_S$): The operation removes connection I_k from model $M_S = (C_S, I_S)$. It returns model $M'_S = (C_S, I'_S)$, where $I'_S = I_S \backslash \{I_k\}$.

  Precondition: Connection I_k must exist in model $M_S = (C_S, I_S)$.

- **Function Creating** ($\gamma_F : (C_i, F_j) \to C'_i$): The operation adds function F_j to class C_i. It returns class C'_i, where $C'_{i_N} = C_{i_N}$, $C'_{i_P} = C_{i_P}$, and $C'_{i_F} = C_{i_F} \cup \{F_j\}$.

  Precondition: Class C_i must exist in model $M_S = (C_S, I_S)$. Function name F_{j_N} and return type F_{j_T} must be set.

Postcondition: There must not exist a function $F_k \in C'_{i_F}$, $F_k \neq F_j$, s.t. $F_{k_N} = F_{j_N}$ (i.e., having an identical name) and $F_{k_R} = F_{j_R}$ (i.e., having an identical set of parameters). Otherwise we say that there exists function F_k with the same *signature* as F_j.

- **Function Renaming** ($\alpha_F : (F_i, m) \to F'_i$): The operation returns function F'_i, where $F'_{i_N} = m$, $F'_{i_R} = F_{i_R}$, and $F'_{i_T} = F_{i_T}$.

  Precondition: Function F_i must exist in some set C_{j_F} of class C_j in model $M_S = (C_S, I_S)$.

  Postcondition: There must not exist a function $F_k \in C_{j_F}$, $F_k \neq F'_i$ having the same signature as F'_i.

- **Function Removing** ($\delta_F : (C_i, F_j) \to C'_i$): The operation removes function F_j from class C_i. It returns class C'_i, where $C'_{i_N} = C_{i_N}$, $C'_{i_P} = C_{i_P}$ and $C'_{i_F} = C_{i_F} \setminus \{F_j\}$.

  Precondition: Class C_i with function F_j must exist in model $M_S = (C_S, I_S)$.

- **Function Return Type Changing** ($\beta_F : (F_i, t) \to F'_i$): The operation returns function F'_i, where $F'_{i_T} = t$, $F'_{i_N} = F_{i_N}$, and $F'_{i_R} = F_{i_R}$.

  Precondition: Function F_i must exist in some set C_{j_F} of class C_j in model $M_S = (C_S, I_S)$.

- **Function Parameter Creating** ($\gamma_R : (F_i, R_j) \to F'_i$): The operation adds parameter R_j to function F_i. It returns function F'_i, where $F'_{i_N} = F_{i_N}$, $F'_{i_T} = F_{i_T}$ and $F'_{i_R} = F_{i_R} \cup \{R_j\}$.

  Precondition: Function F_i must exist in some set C_{k_F} of class C_k in model $M_S = (C_S, I_S)$. Parameter R_j must have name R_{j_N} and type R_{j_T}. Name R_{j_N} must not be present in the set of names of parameters in F_{i_R}.

  Postcondition: There must not exist a function $F_l \in C_{k_F}$, $F_l \neq F'_i$ having the same signature as F'_i.

- **Function Parameter Renaming** ($\alpha_R : (R_j, m) \to R'_j$): The operation returns parameter R'_j, where $R'_{j_N} = m$ and $R'_{j_T} = R_{j_T}$.

  Precondition: Parameter R_j must exist in some set F_{i_R} of function F_i from set C_{k_F} of class C_k in model $M_S = (C_S, I_S)$. Name m must not be present in the set of names of $F_{i_R} \setminus \{R_j\}$.

  Postcondition: Let F'_i be the modified function. There must not exist a function $F_l \in C_{k_F}$, $F_l \neq F'_i$ having the same signature as F'_i.

- **Function Parameter Type Changing** ($\beta_R : (R_j, t) \to R'_j$): The operation returns parameter R'_j, where $R'_{j_T} = t$ and $R'_{j_N} = R_{j_N}$.

  Precondition: Parameter R_j must exist in some set F_{i_R} of function F_i from set C_{k_F} of class C_k in model $M_S = (C_S, I_S)$.

  Postcondition: Let F'_i be the modified function. There must not exist a function $F_l \in C_{k_F}$, $F_l \neq F'_i$ having the same signature as F'_i.

- **Function Parameter Removing** ($\delta_R : (F_i, R_j) \to F'_i$): The operation removes parameter $R_j \in F_{i_R}$ from function F_i. It returns function F'_i, where $F'_{i_N} = F_{i_N}$, $F'_{i_T} = F_{i_T}$ and $F'_{i_R} = F_{i_R} \backslash \{R_j\}$.

  Precondition: Function F_i with parameter R_j must exist in some set C_{k_F} of class C_k in model $M_S = (C_S, I_S)$.

  Postcondition: There must not exist a function $F_l \in C_{k_F}$, $F_l \neq F'_i$ having the same signature as F'_i.

- **Attribute Creating** ($\gamma_P : (C_i, P_j) \to C'_i$): The operation adds attribute P_j to class C_i. It returns class C'_i, where $C'_{i_N} = C_{i_N}$, $C'_{i_F} = C_{i_F}$, and $C'_{i_P} = C_{i_P} \cup \{P_j\}$, e.g., only the set of attributes is changed. ***Precondition:*** Attribute P_j must have name P_{j_N} and type P_{j_T}. Class C_i must exist. There must not exists attribute P_l, $P_l \in C_i$, $P_l \neq P_j$ where $P_{l_N} = P_{j_N}$.

- **Attribute Renaming** ($\alpha_P : (P_j, m) \to P'_j$): The operation returns attribute P'_j, where $P'_{j_N} = m$, $P'_{j_T} = P_{j_T}$. ***Precondition:*** Attribute P_j must exist. There must not exists attribute P_l, $P_l \in C_i$, $P_j \in C_i$, $P_l \neq P_j$ where $P_{l_N} = m$.

- **Attribute Type Changing** ($\beta_P : (P_j, t) \to P'_j$): The operation returns attribute P'_j, where $P'_{j_T} = t$, $P'_{j_N} = P_{j_N}$. ***Precondition:*** Attribute P_j must exist.

- **Attribute Removing** ($\delta_P : (C_i, P_j) \to C'_i$): The operation removes attribute $P_j \in C_{i_P}$ from class C_i. It returns class C'_i, where $C'_{i_N} = C_{i_N}$, $C'_{i_P} = C_{i_P}$, $C'_{i_F} = C_{i_F}$, and $C'_{i_P} = C_{i_P} \backslash \{P_j\}$. ***Precondition:*** Class C_i and attribute P_j must exist.

- **Attribute Moving** ($\epsilon_P : (C_i, C_j, P_k) \to (C'_i, C'_j)$): The operation moves attribute $P_k \in C_{i_P}$ from class C_i to class C_j. It returns class C'_i, where $C'_{i_N} = C_{i_N}$, $C'_{i_F} = C_{i_F}$, and $C'_{i_P} = C_{i_P} \backslash \{P_k\}$ and class C'_j, where $C'_{j_N} = C_{j_N}$, $C'_{j_F} = C_{j_F}$, and $C'_{j_P} = C_{j_P} \cup \{P_k\}$. ***Precondition:*** Classes C_i, C_j and attribute P_k must exist. There must not exist attribute P_l, $P_l \in C_j$, $P_l \neq P_k$ where $P_{l_N} = P_{k_N}$.

7.6 Atomic Resource Model Operations

Let $M_R = (V_R, E_R)$ be a Resource Model. Each vertex $V_i \in V_R, i \in [1, n]$ has a name V_{i_N} and a set of functions V_{i_F}. Every function F_j, $j \in [0, o]$ has a name F_{j_N}, a return type F_{j_T}, and a set of parameters F_{j_R}. Each function parameter R_k, $k \in [0, m]$ has a name R_{k_N} and a type R_{k_T}. E_R is a set of edges, where $E_l(V_p, V_q)$, $l \in [1, s]$, $E_l \in E_R$ is an ordered edge between vertices V_p and V_q, s.t. $V_p, V_q \in V_R$.

The Resource Model atomic operations are defined as follows:

- **Vertex Creating** ($\gamma_V : (M_R, V_i) \to M'_R$): The operation creates vertex V_i in model $M_R = (V_R, E_R)$. It returns model $M'_R = (V'_R, E_R)$, where $V'_R = V_R \cup \{V_i\}$. In addition, if $V_R \neq \emptyset$, then V_i is subsequently connected to a $V_j \in V_R$ via edge $E_k(V_j, V_i)$. So, then the operation returns model $M'_R = (V'_R, E'_R)$, where $V'_R = V_R \cup \{V_i\}$ and $E'_R = E_R \cup \{E_k\}$.

  Precondition: Vertex name V_{i_N} must be set (eventually to a default value).

- **Vertex Renaming** ($\alpha_V : (V_i, m) \to V'_i$): The operation returns vertex V'_i, where $V'_{i_N} = m$, $V'_{i_F} = V_{i_F}$.

  Precondition: Vertex V_i must exist in model $M_R = (V_R, E_R)$. (Note that there is no restriction on unique vertex names – the operation depends on the instances, not names.)

- **Vertex Removing** ($\delta_V : (M_R, V_i) \to M'_R$): The operation removes vertex $V_i \in V_R$ from model $M_R = (V_R, E_R)$. It returns model $M'_R = (V'_R, E_R)$, where $V'_R = V_R \backslash \{V_i\}$. If there exists an edge $E_k(V_j, V_i) \in E_R$ for a $V_j \in V_R$, it is first removed. So, then the operation returns model $M'_R = (V'_R, E'_R)$, where $V'_R = V_R \backslash \{V_i\}$ and $E'_R = E_R \backslash \{E_k\}$.

  Precondition: Vertex V_i must exist in model $M_R = (V_R, E_R)$. There must not exist an edge $E_k(V_i, V_j) \in E_R$ for any $V_j \in V_R$, i.e., V_i must be a *leaf* node.

  Next, if there exists a subtree in the model, where V_i is its root, the whole subtree must be removed. This is done by calling operations *edge removing* and *vertex removing* recursively. An example of this situation is depicted in Figure 7.4.

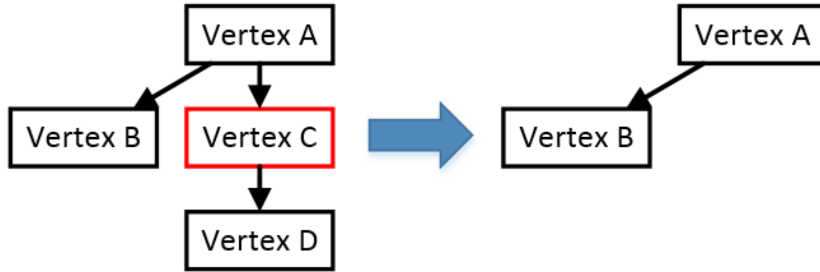

Figure 7.4: An example of removal of vertex C from a Resource Model

- **Edge Creating** ($\gamma_E : (M_R, E_k) \to M'_R$): The operation creates edge $E_k(V_i, V_j)$ between vertices V_i and V_j in model $M_R = (V_R, E_R)$. It returns model $M'_R = (V_R, E'_R)$, where $E'_R = E_R \cup \{E_k\}$.

  Precondition: Vertices V_i and V_j must exist in model $M_R = (V_R, E_R)$.

  Postcondition: M'_R must have a tree structure.

- **Edge Removing** ($\delta_E : (M_R, E_k) \to M'_R$): The operation removes edge E_k from model $M_R = (V_R, E_R)$. It returns model $M'_R = (V_R, E'_R)$, where $E'_R = E_R \backslash \{E_k\}$.

  Precondition: Edge $E_k(V_i, V_j)$ must exist in model $M_R = (V_R, E_R)$. Either V_i or V_j must be a *leaf* node, i.e., there must not exist an edge outgoing from V_i or V_j.

  Postcondition: M'_R must have a tree structure.

- **Function Creating** ($\gamma_F : (V_i, F_j) \to V'_i$): The operation adds function F_j to vertex V_i. It returns vertex V'_i, where $V'_{i_N} = V_{i_N}$ and $V'_{i_F} = V_{i_F} \cup \{F_j\}$.

  Precondition: Vertex V_i must exist in model $M_R = (V_R, E_R)$. Function name F_{j_N} and return type F_{j_T} must be set.

  Postcondition: There must not exist a function F_l in V'_{i_F}, $F_l \neq F_j$ with the same signature as F_j.

- **Function Renaming** ($\alpha_F : (F_i, m) \to F_i'$): The operation returns function F_i', where $F_{i_N}' = m$, $F_{i_R}' = F_{i_R}$, $F_{i_T}' = F_{i_T}$.

  Precondition: Function F_i must exist in some set V_{j_F} of node V_j in model $M_R = (V_R, E_R)$.

  Postcondition: There must not exist a function F_l in $V_{j_F} \backslash \{F_i'\}$ with the same signature as F_i'.

- **Function Removing** ($\delta_F : (V_i, F_j) \to V_i'$): The operation removes function $F_j \in V_{i_F}$ from vertex V_i. It returns vertex V_i', where $V_{i_N}' = V_{i_N}$ and $V_{i_F}' = V_{i_F} \backslash \{F_j\}$.

  Precondition: Vertex V_i with function F_j must exist in model $M_R = (V_R, E_R)$.

- **Function Return Type Changing** ($\beta_F : (F_i, t) \to F_i'$): The operation returns function F_i', where $F_{i_T}' = t$, $F_{i_N}' = F_{i_N}$, and $F_{i_R}' = F_{i_R}$.

  Precondition: Function F_i must exist in model $M_R = (V_R, E_R)$.

- **Function Parameter Creating** ($\gamma_R : (F_i, R_j) \to F_i'$): The operation adds parameter R_j to function F_i. It returns function F_i', where $F_{i_N}' = F_{i_N}$, $F_{i_T}' = F_{i_T}$ and $F_{i_R}' = F_{i_R} \cup \{R_j\}$.

  Precondition: Function F_i must exist in some set V_{k_F} of node V_k in model $M_R = (V_R, E_R)$. Parameter R_j must have name R_{j_N} and type R_{j_T}. Name R_{j_N} must not be present is the set of names of F_{i_R}.

  Postcondition: There must not exist a function F_l in $V_{k_F} \backslash \{F_i'\}$ with the same signature as F_i'.

- **Function Parameter Renaming** ($\alpha_R : (R_j, m) \to R_j'$): The operation returns parameter R_j', where $R_{j_N}' = m$, $R_{j_T}' = R_{j_T}$.

  Precondition: Parameter R_j must exist in some set F_{i_R} of function F_i from set V_{k_F} of node V_k in model $M_R = (V_R, E_R)$. Name m must not be present is the set of names of $F_{i_R} \backslash \{R_j\}$.

  Postcondition: Let F_i' be the modified function. There must not exist a function F_l in $V_{k_F} \backslash \{F_i'\}$ with the same signature as F_i'.

- **Function Parameter Type Changing** ($\beta_R : (R_j, t) \to R_j'$): The operation returns parameter R_j', where $R_{j_T}' = t$, $R_{j_N}' = R_{j_N}$.

  Precondition: Parameter R_j must exist in some set F_{i_R} of function F_i from set V_{k_F} of node V_k in model $M_R = (V_R, E_R)$.

  Postcondition: Let F_i' be the modified function. There must not exist a function F_l in $V_{k_F} \backslash \{F_i'\}$ with the same signature as F_i'.

- **Function Parameter Removing** ($\delta_R : (F_i, R_j) \to F_i'$): The operation removes parameter $R_j \in F_{i_R}$ from function F_i. It returns function F_i', where $F_{i_N}' = F_{i_N}$, $F_{i_T}' = F_{i_T}$, and $F_{i_R}' = F_{i_R} \backslash \{R_j\}$.

  Precondition: Function F_i with parameter R_j must exist in model $M_R = (V_R, E_R)$.

  Postcondition: There must not exist a function F_l in $V_{k_F} \backslash \{F_i\}$ with the same signature as F_i'.

- **Attribute Creating** ($\gamma_P : (V_i, P_l) \rightarrow V_i'$): The operation adds parameter P_l to vertex V_i. It returns vertex V_i', where $V_{i_N}' = V_{i_N}$ and $V_{i_P}' = V_{i_P} \cup \{P_l\}$. ***Precondition:*** Vertex V_i must exist. Attribute name P_{l_N} and type P_{l_T} must be set. There must not exists attribute P_j, $P_j \in V_i$, $P_j \neq P_l$ where $P_{l_N} = P_{j_N}$.

- **Attribute Removing** ($\delta_P : (V_i, P_l) \rightarrow V_i'$): The operation removes attribute $P_l \in V_{i_P}$ from vertex V_i. It returns vertex V_i', where $V_{i_N}' = V_{i_N}$ and $V_{i_P}' = V_{i_P} \backslash \{P_l\}$. ***Precondition:*** Vertex V_i and attribute P_l must exist.

- **Attribute Renaming** ($\alpha_P : (P_l, m) \rightarrow P_l'$): The operation returns attribute P_t', where $P_{l_N}' = m$, $P_{l_T}' = P_{i_T}'$, and $P_{l_B}' = P_{l_B}$. ***Precondition:*** Attribute P_l must exist. ***Postcondition:*** There must not exists attribute P_j, $P_j \in C_i$, $P_l \in V_i$, $P_l \neq P_j$ where $P_{j_N} = m$.

- **Attribute Type Changing** ($\beta_P : (P_l, u) \rightarrow P_t'$): The operation returns attribute P_l', where $P_{l_T}' = u$, $P_{l_N}' = P_{l_N}$, and $P_{l_B}' = P_{l_B}$. ***Precondition:*** Attribute P_t must exist.

- **Attribute Moving** ($\epsilon_P : (V_i, V_j, P_k) \rightarrow (V_i', V_j')$): The operation moves attribute $P_k \in V_{i_P}$ from vertex V_i to vertex V_j. It returns class V_i', where $V_{i_N}' = V_{i_N}$, $v_{i_F}' = V_{i_F}$, and $V_{i_P}' = v_{i_P} \backslash \{P_k\}$ and vertex V_j', where $V_{j_N}' = V_{j_N}$, $V_{j_F}' = V_{j_F}$, and $V_{j_P}' = V_{j_P} \cup \{P_k\}$. ***Precondition:*** Vertices V_i, V_j and attribute P_k must exist. There must not exists attribute P_l, $P_l \in V_j$, $P_l \neq P_k$ where $P_{l_N} = P_{k_N}$.

Note that other functions, like, e.g., deleting a whole subtree, moving a whole subtree, etc. are not atomic, but composite. Also note that we do not have atomic functions for adding/deleting of an edge, because they would require adding/deleting of other parts of the tree so they would not be atomic, but can be defined as composite operations.

7.7 Operation Propagation Policies

Before we focus on the propagation algorithms, let us note that although the propagation operations are defined on the models, sometimes the propagation of the changes is not required. The designer can decide to suppress them. For this purpose we propose so-called *propagation policies*, which can affect the final result of the propagation:

- **Propagate**: This policy allows to perform the change directly. This is the default value.

- **Omit**: This policy does not perform the change. The subsequent propagation is omitted.

- **Prompt**: The system asks the user to propagate or omit the propagation if there occurs any.

The policies can be applied before every propagation from the source to the target model.

7.8 Propagation Algorithms

In this section we describe the algorithms of propagation of changes from the source PIM to the target Resource Model. Algorithms describe propagation of atomic operations from PIM to Resource Model. As we will see, we will call multiple atomic operations in the algorithms, so in fact the algorithms correspond to composite operations.

Getting Related Resource Model Items from PIM Items In the following algorithms we will need a helper function $GetRelationsOf()$ to get related items of a PIM item from the Resource Model having a specified Resource Model type. For example, calling of $GetRelationsOf(C,'VertexType')$ returns all related Resource Model items of PIM class C whose type is *VertexType*.

Algorithm 2 GetRelationsOfType

Require: PIM item P, Resource Model item type T
Ensure: Returns collection of Resource Model items related on P of type T

Class Renaming Algorithm 3 gets as parameters the class to be renamed and a new name of the class. First, it changes the name of the PIM class. Next, the algorithm finds all related resource vertices of class C and changes their names. An example is depicted in Figure 7.5.

Algorithm 3 ClassRenaming

Require: Class C, new name N of the class
Ensure: Renaming of the related vertex
1: $C.Name \leftarrow N$
2: $relatedVertices \leftarrow GetRelationsOf(C,'VertexType')$
3: **for all** $rv \in relatedVertices$ **do**
4: $rv.Name \leftarrow N$
5: **end for**

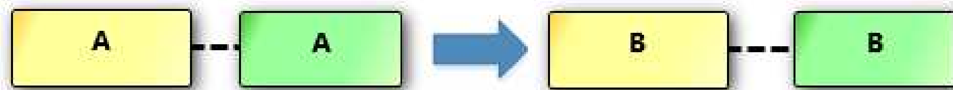

Figure 7.5: An example of renaming class A to B

Class Creating Algorithm for creating class adds a new class to the PIM. It has no impact on the target Resource Model, because the new class is added separately and has no association with the existing model (if any).

Connection Creating Algorithm 4 is the key algorithm. Adding a new connection between two classes C and D in the PIM can absolutely change the structure of the Resource Model. The algorithm has two parameters – classes C and D – between which the connection will be created. First, it checks that there

does not exist a connection between classes C and D. If yes, the algorithm ends. Otherwise, the algorithm gets the related vertices from the Resource Model for both classes C and D. The algorithm iterates over both collections of vertices. If one of the vertices is an ancestor of the second one (line 9), then the ancestor vertex is set to variable rc and a copy of the subtree of rd is created and set as a new child of rc. Else, if rc and rd are not in sibling relation (they have no common ancestor), vertex rd is set as a child of vertex rc. This situation is depicted in Figure 7.6. In this case, the user has to decide which vertex will be set as a parent and a child respectively to match the tree hierarchy of the Resource Model. Finally, if rc and rd are neither in ancestor-descendant relation, nor in sibling relation, the algorithm copies subtrees of both vertices and sets them as the child of the other vertex. This situation is depicted in Figure 7.7.

Helper method $CheckIfConnectionBetweenClassesExists$ $(Class_1, Class_2)$ returns true if there exists a connection (association) between the classes, otherwise it returns false. Method $CreateConnectionBetweenClasses(Class_1, Class_2)$ creates a connection between the two classes. Helper method $IsVertexAncestorOfOther(Vertex_1, Vertex_2)$ returns true if one of the vertices is the ancestor of the other, otherwise it returns false. Method $AreVerticesSiblings(Vertex_1, Vertex_2)$ returns true if the vertices have a common ancestor, otherwise it returns false. Helper method $CreateCopyOfSubtree(Vertex)$ creates a copy of the tree rooted at $Vertex$ and returns the vertex which is the root of the tree copy. And, finally, method $AddChildToVertex(Vertex_1, Vertex_2)$ creates a connection between the vertices, where the first parameter will be the parent of the second one. During copying of the parts in the Resource Model, the relations between items from the PIM and the Resource Model are copied too, where the relation between class C and the newly created vertex exists.

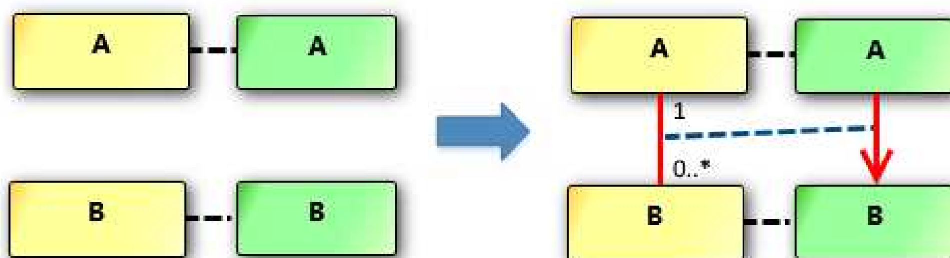

Figure 7.6: An example of connection creating when classes A and B are not siblings

Algorithm 4 CreatingConnectionBetweenClasses

Require: Existing classes C and D
Ensure: Creation of connection CO between C and D

1: **if** $CheckIfConnectionBetweenClassesExists(C, D)$ **then**
2: return
3: **end if**
4: $CreateConnectionBetweenClasses(C, D)$
5: $relatedVertexecOfC \leftarrow GetRelations(C)$
6: $relatedVertexecOfD \leftarrow GetRelations(D)$
7: **for all** $rc \in relatedVertexecOfC$ **do**
8: **for all** $rd \in relatedVertexecOfD$ **do**
9: **if** $IsVertexAncestorOfOther(rc, rd)$ **then**
10: // if true, rc is set as parent
11: $treeRoot_rd \leftarrow CreateCopyOfSubtree(rd)$
12: $AddChildToVertex(rc, treeRoot_rd)$
13: **else if** not $AreVerticesSiblings(rc, rd)$ **then**
14: $AddChildToVertex(rc, rd)$
15: **else**
16: $treeRoot_rc \leftarrow CreateCopyOfSubtree(rc)$ // creates references between models too
17: $AddChildToVertex(rd, treeRoot_rc)$
18: $treeRoot_rd \leftarrow CreateCopyOfSubtree(rd)$
19: $AddChildToVertex(rc, treeRoot_rd)$
20: **end if**
21: **end for**
22: **end for**

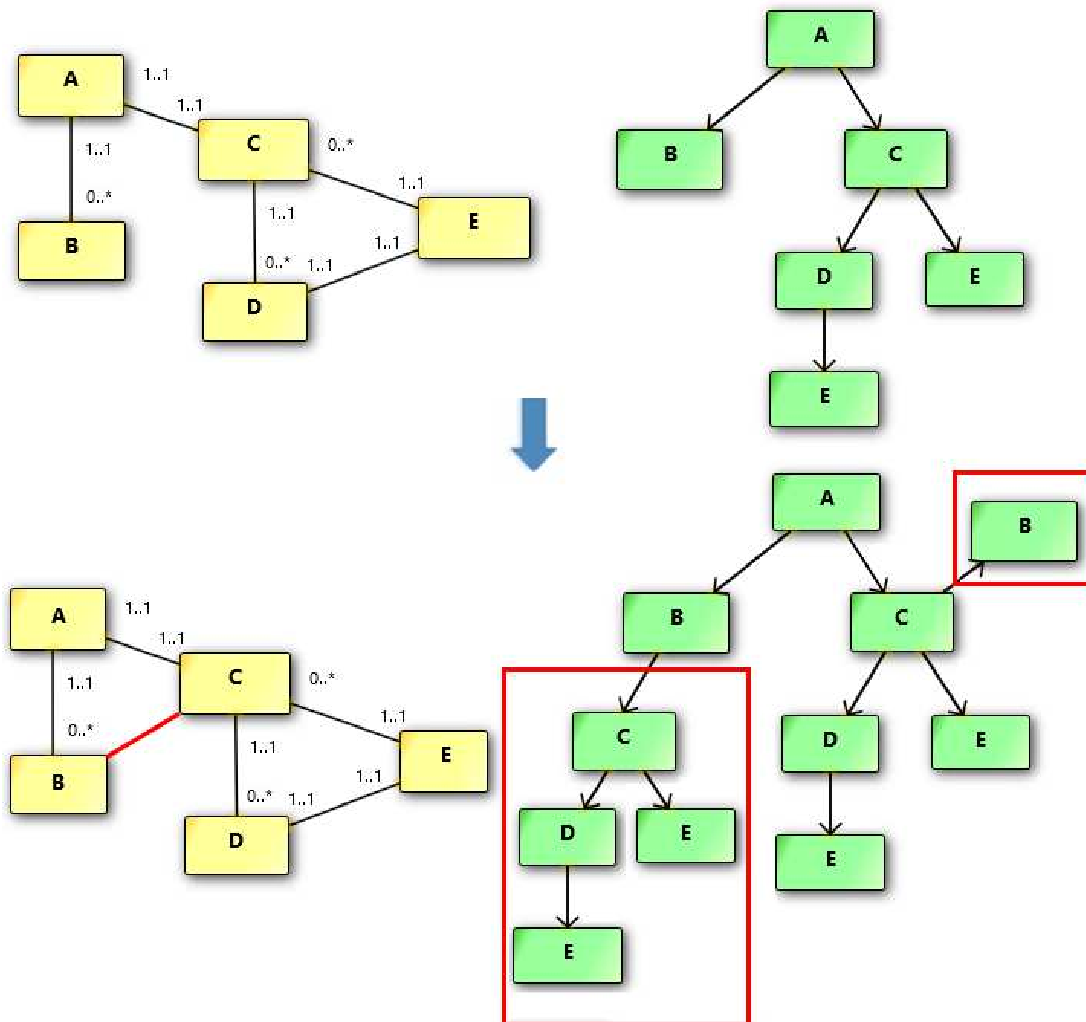

Figure 7.7: An example of connection creating when classes B and C are in a siblings relation. The algorithm first creates copies of both subtrees of vertices B and C and then adds them as children of the opposite ones.

Class Removing Algorithm 5 removes an existing class C from the PIM diagram. Subsequently it removes all corresponding vertices of class C in the Resource Model. It removes corresponding connections between vertices too. An example is shown in Figure 7.8.

Algorithm 5 ClassRemoving

Require: Class C to be removed
Ensure: Removing of class C and related vertices in Resource Model
1: $relatedVertices \leftarrow GetRelationsOf(C,'VertexType')$
2: $RemoveClass(C)$
3: **for all** $rv \in relatedVertices$ **do**
4: $\quad RemoveSubtree(rv)$
5: **end for**

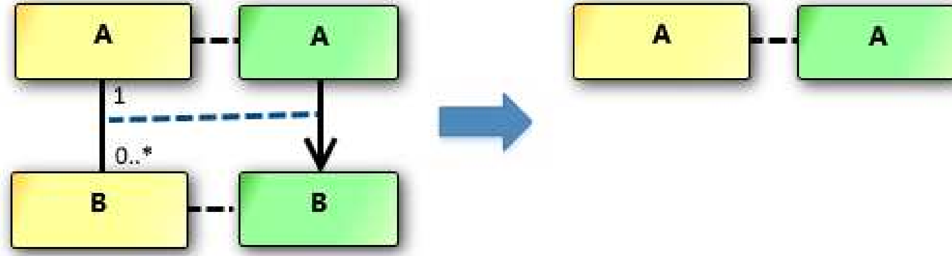

Figure 7.8: An example of removing a class B

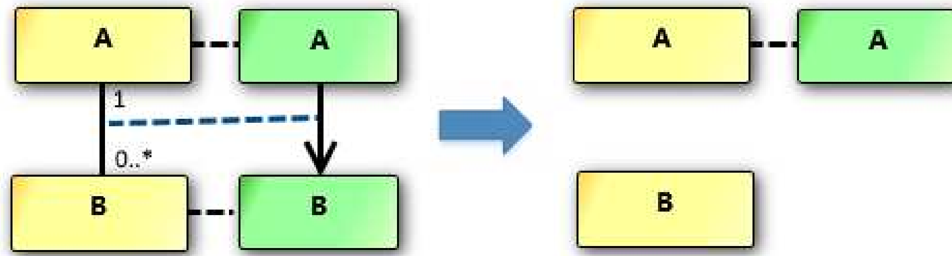

Figure 7.9: An example of removing a connection (association) between classes A and B

Connection Removing Algorithm 6 removes connection C between two classes. Removing of a connection must be propagated to the target Resource Model. An example is depicted in Figure 7.9. There is defined no operation to remove a connection in the Resource Model. So, the whole subtree of the vertices related to class C must be removed. For this purpose we have defined a helper function $RemoveSubtree(Vertex)$ that recursively removes a subtree represented by the vertex and all relations to the corresponding PIM items by calling $RemoveVertex$ function via the depth-first search algorithm.

Algorithm 6 ConnectionRemoving

Require: Connection CO to be removed
Ensure: Removing of connection CO and related connections in Resource Model

1: $relatedConnections \leftarrow GetRelationsOf(CO,'ConnectionType')$
2: $RemoveConnection(CO)$
3: **for all** $rv \in relatedConnections$ **do**
4: $\quad RemoveSubtree(rv.ConnectionEnd)$
5: **end for**

Attribute Renaming, Adding, Renaming These operations are not propagated, because there are no attributes defined in the Resource Model.

Function Creating Algorithm 7 for adding a new function to a PIM class first adds a new function F to class C. Next, it adds newly created functions to all related vertices of class C in the Resource Model. Finally, it creates relations between function F and functions created in the Resource Model by function

$CreateRelation(PIMModelItem, ResourceModelItem)$. An example is depicted in Figure 7.10.

Algorithm 7 FunctionCreating

Require: Function F to be added to class C
Ensure: Creating of the functions in related vertices of the class
1: $CreateFunctionClass(C, F)$
2: $relatedVertices \leftarrow GetRelationsOf(C,' VertexType')$
3: **for all** $rv \in relatedVertices$ **do**
4: $f \leftarrow newFunction()$
5: $CreateFunction(rv, f)$
6: $CreateRelation(F, f)$
7: **end for**

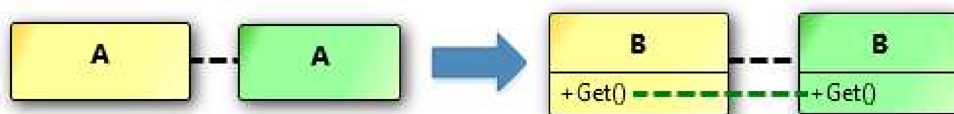

Figure 7.10: An example of creating function *Get*

Function Renaming Function renaming in Algorithm 8 has as parameters function F and the new name N. As in Algorithm 3 it sets the new name of function F and all related functions in the target Resource Model. An example is depicted in Figure 7.11.

Algorithm 8 FunctionRenaming

Require: Function F, new name N of the function
Ensure: Renaming of the function and its related functions in Resource Model
1: $F.Name \leftarrow N$
2: $relatedFunctions \leftarrow GetRelationsOf(F,' FuntionType')$
3: **for all** $rf \in relatedFunctions$ **do**
4: $rf.Name \leftarrow N$
5: **end for**

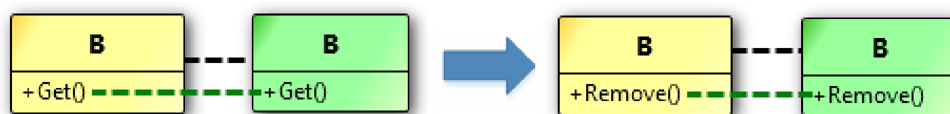

Figure 7.11: An example of renaming function *Get* to *Remove*

Function Return Type Updating As described in Algorithm 9, it first changes the return type T of function F. Next, it changes the type of all related functions in the target Resource Model. An example is depicted in Figure 7.12.

Algorithm 9 FunctionReturnTypeUpdating

Require: Function F, new return type T of the function
Ensure: Update of the function's return type and its related functions in Resource Model

1: $F.ReturnType \leftarrow T$
2: $relatedFunctions \leftarrow GetRelationsOf(F,' FuntionType')$
3: **for all** $rf \in relatedFunctions$ **do**
4: $\quad rf.ReturnType \leftarrow T$
5: **end for**

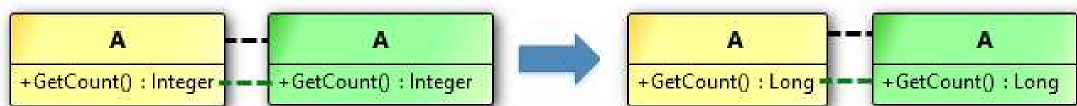

Figure 7.12: An example of updating function return type *int* to *long*

Function Removing Algorithm 10 removes a given function F from its class. Additionally it removes all related functions the in the Resource Model and relations by function $RemoveFunction(ResourceModelFunction)$. An example is depicted in Figure 7.13.

Algorithm 10 FunctionRemoving

Require: Function F to be removed
Ensure: Removing of function F and related functions in Resource Model

1: $relatedFunctions \leftarrow GetRelationsOf(F,' FuntionType')$
2: $RemoveFunction(F)$
3: **for all** $rf \in relatedFunctions$ **do**
4: $\quad RemoveFunction(rf)$
5: **end for**

Figure 7.13: An example of removing function *Get* from class *A*

Function Parameter Adding Algorithm 11 adds a new parameter P to function F. Subsequently it adds this parameter to all related functions in the target Resource Model. An example is depicted in Figure 7.14.

Algorithm 11 FunctionParameterAdding

Require: Parameter P to be added to function F
Ensure: Adding parameter P to function F and related functions in Resource Model
1: $AddParameter(F, P)$
2: $relatedFunctions \leftarrow GetRelationsOf(F,' FuntionType')$
3: **for all** $rf \in relatedFunctions$ **do**
4: $p \leftarrow newParameter()$
5: $AddParameter(rf, p)$
6: $CreateRelation(P, p)$
7: **end for**

Figure 7.14: An example of adding function parameter *surname* to function *SetName*

Function Parameter Type Updating As described in Algorithm 12, it first changes type T of function parameter F. Next, it changes the type of all related parameters in the target Resource Model.

Algorithm 12 FunctionParameterTypeUpdating

Require: Function parameter P, new type T of the parameter
Ensure: Update of the parameters's return type and its related parameters in Resource Model
1: $P.ReturnType \leftarrow T$
2: $relatedParameters \leftarrow$
3: $GetRelationsOf(F,' FuntionParameterType')$
4: **for all** $rp \in relatedParameters$ **do**
5: $rp.Type \leftarrow T$
6: **end for**

Function Parameter Renaming Algorithm 13 changes the name of parameter P to a new value N. Next, it changes the name of all related parameters in the target Resource Model. An example is depicted in Figure 7.15.

Algorithm 13 FunctionParameterRenaming

Require: Function parameter P, new name N of the parameter
Ensure: Update of the parameters's name and its related parameters in Resource Model

1: $P.Name \leftarrow N$
2: $relatedParameters \leftarrow$
3: $\quad GetRelationsOf(F,' FuntionParameterType')$
4: **for all** $rp \in relatedParameters$ **do**
5: $\quad rp.Name \leftarrow N$
6: **end for**

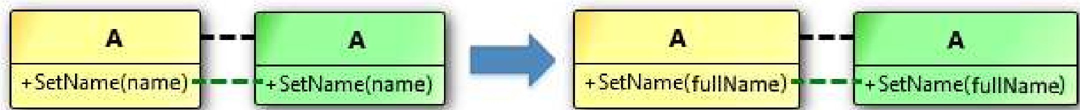

Figure 7.15: An example of renaming parameter *name* of function *SetName* to *fullName*

Function Parameter Removing Algorithm 14 removes parameter P from the PIM and next removes all related parameters in the Resource Model and their relations. An example is depicted in Figure 7.16.

Algorithm 14 FunctionParameterRemoving

Require: Parameter P to be removed
Ensure: Removing of parameter P and related parameters in Resource Model

1: $relatedParameters \leftarrow$
2: $\quad GetRelationsOf(F,' FuntionParameterType')$
3: $RemoveParameter(P)$
4: **for all** $rp \in relatedParameters$ **do**
5: $\quad RemoveParameter(rp)$
6: **end for**

Figure 7.16: An example of removing parameter *surname* from function *SetName*

Attribute Creating Algorithm 15 adds new attribute P to PIM class C. Next, it tries to add the newly created attributes to all related vertices of class C in the Resource Model. There must be done a check if there already exists an attribute with the same name by method $ExistsAnotherAttributeWithSetName$(*Attribute*, $TargetClass$). If so, an exception is raised and the algorithm ends. Finally, the algorithm creates relations between attribute P and attributes created in the Resource Model using function $CreateRelation(PIMModelItem,$ *Resource-ModelItem*).

Algorithm 15 AttributeCreating

Require: Attribute P to be added to class C
Ensure: Creating of the attributes in related vertices of the class
1: $CreateAttributeClass(C, P)$
2: $relatedVertices \leftarrow GetRelationsOf(C,' VertexType')$
3: **for all** $rv \in relatedVertices$ **do**
4: **if** $ExistsAnotherAttributeWithSetName(rv, P.Name)$ **then**
5: // raise a warning
6: $raise\ AttributeWithNameAlreadyExists(N)$
7: **else**
8: $p \leftarrow newAttribute(P.Name)$
9: $CreateAttribute(rv, p)$
10: $CreateRelation(P, p)$
11: **end if**
12: **end for**

Attribute Removing Algorithm 16 removes a given attribute P from its class. Additionally it removes all related attributes in the Resource Model and relations by function $RemoveAttribute(ResourceModelAttribute)$.

Algorithm 16 AttributeRemoving

Require: Attribute P to be removed
Ensure: Removing of the attribute P and related attributes in Resource Model
1: $relatedAttributes \leftarrow GetRelationsOf(P,' AttributeType')$
2: $RemoveAttribute(P)$
3: **for all** $rp \in relatedAttributes$ **do**
4: $RemoveAttribute(rp)$
5: **end for**

Attribute Type Updating As described in Algorithm 17, the algorithm first changes type T of attribute P. Next, it changes the type of all related attributes in the Resource Model.

Algorithm 17 AttributeTypeUpdating

Require: Attribute P, new type T of the function
Ensure: Update of the attribute type and its related attributes in Resource Model
1: $P.Type \leftarrow T$
2: $relatedAttributes \leftarrow GetRelationsOf(P,' AttributeType')$
3: **for all** $rp \in relatedAttributes$ **do**
4: $rp.Type \leftarrow T$
5: **end for**

Attribute Renaming Algorithm 18 gets as parameters the attribute and a new name of the attribute. First, it changes the name of the PIM attribute. Next, it finds all related attributes of attribute P. For every attribute in the Resource

Model, there must be checked that there is not an attribute with the same name (by method $ExistsAnotherAttributeWithSetName()$). If so, an error is raised.

Algorithm 18 AttributeRenaming

Require: Attribute P, new name N of the attribute
Ensure: Renaming of the related attribute in vertex

1: $P.Name \leftarrow N$
2: $relatedAttributes \leftarrow GetRelationsOf(P,' AttributeType')$
3: **for all** $rp \in relatedAttributes$ **do**
4:    **if** $ExistsAnotherAttributeWithSetName(rp, N)$ **then**
5:       // raise a warning
6:       $raise\ AttributeWithNameAlreadyExists(N)$
7:    **else**
8:       $rp.Name \leftarrow N$
9:    **end if**
10: **end for**

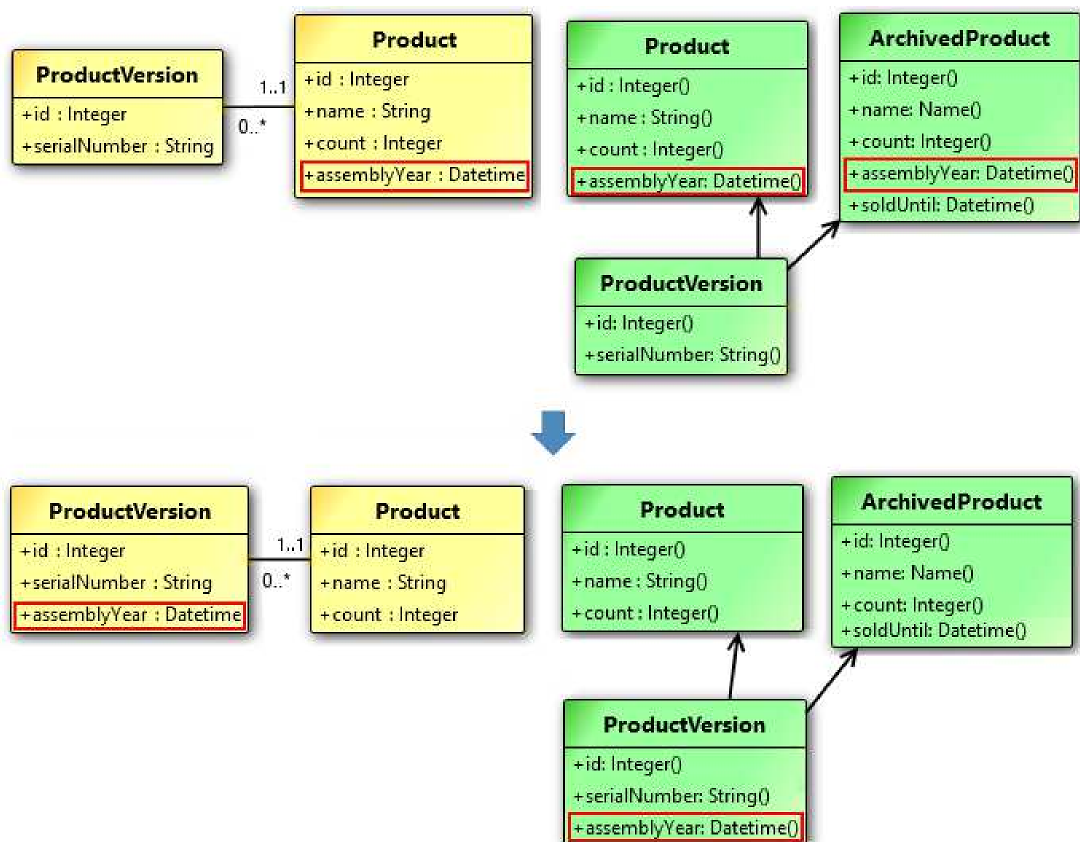

Figure 7.17: An example of moving attribute *assemblyYear* from class *Product* to class *ProductVersion*

Attribute Moving Algorithm 19 gets as parameters the attribute to be moved, the class where the attribute is and the class to which the attribute should be moved. First, the algorithm checks if there exists an attribute with the same name

in the target class using method $ExistsAnotherAttributeWithSetName$(*Attribute*, *TargetClass*). If so, the algorithm raises an exception *AttributeWithNameAlreadyExistsInClass*(*Attribute*, *TargetClass*) and ends. If not, the attribute is moved. Next, the algorithm tries to move all related Resource Model attributes to the Resource Model classes related to target PIM class. If there is no collision with the attribute name, the Resource Model attribute is moved. Otherwise the algorithm raises an error.

An example of the moving propagation is depicted in Figure 7.17. Attribute *assemblyYear* is moved from PIM class *Product* to class *ProductVersion* and subsequently moved in Resource Model.

Algorithm 19 AttributeMoving

Require: Attribute P, target class C_t
Ensure: Moving of attribute P from class C_s to class C_t

```
 1: C_s ← P.parent
 2: if ExistsAnotherAttributeWithSetName(C_t, P.Name) then
 3:     // raise a warning
 4:     raise AttributeWithNameAlreadyExistsInClass(P, C_t)
 5: else
 6:     move attribute P from C_s to C_t
 7: end if
 8: relatedAttributes ← GetRelationsOf(P, 'AttributeType')
 9: relatedtargetClasses ← GetRelationsOf(C_t, 'ClassType')
10: for all rp ∈ relatedAttributes do
11:     for all rtc ∈ relatedtargetClasses do
12:         if ExistsAnotherAttributeWithSetName(rtc, rp.Name) then
13:             // raise a warning
14:             raise AttributeWithNameAlreadyExistsInClass(rp, rtc)
15:         else
16:             move attribute rp from rp.parent to rtc
17:         end if
18:     end for
19: end for
```

7.8.1 Cardinalities

Cardinalities are used when there must be defined a (strict) relation between objects. Cardinalities can be expressed over the PIM too (see Figure 7.3) and they can be used for specific evolution process of the related models. However, cardinalities are not defined in the Resource Model. Since the model itself follows the tree structure, a cardinality has no significance here.

One situation where the cardinality can be used is during generation of resource addresses from the model because the cardinalities can reduce number of generated addresses when there is relation `[1..1]`.

7.8.2 API Versioning

Thanks to the defined algorithms, it is possible to record or generate operations done during the evolution process. These operations can be expressed, e.g., in textual or graphical form or a diff, comprehensible for the API consumer. Since all changes are tracked, the situation when a change in the new API version is not documented should be avoided.

Suppose adding of the new class *Comment* and its connection to the class *User* in the PIM model. These two operations are propagated to the related Resource Model. Except for the transformation of the target model, they can generate the following description:

- New class *Comment* added.
- New resource `shop/user/{:uid}/comment:cid` combined with methods *POST, GET* added.

These descriptions can be included in the new API version next to the data structure models.

7.8.3 View Model

In real-world applications it is common that models defined in PIM do not fully correspond to the models defined in PSM. For instance, there can be different attributes in two corresponding classes, there may be a relation between classes in PIM not represented in PSM, or some classes from PIM might not be represented in the PSM. Due to this feature, it is common that in PIM a special class is created and this class projects only selected attributes or even attributes having different types than in the original class. These classes are usually called *View Model Classes*. For instance, suppose there exists PIM class *User* with attributes *Name*, *Id*, *Email*, *Created* and its corresponding PSM view model *User* only with attributes *Name* and *Email* – PIM attributes *Id* and *Created* are hidden from user view.

Thanks to the above defined extension of the Resource Model with attributes, it is possible to handle this requirement – the Resource Model in fact represents a view of the PIM.

7.8.4 Model Nesting

Another situation can be a combination of several PIM classes into one Resource Model class – so-called *nesting*. An example of the nesting is depicted in Figure 7.18, where PIM classes *Product* and *ProductVersion* are nested into one Resource Model class *Product*. This situation is very common, because the internal architecture and the structures of the system do not have to correspond to the structures offered by the API to the consumers.

To be able to provide changes in the PIM classes and their related PSM classes, we need to define algorithms which solve potential problems, such as, e.g., attribute name collision. Figure 7.19 depicts a situation when attribute *year* is added to PIM class *Product* and this change is correctly propagated to Resource Model class *Product*. Figure 7.20 illustrates a situation of invalid propagation.

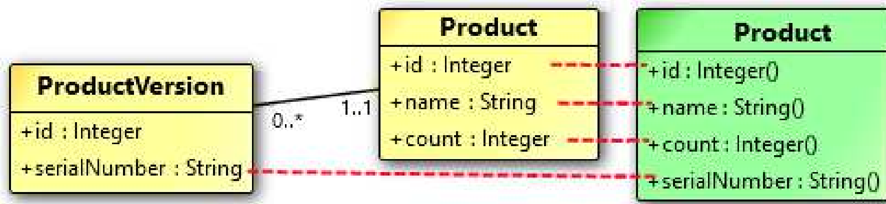

Figure 7.18: An example of nesting PIM classes *Product* and *ProductVersion* into Resource Model class *Product*

Attribute *name* is added to PIM class *Product* and cannot be propagated to Resource Model class *Product*, because there already exists attribute *name*.

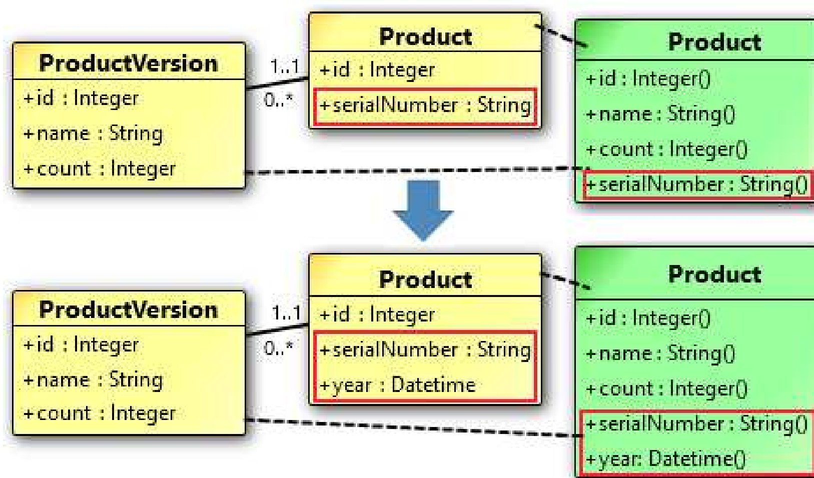

Figure 7.19: An example of correct propagation of adding attribute *year* to class *Product*

7.8.5 Resource Parameters Evolution

In the REST resource, particular data can be specified by its unique identifier (e.g., ID or hash). This identifier is specified after the particular data item. For example, in the resource `shop/user/{:uid}/order/{:oid}`, identifier `{:uid}` specifies ID of the user and `{:oid}` specifies id of the order. It is possible to specify more attributes as identifiers (or keys). In most cases names of these identifiers are mapped directly to the class attributes. So a change of the attribute name influences the resource and correct change propagation must be provided as well.

7.8.6 Applying the Solution on Existing Clients

There can already exist clients before the adoption of this proposal. So the ability to apply or incorporate the proposed solution is an important task. There are no strict contracts in the REST API between server and client like while using web services (e.g., WSDL). It is sufficient to have a list of resource addresses, a description of structures, or JSON schemas. The full list of possible resource addresses can be generated from the particular Resource Model by traversing the model using the depth-first search algorithm (see Section 7.3 for an example). The particular format of the parameters in the address, e.g., of the *id*, can be

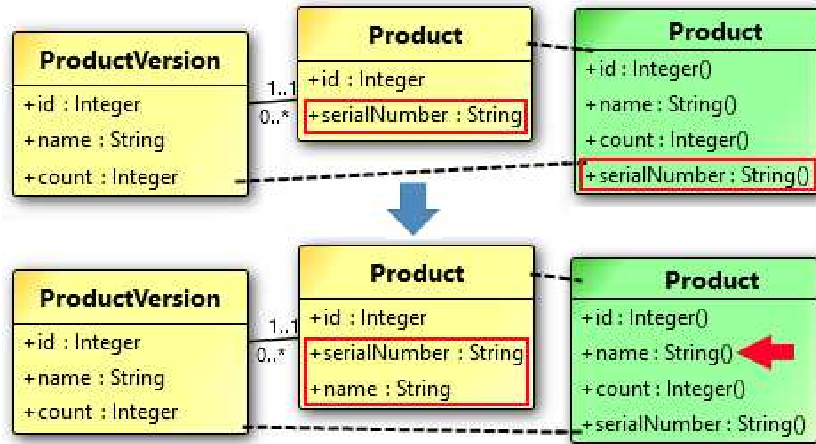

Figure 7.20: An example of invalid propagation of adding attribute *name* to class *Product*

specified for the particular programming language or the framework of the existing client. Next, the implementation can be extended with generating of basic JSON schemas of the resources that can be generated directly from the PIM models.

7.9 Implementation and Experiments

As we have mentioned, since there are no existing solutions providing similar functionality, it is not possible to compare our approach to the competitors. So instead we provide a proof of the concept using real user experience and tests. The presented solution was implemented as an extension of the *DaemonX*. Figures 7.2, 7.21, 7.24, 7.25 are screenshots of the application.

As the source model we use the existing PIM plug-in. For the purpose of this thesis, there were implemented two additional plug-ins. The first one is called *ResourceModel* and it represents the model of the REST resource, which is described in Section 7.3. The second one is an evolution plug-in providing propagation of changes done in the source PIM to the target Resource Model that implements algorithms described in Section 7.4.

7.9.1 Experimental Data

As testing data we used an existing real-world REST API[1] provided by the GitHub and API[2] provided by Twitter [125].

[1] `https://developer.github.com/v3/pulls/`
[2] `https://dev.twitter.com/overview/api`

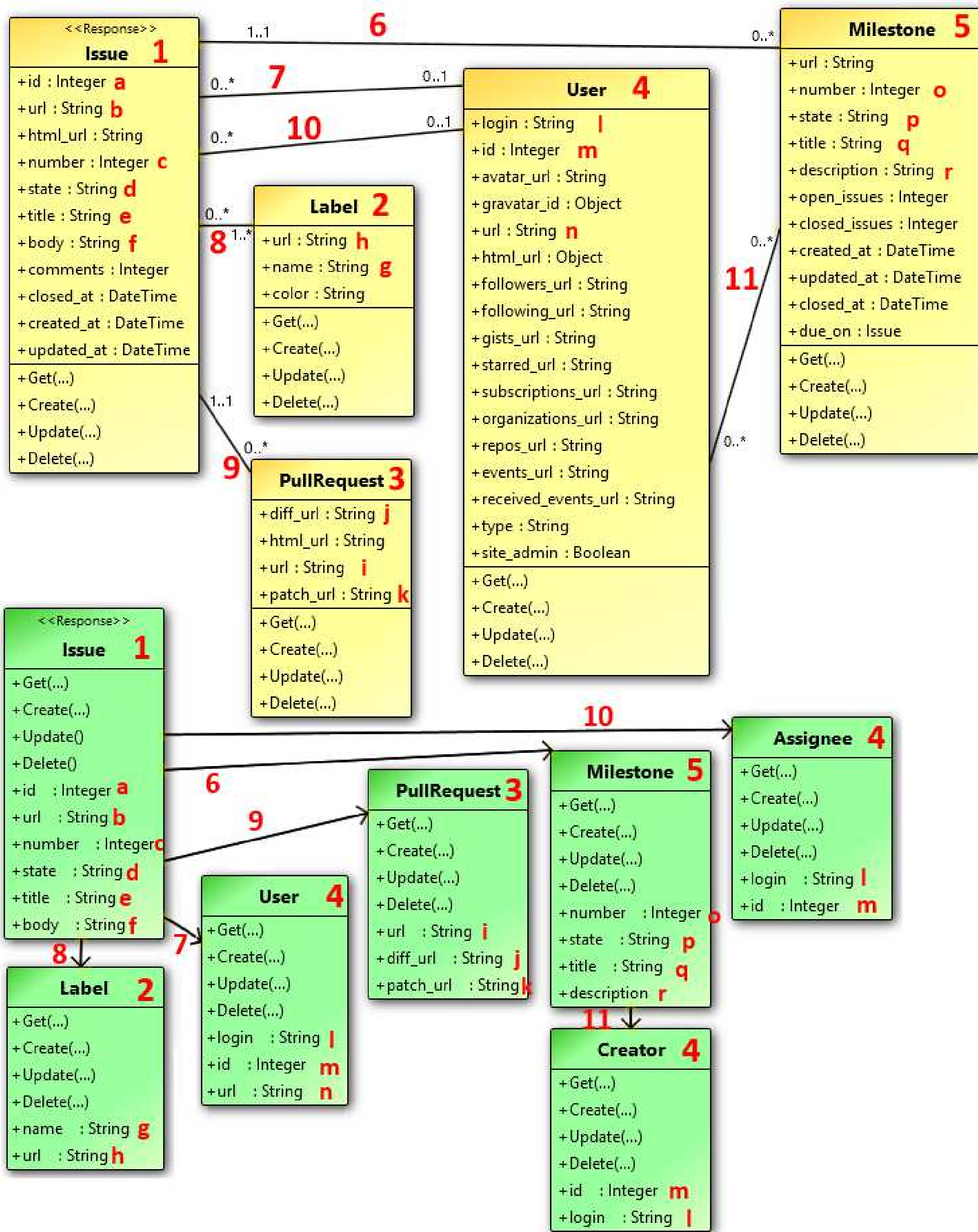

Figure 7.21: An example of the PIM model and its corresponding Resource Model used for experiments

These APIs were selected because they are well-defined, documented, and commonly used. Data structures used for sending request and especially response messages are complex which is convenient for our experiments too. The model representing one of GitHub API responses consists of 5 classes, 7 associations between these classes, and 48 class attributes. There are no functions except for *GET*, *POST*, *PATCH*, and *PUT* methods. In addition, for the purpose of the tests, we have added multiple functions with various parameters to be able to test all operations defined over our models.

In Figure 7.21 there is depicted the PIM model used for experiments and corresponding Resource Model after transformation from the JSON definition

taken from the GitHub API description. Due to mapping complexity, instead of dashed lines, mapping is represented with red numbers – the same numbers in PIM and Resource Model mean that there exists a mapping between these items. As we can see, PIM class *User* is mapped in the Resource Model to multiple vertices, namely *User*, *Assignee*, and *Creator*. Other PIM classes are mapped to the corresponding Resource Model vertices.

7.9.2 Experiments

As we have mentioned, the provided resource API was analyzed and the corresponding PIM and Resource Model were created. Next, these two models were mutually mapped. And, finally, all PIM operations were applied as described later in this section. Thanks to the implemented algorithms, all changes were propagated to the target Resource Model and then checked by the following scenario:

1. Make a change in the source PIM.
2. Let the changes propagate to the target Resource Model.
3. Let the resource API generate from the evolved Resource Model.
4. Validate the new resource API.

In particular, the following changes were done in the PIM:

- **Class Creating**: Impact of creating of a new class in PIM does not have an impact on the Resource Model because it is does not related to it.
- **Class Removing**: This operation when propagated to the Resource Model class has an impact on resource address, because a part of tree will be removed.
- **Class Renaming**: This change, when propagated to the Resource Model class, has an impact on the resource address, because the name of the class must be replaced by the new one in the resource addresses.
  This propagation example is depicted in Figure 7.22. Class *Label* is renamed into *DetailInformation*. This class is mapped to two vertices *Label* in the Resource Model. After the propagation, both vertices are renamed to *DetailInformation* (see Figure 7.23).
- **Connection Creating**: Adding of an association between PIM classes impacts the target model only if the propagation is permitted. In this case, the target model changes its structure and a new resource address containing this newly added class will be generated.
  An example of this case is depicted in Figure 7.24 where a connection is added between classes *Label* and *Milestone*. The result of the propagation to the Resource Model is depicted in Figure 7.25). The algorithm added a vertex *Label* as a child of vertex *Milestone* and vertex *Milestone* with child *Creator* as a child of vertex *Label*. In this example we can see that the added relation *Milestone* $\rightarrow$ *Label* makes sense, but the second one does not. In this situation, the prompt policy can be applied to propagate only the reasonable subset.

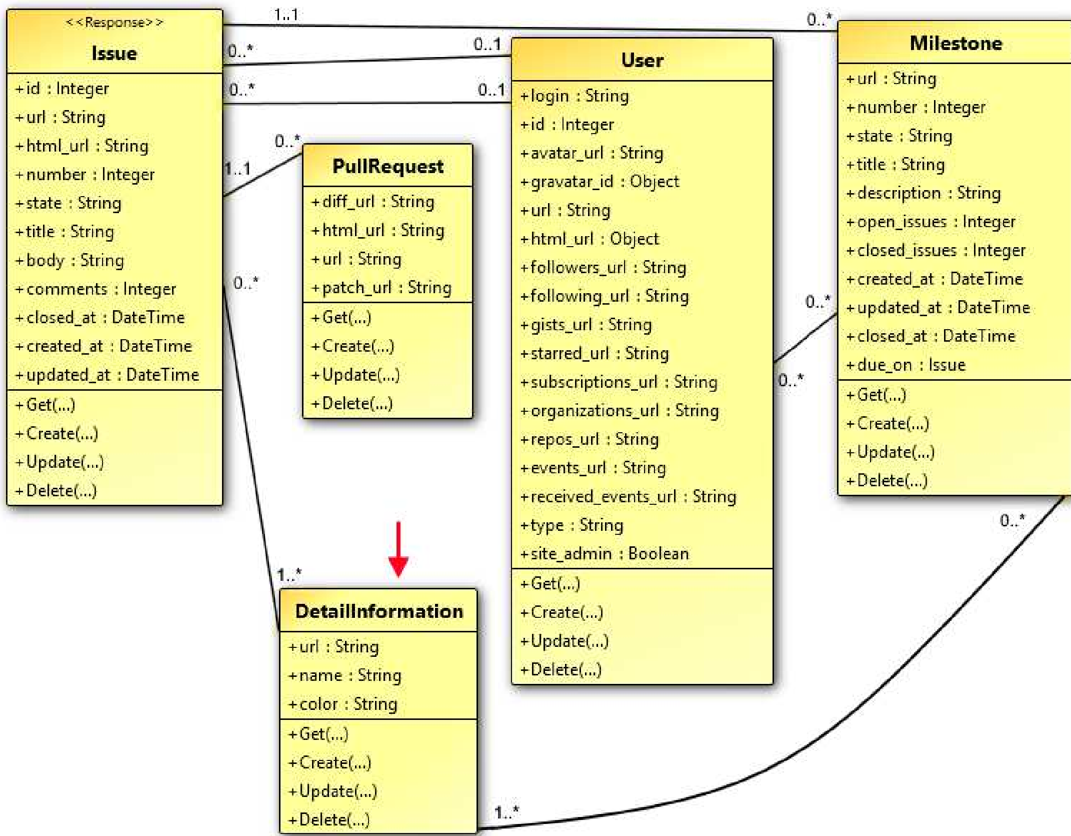

Figure 7.22: An example of renaming class *Label* to *DetailInformation*

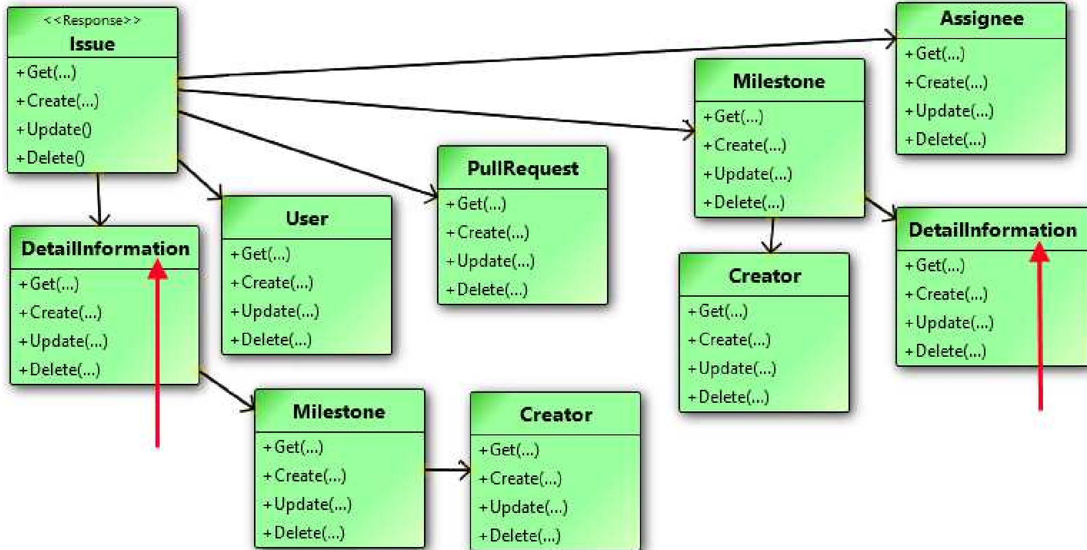

Figure 7.23: An example of propagation of class renaming from Figure 7.22 to the Resource Model

- **Connection Removing**: Association removal affects the target model in the way that it removes a part of the diagram which causes that there will be generated less resource addresses from the model.
- **Function Creating**: Adding of a new function to the PIM class will cause creation of new resource addresses in the Resource Model.
  This situation is depicted in Figure 7.26. Function *SetUrl* is added to PIM

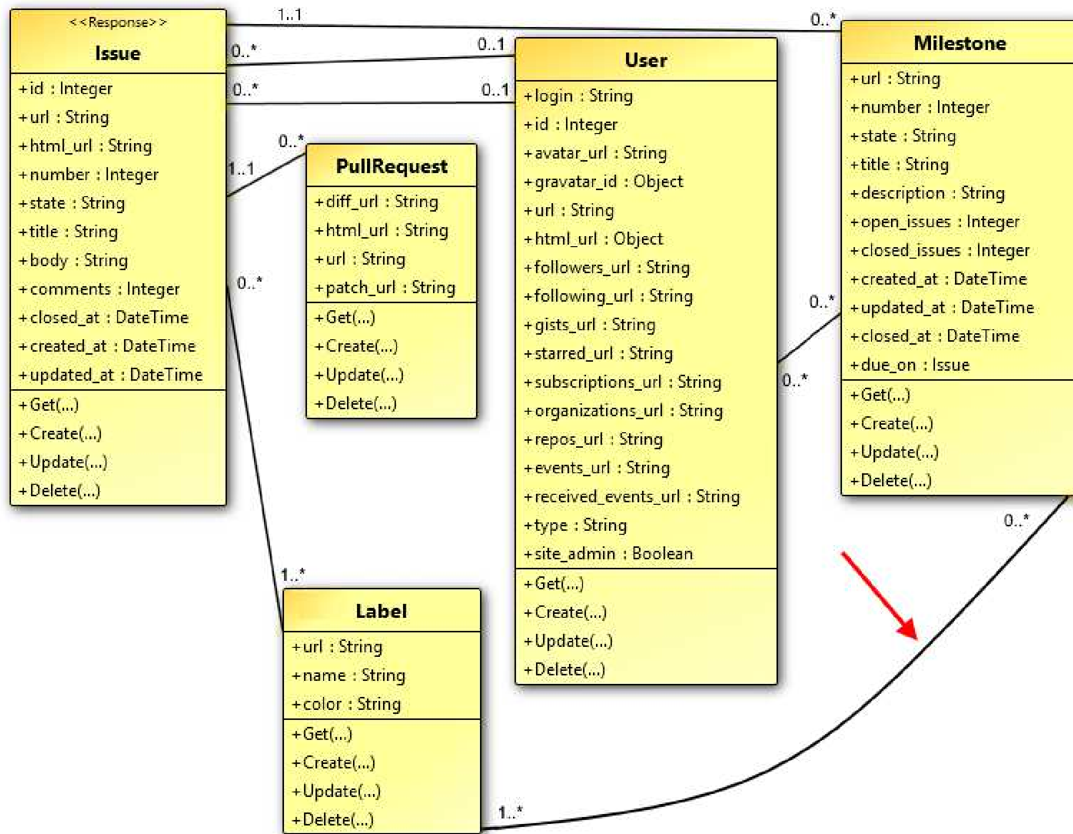

Figure 7.24: An example of adding a connection between classes *Label* and *Milestone*

class *User*. This class is mapped to vertices *User*, *Assignee*, and *Creator* in the Resource Model. After the propagation, *SetUrl* is added to these vertices (see Figure 7.27).

- **Function Removing**: Removing a function in the PIM class will cause removal of the respective addresses from the Resource Model.
- **Function Renaming**: A change of a function in the PIM class will cause a change of the respective addresses in the Resource Model.
- **Function Return Type Changing**: A change of a function return type does not have any impact on the resource address, because it is not present there.
- **Function Parameter Adding**: Adding of a new parameter has an impact on all related functions in the Resource Model and changes resource addresses everywhere, where the function is used.
- **Function Parameter Removing**: As in the previous case, this operation has an impact on all related functions in the Resource Model.
- **Function Return Type Changing**: A change of the function return type does not have any impact on the resource address, because it is not present there.
- **Function Parameter Renaming**: This operation changes a part of the resource address, because of a new parameter name.

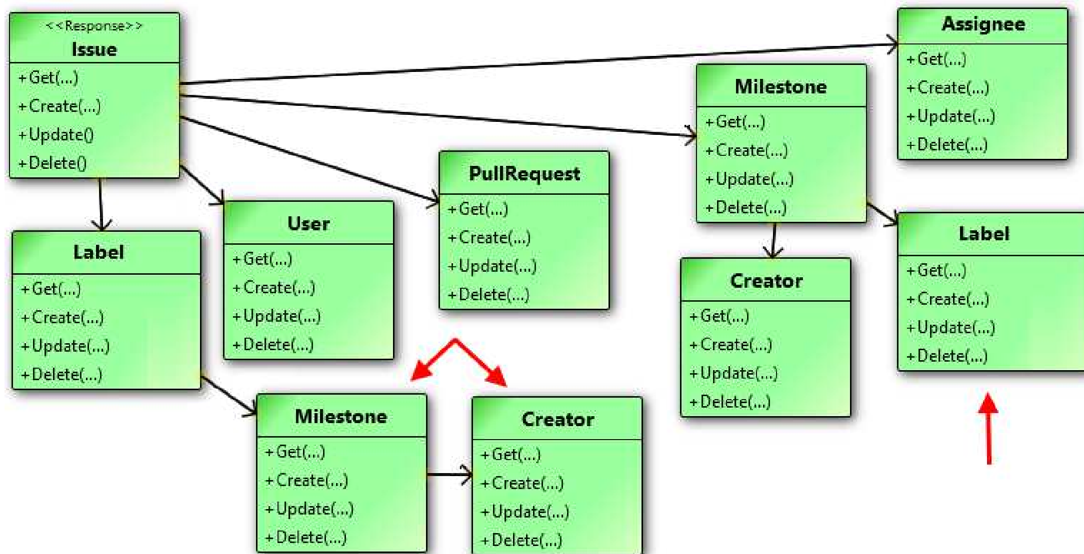

Figure 7.25: An example of propagation of adding a connection from Figure 7.24 to the Resource Model

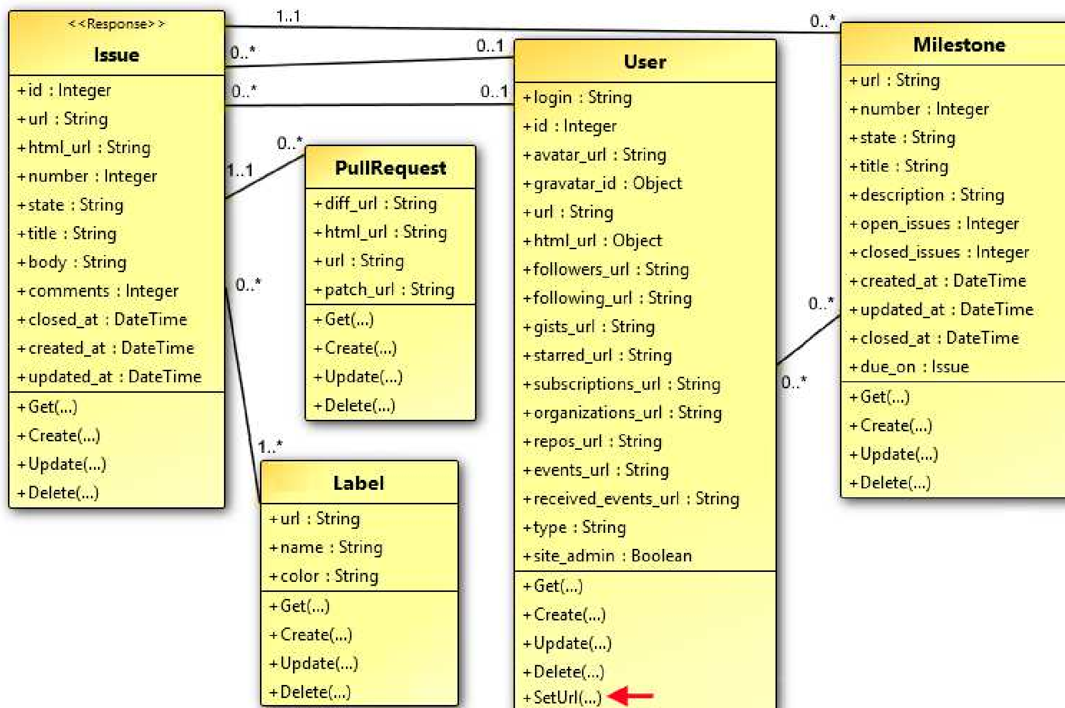

Figure 7.26: An example of adding a function *SetUrl* to a PIM class *User*

- **Function Parameter Type Changing**: This operation does not have an impact on the resource address, because it is not present in the resource address.

- **Attribute Creating**: Adding of a new attribute to the PIM class will cause creation of a new attribute in the Resource Model. To be able to perform this operation, all preconditions must be satisfied. We have added new attribute *alias* of type *String* to PIM class *User* (see Figure 7.21). Since this class is mapped to Resource Model vertices *User*, *Creator*, and *Assignee*, attribute creation was propagated to these vertices. Since there

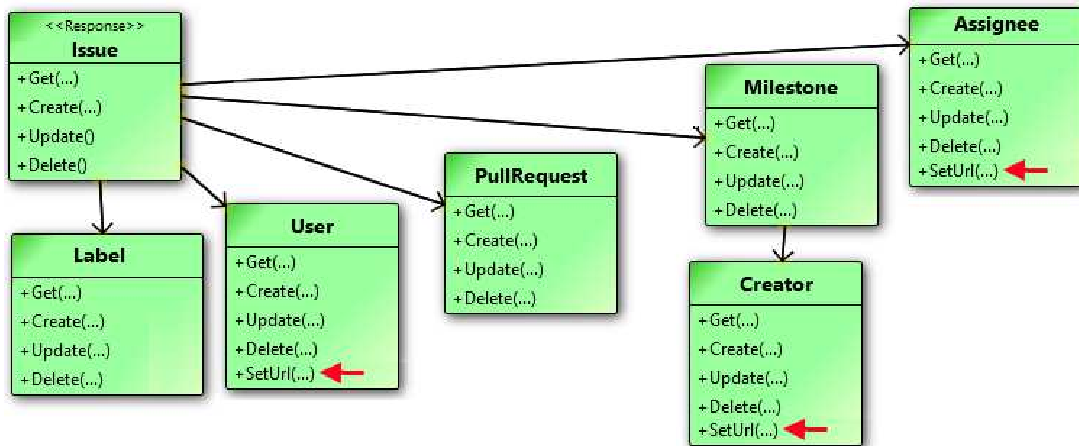

Figure 7.27: An example of propagation of adding function *SetUrl* from Figure 7.26 to the Resource Model

were no attributes with the same name, name collision did not occur.

- **Attribute Removing**: Removing an attribute from a PIM class will cause removal of the related attributes from the Resource Model. As in the previous case, we focused on PIM class *User*. First we removed attribute *login*. Since this attribute is mapped to attributes in Resource Model vertices *User*, *Creator*, and *Assignee*, they were removed from these classes too. Next, we removed attribute *type*. Since this attribute has no relations in the Resource Model, no operation was propagated.
- **Attribute Renaming**: A change of an attribute in the PIM class will cause a change of the respective attributes in the Resource Model. To be able to perform this operation, preconditions must be satisfied. In this case we renamed attribute *state* of PIM class *Milestone* into *status*. Due to the relation to the attribute *state* in Resource Model vertex *Milestone* and since there is no attribute with the same name in the class or vertex, this change was successfully propagated. Subsequently we renamed attribute *html_url* in PIM class *User* into name *url*. Since an attribute with the same name already existed, the operation was canceled.
- **Attribute Type Changing**: A change of an attribute data type does not have any impact on the generated resources, because attribute type is not present in the resource. In this case we changed the data type of attribute *state* in class *Milestone* from *String* to enum *MilestoneStateEnum*. Since there is no restriction on the type change, this change was propagated successfully.
- **Attribute Moving**: Moving an attribute from one PIM class to another will cause moving of all related attributes in the Resource Model. To be able to perform this operation, preconditions must be again satisfied. The algorithm checks all possible collisions of the attribute names. E.g., moving an attribute to a class where there already exists an attribute with the same name or moving multiple attributes to one class represent the same evolution process. Suppose moving of attribute *url* from PIM class *User* to PIM class *Issue*. Since there already exists an attribute with the same name, the operation was canceled. Next, suppose moving of attribute *title*

from PIM class *Issue* to PIM class *Label*. Since there is no attribute with the same name in the target class, the operation can be propagated to all related Resource Model vertices. Again, there exists no attribute *title* in Resource Model vertex *Label*, the attribute moving can be propagated.

The presented experiment uses a PIM model with 5 classes and 7 associations. Related Resource Model contains of 7 vertices and 6 edges. From the Resource Model 7 resources can be generated (and multiplied by particular HTTP methods). All operations defined over PIM model were applied and caused 25 changes in the Resource Model. This lead to 100 changes in related resources, depending on used HTTP methods. From the experiment we can observe the following results:

- The number of changes depends on the Resource model and relations to the PIM model.
- During the experiments, 17 operations over PIM model caused 25 changes in Resource Model and 100 in the resources.
- Attribute operations have no impact on the generated resources.
- Removing of a PIM class can significantly change the Resource Model, e.g., removing of PIM class that is related to the root vertex of the Resource model.

We can conclude with the finding that all the changes were propagated correctly and completely, whereas we saved lots of manual effort and avoided possible errors. An interested reader may download the implementation as well as the sample models from this resource[3].

7.10 Conclusion

The main aim and contribution of this chapter is to provide the ability of maintaining REST resources during time and their continuous development during evolution of the system. Our target is to reduce the possibility of errors during manual update and to reduce the amount of work which must be done because of a change of the resource related to the change of the PIM. For this purpose we provide algorithms to propagate changes again based on the MDA approach. As the source model, we use the well-known PIM and as the target model we defined new PSM model representing a REST service resource, called the Resource Model. The Resource Model representing the REST service can be subsequently used for generating a stub for a particular client and server implementation or a programming language. Next, the model can be combined with existing solutions like RAML or Swagger which would then provide a full MDA managed and documented evolution of the REST API. Finally, this proposal can be integrated together with other PSM models (e.g., database, XML, object etc.) to provide a complex MDA solution, exactly like we implement it in our evolution and change-management framework *DaemonX*. But there is no limitation to implement presented algorithms in transformation languages such as QVT [48].

[3]`http://www.ksi.mff.cuni.cz/~polak/daemonx/`

7.10.1 Future Work

Even though the approach is complex and robust, there still exists issues that can be solved:

- *Enlarge set of operations*: Even the model supports set of all commonly used operations for its update, there can be defined and analyzed shortcut operations, such as *Connection moving* or *Property moving*.
- *Query evolution*: Current model can be used as a source for a query model over REST API and subsequent definition of transformation between these two models. This solution can be further intercorporated with existing solutions.

8. Schema Mapping

In this chapter we explore the application of XML schema similarity mapping in the area of conceptual modeling of XML schemas. This approach extends the DaemonX framework and the approaches from the previous chapters with the following idea: We have a system with multiple models and schemas and we need to integrate an existing schema from an external source into our system. We expand upon our previous efforts to map XML schemas to a common PIM schema using similarity evaluation based on exploitation of a decision tree. In particular, in this approach a more versatile method is implemented and the decision tree is trained using a large set of user-annotated mapping decision samples. Several variations of training that could improve the mapping results are proposed. The approach is implemented within a modeling and evolution management framework called eXolutio [71], a predecessor of DaemonX devoted purely to XML technologies, and its variations are evaluated using a wide range of experiments. The approach was presented in [66].

8.1 Introduction

The XML has become one of the leading formats for data representation and data exchange in the recent years. Due to its extensive usage, large amounts of XML data from various sources are available. Since it is common that sooner or later user requirements change, it is very useful to adapt independently created XML schemas that represent the same reality for common processing. However, such schemas may differ in structure or terminology. This leads us to the problem of *XML schema matching* that maps elements of XML schemas that correspond to each other. Schema matching is extensively researched and there exists a large amount of applications, such as data integration, e-business, schema integration, schema evolution and migration, data warehousing, database design and consolidation, web site creation and management, biochemistry and bioinformatics, etc.

Matching a schema manually is a tedious, error-prone and expensive work. Therefore, automatic schema matching brings significant savings of manual effort and resources. But it is a difficult task because of the heterogeneity and imprecision of input data, as well as high subjectivity of matching decisions. Sometimes the correct matches have to be marked only by a domain expert. As a compromise, in (semi)automatic schema matching the amount of user intervention is significantly minimized. For example, the user can provide information before matching/mapping, during the learning phase. Or, after creation of a mapping the user can accept or refuse suggested mapping decisions which could be later reused for improvement of further matching.

In our case schema matching is used as the key step during the schema integration process. In particular, we match elements from independent XML schemas against elements in the common PIM schema to establish the respective PSM-to-PIM mapping. In particular, this work uses a (semi)automatic approach to schema matching. We explore the applicability of decision trees for this specific use case. A decision tree is constructed from a large set of training samples

and it is used for identification of correct mapping. For our target application various modifications of the training process are proposed and experimentally evaluated on the basis of several common hypotheses. The proposed approach extends previous work [116] and it was implemented and experimentally tested in the modeling and evolution management tool *eXolutio* which is a predecessor of *DaemonX* focussing purely on XML data.

8.2 Related Works

In this section existing schema matching approaches are described. As we have mentioned, since the number of the approaches is high, we have selected only the key classical and the most popular representatives. Next, we present works that analyze and compare various existing approaches and their characteristics.

8.2.1 COMA

COMA matcher [31] is an example of a *composite* approach. Individual matchers are selected from an extensible library of match algorithms. The process of matching is *interactive* and *iterative*. A match iteration has the following three phases: (1) *User feedback and selection of the match strategy*, (2) *Execution of individual matchers*, and (3) *Combination of the individual match results.*

Interactive mode The first step in the iteration is optional. The user is able to provide *feedback* (to confirm or reject previously proposed match candidates or to add new matches) and to define a *match strategy* (selection of matchers, strategies to combine individual match results). In *automatic mode* there is only one iteration and the match strategy is specified by input parameters.

Reuse of match results Since many schemas to be matched are very similar to the previously matched schemas, match results (intermediate similarity results of individual matchers and user-confirmed results) are stored for later reuse.

Aggregation of individual matcher results Similarity values from individual matchers are aggregated to a combined similarity value. Several aggregate functions are available, for example *Min*, *Max* or *Average.*

Selection of match candidates For each schema element its best match candidate from another schema is selected, i.e., the ones with the highest similarity value according to criteria like *MaxN* (n elements from schema S with maximal similarity are selected as match candidates), *MaxDelta* (an element from schema S with maximal similarity is determined as match candidate plus all S elements with a similarity differing at most by a tolerance value d which can be specified either as an absolute or relative value), or *Threshold* (all S elements showing a similarity exceeding a given threshold value t are selected).

The COMA++ [6], an extension of COMA, supports a number of other features like merging, saving and aggregating match results of two schemas.

8.2.2 Similarity Flooding

Similarity Flooding [86] can be used to match various data structures – data schemas, data instances or a combination of both. The algorithm is based on the idea that the similarity of an element is propagated to its neighbors. The input data is converted into *directed labeled graphs.* Every edge in the graphs is represented as a triple (s, l, t), where s is a source node, t is a target node, and l is a label of the edge. The algorithm has the following steps: (1) *Conversion of input schemas to internal graph representation*, (2) *Creation of auxiliary data structures*, (3) *Computation of initial mapping*, (4) *Iterative fix-point computation* and (5) *Selection of relevant match candidates.* The accuracy of the algorithm is calculated as the number of needed adjustments. Output mapping of elements is checked and if necessary, corrected by the user.

Matcher The main matcher is structural and is used in a *hybrid* combination with a simple name matcher that compares common affixes for initial mapping. The matcher is iterative and based on *fixpoint computation* with *initial mapping* as a starting point.

Fixpoint computation The similarity flooding algorithm is based on an iterative computation of σ-values. The computation of the σ-values for a map pair (x, y) is performed iteratively until the Euclidean length of the residual vector $\Delta(\sigma^n, \sigma^n - 1)$ becomes less than ϵ for some $n > 0$ (i.e. the similarities stabilize):

$$\sigma^{i+1}(x,y) = \sigma^i(x,y) + \sum_{\substack{(a,l,x)\in E_A \\ (b,l,y)\in E_B}} \sigma^i(a,b)w((a,b),(x,y)) + \sum_{\substack{(x,l,c)\in E_A \\ (y,l,d)\in E_B}} \sigma^i(c,d)w((x,y),(c,d)) \tag{8.1}$$

where $\sigma^i(x, y)$ is the similarity value in i-th iteration of nodes x and y and σ^0 is the value computed in the initial mapping.

Similarity Flooding can be further improved for example by usage of another matcher for initial mapping or auxiliary source of information – e.g. dictionary.

8.2.3 Decision Tree

In [36] a new method of combining independent matchers was introduced. It is based on the term *decision tree.*

Definition 22. *A* decision tree *is a tree $G = (V, E)$, where V_i is the set of internal nodes (independent match algorithms), V_l is the set of leaf nodes (output decision whether elements do or do not match), $V = V_i \cup V_l$ is the set of all nodes, E is the set of edges (conditions that decide to which child node the computation will continue).*

The decision tree approach does not have the following disadvantages of aggregation of result of independent matchers used, e.g., in COMA:

- **Performance:** In the composite approach with an aggregate function, all of the match algorithms have to run. The time required is worse than with a decision tree.

- **Quality:** Aggregation can lower the match quality, e.g., if we give higher weights to several matchers of the same type that falsely return a high similarity value.
- **Extendability** is worse, because adding a new matcher means updating the aggregation function.
- **Flexibility** is limited, because an aggregation function needs manual tuning of weights and thresholds.
- **Common Threshold:** Each match algorithm has its own value distribution, thus it should have own threshold.

On the other hand, the main disadvantage of Decision Trees is the need of a set of training data.

8.2.4 XML Schema Clustering with Semantic and Hierarchical Similarity Measures

Paper [89] presents a schema clustering process by organising the heterogeneous XML schemas into various groups. The methodology considers not only the linguistic and the context of the elements but also the hierarchical structural similarity. The authors present the *XMine* methodology that quantitatively determines the similarity between the heterogeneous XML schemas by considering the semantic, as well as the hierarchical structural similarity of elements. Similar schemas are clustered into the separate meaningful classes.

Whilst there are several XML documents and schema clustering techniques available, the paper enhances this task by adding the hierarchical similarity in clustering by addressing the element level hierarchical positions. The XMine methodology can deal with the varying structures of schemas and with the varying aspects of semantic differences in the schema elements.

The XMine Methodology

The XMine methology consists of three phases:

1. **Preprocessing**: In this phase, common and similar features between various schemas are analyzed. It consists of four subphases, based on different analyzers that are run – *structure analyzer* (it analyzes the structure of a schema and transforms it into a labelled and directed acyclic tree graph), *element analyzer* (it measures the similarity between arbitrary elements in different schemas primarily based on element names), *maximally similar paths finder* (it determines the common and similar hierarchical structure of elements defined in the schema). At the end, an overall degree of similarity between schemas is computed by taking the element and structural similarity into consideration.
2. **Data Mining**: The schemas similar in structure and semantics are grouped together to form a hierarchy of schema classes using an agglomerative clustering algorithm. The reasons to use the hierarchical method are as follows:

Firstly, similarity of the clusters is based on the number of common elements that the schemas share. Secondly, the algorithm repeatedly merges the clusters to form a final solution. Therefore this clustering process can be analyzed in the post-processing phase to form a hierarchy of schema classes. Thirdly, the used algorithm should be resistant to noise and outliers. Since the data collection can have a schema that may not be related to other schemas, outliers may be present. This algorithm uses a k-nearest neighbour graph in the partitioning phase that ensures reducing of the effects of noise and outliers. Fourthly, the used algorithm should not require the number of clusters to be pre-determined because the relationships between data are unknown. Finally, because the volume of query data can be very large, the algorithm should be scalable.

3. **Postprocessing**: In this phase, the discovered schema patterns are visualized as a tree of clusters called *dendogram*. The dendogram shows the clusters that are merged together and the distance between these merged clusters. This facilitates the generalization and specialization processes of the clusters to develop an appropriate schema class hierarchy. Each cluster, that contains a set of similar schemas, forms a node in the hierarchy, where all nodes (or clusters) are at the same conceptual level.

For the experiments the authors use data from different domains that have structural and semantic differences. The validity and quality of the XMine clustering solution are verified using two common evaluation methods: (1) the intra-cluster and inter-cluster quality and (2) the FScore measure.

The authors claim that XMine comes closer to schema only approaches such as COMA. However, the main difference between these approaches and XMine is that the structural similarity is derived from the maximal similar paths obtained by using the adapted sequential pattern mining algorithm.

Conclusion

The paper presents the XMine methodology that accurately clusters the schemas by considering both structural and semantic information of elements. XMine includes structural information in the similarity measurement by finding the maximal similar paths between schemas.

The evaluation shows the effectiveness of XMine in categorizing a set of heterogeneous schemas into relevant classes that facilitate the generalization of an appropriate schema class hierarchy. The authors declare that this schema clustering approach can also easily be applicable on the document instances after representing each document as a tree. Moreover, the methodology is applicable to general web documents after performing XHTML conversion, and then representing the documents as the trees.

8.2.5 Minimizing User Effort in XML Grammar Matching

In paper [122] the authors address the XML grammar matching/comparison problem, i.e., the comparison of DTDs and/or XML Schemas based on their most common characteristics. The goal is to develop an effective XML grammar matching

method minimizing the amount of manual work needed to perform the match task that requires:

1. Considering various characteristics and constraints of the XML grammars being matched, in comparison with existing 'grammar simplifying' approaches.

2. Allowing a flexible and extensible combination of different matching criteria, adaptable to various application scenarios, in comparison with existing static methods.

3. Effectively considering the semi-structured nature of XML, as the most prominent and distinctive feature of an XML grammar [4], in comparison with existing heuristic or generic approaches.

XML Grammar Tree Representation

The authors provide a XML grammar tree model representation that accurately captures the structural properties of XML grammars and considers their most common characteristics. To reduce time complexity of the algorithms, e.g., for edit distance computing, the tree grammar is transformed to a fully ordered tree.

Another problem that must be solved is recursion and references. The authors solve this problem by creation of a new leaf node n' of each recursive node n, such as n' has the same components as n and data-type set to *Recursive*.

The XML grammar matching approach consists of four components:

1. The **XML Grammar Tree Comparison** component for computing the distance (and consequently the similarity) between two XML grammar trees.

2. The **Extensible Matchers** component encompassing several independent matching algorithms exploited via the *Edit Distance* component to capture the similarities between XML grammar nodes based on their characteristics (label, data-type, cardinality constraints, alternativeness constraints, and node ordering).

3. The **Mapping Identification** component, interacting with the *Tree Edit Distance* component to identify the edit script, and consequently the edit distance mappings, between the compared XML grammar trees.

4. The **UserFeed** component to consider user predefined mappings and user feedback in producing matching results. Where *UserFeed* is an operation that transforms an XML grammar tree A into A', such as in the destination tree $A0$, nodes corresponding to predefined matches are eliminated, along with their corresponding sub-trees.

The authors also present an XML grammar matching framework in the experimental *XML Structural and Semantic Similarity* (XS3) prototype. The main criterion used to assess the effectiveness of automatic schema matching methods is the amount of manual work and user effort required to perform the matching task. In this context, most existing approaches propose to first manually

solve the match task, in order to exploit the obtained results as a reference to evaluate the quality of the matches produced by the system. Thus, similarly to information retrieval, the Precision and Recall metrics can be utilized in comparing real-world and system generated matches. The results of the experiments were compared with well-known solutions, such as Cupid [81], Similarity flooding, COMA, XClust [76] or Relaxation Labeling [138].

Discussion

In this paper, the authors propose a framework for XML grammar matching and comparison based on the concept of edit distance. They claim that this is the first attempt to exploit tree edit distance in an XML grammar matching context. The presented method aims at minimizing the amount of manual work needed to perform the match task.

8.2.6 XML Matchers: Approaches and Challenges

In paper [2] the authors provide a detailed description and classification of existing XML Matchers. They present an *XML Matcher template* which describes the main components of an XML Matcher and discuss how each of these components has been implemented in some popular XML Matcher. This helps to describe how XML Matchers work in practise and to compare them. Next, they discuss selected commercial prototypes designed to find matchings between DTDs/XSDs.

Schema Matchers and Algorithms

The authors divide matchers into various groups, e.g., *Schema matchers* vs. *Instance matchers* and *Simple matchers* vs. *Complex matchers* (these terms are describer later in Section 8.3) and describe differences between them and mention representatives of particular types.

DTD and XSD in XML Matchers

The authors discuss how features of DTDs/XSDs may impact the semantics of schemas encoded by means of these language. They compare them with other representations, e.g., *Entity-Relational* (E/R), and analyze specific feature of DTD/XSD, such as hierarchical structure. Next, they compare DTD and XSD directly and mention abilities of XSD not presented in DTD that can improve matching results.

A Template to Classify XML Matchers

The main contribution of the approach is a template to classify XML Matchers. The authors mention that there is no official definition of the XML Matcher at all. Different researches and papers present own definitions of matchers and they use different methodologies and tools to find semantic matching between DTD/XSD. The authors present an abstract model which plays the role of a template whose structure is applicable to all existing XML Matchers. It consists of several components – for instance, a component could specify the strategy adopted for representing input DTDs/XSDs. Once these components have been

defined, they can represent each existing approach as a set of them. Such a template acts as universal container which is generally sufficient to describe the main features of most of the existing XML Matchers.

Definition 23. *A XML Matcher template is a tuple* $T =< IFD, IRD, M, k, \sigma, \lambda, \phi >$ *where:* IFD *and* IRD *are the Input Format and the Internal Representation domains, respectively.* $M : IFD \rightarrow IRD(pre-processingfunction)$ *is a function which receives an element of* IFD *and returns an element of* IRD*. For each* $S \in IFD$*, we denote as* $M(S)$ *the corresponding element in* IRD*.* k *is an integer greater than or equal to 1.* $\sigma = [\sigma_1, ..., \sigma_k]$ *is a group of k functions. For each* $i = 1...k$ *and for each* $S \in IFD, \sigma_i : M(S) \times M(S) \rightarrow [0, 1]$ *is called* similarity function. $\lambda : [0, 1]^k \rightarrow [0, 1]$ *is an aggregating function;* $[0, 1]^k$ *k denotes the hypercube in* $\mathbb{R}^k$*, i.e., a k-th dimensional array whose components range from 0 to 1.* $\phi : IFD \times IFD \rightarrow \mathbb{R}^+$ *is a* schema similarity function *;* $\mathbb{R}^+$ *is the set of non-negative real numbers.*

Challenges for XML Matchers

The authors refer that even there was done a huge amount of work at both academic and industrial levels in schema matching area, there are still open problems and gaps that should be solved and present some examples of interesting and problematic topics:

- **Schema Clustering and Data Integration on Web**: The traditional integration techniques generally assume that sources belong to the same domain, which is not true at all, especially on the web. An example can be integration of government services from various sectors into one portal. A better solution is to first classify the related schemas into homogenous domains to obtain a DTD/XSD representing the domain as a whole. Then to cluster the DTDs/XSDs of the detected domains to obtain more abstract domains, each represented by a unique DTD/XSD. This process could be repeated until a unique abstract DTD/XSD representing all services is found.
- **Uncertainty Management in XML Matchers**: The output generated by a matcher is often uncertain and additional evaluation is needed. The authors mention an example with possible information loss because of invalid matching. The solution of this problem can be a human advice. In order to manage uncertainty in the schema matching process, the authors refer to concept of *probabilistic mapping*, i.e., they suggested to associate a score with each discovered matching, stating its probability of being correct.

Discussion

The authors provide a detailed analysis of approaches explicitly designed to find matches between DTDs/XSDs. They discuss data representations different from DTD/XSD, such as E/R. The approach introduces a template, called XML Matcher Template, to describe the main components of an XML Matcher, their role and their interactions. They use this template to characterize and compare a set of XML Matchers that gained a large popularity in the literature. Next, the paper contains an analysis of the existing tools for handling schema matching

tasks (for the comparison the XML Matcher template is used too). Finally, they present two important challenges related to XML Matchers, namely the clustering of large collections of DTDs/XSDs and the uncertainty management in XML Matchers.

8.2.7 Comparison of the Related Works

In the first three papers [31, 86, 36] we discuss various matchers, namely COMA, Similarity flooding and Decision tree approaches. The approach [89] presents the XMine methodology that quantitatively determines the similarity between heterogeneous XML schemas by considering the semantic, as well as the hierarchical structural similarity of elements. Similar schemas are clustered into the separate meaningful classes. Whilst there are several XML documents and schema clustering techniques available, this paper enhances this task by adding the hierarchical similarity in clustering via addressing the element level hierarchical positions. The authors claim that the XMine methodology can deal with varying structures of schemas and with varying aspects of semantic differences in the schema elements. The approach [122] focuses on minimizing user effort during the matching process. The paper analyzes various types of matchers and presents a XML Matchers template that should help during the comparison of various matchers. The last paper [2] introduces a template, called XML Matcher Template, to describe the main components of an XML Matcher, their role and their interactions. They use this template to characterize and compare a set of various commonly used XML Matchers which gained a large popularity in the literature.

A general comparison of the first three discussed methods is introduced in Table 8.1. The decision tree approach seems to be the most promising for our application – it is dynamic and versatile. Furthermore it has desirable values of compared properties – it is highly extensible, quick and has a low level of user intervention and a low level of required auxiliary information. As continuing work, the authors currently investigate the extension of our method to deal with user derived data-types.

	Extensibility	Speed	User intervention	Auxiliary info
COMA	Low	Low	High	Low
Similarity Flooding	None	High	None	None
Decision Tree	High	High	Low	Low

Table 8.1: A comparison of the selected existing solutions

Based on existing works, we have explored various approaches to schema matching and selected the most promising possible approach for our application – schema matching using a decision tree. The main motivation of this work was to define a dynamic, versatile, highly extensible solution that requires a low level of user intervention and a low level of required auxiliary information.

8.3 Schema Matching

The (semi)automatic or automatic process of finding correspondences between elements of two schemas is called *schema matching*. In this thesis the term

schema matching is used for simplicity in a general way, but there are various specific types.

- *Schema-to-schema matching* has as an input two XML schemas.
- *Instance-to-instance matching* has as an input two XML documents.
- *Schema-to-instance matching* has as an input an XML document and an XML schema.

Similarity is a measure that expresses the level of correspondence. Its value is from interval $[0, 1]$, where 0 means no similarity and 1 means that the compared items are equal in the selected measured aspects. A *Matcher* is an algorithm that evaluates similarity of schemas according to particular criteria.

8.3.1 Applications of Schema Matching

Schema matching is extensively researched and has a lot of applications [113] in various sectors like *data integration*, *e-business*, *biochemistry and bioinformatics*, *ontology matching* or *data warehousing.*

Data Integration In this area the task is to create a single *mediated schema* from a set of independently designed schemas that allows a uniform access to it. The independently developed schemas have often different structure and terminology, but they describe the same real-world model. Schema matching is the first step in the data integration process. It is used, e.g., in [124]. Conceptual modeling enables a slight variation of schema integration. Independently developed schemas are integrated using a given conceptual schema.

E-business In business transactions messages with different format are often exchanged and they have to be transformed – we need a conversion between different names, data types, ranges of values and structure. Schema matching is used for integration of different representations of the same concept developed by different parties involved in the business transactions. Examples of this application are [59, 63].

XML schemas are used in business transactions among enterprises that exchange business documents with their partners. Many enterprises and organizations have defined their own XML schemas to describe the structure and content of their business documents (i.e., XML instances) to be used in the transactions. Some organizations have also published standard XML schemas to be shared in the transactions within specific industry domains (e.g., e-manufacturing, e-government, or e-health industries). The popularity of XML leads to an integration problem. Different enterprises or organizations often choose different XML representations for the same or similar concepts. One of the most critical steps to achieving the seamless exchange of information between heterogeneous e-business systems is schema matching.

Biochemistry and Bio-informatics Data management in bio-informatics and biochemistry is used for genome research, network analysis of molecular interactions, interaction maps of proteins. Information systems contain usually very large data sets. The volume of data grows exponentially as new types of data emerge. In addition, the semantics of biological data is very rich. Schema matching enables to share and reuse huge amount of data from previous experiments from heterogenous sources. It is studied, e.g., in [32, 117, 69].

Ontology Matching An *ontology* is a representation of knowledge about a certain domain. It uses *concepts*, *attributes* and *relations* to express this knowledge. The concepts are entities and relations express relationships among them. Ontologies are organized into a taxonomy tree and can be specified by languages such as OIL [37], RDF, OWL [126] or SHOE [53]. Ontology matching is the problem of finding semantic mapping between elements of ontologies. Ontology matching is explored, e.g., in GLUE [33], GOMMA [46] and LogMap [67].

Data Warehousing Data warehousing is mainly used for reporting and data analysis. Data is extracted from a set of data sources and has to be transformed into the warehouse format. Schema matching is used to find semantic correspondences between elements of source and warehouse schemas.

8.4 Proposed Solution

First, we will briefly describe the algorithm for construction of decision tree proposed in previous work [116] which was used for PSM-to-PIM mapping in our preliminary implementation called *eXolutio* [71] (as described later in Section 8.5). Then we will follow with the description of the *C5.0* algorithm [111] that we utilized for better training of the decision tree. As we will show, it solves several problems of the original algorithm.

8.4.1 Original Decision Tree Construction

Definition 24. *(Decision tree). A decision tree is a graph $G = (V, E)$, where V_i is a set of internal nodes – independent match algorithms. V_l is a set of leaf nodes – output decision whether elements do or do not match. $V = V_i \cup V_l$ is a set of all nodes. E is a set of edges – conditions that decide to which child node the computation will continue.*

Example 10. *An example of a decision tree is depicted in Figure 8.1. It has the following sets of nodes: $V_i = \{$Leaf Comparator, 3-grams, Jaccard$\}$ and $V_l = \{$mismatch, match, match, mismatch$\}$.*

During the traversal of this tree we are for example at the root node where Leaf Comparator matcher can return the following similarity values:

- 0*: the computation will continue to its left child that is leaf and mapping pair is identified as mismatch,*
- 1*: the computation will continue to its right child that is internal node and the traversal of tree will continue.*

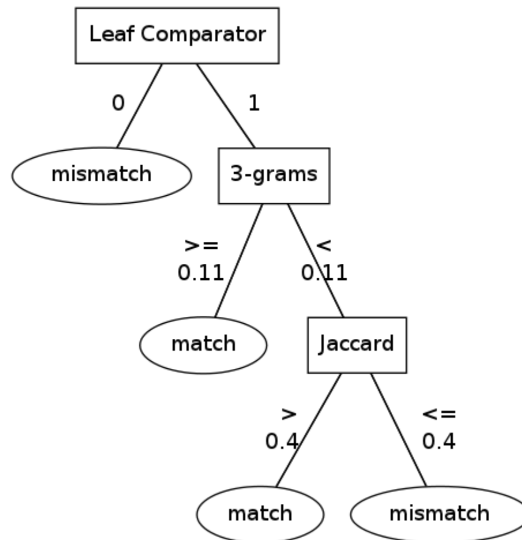

Figure 8.1: Sample decision tree

The decision tree in [116] is constructed as follows: The matchers are split into three groups (called *feature groups*) according to the main feature that they compare: *class name* (if the matcher compares names of the model classes), *data type* (if the matcher compares similarity of data types of the given elements) and *structural similarity* (if the similarity is measured by the analysis of the models structures – relations of the nodes). In each feature group the matchers are assigned with a priority according to their efficiency. Then the matchers are sorted in ascending order according to importance of the group (where for example in our case the class name group is the most important one) and their priority within the group. Finally, the decision tree is built as follows: The first matcher is selected as the root of the tree and other matchers are taken in sequence and added to the tree. If we want to add matcher M to the actual node n (i.e., use function $addMatcherToTree(M, n)$), there are the following possible situations:

- If node n has no child, method M is added as a child of n.
- If node n has children $c_1, ..., c_n$ from the same feature group that M belongs to and it has the same priority, then matcher M is added as the next child of node n.
- If node n has children $c_1, ..., c_n$ from the different feature group that M belongs to or it has a different priority, then for each node $c_i; i \in (1, n)$ we call $addMatcherToTree(M, c_i)$.

Though we have used this algorithm as the preliminary approach in our implementation, it has several drawbacks. First, it does not propose a method for automatic determination of conditions on edges and thresholds for continuous matchers. They have to be either set by the user or the default values are used. Furthermore, the decision tree does not suggest mapping results automatically. It computes an aggregated similarity score. During the traversal of the decision tree for each of the feature groups the maximum similarity value returned by the matcher from this group is stored. Then the aggregated similarity score is computed as an average of the maximal similarity value for each of the feature

groups. For each PSM element it returns possible match candidates – PIM elements evaluated by the aggregate similarity score sorted in the descending order. This helps to find matches, but it is not done automatically – the user has to evaluate each mapping.

Thus we decided to generate the decision tree using machine learning techniques. This approach solves the above mentioned problems, and thus enables to use the advantages of the decision tree approach and minimize the previous disadvantages.

8.4.2 Decision Tree Training via C5.0

Currently, there are several algorithms for the induction of a decision tree from training data, such as *ID3* [109], *CLS* [60], *CART* [13], *C4.5* [110], or *SLIQ* [85], to name just a few. The C5.0 algorithm is exploited and utilized in this thesis because this algorithm and its predecessors are widely used and implemented in various tools such as, e.g., Weka [56]. First, we introduce a notation that is used in the rest of the section:

S	– a set of training samples.
$S(v, M)$	– a set of examples from S that have value v for matcher M.
$S((i_1, i_2), M)$	– a set of examples from S that have value from interval (i_1, i_2) for matcher M.
$S = \{C_1, C_2\}$	– the decision tree algorithm classifies S into two subsets with possible outcomes $C_1 = match$ and $C_2 = mismatch$.
$Info(S)$	– entropy of the set S.
$freq(C_i, S)$	– the number of examples in S that belong to class C_i.
$\lvert S\rvert$	– the number of samples in the set S.
$Gain(M, S)$	– the value of information gain for matcher M and set of samples S.
$Info_M(S)$	– entropy for matcher M.

The entropy of the set of training samples S is computed as follows:

$$Info(S) = -\sum_{i=1}^{2} \left(\frac{freq(C_i, S)}{|S|} \log_2 \left(\frac{freq(C_i, S)}{|S|} \right) \right) \tag{8.2}$$

Set S has to be partitioned in accordance with the outcome of matcher M. There are two possibilities:

1. Matcher M has n discrete values. In that case the entropy for matcher M and set S is computed as follows (using the above defined notation):

$$Info_M(S) = \sum_{i=1}^{n} \left(\frac{|S(i, M)|}{|S|} Info(S(i, M)) \right) \tag{8.3}$$

2. Matcher M has values from continuous interval $[a, b]$, that is why threshold $t \in [a, b]$ that brings the most information gain has to be selected by Algorithm 20. Entropy for matcher M and set S is then computed according to the following formulae:

$$
\begin{aligned}
Info_M(S) = &\left(\frac{|S([a,t],M)|}{|S|} Info([a,t],M)\right) \\
&+ \left(\frac{|S((t,b],M)|}{|S|} Info((t,b],M)\right)
\end{aligned} \quad (8.4)
$$

The gain value for a set of samples S and matcher M is computed as follows:

$$
Gain(M,S) \quad = \quad Info(S) - Info_M(S) \quad (8.5)
$$

Then the decision tree is constructed by Algorithm 21 (which, as we have mentioned, differs from the already described algorithm described in Section 8.4.1).

Algorithm 20 Selection of threshold for continuous values $v_1, ..., v_n$ for matcher M and set of samples S

Require: list of values $(v_1, ..., v_n)$, matcher M, set of samples S
1: $(u_1, ..., u_m) \leftarrow SortAscDistinct(v_1, ..., v_n)$
2: **for** $i \leftarrow 1, m-1$ **do**
3: $\quad A[i] \leftarrow AvgU[i], U[i+1]$
4: $\quad L[i] \leftarrow U[i]$
5: $\quad H[i] \leftarrow U[i+1]$
6: **end for**
7: **for** $i \leftarrow 1, m-1$ **do**
8: $\quad gain_i \leftarrow GainM, S([u_1, A[I]], (A[I], u_m], M)$
9: **end for**
10: $maxGain \leftarrow \max_{i=1}^{m-1} gain_i$
11: $max \leftarrow i | gain_i = maxGain$
12: $t \leftarrow L[max]$
13: $threshCost \leftarrow$ cost of splitting interval into two subintervals $[u_1, t]$ and
14: $(t, u_m]$
15: $result.gain \leftarrow maxGain - threshCost$
16: $result.threshold \leftarrow t$
17: **return** $result$

There are the following possibilities for the content of the set of training samples S in the given node *parent* of the decision tree:

1. If S is empty, then the decision tree is a leaf identifying class C_i – the most frequent class at the parent of the given node *parent*. This leaf is added as a child to node *parent*.

2. If S contains only examples from one class C_i, then the decision tree is a leaf identifying class C_i. This leaf is added as a child to node *parent*.

3. If S contains examples from different classes, then S has to be divided into subsets. Matcher M with the highest value of information gain is selected. There are two possibilities:

(a) If matcher M has n discrete mutually exclusive values $v_1, ..., v_n$, then set S is partitioned into subsets S_i where S_i contains samples with value v_i for matcher M.

(b) If matcher M has values $(v_1, ..., v_n)$ from continuous interval $[a, b]$, then threshold $t \in [a, b]$ has to be determined. Subsets S_1, S_2 contain samples with values from interval $[a, t]$, $[t, b]$, respectively, for matcher M.

Matcher M is added as a child to node *parent*. For all the subsets S_i subtrees are constructed and added to node M as children.

Algorithm 21 Construction of a decision tree T from a set S of user-evaluated training samples

Require: set of samples S, *parent*, *condition*
1: T empty tree
2: **if** S is empty **then**
3: $c \leftarrow$ the most frequent class at the parent of the given node *parent*
4: *AddLeafparent*, c, *condition*
5: **else if** S contains only results from one class C_i **then**
6: $c \leftarrow C_i$
7: *AddLeafparent*, c, *condition*
8: **else**
9: $M \leftarrow$ matcher with the highest value of information gain $Gain(M, S)$
10: *AddNodeparent*, M, *condition*
11: **if** M has n discrete mutually exclusive values $v_1, ..., v_n$ **then**
12: $S' \leftarrow \{S_1, ..., S_n\} | S_i = S(v_i, M)$
13: $c_i \leftarrow v_i$
14: **else if** M has values $(v_1, ..., v_n)$ from continuous interval $[a, b]$ **then**
15: $t \leftarrow ComputeThreshold(v_1, ..., v_n), M, S$
16: $S_1 \leftarrow S([a, t], M)$
17: $c_1 \leftarrow [a, t]$
18: $S_2 \leftarrow S((t, b], M)$
19: $c_2 \leftarrow (t, b]$
20: $S' \leftarrow \{S_1, S_2\}$
21: **end if**
22: **for all** $S_i \in S'$ **do**
23: $T_i \leftarrow BuildTreeS_i$, M, c_i
24: **end for**
25: **end if**
26: **return** T

The threshold for matcher M with values $(v_1, ..., v_n)$ from continuous interval $[a, b]$ is selected as follows:

- Values are sorted in the ascending order, duplicates are removed. Let us denote them $u_1, ..., u_m$.
- All possible thresholds $A_i \in [u_i, u_{i+1}]$ have to be explored.

- For each interval $[u_i, u_{i+1}]$ the midpoint A_i is chosen as a split to two subsets $[u_1, A_i]$ and $(A_i, u_m]$.

- For each midpoint the information gain is computed and the midpoint A_{max} with the highest value of information gain is selected.

- The threshold is then returned as a lower bound of interval $[u_{max}, u_{max+1}]$.

Example 11. *For simplicity only three matchers are used:* `Matched Thesauri` *(which uses previous confirmed matching results for the evaluation),* `Levenshtein Distance` *(which computes the shortest edit distance from one string to another for operations insert, update and delete of a character) and* `N-gram` *(which computes the number of the same N-grams in two string where an N-gram is a sequence of N characters in a given string). The C5.0 algorithm works in the following steps:*

- *In the beginning, the training set S contains 14 samples (see Table 8.2).* `Matched Thesauri` *has discrete values* 0 *and* 1*.* `Levenshtein Distance` *and* `N-gram` *have values from continuous interval* $[0, 1]$*. The gain values are computed for all matchers.* `Matched Thesauri` *has the highest gain value of* 0.371*, that is why* `Matched Thesauri` *is selected as the root of the constructed decision tree. Set S is divided into two parts* $S(0,$ `Matched Thesauri`$)$ *and* $S(1,$ `Matched Thesauri`$)$*.*

- *Set* $S_{MT1} = S(1,$ `Matched Thesauri`$)$ *(see Table 8.3) contains samples that have value* 1 *for matcher* `Matched Thesauri` *and it contains only samples from the same match class. New leaf* ***match*** *is added as a child to node* `Matched Thesauri`*.*

- *Set* $S_{MT0} = S(0,$ `Matched Thesauri`$)$ *(Table 8.4) consists of samples with value* 0 *for matcher* `Matched Thesauri` *and results from various classes, so this set has to be further divided. Gain values are computed and matcher with the highest gain value, i.e.,* `N-gram`*, is added as a child of node* `Matched Thesauri`*. The threshold value for* `N-gram` *matcher with continuous range is* 0.071 *and set* S_{MT0} *is divided into two subsets* $S_{N1} = S([0, 0.071],$ `N-gram`$)$ *(see Table 8.6) and* $S_{N2} = S((0.071, 1],$ `N-gram`$)$ *(see Table 8.5).*

- *There are only mismatch results in set* S_{N1}*, so leaf* ***mismatch*** *is added as a child to node* `N-gram`*.*

- *Set* S_{N2} *also contains results from one class – match. Another leaf* ***match*** *is added to node* `N-gram`*.*

<table>
<tr><th colspan="3">Matched Thesauri</th><th colspan="3">Levenshtein Distance</th><th colspan="3">N-gram</th><th rowspan="2"></th></tr>
<tr><th colspan="3">Discrete</th><th colspan="3">Continuous</th><th colspan="3">Continuous</th></tr>
<tr><td colspan="3">1</td><td colspan="3">0</td><td colspan="3">0</td><td>match</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.167</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.1</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">1</td><td colspan="3">0.2</td><td colspan="3">0.063</td><td>match</td></tr>
<tr><td colspan="3">1</td><td colspan="3">0</td><td colspan="3">0</td><td>match</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.1</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.5</td><td colspan="3">0.25</td><td>match</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.429</td><td colspan="3">0.25</td><td>match</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.125</td><td colspan="3">0.071</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.6</td><td colspan="3">0.385</td><td>match</td></tr>
<tr><td colspan="3">1</td><td colspan="3">0.2</td><td colspan="3">0.067</td><td>match</td></tr>
<tr><td></td><td>match</td><td>misc</td><td>0.183</td><td>match</td><td>misc</td><td>0.032</td><td>match</td><td>misc</td><td rowspan="4"></td></tr>
<tr><td>0</td><td>3</td><td>7</td><td>≤</td><td>2</td><td>7</td><td>≤</td><td>2</td><td>6</td></tr>
<tr><td>1</td><td>4</td><td>0</td><td>></td><td>5</td><td>0</td><td>></td><td>5</td><td>1</td></tr>
<tr><td colspan="3">0.371</td><td colspan="3">0.324</td><td colspan="3">0.115</td></tr>
</table>

Table 8.2: Base training set S of Example 11

The final trained decision tree is displayed in Figure 8.2.

Matched Thesauri	Levenshtein Distance	N-gram	
Discrete	Continuous	Continuous	
1	0	0	match
1	0.2	0.063	match
1	0	0	match
1	0.2	0.067	match

Table 8.3: Base training set S_{MT1} of Example 11 when `Matched Thesauri` $= 1$

<table>
<tr><th colspan="3">Matched Thesauri</th><th colspan="3">Levenshtein Distance</th><th colspan="3">N-gram</th><th rowspan="2"></th></tr>
<tr><th colspan="3">Discrete</th><th colspan="3">Continuous</th><th colspan="3">Continuous</th></tr>
<tr><td colspan="3">0</td><td colspan="3">0.167</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.1</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.1</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.5</td><td colspan="3">0.25</td><td>match</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0</td><td colspan="3">0</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.429</td><td colspan="3">0.25</td><td>match</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.125</td><td colspan="3">0.071</td><td>misc</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.6</td><td colspan="3">0.385</td><td>match</td></tr>
<tr><td></td><td>match</td><td>misc</td><td>0.298</td><td>match</td><td>misc</td><td>0.161</td><td>match</td><td>misc</td><td rowspan="4"></td></tr>
<tr><td>0</td><td>3</td><td>7</td><td>≤</td><td>0</td><td>7</td><td>≤</td><td>0</td><td>7</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>></td><td>3</td><td>0</td><td>></td><td>3</td><td>0</td></tr>
<tr><td colspan="3">0</td><td colspan="3">0.762</td><td colspan="3">0.781</td></tr>
</table>

Table 8.4: Base training set S_{MT0} of Example 11 when `Matched Thesauri` $= 0$

Matched Thesauri	Levenshtein Distance	N-gram	
Discrete	Continuous	Continuous	
0	0.5	0.25	match
0	0.429	0.25	match
0	0.6	0.385	match

Table 8.5: Base training set S_{N2} of Example 11 when `N-Gram` > 0.071

Matched Thesauri	Levenshtein Distance	N-gram	
Discrete	Continuous	Continuous	
0	0.167	0	misc
0	0.1	0	misc
0	0	0	misc
0	0.1	0	misc
0	0	0	misc
0	0	0	misc

Table 8.6: Base training set S_{N1} of Example 11 when `N-Gram` ≤ 0.071

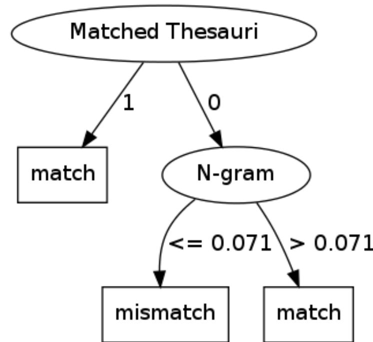

Figure 8.2: Final decision tree

8.5 Implementation and Experiments

For the purpose of evaluation of the described approach we have performed an extensive set of experiments. This section contains description of only one of the experiments and a discussion of its results. The complete set of experiments can be found in [65]. Particular experiments differ in used sets of matchers, training sets, etc. The proposed approach was implemented in the *eXolutio* tool and replaces the original approach [116] (whose disadvantages were described in Section 8.4.1). *eXolutio*, as a predecessor of *DaemonX* focussing purely on XML technologies, is based on the MDA approach too and models XML schemas at two levels – PIM and PSM. *eXolutio* allows the user to manually design a common PIM schema and multiple PSM schemas with interpretations against the PIM schema. Mapping between the two levels allows to propagate a change to all the related schemas.

All experiments were run on a standard personal computer with the following configuration: `Intel(R) Core(TM) i5-3470 3.20 GHz processor, 8 GB RAM, OS 64-bit Windows 7 Home Premium SP1.`

The following sets of XML schemas have been used for training of the decision tree:

- BMEcat is a standard for exchange of electronic product catalogues[1].
- OpenTransAll is a standard for business documents[2].
- OTA focuses on the creation of electronic message structures for communication between the various systems in the global travel industry[3].

A PIM schema used for experiments describes a common interface for planning various types of holidays. It can be found in [65].

For evaluation the following XML schemas were used:

- Artificial XML schema 01_Hotel designed for the purpose of this work. It describes basic information about hotels.
- Realistic XML schemas: 02_HotelReservation[4], 03_HotelAvailabilityRQ[5]

The domain thesaurus contains sets of words that are related semantically – for example they are synonyms or abbreviations common for the given domain. The user is enabled to expand the following one or create a completely new thesaurus. The thesauri are used during matching by Dictionary matcher. The domain thesaurus for the domain of hotels is as follows (related words are marked by ∼):

- address ∼ location,
- accommodation ∼ hotel,
- boarding ∼ meal,
- count ∼ amount,
- lengthOfStay ∼ numberOfNights.

The presented experiment results from the following observation: Efficiency of methods used to measure similarity between elements depends on the type of elements – if they are classes or if they are attributes. In this experiment two sets of decision tree are used: *two separate trees for classes and for attributes* and *one common tree for classes and attributes*.

Experiment Setup

- **Used schemas**: 01_Hotel, 02_HotelReservation, 03_HotelAvailabilityRQ
- **Decision tree**:
  - Separate decision tree for classes (in Figure 8.3) and for attributes (in Figure 8.4)
  - Common decision tree (in Figure 8.5)
- **Decision tree training set**:
  - Set of XSD Schemas: OTA

[1] www.bmecat.org
[2] www.opentrans.de
[3] www.opentravel.org
[4] http://kusakd5am.mff.cuni.cz/hb/schema/reservation
[5] http://itins4.madisoncollege.edu/IT/152121advweb/XMLExamples/unit3/schemaSimple/HotelAvailabilityRQ.xsd

- Sample count:
  * Separate decision tree for attributes: 27,942 match pairs
  * Separate decision tree for classes: 27,793 match pairs
  * Common decision tree: 55,815 match pairs

- **Thesaurus for Dictionary**: None
- **Thesaurus for Matched Thesauri**: None
- **Matchers**: `Children` (which compares the structural similarity of child nodes or neighboring nodes of classes), `DataType` (which compares data types), `Dictionary` (that looks for synonyms of the input string), `Length Ratio` (which computes the ratio of lengths of two input strings), `Levenshtein Distance`, `Matched Thesauri`, `Prefix` (which compares whether the string s_1 is a prefix of the string s_2 or the other way around)

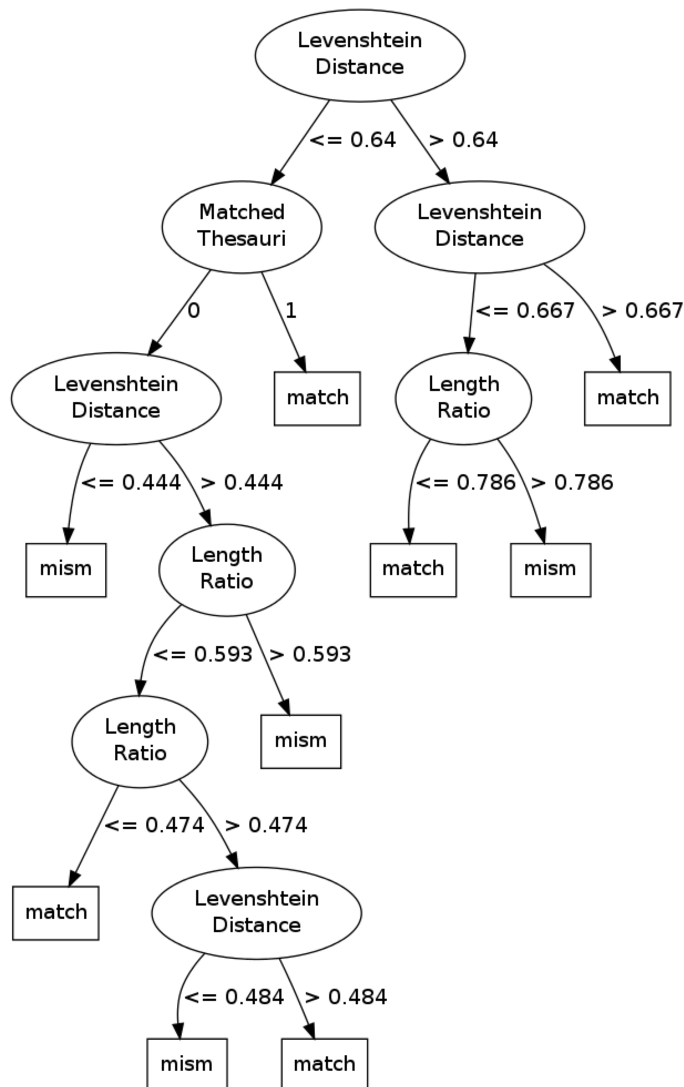

Figure 8.3: Separate decision tree for classes for experiment *SeparateTrees*

Both sets of decision trees are induced from the same set of training samples `OTA`, particularly match pairs of XML schema `OTA_HotelAvailGetRQ.xsd` and XML schema `OTA_HotelAvailGetRS.xsd`. A

separate decision tree for classes and attributes is trained only from match pairs of classes and attributes respectively. The common tree is trained from both sets together. The final decision trees are shown in Figures 8.4, 8.3 and 8.5.

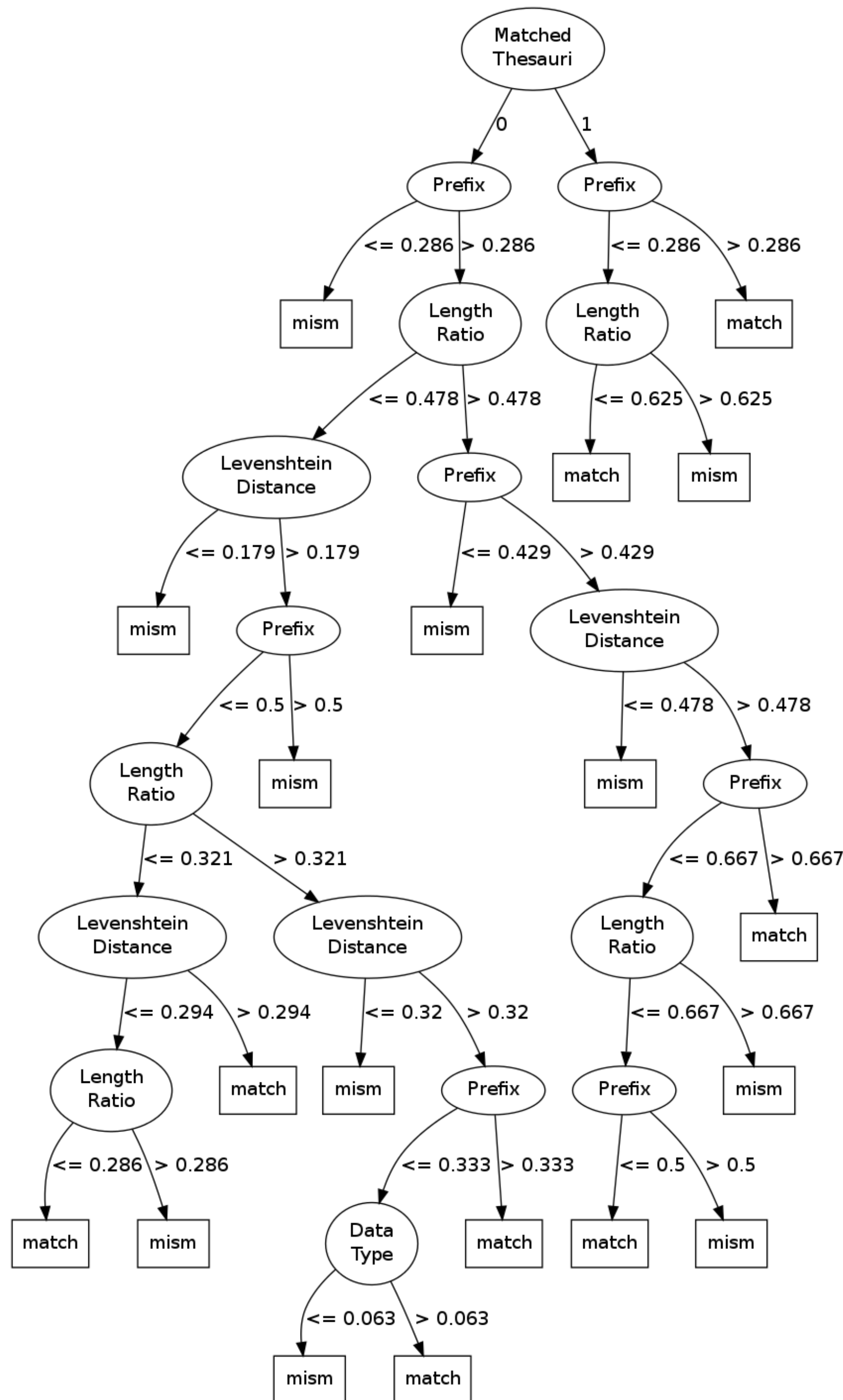

Figure 8.4: Separate decision tree for attributes for experiment *SeparateTrees*

The root of the separate decision tree for attributes is `Matched Thesauri`, all the other trees in this experiment have `Levenshtein Distance`. The matcher at the second level is the same for both branches and they have the same threshold. Especially the subtree for mapping pairs that are contained in `Matched Thesauri` is interesting. We would assume that this subtree should be smaller or even a

leaf with the value 'match'. This could be explained by errors in user annotation of mapping results – the same match pair is annotated with different matching decision than the previous one or some mapping pairs have different meaning in different context.

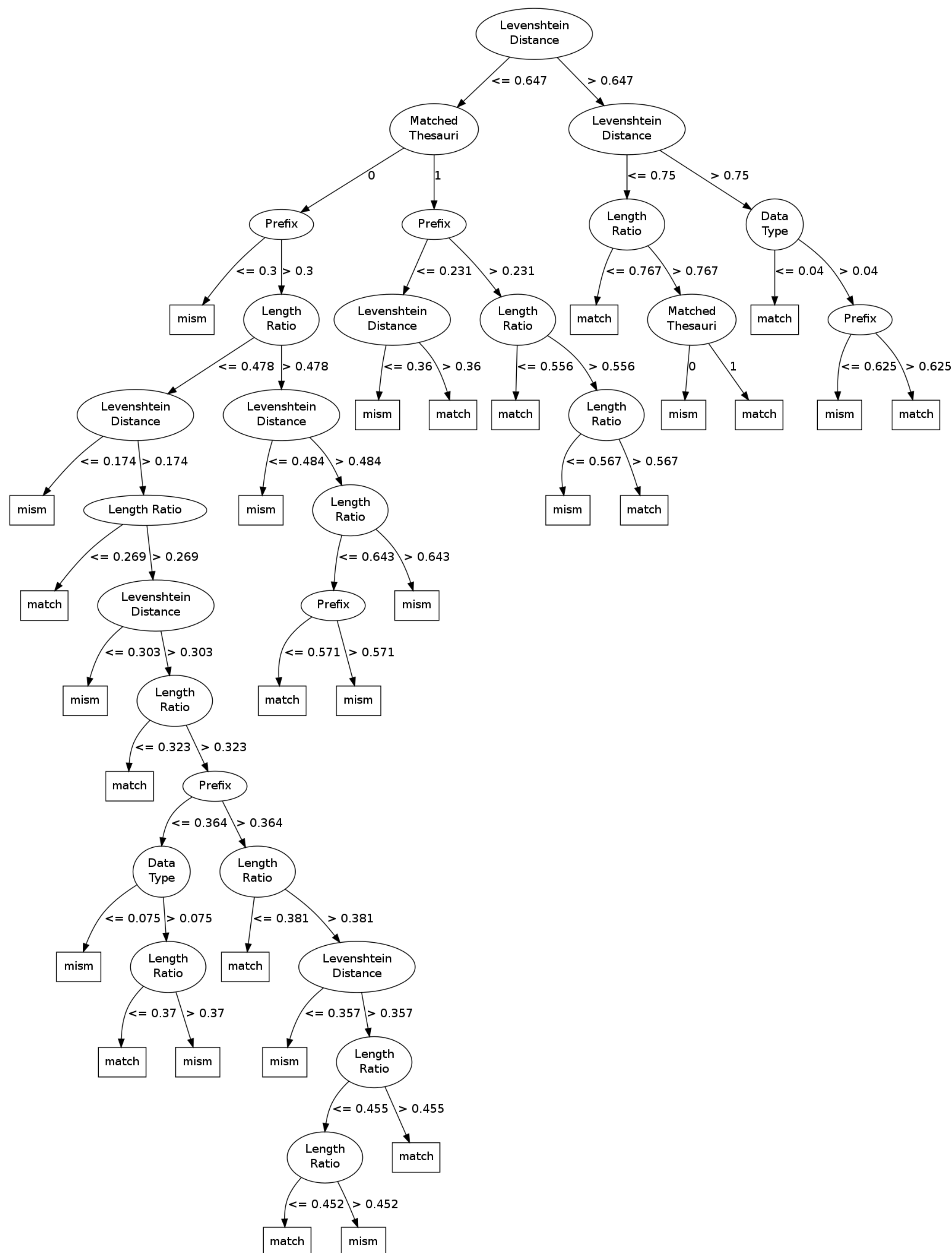

Figure 8.5: Common decision tree for experiment *SeparateTrees*

The separate decision tree for classes is relatively simple. It contains only matchers `Levenshtein Distance`, `Matched Thesauri` and `Length Ratio`, other matchers are not used. It corresponds with the original observation that some methods are more effective for certain types of elements. Matchers whose sim-

ilarity values do not distinguish mapping pairs enough are not included. Pairs that are contained in the thesaurus are directly suggested as matches.

The threshold value for `Matched Thesauri` matcher in the root of the common tree and the separate tree for classes is nearly similar. The common decision tree is the most complex one from the above mentioned. The common tree also contains two subtrees for `Matched Thesauri`. The first one is at the second level and it contains two full subtrees for both the values.

The right subtree for pairs that are contained in thesauri is more complex than the tree in the separate tree for attributes. This could be caused by a larger number of training samples that allows for more detail resolution. The second one, i.e., `Matched Thesauri`, is directly a parent of the leaves.

In Figures 8.6, 8.7, 8.8 and 8.9 there are displayed the histograms of the match quality measures – Precision, Recall, F-Measure and Overall respectively. All the measures are at first computed for both types of elements together and then separately for attribute and class elements.

In Figure 8.6 Precision is high for classes in all schemas and for both types of trees. The quality of mapping decision differs significantly with the type of element, but the training set contains a similar number of match pairs for classes (27,793 match pairs) and attributes (27,942 match pairs).

The separate tree for classes did not suggest any mapping pair as a match for schema `03_HotelAvailabilityRQ`, just as the separate tree for attributes for schema `02_HotelReservation`. Attributes in schema `03_HotelAvailabilityRQ` are difficult to identify for all the decision trees. All Precision values are from the interval [0.545, 0.769] – they identified almost the same number of relevant results as irrelevant.

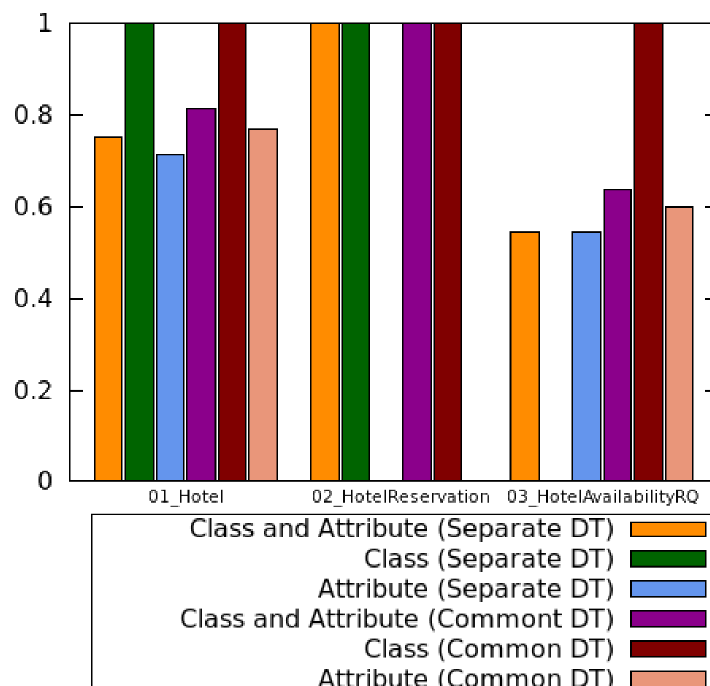

Figure 8.6: Precision for experiment *SeparateTrees*

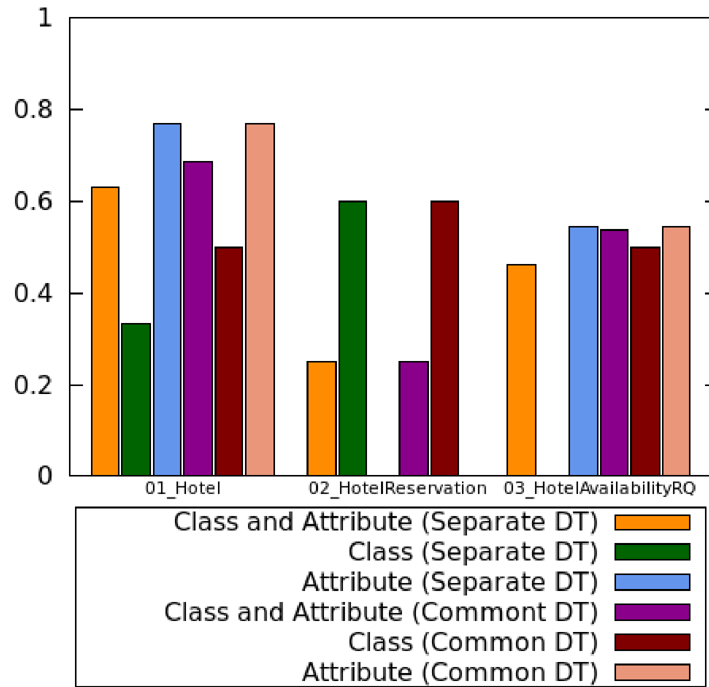

Figure 8.7: Recall for experiment *SeparateTrees*

Recall is lower than Precision in all the cases except for schema `01_Hotel` and the separate tree for attributes in Figure 8.7. There were no true positives attributes for schema `02_HotelReservation` for both trees and no true positives classes for schema `03_HotelAvailabilityRQ` for separate tree. Values of Recall are lower for attributes than Recall for classes.

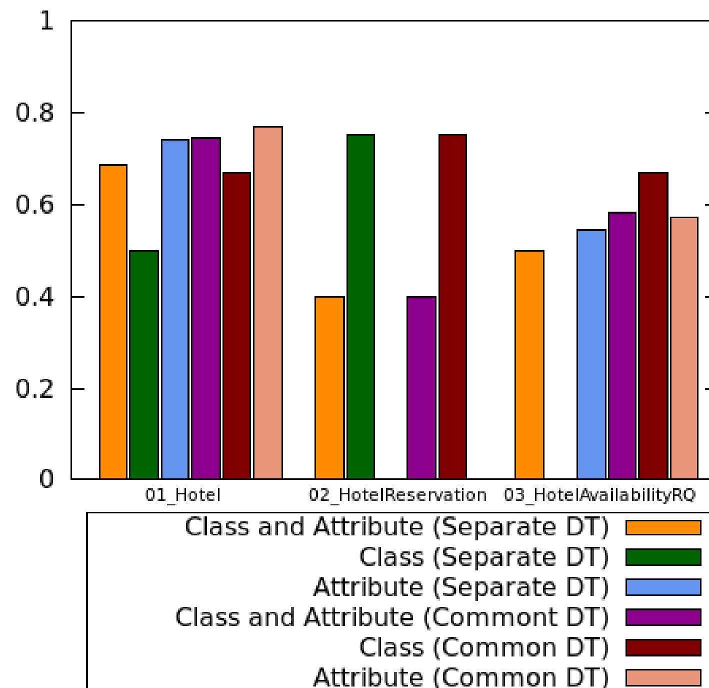

Figure 8.8: F-Measure for experiment *SeparateTrees*

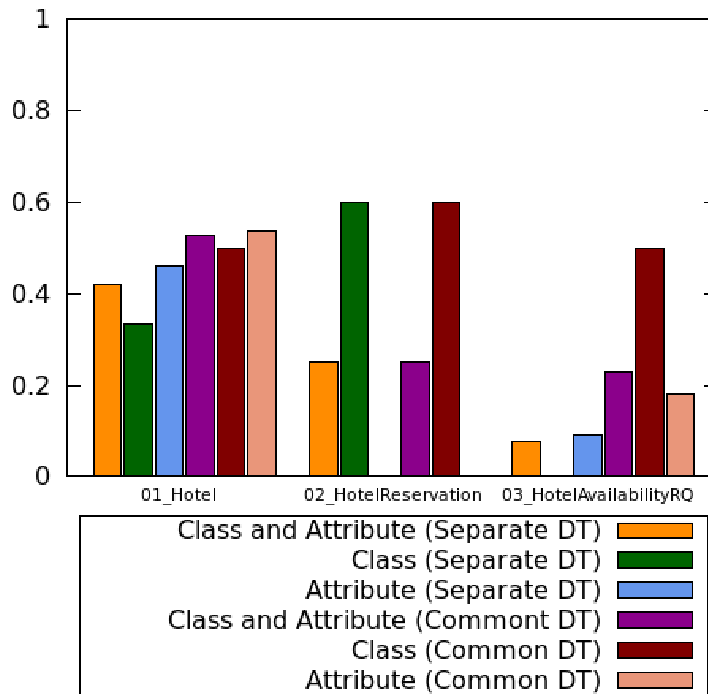

Figure 8.9: Overall for experiment *SeparateTrees*

In Figure 8.8 the values for F-Measure are equal for schema `02_HotelReservation` for classes for both trees. Post-match effort for adding false negatives (FN) and removing false positives (FP) is quite high in all cases in Figure 8.9. The highest value of Overall is 0.6.

The hypothesis was not confirmed, all similarity measures are higher for the common decision tree that is trained from a bigger set of training examples, the quality of the decision tree seems to depend more on the size of the set of training samples. The best score was achieved for Precision. Both trees in this experiment had a larger number of FN than FP. They miss a match suggestion more than they incorrectly suggest it as a match pair. It could be improved by adding an auxiliary source of information or a new matcher.

Examples of matching results from this experiment are shown in Table 8.7. Match pair `numberOfNights` – `LengthOfStay` is difficult to identify without an auxiliary source of information for both sets of decision tree. Match pairs `CheckOutDate` – `CheckOut`, `ContactInfo` – `Contact` and `BedType` – `ReservationType` were identified correctly by the common tree and incorrectly by separate decision trees.

Further experiments with various hypotheses, e.g., training with sets of different sizes, with different sets of matchers, or with usage of auxiliary information, can be found in [65].

8.6 Conclusion

Schema matching, i.e., the problem of finding correspondences, relations or mappings between elements of two schemas, has been extensively researched and has a lot of different applications. In this chapter a particular application of schema matching in MDA is explored. We have implemented our approach within a

	XSD	PIM	DT type	DT	User	Result
C	ContactInfo	Contact	Separate	Mismatch	Match	FN
C	ContactInfo	Contact	Common	Match	Match	TP
A	Fax	FaxNumber	Separate	Match	Match	TP
A	Fax	FaxNumber	Common	Match	Match	TP
A	numberOfNights	LengthOfStay	Separate	Mismatch	Match	FN
A	numberOfNights	LengthOfStay	Common	Mismatch	Match	FN
A	CheckOutDate	CheckOut	Separate	Mismatch	Match	FN
A	CheckOutDate	CheckOut	Common	Match	Match	TP
A	BedType	ReservationType	Separate	Match	Mismatch	FP
A	BedType	ReservationType	Common	Mismatch	Mismatch	TN

Table 8.7: Examples of mapping results for experiment *SeparateTrees*

modeling and evolution management tool *eXolutio* which is based on the idea of MDA. The mapping is necessary, because changes in one place can be then propagated to all the related schemas. The presented schema matching approach is used to identify mappings between PIM and PSM level of MDA, representing an interpretation of a PSM element against a PIM element.

Since there exists a huge set of existing works, we have explored various approaches to schema matching and selected the most promising possible approach for our application – schema matching using a decision tree. This solution is dynamic, versatile, highly extensible, quick and has a low level of user intervention and a low level of required auxiliary information. We have extended the previous work by utilization of the C5.0 algorithm for training of decision tree from a large set of user-annotated schema pairs. Our approach is now more versatile, extensible and reusable. Further we evaluated our approach on a wide range of experiments and implemented a module that is easily extensible. We also implemented a user-friendly interface for evaluation of mappings suggested by the decision tree, i.e., a solid background for further experiments.

8.6.1 Future Work

In this chapter we focused on schema matching issue. Even we analyzed multiple approaches, presented a complex solution based on decisions tree and evaluated the approach on multiple tests, there are still open issues that can be solved:

- *Expand set of available matchers*: There exists matchers with more powerful functions, e.g., with a matcher that uses the *WordNet*[6] thesaurus for synonyms. Further possibilities are for example string matchers[7] and the *Soundex* matcher[8]
- *User interface improvements*: We could add an interface for evaluation of matches during the preparation phase of decision tree training or dynamic editing of trained decision tree – remove, move, add matcher node, change results in leaves or threshold on edges.

[6] `http://wordnet.princeton.edu/`
[7] `http://secondstring.sourceforge.net/`
[8] `http://www.archives.gov/research/census/soundex.html`

- *Definition of general algorithms for arbitrary schemas and their incorporation into DaemonX*: This enables a possibility to generate models from existing schemas and their (semi)automatic mapping to PIM models.

9. Experiments

In the previous chapters we focused on models representing various technologies and languages for data management and data storage and on algorithms for model transformation. In this chapter we present a complex example of evolution process of an IS. We model complex situations that, starting from a single point, influence the whole system, e.g., change the database schema, change the producer resource or producer data structures, etc. Using the described situations we present and measure how each task can be accomplished by our algorithms (semi)automatically. The approach was accepted for [83].

9.1 Description of Experiments

During our research of particular models and algorithms we used various data for experiments related to the specific technologies, e.g., a complex database schema [1] or a benchmark for XPath query evaluation [42]. For these data we created sets or distinct scenarios, situations and queries of various complexity to provide correctness and ability of our algorithms.

The main aim of experiments in this chapter is to evaluate how the algorithms work on real-world data and scenarios instead of synthetic situations and to measure how they can reduce the need for manual check and update while evolving an application model. We also created a set of hypotheses that can be confirmed or rejected based on experiments results:

- Complexity of the change propagation depends on the type of change (operation).
- Complexity of the change propagation depends on the number of relations of changed models.
- An initial change causes at least one change in the related model(s) or informs the user about a situation that must be resolved.

9.1.1 Experimental Data

For the experiments we selected a real-world framework that covers our assumptions for target applications. We used various parts of the MediaWiki [84] (also known as WikiMedia Foundation) project – an open-source web framework with the best known representative Wikipedia, a world-wide popular collaborative encyclopedia. We chose this project for several reasons:

- It is a well-known project used by an immense number of users, especially in the case of the popular Wikipedia site.
- It provides source codes in GitHub and a good documentation of versions of its releases with a description of changes and migrations.
- It provides a database schema, API description and various formats, e.g., XML, JSON, etc.

- It was used in various papers and evaluations as a real-world use case, e.g., [29].

In particular, we used MediaWiki database schema[1] and XML export schema[2]. From these schemas and resources we created models for the experiments. As the source of our process we used a PIM model instead of a database model because of the MDA approach (the model itself was, however, transformed from the mentioned database schema).

A simple relation diagram of particular models and their interconnections is depicted in Figure 9.1. Each box represents a model and an arrowhead line represents the direction of the change propagation. (Although the presented diagram forms a tree, there is no restriction, e.g., for *Directed Acyclic Graphs* (DAGs) or other types of graphs). The case of a possible infinite loop or multiple updates of a single model must be handled by the particular manager of the transformation process and it depends on the initial model of the transformation.

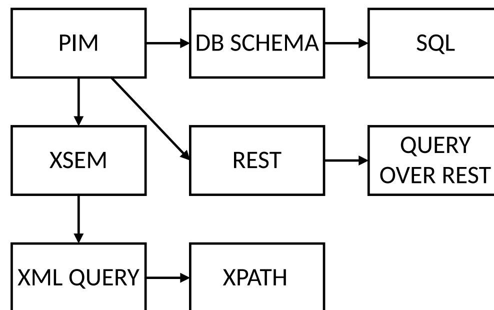

Figure 9.1: A diagram of the models

The model for the REST API was inspired by the MediaWiki API documentation[3] [4]. Since MediaWiki API uses an API based on querying via query parameters, it is not sufficient for REST and for the experiments, so we created a set of test resources based on the database schema.

An XML schema model was created from the MediaWiki description[5]. Next we created a set of XPath queries over this schema to be able to test evolution of XML query models.

We also inspected the project's version control history[6] and selected multiple database schema changes and migration scripts for our experiments. We applied changes described in migration scripts to our source PIM model and analyzed changes of related models. Finally, we compared results of the transformations with results of the migration in the MediaWiki repository.

9.1.2 Experimental Evaluation

The assessment of our experiments is a statistics of the number of manual updates compared to the number of executed operations and a comparison of transformation results and changes in the MediaWiki repository. Every change was analyzed

[1] https://www.mediawiki.org/wiki/Manual:Database_layout
[2] https://www.mediawiki.org/wiki/Help:Export
[3] https://www.mediawiki.org/api/rest_v1/
[4] https://www.mediawiki.org/wiki/API:Main_page
[5] https://www.mediawiki.org/wiki/Help:Export
[6] https://github.com/wikimedia/mediawiki

for the number of expected manual changes and subsequent updates of related parts needed for evolution. As an evaluation tool we use *DaemonX* that automatically records (and counts) all executed operations needed for the evolution management process (transformation).

9.1.3 Database Model

The schema of the database consists of 48 tables divided into 13 logically separated parts. As mentioned before, a PIM model representing this schema was created and so was the database schema and the model of SQL queries. From the PIM model we generated the XSEM model and created queries in XPath model over this model. Finally we created a model representing REST resources based on the PIM model. We have selected 30 various migration scripts of database schema for our evaluation. We present the key findings in the rest of this chapter.

9.2 Particular Experiments

Every experiment in this section contains a respective description, a database script from MediaWiki repository with a URL to particular commit in repository describing the change, and a table with changes in particular models, operations executed in models and evaluation of the number of manual changes in the models. The results of the experiments are discussed at the end of this section.

Adding PIM Class `page_restrictions` The respective initial SQL query of adding a new class `page_restrictions` is depicted in Listing 9.1. As was mentioned before, we used SQL scripts as an initial change in the source PIM diagram that triggers the evolution process.

Listing 9.1: Adding class `page_restrictions`

```
CREATE TABLE page_restrictions (
  pr_page      INTEGER       NULL
    REFERENCES page (page_id) ON DELETE CASCADE,
  pr_type      TEXT         NOT NULL,
  pr_level     TEXT         NOT NULL,
  pr_cascade   SMALLINT     NOT NULL,
  pr_user      INTEGER      NULL,
  pr_expiry    TIMESTAMPTZ  NULL)
```

Adding a new class has basically no impact on the system. A newly added class has no relations to other classes and no propagation is needed. However, automation that can be provided to the user is propagation of creation of a class to other models, e.g., the database model, the XML schema model, or the resource model and creation of relations between these newly created objects. To provide this option, the user should be notified about this possibility with respective actions. The results with the particular operation in every related model and the number of required manual updates are provided in Table 9.1. GitHub repository commits of the particular change can be found here [7] [8].

[7]https://github.com/wikimedia/mediawiki/blob/bbfad4fc598c2b46cb97566012d8534b4af05697/maintenance/postgres/archives/patch-page_restrictions.sql

[8]https://github.com/wikimedia/mediawiki/search?l=PHP&q=page_restrictions&typ

Direction	Operation	Number of operations
PIM → DB	Adding of a table	1
DB → SQL	Possible generation of a query, that returns all columns of the table e.g., `SELECT * FROM table_name`	1
PIM → XML	Adding of an element	1
XML → XQ	Generation of a query, that returns particular element from XML, e.g., `/path_to_element/element`	1
PIM → REST	Adding of a new resource	1
	Total	5

Table 9.1: Adding class `page_restrictions`

Adding PIM Property `job.job_token` Adding of a new property into a class differs from adding of a class. A new property is added into a class that can have references to other models. So all related models must be inspected and updated alternatively. But, in some situations, this propagation can be unwanted, e.g., adding of a private attribute that should not be returned in a query, such as password.

A sample SQL command which adds property `job.job_token` is depicted in Listing 9.2 (we omit other newly added columns from the migration) and the respective results are provided in Table 9.2. In this situation, all models related to class `job` must be inspected and updated. As was mentioned, adding can be unwanted in all related models and the user must decide if the respective model should be updated or not. In our situation we propagated the new property to all respective models. GitHub repository commits of the particular change can be found here [9] [10].

Listing 9.2: Adding property `job.job_token`

```
ALTER TABLE /*_*/job
  ADD COLUMN job_token varbinary(32)
   NOT NULL default '';
```

Direction	Operation	Number of operations
PIM → DB	Adding of a column	1
DB → SQL	Update of queries – notification to user	11
PIM → XML	Adding an element / attribute	1
XML → XQ	Update of queries	1
PIM → REST	Adding of resources	0
	Total	14

Table 9.2: Adding property `job.job_token`

Adding PIM Property `job.job_token_timestamp` This example presents another case of adding a property into the same class `job`. It differs from the

e=&utf8=%E2%9C%93

[9] https://github.com/wikimedia/mediawiki/blob/aeedfb8526e9d221553e430437a7572a6da2ba65/maintenance/archives/patch-job_token.sql

[10] https://github.com/wikimedia/mediawiki/blob/69ae945e8d39972a07bea89ddb64bc0189b43ac2/includes/jobqueue/JobQueueDB.php

previous example in propagation to related models – in this situation we did not propagate the new property to all of them because it was not required. The sample SQL command is depicted in Listing 9.3 and the respective results in Table 9.3. GitHub repository commits of the particular change can be found here [11] [12].

Listing 9.3: Adding property `job.job_token_timestamp`

```
ALTER TABLE /*_*/job
  ADD COLUMN job_token_timestamp varbinary(14)
    NULL default NULL;
```

Direction	Operation	Number of operations
PIM → DB	Adding of a column	1
DB → SQL	Update of queries – notification to user	7
PIM → XML	Adding of an element/attribute	1
XML → XQ	Update of queries	1
PIM → REST	Adding of a resource	0
	Total	10

Table 9.3: Adding property `job.job_token_timestamp`

Adding of a PIM Relation Between Classes – Adding a Foreign Key Constraint `ufg_user` on Column `user_former_groups.user_former_groups_fk1` A situation very similar to adding a new property is adding of a relation between two classes. Its impact on related models can be extensive, depending on the type of the relation and its cardinality. After adding of a particular relation, the user can be asked if this change should be propagated to other models. An example can be adding of relation between classes `mwuser` and `user_former_groups` with cardinality $[1, *]$ and propagation to the XML schema model. It must be decided how this relation should be represented – if as a nesting of element `mwuser` in element `user_former_groups` or if this relation should be represented using *key/keyref* relation. This decision can have a significant impact on the schema model structure and can cause multiple related updates such as update of the queries over the schema. The respective SQL command is depicted in Listing 9.4 and the results in Table 9.4. GitHub repository commits of the particular change can be found here [13] [14].

Listing 9.4: Adding of a relation between classes – adding of a foreign key constraint on column `user_former_groups.user_former_groups_fk1`

```
ALTER TABLE &mw_prefix.user_former_groups
  ADD CONSTRAINT &mw_prefix.user_former_groups_fk1
  FOREIGN KEY (ufg_user)
  REFERENCES &mw_prefix.mwuser(user_id)
```

[11] https://github.com/wikimedia/mediawiki/blob/aeedfb8526e9d221553e430437a7572a6da2ba65/maintenance/archives/patch-job_token.sql

[12] https://github.com/wikimedia/mediawiki/blob/69ae945e8d39972a07bea89ddb64bc0189b43ac2/includes/jobqueue/JobQueueDB.php

[13] https://github.com/wikimedia/mediawiki/blob/aeedfb8526e9d221553e430437a7572a6da2ba65/maintenance/oracle/archives/patch-user_former_groups.sql

[14] https://github.com/wikimedia/mediawiki/commit/0c5301a0d1dfc82088c67873553506411e2943e6

```
ON DELETE CASCADE DEFERRABLE
  INITIALLY DEFERRED;
```

Direction	Operation	Number of operations
PIM → DB	Adding of a relation	1
DB → SQL	No update needed	0
PIM → XML	Adding of a relation	1
XML → XQ	No update needed	0
PIM → REST	No update needed	0
	Total	2

Table 9.4: Adding of a relation between classes – adding of a foreign key constraint on column `user_former_groups.user_former_groups_fk1`

Removing PIM Class `user_newtalk` Removing of a class can have a significant impact on the model, especially if it is in relation with other classes of the model, e.g., in a hierarchy. On the other hand, it is common that a class is removed in situations when it is not used in other parts of the application and can be removed without any related updates. A sample command is depicted in Listing 9.5 and respective results in Table 9.5. In the presented example except for the SQL model the class had only one relation in the related models. GitHub repository commits of the particular change can be found here [15] [16].

Listing 9.5: Removing class `user_newtalk`

```
DROP TABLE /*$wgDBprefix*/user_newtalk;
```

Direction	Operation	Number of operations
PIM → DB	Removing a table	1
DB → SQL	Removing from queries	2
PIM → XML	Removing of an element	1
XML → XQ	Removing from queries	1
PIM → REST	Removing of a resource	1
	Total	6

Table 9.5: Removing of class `user_newtalk`

Removing PIM Property `user.user_options` Property removing is similar to class removing, especially when it is used as a key in other classes. In our example property `options` had a relation to a column in DB model that had three relations in the SQL model (i.e., it was used in three queries) that had to be updated respectively. In other models there was only one reference. The respective sample SQL command is depicted in Listing 9.6 and the results in Table 9.6. GitHub repository commits of the particular change can be found here

[15] https://github.com/wikimedia/mediawiki/blob/aeedfb8526e9d221553e430437a7572a6da2ba65/maintenance/archives/patch-drop-user_newtalk.sql

[16] https://github.com/wikimedia/mediawiki/search?p=3&q=user_newtalk&type=Code&utf8=%E2%9C%93

[17] [18] [19].

Listing 9.6: Removing property user.user_options

```
ALTER TABLE /*$wgDBprefix*/user
  DROP COLUMN user_options
```

Direction	Operation	Number of operations
PIM → DB	Removing of a column	1
DB → SQL	Removing from queries	3
PIM → XML	Removing of an element / attribute	1
XML → XQ	Removing from queries	1
PIM → REST	Removing of a resource	0
	Total	6

Table 9.6: Removing property user.user_options

Renaming PIM Class watchlist to oldwatchlist Renaming of a class in model definition seems like an easy task – we just replace the name. But it can have multiple consequences in other parts of the system. Every occurrence of a particular class must be checked and updated. Next, depending on the particular model, there must be a check that the new name of the class does not collide with an already existing class. A sample SQL command is depicted in Listing 9.7 and the respective results in Table 9.7. GitHub repository commits of the particular change can be found here[20].

Listing 9.7: Renaming class watchlist to oldwatchlist

```
ALTER TABLE watchlist RENAME TO oldwatchlist;
```

Direction	Operation	Number of operations
PIM → DB	Renaming of a table	1
DB → SQL	Renaming of a table in queries	2
PIM → XML	Renaming of an element	1
XML → XQ	Renaming of an element in queries	2
PIM → REST	Renaming of a class	1
	Total	0

Table 9.7: Renaming class watchlist to oldwatchlist

Renaming PIM Property revision.rev_id to revision.revision_rev_id_seq Renaming of a property is similar to class renaming. All related models of the property must be analyzed and updated respectively. Except for renaming there must be done an inspection and validation

[17] https://github.com/wikimedia/mediawiki/blob/aeedfb8526e9d221553e430437a7572a6da2ba65/maintenance/archives/patch-rc_moved.sql

[18] https://github.com/wikimedia/mediawiki/search?p=7&q=recentchanges&type=Code&utf8=%E2%9C%93

[19] https://github.com/wikimedia/mediawiki/commit/eda06e8593c12b4359a46cf3b428c1a1a88e40c4

[20] https://github.com/wikimedia/mediawiki/blob/aeedfb8526e9d221553e430437a7572a6da2ba65/maintenance/archives/patch-watchlist.sql

of the affected models, because there can occur situations such as duplicate property names, name collisions in query aliases, etc. If such a situation occurs, the user must be informed and decide how to handle it. A sample SQL command is depicted in Listing 9.8 and the respective results in Table 9.8. GitHub repository commits of he particular change can be found here [21] [22].

Listing 9.8: Renaming property `revision.rev_id` to `revision.revision_rev_id_seq`

```
ALTER TABLE revision
  RENAME rev_rev_id_val TO revision_rev_id_seq;
```

Direction	Operation	Number of operations
PIM → DB	Update of a name	1
DB → SQL	Update of queries	2
PIM → XML	Renaming of an element	1
XML → XQ	Update of queries	2
PIM → REST	User notification – adding of a resource	0
	Total	5

Table 9.8: Renaming property `revision.rev_id` to `revision.revision_rev_id_seq`

9.2.1 Experimental Results

From the results of the described experiments we can conclude with the following observations:

- *Complexity of Propagation and Changes*: Complexity of propagation strongly depends on the particular changes. Operations such as adding a class or a property do not require a complex analysis or related models – the newly added item is not used and based on user decision it can be propagated to other selected models as a new item without any other relations. On the other hand, updates such as renaming, removing, or adding a relation can have an impact on the whole system and in some situations the propagation cannot be processed automatically, because there are multiple possibilities or handling conflicts. Thus the user has to decide what should be the result of change propagation.
- *Results of the Propagation*: The result of the propagation depends on the defined algorithms. In most cases the algorithms provide the change directly, i.e., by predefined rules. But in some cases the result can depend on multiple events such as, e.g., whether the newly added property should be propagated as an XML attribute or as an XML element. This situation must be explicitly determined by the user. Additionally, there can be defined so-called *policies* for particular situations to reduce the number of explicit user decisions.

[21] https://github.com/wikimedia/mediawiki/blob/aeedfb8526e9d221553e430437a7572a6da2ba65/maintenance/postgres/archives/patch-update_sequences.sql

[22] https://github.com/wikimedia/mediawiki/blob/0958f53373671e212f01bf3987e405c6121a2d70/includes/Revision.php

- *Algorithm Complexity*: We did not evaluate complexity of our algorithms. All algorithms work with a limited number of items in the model, the worst time and space complexity of the algorithms is however polynomial. Next, the number of related models is limited and cyclic propagation is forbidden. This implies that the complexity of the evolution process is polynomial too.
- *Analysis of the Change Impact*: Thanks to relations between the models the solution brings another benefit – a possibility to analyze the impact of a change in all related models instead of manual inspection done by user. This analysis can save a significant amount of user's time.

As mentioned before we selected 30 SQL migrations from the project repository which we applied on the source PIM model to propagate changes to all related models as is depicted in Figure 9.1. From Table 9.9 we can see that changes in the PIM model caused exactly 30 changes in the DB model because the PIM model was initially generated from the DB model. 32 updates were done in the XML schema model based on relation between PIM and XML models. Only 25 changes were propagated into the REST model. This is because the REST model did not cover the whole PIM model. Much more interesting is that 94 changes were propagated from the DB model to the SQL model. We can explain this result by the common use of SQL – one DB table and/or column can occur in multiple SQL queries so an update of a single table/column can cause multiple updates in the SQL model. The same statement holds for the XML query model – in this case 54 updates were done.

Direction	Number of operations
PIM → DB	30
DB → SQL	94
PIM → XML	32
XML → XQ	54
PIM → REST	25
Total	235

Table 9.9: Evaluation of all experiments

Totally, from initial 30 various changes in PIM model that were propagated to related models were generated 235 changes that were done automatically without manual user inspection. The only user interaction was to decide in situation of multiple possibilities resolved by algorithms.

9.3 Conclusion

The main aim of this chapter is to sum up our previous work and present the benefits it brings to various segments of software development process in comparison with other tools with similar aims. In particular, the key advantages are:

- Analysis of evolution changes and their (semi-)automatic propagation to related models based on the defined algorithms that significantly reduce manual inspection of application code.

- Analysis of changes and notification of the user in indecisive situations, such as when removal of an item in one model can cause necessary updates in related system parts.
- Logging of the model's changes for possible manual analysis and/or update.
- Generation of adaptation code/scripts, e.g., SQL queries, XPath scripts, or REST requests, for models to be updated.

Even though a manual update is possible, in more complex systems it is a highly error-prone task that can bring multiple manual changes and hidden issues that can be overseen and thus cause problems during the application run. Especially it is significant in popular micro-service systems with multiple services that communicate with each other and have to share and accept the same message structures to work properly. These services are typically written in various programming languages and technologies where it is hard to share a common model definition or a contract, e.g., in JSON schema or XML schema. Moreover, non strictly typed languages such as JavaScript can accept messages with a new structure until the changes cause a code exception, e.g., by calling an undefined property or a different data type.

We evaluated the algorithms using the well-known MediaWiki project that provides SQL schemas, SQL queries, XML schemas, and REST API as well as a rich version control history with updates and migrations that we used in our experiments. The main aim of these experiments was to verify that our algorithms work correctly in various real-world scenarios and to measure how they reduce user effort needed for manual update.

10. Conclusion

Software development and information systems become more and more complex these days. They consist of various technologies, models and layers of abstractions. Development and maintenance of such a system can be a very complex and error-prone task. A solution of this problem can be in usage of the MDA approach which enables to describe models, design an application and divide it to different layers of abstraction. Thanks to this design it is possible to use models for design, visualization and generation of system parts, interconnect particular models and handle changes in one model which can impact other related models. This enables to maintain complex systems with mutual relations and dependencies between its parts.

In this thesis we presented results of our research dealing with definition of models, operations over these models and algorithms for model transformations. Next we focused on the schema mapping problem and introduced improved algorithms. All these topics cover publications [108, 107, 20, 75, 106, 105, 66, 83].

Our goal is to analyze and define models of widely used standards and technologies used in software development these days such as, e.g., XML Schema, XPath, SQL, BPMN, REST, etc. Subsequently we define operations over these models to be able to apply changes on them. Finally, we define relations between particular models and transformation algorithms over these interconnected models. Thanks to all these features it is possible to reduce the necessity to check the whole system for possible inconsistencies caused by the changes. All changes are analyzed, and propagated (semi)automatically:

- From the changes (list of generated operations) the user can see to what extent the change influences the whole system. This can be used for appraisal of the work, business calculations, etc.
- If an operation cannot be proceeded, e.g., because of multiple possible results or transformation options, the user is notified and inquired for a decision from the offered possibilities. An example can be removing of some parts which can cause that relations between a removed item and all its related items will be lost.
- If an operation cannot be proceeded even (semi)automatically, the user is notified that a manual update must be done. Another option is that there can be predefined different options and the user just selects one of them.
- From operations proceeded during the evolution process a log can be generated. This can be used in situations when there exist other related systems not contained in the project, e.g., a third party system using our service or API. The generated log can be send to system administrators as a change log and does not have to be written manually. In *DaemonX* a log is represented by a list of executed operations (commands) which are stored in so-called command stack – because a stack is a straight-forward data structure that can be used for this purpose. If any command is executed, it is put on the top and if an action (command) should be reverted, all commands

above it must be taken from the stack (reverted) first before the required operation itself is reverted.

- Except for operations for changing models, it is possible to define operations for generation of (migration) scripts from the models. An example can be generation of XPath or SQL queries from particular models after transformation and their application, e.g., for database migration.
- Since MDA is based on models, it is natural to visualize these models. This is useful to users and especially developers to be able to design and work with models via a graphical interface to have better overview of the whole model, relations or domain. Since all presented models were experimentally implemented as an extension of the *DaemonX* framework, it is possible to use this feature.

Last but not least, we focused on the problem of similarity of XML schemas (PSMs) which can refer to the same PIM. We present extended algorithms to bring more precise analysis of similar documents. This ability is important in situations when we are not developing or designing platform specific models or schemas from scratch, but we have existing schemas that have to be integrated into our IS. The straightforward solution is to transform the input schema (expressed, e.g., in XML Schema language) into PSM and map it manually. The solution that we presented is to use algorithms that analyze similarity between PIM and PSM models that can facilitate the mapping process.

All mentioned approaches were described in the following chapters:

- In Chapter 4 we defined a novel model representing an XPath query. The approach contains definition of algorithms for model update and transformation. The main contribution is the ability to recognize and analyze changes in XML schema and to update related queries respectively thanks to the defined model representing an XPath query. If the revalidation of the query is not possible, this situation is reported to the designer.
- In Chapter 5 we describe a model of SQL that enables possibility to react on changes done in a relational schema and its propagation to SQL queries. The main contribution is the ability to model SQL queries concurrently with the respective schema, to analyze changes performed in the database model and to update the queries to preserve their compatibility and correctness. Changes in the database schema model are propagated immediately to the SQL query using mutual mapping.
- The business processes evolution management strategy described in Chapter 6 presents an approach to generation of service interfaces in business process models and to analyze the influence of user's changes on the derived XML schema. The main contribution of our approach is the ability to create a conceptual model of the exchanged data as a partial view of the whole problem domain and automatic derivation of an optimal communication XML schema based on the defined metrics. It also involves a (semi)automatic algorithm for updating already derived communication XML schema according to user-specified changes of business rules which reduces possible errors during manual schema update.

- In Chapter 7 we present an approach for maintaining REST resources during time and their continuous development during evolution of the system. We provide algorithms to propagate changes based on the MDA approach. As the source model, we used the PIM and as the target model we defined a new PSM model representing a REST service resource, called the Resource Model. The Resource Model representing the REST service can be subsequently used for generating a stub for a particular client and server implementation or a programming language. Next, the model can be combined with existing solutions like Swagger or RAML which would then provide a full MDA managed and documented evolution of the REST API.
- In Chapter 8 we deal with schema matching, i.e., the problem of finding correspondences, relations or mappings between elements of two schemas, which has been extensively researched and has a lot of different applications. We focused on particular application of schema matching in MDA. A correct mapping is critical in case of a change, because changes in one place are propagated to all the related schemas. The presented schema matching approach is used to identify mappings between PIM and XML PSM level of MDA, representing an interpretation of a PSM element against a PIM element. We have explored various approaches to schema matching and selected the most promising approach for our application – schema matching using a decision tree. This solution is dynamic, versatile, highly extensible, efficient and has a low level of user intervention and a low level of required auxiliary information.
- In Chapter 9 we experimentally evaluate the transformation algorithms presented in the previous chapters using well-known MediaWiki project that provides SQL schemas, SQL queries, XML schemas, and REST API as well as a rich version control history with updates and migrations that we used in our experiments. The purpose of these experiments is to verify that our algorithms work correctly in various real-world scenarios and to measure how they reduce user effort needed for manual update.
- Finally, all the presented models were experimentally implemented in the *DaemonX* framework to verify the algorithms. Its first release was developed as a software project. From the beginning, it was designed as a general framework that supports extensibility by user-defined model and evolution plug-ins. This core feature was used for mentioned experimental implementations that helped to test and verify the proposed algorithms. Moreover, further implemented plug-ins contain additional features, like generation of the particular queries from the models, e.g., XPath, SQL or REST endpoints (resources). Next, thanks to architecture it was possible to change the part implementing basic undo/redo functionality with more advance solution. We briefly describe the framework in Chapter 3.

To conclude, though there are still several open questions and possible extensions of the approaches, we allege that our models and algorithms can be implemented and used in practise. As a future work, there can be defined more complex algorithms for presented models, model transformations and especially for model evolution. An example is splitting of a property into multiple properties, that

is a complex operation composed from multiple atomic operations. Next, there exist other technologies that can be described as models with theirs operations and that can be incorporated into current solution, e.g., RELAX NG [21], JSON Schema [40], etc.

All the presented approaches were supported by several grants and projects, namely by the Charles University Grant Agency (SVV-2013-267312, SVV-2014-260100, SVV-2015-260222, SVV-2017-260451, GAUK-1416213) and the Czech Science Foundation (P202/10/0573).

Bibliography

[1] Adventure Works team. Adventure Works 2014, January 2016. `https://msftdbprodsamples.codeplex.com/`.

[2] Santa Agreste, Pasquale De Meo, Emilio Ferrara, and Domenico Ursino. {XML} matchers: Approaches and challenges. *Knowledge-Based Systems*, 66:190 – 209, 2014.

[3] Lina Al-Jadir and Fatmé El-Moukaddem. Once Upon a Time a DTD Evolved into Another DTD... In *Object-Oriented Information Systems*, volume 2817 of *Lecture Notes in Computer Science*, pages 3–17. Springer Berlin Heidelberg, 2003.

[4] Alsayed Algergawy, Eike Schallehn, and Gunter Saake. Improving {XML} schema matching performance using prüfer sequences. *Data & Knowledge Engineering*, 68(8):728 – 747, 2009.

[5] Amazon. Amazon Web Services. `http://amazonpayments.s3.amazonaws.com/documents/order.xsd`.

[6] David Aumueller, Hong Hai Do, Sabine Massmann, and Erhard Rahm. Schema and Ontology Matching with COMA++. *Proceeding SIGMOD '05 Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 1:906–908, 2005.

[7] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[8] Douglas K. Barry and Torsten Stanienda. Solving the Java Object Storage Problem. *IEEE Computer*, 31(11):33–40, 1998.

[9] Linda Bird, Andrew Goodchild, and Terry Halpin. Object role modelling and xml-schema. In *Proceedings of the 19th International Conference on Conceptual Modeling*, ER'00, pages 309–322, Berlin, Heidelberg, 2000. Springer-Verlag.

[10] API Blueprint. API Blueprint. `http://apiblueprint.org/`, January 2016.

[11] Scott. Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérome Siméon. *XQuery 1.0: An XML Query Language*. W3C, 2007.

[12] Béatrice Bouchou, Denio Duarte, Mírian Halfeld Ferrari Alves, Dominique Laurent, and Martin A. Musicante. Schema Evolution for XML: A Consistency-Preserving Approach. In Jiří Fiala, Václav Koubek, and Jan Kratochvíl, editors, *Mathematical Foundations of Computer Science 2004*, volume 3153 of *Lecture Notes in Computer Science*, pages 876–888. Springer Berlin Heidelberg, 2004.

[13] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984.

[14] K. Selçuk Candan, Huan Liu, and Reshma Suvarna. Resource Description Framework: Metadata and Its Applications. *SIGKDD Explor. Newsl.*, 3(1):6–19, July 2001.

[15] Loredana Caruccio, Giuseppe Polese, and Genoveffa Tortora. Synchronization of queries and views upon schema evolutions: A survey. *ACM Trans. Database Syst.*, 41(2):9:1–9:41, May 2016.

[16] Balder Cate and Maarten Marx. Axiomatizing the Logical Core of XPath 2.0. *Theor. Comp. Sys.*, 44:561–589, April 2009.

[17] Peter Pin-Shan Chen. The Entity-relationship Model – Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[18] J. Choobineh, M. V. Mannino, J. F. Nunamaker, and B. R. Konsynski. An expert database design system based on analysis of forms. *IEEE Transactions on Software Engineering*, 14(2):242–253, Feb 1988.

[19] Martin Chytil. Adaptation of Relational Database Schema. Master's thesis, Charles University in Prague, 2012. `http://www.ksi.mff.cuni.cz/~holubova/dp/Chytil.pdf`.

[20] Martin Chytil, Marek Polák, Martin Nečaský, and Irena Holubová. Evolution of a Relational Schema and Its Impact on SQL Queries. In *Intelligent Distributed Computing VII - Proceedings of the 7th International Symposium on Intelligent Distributed Computing, IDC 2013, Prague, Czech Republic, September 2013*, pages 5–15, 2013.

[21] James Clark and Murata Makoto. RELAX NG Specification. `https://www.oasis-open.org/committees/relax-ng/spec-20011203.html`, December 2001. OASIS Committee Specification and ISO/IEC 19757-2.

[22] Robbie Clutton. API versioning. `http://pivotallabs.com/api-versioning/`, May 2013. Pivotal Labs.

[23] Microsoft Corporation. ASP.NET Web API. `http://www.asp.net/web-api`, February 2015.

[24] Microsoft Corporation. Visual Studio 2015. `https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx`, August 2015.

[25] SAP Corporation. SAP Corporation.

[26] Douglas Crockford. JavaScript Object Notation. `http://www.json.org/`.

[27] Alcino Cunha and Joost Visser. Transformation of structure-shy programs with application to xpath queries and strategic functions. *Science of Computer Programming*, 76(6):516 – 539, 2011.

[28] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *The VLDB Journal*, 22(1):73–98, February 2013.

[29] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008.

[30] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 459–470, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[31] Hong Hai Do and Erhard Rahm. COMA – A system for flexible combination of schema matching approaches. *Proceedings of the 28th international conference on Very Large Data Bases*, 1:610–621, 2002.

[32] Hong Hai Do and Erhard Rahm. Flexible Integration of Molecular-biological Annotation Data: The GenMapper Approach. *9th International Conference on Extending Database Technology*, 1:811–822, 2004.

[33] AnHai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Learning to Map between Ontologies on the Semantic Web. *Proceeding WWW '02 Proceedings of the 11th international conference on World Wide Web*, 1:662–673, 2002.

[34] Eladio Domínguez, Jorge Lloret, Ángel L. Rubio, and María A. Zapata. *Evolving XML Schemas and Documents Using UML Class Diagrams*, pages 343–352. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[35] Christian Drumm, Matthias Schmitt, and Erhard Rahm. Quickmig - automatic schema matching for data migration projects. In *Proc. of the Sixteenth Conference on Information and Knowledge Management (CIKM*, 2007.

[36] Fabien Duchateau, Zohra Bellahsene, and Remi Coletta. A Flexible Approach for Planning Schema Matching Algorithms. *On the Move to Meaningful Internet Systems: OTM 2008*, 1:249–264, 2008.

[37] Dieter Fensel, Ian Horrocks, Frank Harmelen, Deborah McGuinness, and Peter Patel-Schneider. OIL: Ontology Infrastructure to Enable the Semantic Web. *IEEE Intelligent Systems*, 16:200–201, 2001.

[38] Flávio Ferreira and Hugo Pacheco. Xpto an xpath preprocessor with type-aware optimization, 2007.

[39] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000. AAI9980887.

[40] Internet Engineering Task Force. *JSON Schema: core definitions and terminology*. Internet Engineering Task Force, January 2013.

[41] The Eclipse Foundation. Eclipse. https://eclipse.org/.

[42] Massiomo Franceschet. XPath Functional Test. http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/FT.html, 2011.

[43] Pierre Genevès, Nabil Layaïda, and Vincent Quint. Identifying query incompatibilities with evolving xml schemas. *SIGPLAN Not.*, 44:221–230, August 2009.

[44] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of xml paths and types. *SIGPLAN Not.*, 42:342–351, June 2007.

[45] GitHub. GitHub. https://github.com/, December 2014. A web-based Git repository hosting service.

[46] Anika Gross, Michael Hartung, Toralf Kirsten, and Erhard Rahm. GOMMA Results for OAEI 2012. In *Proceedings of the 7th International Conference on Ontology Matching - Volume 946*, OM'12, pages 133–140, Aachen, Germany, Germany, 2012. CEUR-WS.org.

[47] Object Modeling Group. MDA Guide Version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf, June 2003. MDA Guide Version 1.0.1.

[48] Object Modeling Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. http://www.omg.org/spec/QVT/, April 2008.

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

[50] Object Modeling Group. *Unified Modeling Language (OMG UML), Superstructure, V2.4.1.* Object Modeling Group, November 2011. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF.

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

[52] Object Modeling Group. The Meta Object Facility Specification. http://www.omg.org/mof/, August 2015.

[53] Parallel Understanding Systems Group. *Simple HTML Onthology Extension.* Department of Computer Science University of Maryland at College Park.

[54] The Open Group. *SOA Reference Architecture.* The Open Group, 2011. 1-937218-01-0.

[55] Hong Hai, Do. *Schema Matching and Mapping-based Data Integration: Architecture, Approaches and Evaluation.* VDM Verlag, Saarbrücken, Germany, Germany, 2007.

[56] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[57] Terry Halpin. *ORM/NIAM Object-Role Modeling*, pages 81–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[58] Pieter H. Hartel. A Trace Semantics for Positive Core XPath. In *TIME '05*, pages 103–112, Washington, DC, USA, 2005. IEEE.

[59] Hai He, Weiyi Meng, Clement Yu, and Zonghuan Wu. Automatic integration of Web search interfaces with WISE-Integrator. *The VLDB Journal*, 13(3):256–273, 2004.

[60] Earl B. Hunt, Janet Marin, and Philip J. Stone. Experiments in Induction. *Artificial intelligence*, 1966.

[61] ISO. *ISO/IEC 9075-14:2003 Part 14: XML-Related Specifications (SQL/XML)*. ISO, 2006.

[62] ISO/IEC 9075-1:2008. *Part 1: Framework (SQL/Framework)*. Int. Organization for Standardization, 2008.

[63] Kim Jaewook, Yun Peng, Nenad Ivezic, and Junho Shin. An Optimization Approach for Semantic-based XML Schema Matching. *International Journal of Trade, Economics, and Finance*, pages 78–86, 2011.

[64] Karel Jakubec, Marek Polák, Martin Nečaský, and Irena Holubová. Undo/Redo Operations in Complex Environments. In *Proceedings of the 5th International Conference on Ambient Systems, Networks and Technologies (ANT 2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014), Hasselt, Belgium, June 2-5, 2014*, pages 561–570, 2014.

[65] Eva Jílková. Adaptive Similarity of XML Data. Master's thesis, Charles University in Prague, Prague, 2013. `http://www.ksi.mff.cuni.cz/~holubova/dp/Jilkova.pdf`.

[66] Eva Jílková, Marek Polák, and Irena Holubová. Adaptive Similarity of XML Data. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31, 2014, Proceedings*, pages 535–552, 2014.

[67] Ernesto Jiménez-Ruiz and Bernardo Cuenca Grau. *The Semantic Web – ISWC 2011: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, chapter LogMap: Logic-Based and Scalable Ontology Matching, pages 273–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[68] Mike Kelly. Hypertext Application Language. `http://stateless.co/hal_specification.html`, September 2013.

[69] Toralf Kirsten, Hong-Hai Do, Christine Körner, and Erhard Rahm. *Data Integration in the Life Sciences: Second International Workshop, DILS*

2005, San Diego, CA, USA, July 20-22, 2005. Proceedings, chapter Hybrid Integration of Molecular-Biological Annotation Data, pages 208–223. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[70] Meike Klettke. Conceptual XML Schema Evolution - the CoDEX Approach for Design and Redesign. In *BTW Workshops*, pages 53–63, 2007.

[71] Jakub Klímek, Irena Mlýnková, and Martin Nečaský. eXolutio: Tool for XML and Data Management. *In CEUR Workshop Proceedings*, pages 69–80, 2012.

[72] Jakub Klímek, Martin Nečaský, and Irena Mlýnková. Evolution and Change Management of XML Applications. *(submitted)*, 0:0–0, 2011.

[73] Konstantin Korovin. CNF and Clausal Form. In *Logic in Computer Science, lecture notes*, 2006.

[74] Vladimír Kudelas. Adapting Service Interfaces when Business Processes Evolve. Master's thesis, Charles University in Prague, Prague, 2012. `https://is.cuni.cz/webapps/zzp/detail/89552/?lang=en`.

[75] Vladimír Kudelas, Marek Polák, Martin Nečaský, and Irena Holubová. Adapting Service Interfaces when Business Processes Evolve. In *IEEE 8th International Conference on Research Challenges in Information Science, RCIS 2014, Marrakech, Morocco, May 28-30, 2014*, pages 1–12, 2014.

[76] Mong Li Lee, Liang Huai Yang, Wynne Hsu, and Xia Yang. Xclust: Clustering xml schemas for effective integration. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, CIKM '02, pages 292–299, New York, NY, USA, 2002. ACM.

[77] Bernadette Farias Lóscio and Ana Carolina Salgado. *Evolution of XML-Based Mediation Queries in a Data Integration System*, pages 402–414. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[78] Bernadette Farias Lóscio. Managing the Evolution of XML-based Mediation Queries. `http://repositorio.ufpe.br/handle/123456789/1838?show=full`, 2003. Cin – Centro de Informática, Recife Pernambuco Brazi.

[79] Ivan Luković, Sonja Ristić, Pavle Mogin, and Jelena Pavićević. Database schema integration process – a methodology and aspects of its applying. In *Sad Journal of Mathematics (Formerly Review of Research, Faculty of Science, Mathematic Series), Novi Sad, 2006, Accepted for publishing*, 2006.

[80] Ondřej Macek and Martin Nečaský. An Extension of Business Process Model for XML Schema Modeling. *6th World Congress on Services*, pages 383–390, July 2010.

[81] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[82] David Maier. *Theory of Relational Databases.* Computer Science Pr, 1983.

[83] Polák Marek and Irena Holubová. Information system evolution management – a complex evaluation. In *5th European Conference on the Engineering of Computer Based Systems.* Department of Computer Science, University of Cyprus, August 2017.

[84] MediaWiki. MediaWiki. https://www.mediawiki.org/wiki/MediaWiki.

[85] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classier for data mining. *Proceeding EDBT '96 Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, pages 18–32, 1996.

[86] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm. *Proceeding ICDE '02 Proceedings of the 18th International Conference on Data Engineering*, page 117, 2002.

[87] Marco Mesiti, Roberto Celle, MatteoA. Sorrenti, and Giovanna Guerrini. X-Evolution: A System for XML Schema Evolution and Document Adaptation. In Yannis Ioannidis, MarcH. Scholl, JoachimW. Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 1143–1146. Springer Berlin Heidelberg, 2006.

[88] Mirella M. Moro, Susan Malaika, and Lipyeow Lim. Preserving xml queries during schema evolution. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 1341–1342, New York, NY, USA, 2007. ACM.

[89] Richi Nayak and Wina Iryadi. {XML} schema clustering with semantic and hierarchical similarity measures. *Knowledge-Based Systems*, 20(4):336 – 349, 2007.

[90] Martin Nečaský. XSEM - A Conceptual Model for XML. *Proc. of the fourth Asia-Pacific conference on Conceptual Modelling*, pages 37–48, 2007.

[91] Martin Nečaský. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems.* IOS Press, Amsterdam, Netherlands, 2009.

[92] Martin Nečaský, Jakub Klímek, Jakub Malý, and Irena Mlýnková. Evolution and Change Management of XML-based Systems. *Journal of Systems and Software*, 85(3):683 – 707, 2012. Novel approaches in the design and implementation of systems/software architecture.

[93] Martin Nečaský, Irena Mlýnková, and Jakub Klímek. Model-Driven Approach to XML Schema Evolution. *OTM'11 Proceedings of the 2011th Confederated international conference on On the move to meaningful internet systems*, pages 514–523, 2011.

[94] Martin Nečaský, Irena Mlýnková, Jakub Klímek, and Jakub Malý. When Conceptual Model Meets Grammar: A Dual Approach to XML Data Modeling. *Data and Knowledge Engineering*, 72:1 – 30, 2012.

[95] Martin Nečaský and Jaroslav Pokorný. Designing Semantic Web Services using Conceptual Model. In *ACM SAC '08*, pages 2243–2247. ACM, 2008.

[96] Routledge Nicholas, Bird Linda, and Goodchild Andrew. *UML and XML schema*. Australian Computer Society, Darlinghurst, Australia, 2002.

[97] Mark Nottingham. API evolution. `https://www.mnot.net/blog/2012/12/04/api-evolution`, December 2012.

[98] OASIS. *Web Services Business Process Execution Language (WSBPEL) TC*. OASIS, 2007. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

[99] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, Konstantinos Aggistalis, Fotini Pechlivani, and Yannis Vassiliou. Language Extensions for the Automation of Database Schema Evolution. In Jose Cordeiro and Joaquim Filipe, editors, *ICEIS (1)*, pages 74–81, 2008.

[100] George Papastefanatos, Panos Vassiliadis, and Yannis Vassiliou. Adaptive Query Formulation to Handle Database Evolution. In *Proceedings of the CAiSE Forum*, 2006.

[101] Peter Piják. Universal Constraint Language. Master's thesis, Charles University in Prague, Prague, 2011. `https://is.cuni.cz/webapps/zzp/detail/96494/?lang=en`.

[102] Jaroslav Pokorný. XML in Enterprise Systems: Its Roles and Benefits. In Peter Bernus, Guy Doumeingts, and Mark Fox, editors, *Enterprise Architecture, Integration and Interoperability*, volume 326 of *IFIP Advances in Information and Communication Technology*, pages 128–139. Springer Berlin Heidelberg, 2010.

[103] Marek Polák. XML Query Adaptation. Master's thesis, Charles University in Prague, 2011. `http://www.ksi.mff.cuni.cz/~mlynkova/dp/Polak.pdf`.

[104] Marek Polák, Martin Chytil, Karel Jakubec, Vladimír Kudelas, Peter Piják, Martin Nečaský, and Irena Holubová. Data and Query Adaptation using DaemonX. *Computing and Informatics*, 34(1):99–137, 2015.

[105] Marek Polák and Irena Holubová. Advanced REST API Management and Evolution Using MDA. In *DChanges '15: Proceedings of the 3rd International Workshop on (Document) Changes*, DocEng '15, New York, NY, USA, 2015. ACM.

[106] Marek Polák and Irena Holubová. REST API Management and Evolution Using MDA. In *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering*, C3S2E '15, pages 102–109, New York, NY, USA, 2015. ACM.

[107] Marek Polák, Irena Mlýnková, and Eric Pardede. XML Query Adaptation as Schema Evolves. In Henry Linger, Julie Fisher, Andrew Barnden, Chris Barry, Michael Lang, and Christoph Schneider, editors, *Building Sustainable Information Systems*, pages 401–416. Springer US, 2013.

[108] Marek Polák, Martin Nečaský, and Irena Holubová. DaemonX: Design, Adaptation, Evolution, and Management of Native XML (and More Other) Formats. In *The 15th International Conference on Information Integration and Web-based Applications & Services, IIWAS '13, Vienna, Austria, December 2-4, 2013*, page 484, 2013.

[109] J. Ross Quinlan. Induction of Decision Trees. *Machine Learning Volume 1 Issue 1*, pages 81–106, 1986.

[110] J. Ross Quinlan. C4.5: programs for machine learning. *Morgan Kaufmann Publishers Inc. San Francisco, CA, USA*, 1993.

[111] J. Ross Quinlan. *C5.0: An Informal Tutorial.* Rulequest Research, March 2013. Academic Press, New York.

[112] Dave Raggett, Le Arnaud Hors, and Ian Jacobs. *HTML 4.01 Specification.* W3C, 1999. `http://www.w3.org/TR/html4/`.

[113] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal – The International Journal on Very Large Data Bases Volume 10 Issue 4*, pages 334–350, 2001.

[114] Armin Ronacher. Flask microframework. `http://flask.pocoo.org`, December 2014.

[115] SAP. PowerDesigner. `http://go.sap.com/product/data-mgmt/powerdesigner-data-modeling-tools.html`.

[116] Jakub Stárka. Similarity of XML Data. Master's thesis, Charles University in Prague, Prague, 2010. http://www.ksi.mff.cuni.cz/ holubova/dp/Starka.pdf.

[117] Robert Stevens, Patricia Baker, Sean Bechhofer, Gary Ng, Alex Jacoby, Norman W. Paton, Carole A. Goble, and Andy Brass. TAMBIS: Transparent Access to Multiple Bioinformatics Information Sources. *Bioinformatics*, 16(2):184–186, 2000.

[118] Sparx Systems. Enterprise Architect. `http://www.sparxsystems.com/products/ea/`, 2015.

[119] DaemonX Team. DaemonX. `http://daemonx.codeplex.com/`, 2011.

[120] Swagger Team. Swagger. `http://swagger.io/`, January 2016.

[121] XCase Team. XCase. `http://xcase.codeplex.com/`.

[122] Joe Tekli and Richard Chbeir. Minimizing user effort in xml grammar matching. *Inf. Sci.*, 210:1–40, November 2012.

[123] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, October 2004. http://www.w3.org/TR/xmlschema-1/.

[124] Andreas Thor and Erhard Rahm. CloudFuice: A flexible Cloud-based Data Integration System. *Web Engineering Lecture Notes in Computer Science Volume 6757*, pages 304–318, 2011.

[125] Twitter. Twitter API. https://dev.twitter.com/overview/api, March 2015.

[126] W3C. *OWL Web Ontology Language Overview. W3C Recommendation.* W3C, February 2004. https://www.w3.org/TR/owl-features/.

[127] W3C. *SOAP Version 1.2 Part 0: Primer*. W3C, 2007.

[128] W3C. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. W3C, 2007. http://www.w3.org/TR/wsdl20-primer/.

[129] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C, 2008. http://www.w3.org/TR/REC-xml.

[130] W3C. *SPARQL Query Language for RDF*. W3C, 2008. http://www.w3.org/TR/rdf-sparql-query/.

[131] W3C. Web Services Activity. http://www.w3.org/2002/ws/, 2009.

[132] W3C. *XML Path Language (XPath) 2.0 (Second Edition)*. W3C, 12 2010.

[133] W3C. The WebSocket API. https://www.w3.org/TR/2011/WD-websockets-20110929/, 2011.

[134] W3C. AJAX tutorial. http://www.w3schools.com/ajax/default.asp, December 2014.

[135] W3C. *XSL Transformations (XSLT) Version 3.0*. W3C, 11 2015.

[136] M. Weisfeld. *The Object-Oriented Thought Process*. Developer's Library. Addison-Wesley, 2009.

[137] Raml Workgroup. RESTful API modeling language. http://raml.org/.

[138] Shanzhen Yi, Bo Huang, and Weng Tat Chan. Xml application schema matching using similarity measure and relaxation labeling. *Inf. Sci.*, 169(1-2):27–46, January 2005.

List of Tables

3.1 Comparison of the related tools 24
3.2 Comparison of the related tools models 24

4.1 Defined changes 34
4.2 Tree type expressions 37

8.1 A comparison of the selected existing solutions 157
8.2 Base training set S of Example 11 165
8.3 Base training set S_{MT1} of Example 11 when `Matched Thesauri` $= 1$165
8.4 Base training set S_{MT0} of Example 11 when `Matched Thesauri` $= 0$165
8.5 Base training set S_{N2} of Example 11 when `N-Gram` > 0.071 166
8.6 Base training set S_{N1} of Example 11 when `N-Gram` ≤ 0.071 166
8.7 Examples of mapping results for experiment *SeparateTrees* 174

9.1 Adding class `page_restrictions` 180
9.2 Adding property `job.job_token` 180
9.3 Adding property `job.job_token_timestamp` 181
9.4 Adding of a relation between classes – adding of a foreign key constraint on column `user_former_groups.user_former_groups_fk1` 182
9.5 Removing of class `user_newtalk` 182
9.6 Removing property `user.user_options` 183
9.7 Renaming class `watchlist` to `oldwatchlist` 183
9.8 Renaming property `revision.rev_id` to `revision.revision_rev_id_seq` 184
9.9 Evaluation of all experiments 185

List of Figures

1.1 An example of simple IS . . . 6
1.2 An example of the PIM-to-PSM mapping from one PIM model to multiple PSM models . . . 8
1.3 Five-level evolution management framework . . . 10
1.4 Five-level evolution management framework with denoted chapters 16

2.1 PIM model example . . . 17
2.2 An example of relations between PIM and PSM models . . . 19
2.3 Evolution process diagram . . . 19
2.4 An example of related models . . . 20
2.5 An example of propagation from Model 1 . . . 20
2.6 An example of propagation from Model 3 . . . 20

3.1 Schema of MOF layers and a UML example . . . 26
3.2 A screenshot of *DaemonX* with PIM and XSEM PSM models . . 28
3.3 A screenshot of *DaemonX* with BPMN, UML class and relational models . . . 29
3.4 An example of evolution process (PIM and XSEM PSM model) – initial state . . . 29
3.5 An example of evolution process (PIM and XSEM PSM model) – propagation . . . 30
3.6 A screenshot of *DaemonX* – evolution manager window . . . 31
3.7 A screenshot of *DaemonX* with command stack for undo/redo management . . . 32

4.1 Location of the XPath model and change propagation in the context of the five-level evolution management framework . . . 33
4.2 Sample PSM schema . . . 42
4.3 XPath axes and nodes . . . 44
4.4 XPath expression node . . . 45
4.5 Mapping between XSEM and XPath models . . . 46
4.6 Schema example for adding . . . 48
4.7 Query example for adding . . . 48
4.8 Schema example for removing . . . 50
4.9 Query example for removing . . . 50
4.10 Schema example for renaming . . . 51
4.11 XPath example for renaming . . . 51
4.12 Reordering and following axis . . . 52
4.13 Example of reconnection problem – following axis . . . 53
4.14 Initial schema part of the Amazon example . . . 56
4.15 Evolved schema part of the Amazon example . . . 57
4.16 Original and evolved query of the Amazon example . . . 57

5.1 Location of the relational database model and related change propagation in the five-level evolution management framework . . . 59
5.2 An example of PSM database model . . . 72
5.3 An example of a simple model of the *GROUP BY* clause . . . 74

5.4 An example of a visual model of a more complex SQL query . . . 76
5.5 An example of a mapping between database model and query model 76
5.6 An example of adding new *DataSourceItem* to the *DataSource* and to the *From* components 80
5.7 Model of the complex usage of complex *GroupBy* query 82
5.8 Updated model of the complex *GroupBy* query 82

6.1 Location of the business process model and respective change propagation in the five-level evolution management framework 85
6.2 MDA and business process modeling in *DaemonX* 92
6.3 PIM and PSM example of business process modeling 94
6.4 Architecture of *DaemonX* models related to business process modeling 95
6.5 Cardinality examples 97
6.6 Examples of reducing keys in a PSM schema 99
6.7 Replacement problems for specializations (a) and cardinalities (b) 100
6.8 Examples of *replace key* operation 101
6.9 An example of a PIM-View schema for metrics evaluation example 104
6.10 Examples of a PSM schemas for metrics evaluation example . . . 105
6.11 Business process model of the experiment 108
6.12 PIM schema of the experiment 109
6.13 PIM-View schema PIMView CustomersInfo 110
6.14 PSM schema derived from PIMView CustomersInfo 111

7.1 Location of the REST model and related change management in the context of the five-level evolution management framework . . 113
7.2 An example of the Resource Model of a simple e-shop 117
7.3 An example of the PIM-Resource Model mapping 118
7.4 An example of removal of vertex C from a Resource Model 122
7.5 An example of renaming class A to B 125
7.6 An example of connection creating when classes A and B are not siblings 126
7.7 An example of connection creating when classes B and C are in a siblings relation. The algorithm first creates copies of both subtrees of vertices B and C and then adds them as children of the opposite ones. 128
7.8 An example of removing a class B 129
7.9 An example of removing a connection (association) between classes A and B 129
7.10 An example of creating function *Get* 130
7.11 An example of renaming function *Get* to *Remove* 130
7.12 An example of updating function return type *int* to *long* 131
7.13 An example of removing function *Get* from class A 131
7.14 An example of adding function parameter *surname* to function *SetName* 132
7.15 An example of renaming parameter *name* of function *SetName* to *fullName* 133
7.16 An example of removing parameter *surname* from function *SetName* 133

7.17 An example of moving attribute *assemblyYear* from class *Product* to class *ProductVersion* . . . 135
7.18 An example of nesting PIM classes *Product* and *ProductVersion* into Resource Model class *Product* . . . 138
7.19 An example of correct propagation of adding attribute *year* to class *Product* . . . 138
7.20 An example of invalid propagation of adding attribute *name* to class *Product* . . . 139
7.21 An example of the PIM model and its corresponding Resource Model used for experiments . . . 140
7.22 An example of renaming class *Label* to *DetailInformation* . . . 142
7.23 An example of propagation of class renaming from Figure 7.22 to the Resource Model . . . 142
7.24 An example of adding a connection between classes *Label* and *Milestone* . . . 143
7.25 An example of propagation of adding a connection from Figure 7.24 to the Resource Model . . . 144
7.26 An example of adding a function *SetUrl* to a PIM class *User* . . . 144
7.27 An example of propagation of adding function *SetUrl* from Figure 7.26 to the Resource Model . . . 145

8.1 Sample decision tree . . . 160
8.2 Final decision tree . . . 166
8.3 Separate decision tree for classes for experiment *SeparateTrees* . . . 168
8.4 Separate decision tree for attributes for experiment *SeparateTrees* 169
8.5 Common decision tree for experiment *SeparateTrees* . . . 170
8.6 Precision for experiment *SeparateTrees* . . . 171
8.7 Recall for experiment *SeparateTrees* . . . 172
8.8 F-Measure for experiment *SeparateTrees* . . . 172
8.9 Overall for experiment *SeparateTrees* . . . 173

9.1 A diagram of the models . . . 178