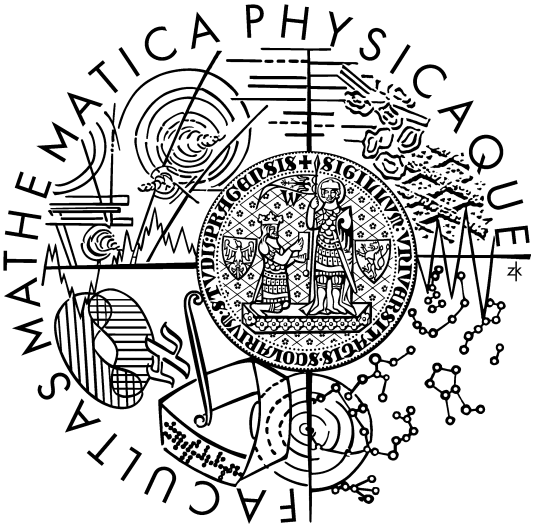
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jakub Klímek

Evoluce XML schémat

XML schema evolution

Katedra softwarového inženýrství

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

Studijní program: Informatika, softwarové systémy

2009

Na tomto místě bych rád poděkoval svému vedoucímu softwarového projektu a diplomové práce Mgr. Martinu Nečaskému, Ph.D za neocenitelnou podporu a hodnotné rady. Dále bych chtěl poděkovat svým rodičům za to, že mi umožnili plně se soustředit na studium a přípravu této práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 5.8.2009 Jakub Klímek

Contents

Název práce: *Evoluce XML schémat*
Autor: *Jakub Klímek*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *Mgr. Martin Nečaský, Ph.D.*
e-mail vedoucího: *necasky@ksi.ms.mff.cuni.cz*

Abstrakt: *V předložené práci studujeme možnosti v oblasti evoluce XML schémat. Práce obsahuje přehled existujících technik pro konceptuální modelování XML dat, sjednocování schémat a evoluci XML schémat. Je zde prezentována nová poloautomatická technika pro napojení XML schémat na konceptuální model založená na modelu XSEM pro konceptuální modelování XML dat využívajícím MDA, která velmi zjednodušuje a usnadňuje návrh a údržbu sady XML schémat v informačním systému. Dále práce obsahuje sadu evolučních operací, pomocí kterých je možné měnit konceptuální model a změny automaticky propagovat do napojených XML schémat.*

Klíčová slova: *XML, XSEM, evoluce, mapování*

Title: *XML schema evolution*
Author: *Jakub Klímek*
Department: *Department of Software Engineering*
Supervisor: *Mgr. Martin Nečaský, Ph.D.*
Supervisor's e-mail address: *necasky@ksi.ms.mff.cuni.cz*

Abstract: *In the present work we study possibilities in the area of XML schema evolution. The thesis contains a survey of conceptual modeling of XML data, schema matching and XML schema evolution. A new semi-automatic technique of connecting XML schemas to a conceptual model based on XSEM, a model for conceptual modeling of XML data based on MDA, is presented. It makes the design and maintenance of a set of XML schemas in an information system simpler and easier. In addition, the thesis contains a set of evolution operations allowing changes in the conceptual model to be automatically propagated to the connected XML schemas.*

Keywords: *XML, XSEM, evolution, mapping*

Chapter 1

Introduction

In a typical modern business, data exchange throughout the IT infrastructure is achieved by using XML [27] in some way (i.e. Web services [8], e-commerce). Also, XML is used as a data storage format in various databases [5]. To assure compatibility, one or more XML formats among communicating parties must be established. To prevent chaos, these are described by *XML schemas* written in an *XML schema language* like DTD [27], XML Schema [28] or Relax NG [7]. The advantage is that each time an XML document is processed, a check against a specified XML schema can be performed to ensure its validity.

Imagine an information system made of several components which communicate among each other using XML. Because each component of the system performs a different task, many different XML formats will be used. When the XML formats are specified with XML schemas, everything works fine until a need for change arises.

When a larger number of existing XML schemas needs to be changed in time as the business they are part of evolves, the problem with this approach becomes obvious. For example, changing a representation of a client's name from being represented by one value into being represented by first name and last name, becomes time consuming, frustrating and error-prone action. It requires a domain expert to go through every single XML schema to check whether the changed object is present in that schema and if it is, to change it. As the number of affected schemas grows, the probability of a mistake grows as well.

The process of incorporating changes into a set of XML schemas is what we call *XML schema evolution*.

1.1 Motivation

As an example of the problem of XML schema evolution, let us have a company that receives orders and let us focus on a part of the system that processes purchases as seen on Figure 1.1. Let the messages used in the process be XML messages each with a separate XML schema. The XML schemas are visualized in Figures 1.2 and 1.3[1]. The process goes as follows:

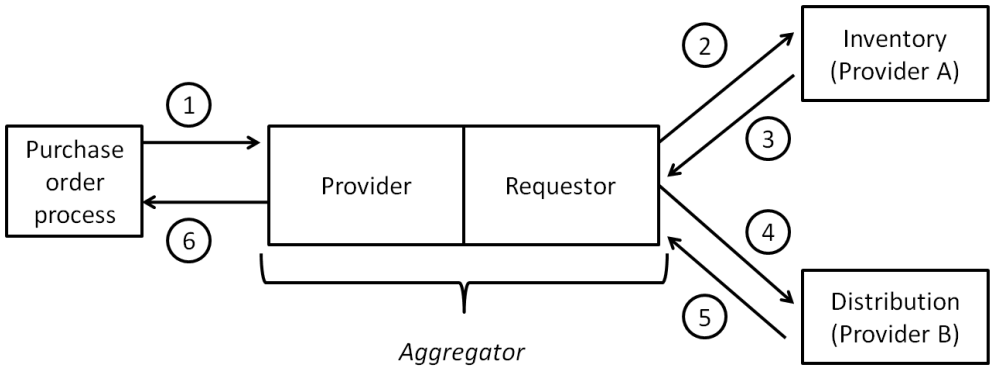


Figure 1.1: Example of a system with 6 XML schemas

1. A client process sends an order to the *aggregator*. The order contains a list of items purchased and an address to which to send the order. (Figure 1.2(a))

2. The *aggregator* sends the list of items to the *inventory*. (Figure 1.2(b))

3. The *inventory* checks if the items are available and, if they are, makes a reservation. Then it sends a response containing a reservation number to the *aggregator*. (Figure 1.2(c))

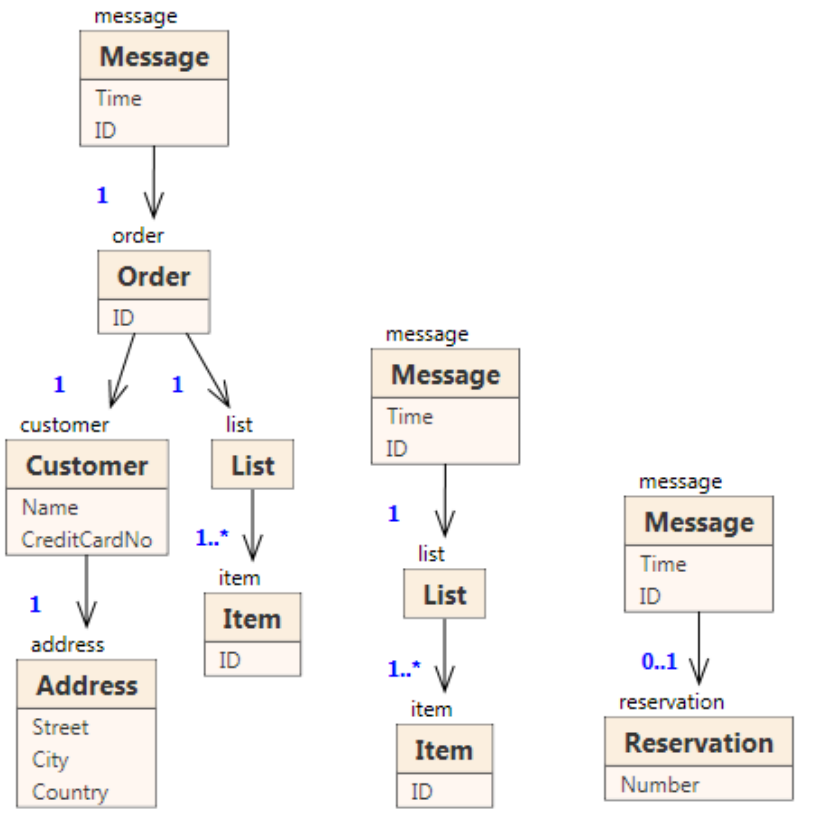
4. If the items are in stock the *aggregator* sends a message containing the reservation number and the address to the *distribution*. (Figure 1.3(a))

5. *Distribution* registers the order for expedition and sends a confirmation to the *aggregator*. (Figure 1.3(b))

6. The *aggregator* sends a confirmation or an "out-of-stock" message back to the client. (Figure 1.3(c))

This example is simplified, i.e. in reality, the elements would have more attributes, but it is enough to demonstrate the approaches presented in

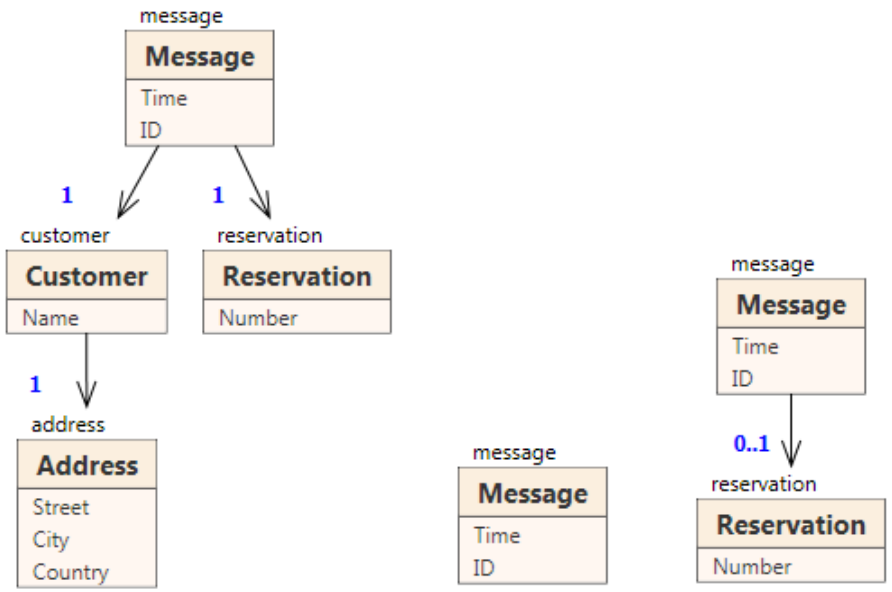
[1]The figures show graphical representations of XML schemas called PSM diagrams, explained later in 3.1.5. The corresponding XML Schemas (references in brackets) are in Appendix A.



(a) Message 1 (A.3) (b) Message 2 (A.4) (c) Message 3 (A.5)

Figure 1.2: Example schemas of messages - part I

this thesis. All the messages in the process deal with the same data, although some of them use only a subset of the data. We could say that the XML schemas specify *views* on the data. To be specific, message 1 contains a list of items and an address, message 2 contains only the list of items, but not the address, as it is not needed in the *inventory*. It provides a simplified view on the data that best suits the purpose of the message. These XML schemas need to be designed and maintained, which today is done mostly manually. For example, to change the representation of a customer's name from one value *name* to a pair of values *forename* and *surname*, a domain expert would have to identify the messages containing this value (Messages 1 and 4 in our case) and change it in each message manually. Remember that our example is only a small part of a company's infrastructure, so in real life, it could be dozens of schemas instead of two in our case and also the changes made to them can be far more complicated.



(a) Message 4 (A.6) (b) Message 5 (A.7) (c) Message 6 (A.8)

Figure 1.3: Example schemas of messages - part II

1.2 Aim of this thesis

One of the first possible steps to a solution of the problem of XML schema evolution is an introduction of a *conceptual model*, to which all the schemas would be mapped and which would connect them. The schemas from our example can be connected to an appropriate conceptual diagram that could look like the one in Figure 1.4. Then they can be evolved automatically using a single command issued at the conceptual level. The changes are then propagated to the XML schemas through these connections. This eliminates the process of identifying affected schemas, as this is done automatically using the connections to the conceptual model. The process is also much faster and more reliable, because it is no longer possible to overlook an occurrence of the changed entity or overwrite something else in the process.

The most common situation in today's businesses is as follows. Some kind of a conceptual diagram describing the problem domain (all of the company's data) is present, and a set of XML schemas describing all kinds of messages, queries and databases working with the same data is present as well. What is missing are the links connecting individual elements and types in those separate XML schemas to the elements of the conceptual model. As implied before, we will need these links to propagate changes to the schemas automatically. Because of this fact, the first step in XML schema evolution should be to connect the XML schemas to a conceptual model. The first

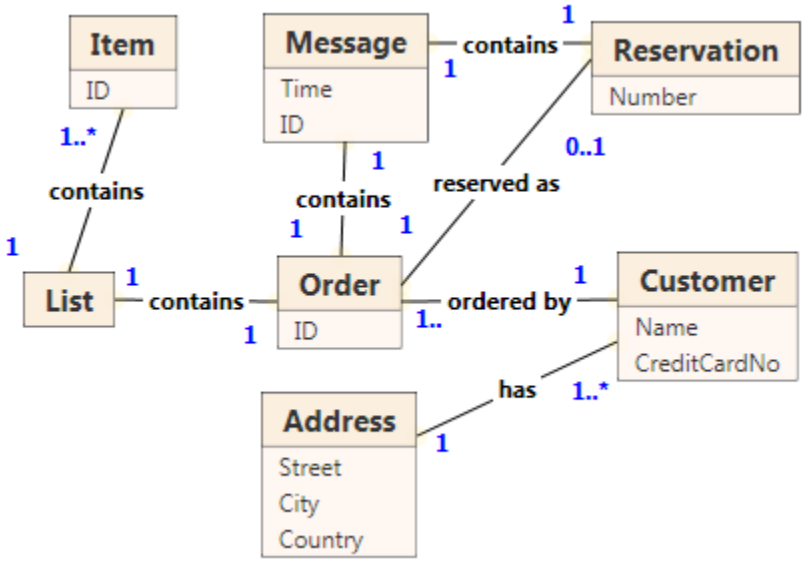


Figure 1.4: Example of a conceptual diagram

aim of this thesis is to propose a semi-automatic method of building those connections when the conceptual model and the XML schemas are present.

When our first aim is accomplished, we will be in a situation in which we have one conceptual model and a set of XML schemas connected to it. The user will be able to make changes to the conceptual model and propagate the changes to the XML schemas. Also, changes made to the XML schemas will be propagated to the conceptual model and from there back to the other affected XML schemas. Our second aim is to propose a set of operations on both the conceptual and XML schema levels to satisfy the most common needs when evolving XML schemas, including the propagation of changes between the levels.

1.3 Outline

The structure of the rest of this thesis follows the process of XML schema evolution. In Chapter 2, we provide a very brief survey of XML technologies. In Chapter 3, an introduction to a 5-level XML evolution architecture and conceptual modeling of XML data is presented, as those parts are important in the process of connecting XML schemas to a conceptual diagram. Another important part is described in Chapter 4, which contains a brief survey of some basic techniques used in schema matching, some of which are used in the approach presented in this thesis. It also contains a brief survey of related work in the area of schema matching, reverse engineering of XML schemas and XML schema evolution. Chapter 5 introduces the

technique of connecting an XML schema to a conceptual model suggested as the initial step in XML schema evolution. Chapter 6 continues in describing the process. It contains detailed description of operations that can be performed as evolution steps. Finally, Chapter 7 concludes and provides future direction of research in the area.

Chapter 2

XML Technologies

In this chapter, a brief introduction to XML and XML schemas is presented.

2.1 XML

XML (*eXtensible Markup Language*) [27] is a markup language designed to structure, transport and store data. Today, it is used almost everywhere because of its simplicity, platform independence and because it enables the user to define his own elements, making it universal and customizable. XML is a W3C recommendation[1]. An example of a XML document is in Figure 2.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
   <customer>
      <name>John Doe</name>
         <address>
            <street>1600 Pennsylvania Avenue</street>
            <city>Washington, DC</city>
            <country>USA</country>
         </address>
   </customer>
   <list>
      <item name="apple"  qty="1"/>
      <item name="orange" qty="3"/>
   </list>
</order>
```

Figure 2.1: XML document example

[1] http://www.w3c.org/XML

2.1.1 Constructs

Basic building blocks of XML documents are *elements* and *attributes* (and other secondary structures not important for this thesis). An element consists of two tags - an opening tag and a closing tag. A tag is a text enclosed in an opening and closing angle bracket. There can be other elements nested in an element, therefore an XML document has a tree structure. An attribute is a piece of information describing the element. It consists of an attribute name, an equals sign and an attribute value enclosed in single or double quotes.

2.1.2 Syntax

There are a few easy syntax rules for XML documents[2]:

All XML elements must have a closing tag.

An element consists of two tags - an opening tag and a closing tag[3]:

```
<order> ... </order>
```

There is one exception. It is possible to have an empty element. In that case, it is OK to write:

```
<order/>
```

XML tags are case sensitive

XML is case sensitive, therefore it is not possible to write:

```
<Order> ... </order>
```

XML elements must be properly nested

It is not possible to write:

```
<order><customer></order></customer>
```

The correct order is:

```
<order><customer></customer></order>
```

[2] `http://www.w3schools.com/xmL/xml_syntax.asp`

[3] The XML declaration is not part of the XML document, therefore it has no closing tag.

XML documents must have a root element

In other words, the whole document needs to be one element with other elements inside it. For example, see Figure 2.1.

2.1.3 Correctness

There are two levels of correctness for XML documents.

Well-formed

A *well-formed* XML document is a document conforming to XML syntax rules.

Valid

A *valid* XML document additionally conforms to syntactic rules defined in an XML schema written in any one of the XML schema languages like DTD [27], XML Schema [28] or Relax NG [7].

2.2 XML schemas

An XML schema is a description of a type of XML document. It contains constraints on the content and structure of an XML document that conforms to this schema. This can be useful when several parties want to communicate, because they can establish a common format to which they must adhere. Each XML document exchanged can then be checked against this common schema.

There are several XML schema languages, in which the constraints can be expressed. These include a relatively limited DTD [27] (*Document Type Definition*), which is native to the XML specification, and newer languages like XML Schema [28] and Relax NG [7].

In a typical IT infrastructure, there is usually a set of XML schemas describing all kinds of messages used in the system, as well as, for example, a database structure. Managing these sets of XML schemas is the theme of this thesis.

Examples of schemas in XML Schema can be found in Appendix A.

Chapter 3

XML Evolution Architecture

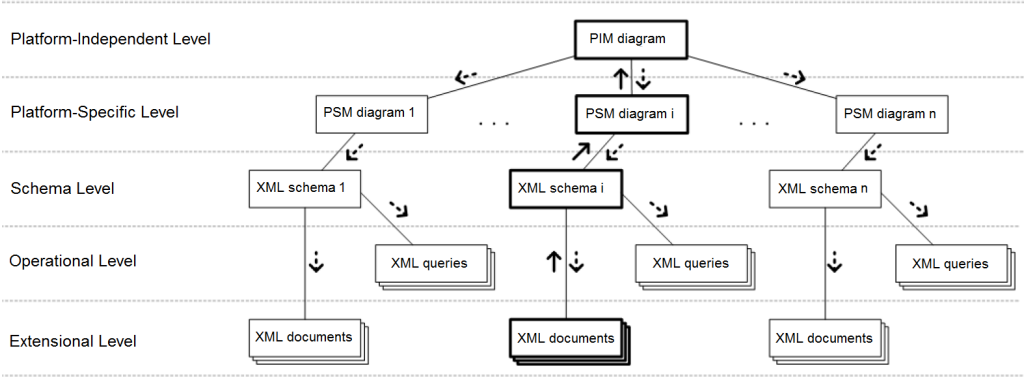


Figure 3.1: Five level XML evolution architecture

In this thesis, we view the problem of XML evolution as having five levels as it can be seen in Figure 3.1. Components from all levels are connected with components from one level above and one level below. This allows for a change anywhere in the system to be propagated through these connections to all affected places automatically. The levels in the figure can be divided into two groups. The *platform-independent* and *platform-specific* levels are called *conceptual levels*. The remaining three levels, the *schema*, *operational* and *extensional*, containing the actual files present in the system, are called *logical levels*. In this thesis, we will focus on the conceptual levels, as they are paramount for the XML schema evolution, as mentioned before. The following sections describe these levels in detail.

3.1 Conceptual levels

The two conceptual levels are platform-independent and platform-specific. They are based on MDA - *Model-Driven Architecture*.

3.1.1 Model-Driven Architecture

Model-Driven Architecture (MDA) [16] is a general approach to modeling software systems and can be profitably applied to data modeling as well. MDA distinguishes several types of models that are used for modeling at different levels of abstraction. For this thesis, two types of models are important. A *Platform-Independent Model* (PIM) allows modeling data at the conceptual level. A PIM diagram is abstracted from a representation of the data in concrete data models such as relational or XML. A *Platform-Specific Model* (PSM) models how the data is represented in a target data model (such as XML). For each target data model, we need a special PSM that is able to capture its implementation details. A PSM diagram then models a representation of the problem domain in this particular target data model, it provides a mapping between the conceptual diagram and a target data model schema.

Because we want to model XML representations of data, we need a conceptual model based on MDA, that would allow us to model data on the PIM level as well as various XML formats on the PSM level. In section 3.1.2, we will describe some conceptual models for XML which do not apply MDA sufficiently and therefore are not suitable for this thesis. In section 3.1.3, we will describe a conceptual model called XSEM [18], which is a proper model for this thesis.

3.1.2 Earlier approaches

In practice, two conceptual modeling languages are usually considered: *Entity-Relationship Model* (ER) [6] and *Unified Modeling Language* (UML) [20]. The main disadvantage of approaches in the area is that they do not apply MDA sufficiently which brings problems.

ER–based approaches

ER is used for conceptual modeling of relational databases. It contains two modeling constructs. *Entity types* are for modeling real–world concepts. *Relationship types* are for modeling associations among concepts. Both can have attributes that model characteristics of a concept or association. Approaches in this category extend ER to be suitable for conceptual modeling of XML schemas. They consider the basic ER constructs and add new ones. *EER* [2] adds constructs for modeling specifics of DTD. *XER* [24] allows modeling specifics of XML Schema. There are also approaches extending ER with constructs that do not strictly follow any target XML schema language. Examples of such approaches are *EReX* [14], *ERX* [22] or *X-Entity* [12].

The authors of these approaches do not consider MDA, but their proposed models are in fact PSMs. This has two negative impacts: (1) At the conceptual level, the designer considers how the data is represented in a XML schema instead of considering the data itself. This does not belong to the conceptual level where one should model the domain independently of target XML schemas. (2) For two different XML schemas two independent conceptual diagrams must be designed without any interrelation. When a concept is represented in both, it must be modeled twice. This makes the conceptual diagrams non–transparent and goes against the principles of conceptual modeling.

UML-based approaches

UML is a language composed of several sublanguages designed for modeling aspects of software systems. For data modeling, a part called *UML class diagrams* is applied. The basic constructs are *classes* and *associations* whose semantics is similar to ER entity and relationship types. Classes have attributes. Neither ER nor UML can be directly applied for modeling XML schemas and must be extended. There already are approaches based on UML [4][17][23] which apply MDA. As a PIM they apply the UML class diagrams. As a PSM they propose *profiles*. A profile is a set of *stereotypes* - constructs that can be applied to a construct in a PIM diagram and that specify how this PIM construct is represented in an XML schema. There are only minor differences in the profiles proposed by the approaches in this group. A typical representative of this approach is Enterprise Architect [26].

These approaches apply MDA but have significant drawbacks. They are dependent on a certain XML schema language: proposed PSMs are usually intended for XML Schema. Moreover, they consider automatic derivation of PSM diagrams from a PIM diagram. In practice, we need to specify several different XML schemas that represent our problem domain for various situations. Therefore, it would be more practical if a designer could derive more PSM diagrams from the same PIM diagram according to user requirements. This cannot be done automatically, manual participation of the designer in the process is necessary.

XML schema visualization

XML schema visualization is based on visualizing the constructs of a particular XML schema language, usually XML Schema, and does not consider MDA at all. It is widely applied in practice and implemented in commercial XML schema design tools, e.g. Altova XML Spy [1]. They do not provide any shift of XML schema languages toward conceptual modeling and do not eliminate problems with applying XML schema languages for designing

XML schemas.

3.1.3 XSEM

The approaches presented in this thesis are based on XSEM [18, p. 47-108], a conceptual model for XML. It utilizes UML class diagrams to apply MDA to model XML data on two levels: PIM and PSM. For example, a PIM can be a description of a company domain, which usually already exists. A PSM diagram is a visualization of a single XML schema describing a specific type of an XML message used in the company's IT infrastructure. While the PIM is usually only one, there can be any number of PSM diagrams representing different *views* on the same company data as used under different circumstances. An example of this can be a company receiving an order via its e-shop and the company's managers requesting a list of customers. Both of these actions are usually implemented as XML messages described by separate XML schemas, but using the same data (i.e. customer's identification).

The main feature is that all the XSEM PSM components are formally interrelated with the components of the PIM level. This allows for describing semantics of the PSM components by components from the PIM level. Using this, a software implementing XSEM can maintain connections between corresponding PIM and PSM components. These connections enable a change in a PIM component (class, association, etc.) to be propagated to all the affected PSM components (i.e. PSM Class representing XML Schema's complex type) in PSM diagrams (which directly represent XML schemas). Even more interestingly, a change in a PSM component can be propagated to the PIM level, where all the other derived[1] PSM components can be discovered and updated as well. In other words, when someone decides that a customer's name is to be represented by two values (first and last name) instead of one string, it is possible to automatically find all relevant usages of this value and change all the affected XML schemas and the conceptual diagram at the same time. This approach has already been implemented in *XCase - A Tool for XML Data Modeling*[2] described later in 3.1.6.

3.1.4 XSEM PIM components

In this section, the PIM components used by XSEM are described. XSEM uses UML class diagrams for PIM, so the basic constructs remain the same. The example of a conceptual diagram in Figure 1.4 is in fact a PIM diagram. A more detailed description follows:

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

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

PIM Classes

PIM classes are the basic constructs of the PIM. They have a name and they can have attributes and operations. As operations are not relevant to conceptual modeling of XML, we will further ignore them. Attributes are described in the next paragraph. Classes can be connected through *associations*, which are also described later in this section.

PIM Attributes

A PIM attribute belongs to a PIM class. It can have a name, data type, default value and multiplicity. The multiplicity consists of two values, lower and upper, describing the lowest and the highest number of occurrences the attribute can have within one PIM class.

PIM Associations

PIM associations connect PIM classes. They can have a name and multiple *association ends*, which means that they can connect any number of PIM classes, including 1 (self-reference). One association end is assigned to each of the connected PIM classes and has lower and upper multiplicities. Also, it has a role assigned, which is a textual description of the meaning of the PIM association in context of the PIM class, which the association end is assigned to.

3.1.5 XSEM PSM components

In this section, an overview of PSM components as defined by XSEM and implemented by XCase is given. A PSM diagram is a visual representation of an XML document structure. Because of this fact, its shape is a forest. Simply put, the user constructs the desired XML schema from the classes already present in the PIM. This process guarantees that all the PSM components have been *derived* properly from their conceptual counterparts, maintaining this connection for further use. This includes potential changes, that can be propagated to all affected components automatically. When a PSM diagram is finished, it can be exported to some XML schema language. The visualizations are from XCase, which is described later in 3.1.6. A more detailed description of the PSM components follows.

PSM Classes

Each PSM class in a PSM diagram must be derived from a PIM class. We say that the PSM class *represents* this PIM class. A PSM class models how instances of the represented PIM class are expressed in the modeled XML

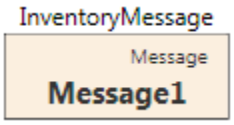


Figure 3.2: PSM class

schema. The PSM class has a name and an element label as depicted in Figure 3.2. If the PSM class name differs from the name of the represented PIM class, the PIM class name is displayed as well. In this case, a PSM class named *Message1* representing a PIM class *Message* has an element label *InventoryMessage*.

Root classes of a PSM diagram are created by deriving directly from a PIM class in the PIM. Child PSM classes C^i_{psm} (to represent PIM classes C^i_{pim}) are added under a parent PSM class C_{psm} (representing a PIM class C_{pim}). This is done by choosing a PIM paths in the PIM from C_{pim} to C^i_{pim}. The child classes are called the *content* of a PSM class. PSM classes are connected by *PSM associations*.

In XML documents, an instance of a PSM class is modeled as a sequence of elements representing its content. If the PSM class has an element label, this sequence is enclosed in an XML element with the name from the element label. Otherwise, the content and the PSM attributes are included to a parent of this PSM class, if there is any.

PSM Attributes

Each PSM class can have PSM attributes. These attributes can either be derived from attributes of the represented PIM class, or they can be *PIM-less*, indicating that they only exists in the XML schema and not on the conceptual level. Also, a PSM attribute can have an alias - a name that it should have in an XML document, which can be different from the name of the represented attribute. Visualization of this situation is in Figure 3.3. There is again the PSM class *Message1* which has two PSM attributes, *Time* and *ID*. Because the attributes have aliases *ArrivalTime* and *MessageID* respectively, they will be present under those aliases in the resulting XML schema. In XML documents, PSM attributes represent XML attributes of XML elements.

Other PSM Constructs

There are other PSM constructs - attribute containers, content containers, content choices, class unions and structural representatives, but they model XML specifics, which have no semantic equivalent on the PIM level, and

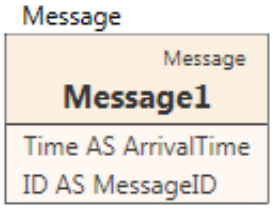


Figure 3.3: PSM attributes

therefore they are not important to this thesis. For their detailed description see [18].

3.1.6 XCase - Tool for XML Data Modeling

XCase[3] is a tool for conceptual XML data modeling implementing the described XSEM model. Since this was the first tool for conceptual modeling of XML with XSEM, its main purpose was to examine possibilities of XSEM as well as conceptual modeling for XML in general.

User work is organized into projects. Each project contains a PIM and a number of PSM diagrams. XCase serves as a full-fledged UML editor. To design PSM diagrams, UML metamodel was extended to support XSEM constructs. An automatic translation of XML schemas from their representation as PSM diagrams into XML Schema language is also part of XCase.

Features

XML schema visualization XSEM model was designed to visualize XML schemas. Working with the visual representation is easier than directly editing the XML Schema files. Moreover, being fully familiar with the schema languages is not required.

Avoiding duplications when using the same PIM concepts in different schemas All PSM diagrams in the XCase project are bound to a common PIM. The PIM components, their attributes and their relations are defined only once in the PIM. When such a component is created, it can be derived to a PSM diagram. A connection between the PIM component and its PSM representation is created.

XML schema design independent from schema language Modeling XML schemas with PSM diagrams is not bound to any specific schema

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

language. The current version of XCase allows users to translate the PSM diagrams to XML Schema, but export to other languages such as Relax NG [7] would also be possible.

Consistency checking During the design process, the connections between the PIM components and their representations in the PSM diagrams are maintained and can be used to check consistency, to locate usages of the PIM components in the PSM diagrams or to propagate changes. A user can alter both PIM and PSM diagrams at any time without worrying about a loss of consistency.

XML schema evolution All the PIM and PSM diagram visualizations in this thesis are modeled in XCase. Currently, it has no reverse engineering support and only a basic set of evolution operations is implemented.

3.2 Logical levels

Now that the conceptual levels of the XML evolution architecture have been presented, the logical levels will be described briefly. The propagation of changes from the conceptual levels to the logical levels is not discussed in this thesis.

3.2.1 Schema level

The schema level contains the actual XML schemas written in an XML schema language. Those can be automatically generated from PSM diagrams [18, p. 109-128] in the conceptual level. Specifically, they can be generated every time the conceptual model evolves.

3.2.2 Operational level

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

3.2.3 Extensional level

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

Chapter 4

Related work

Since creating a PSM diagram from an XML schema is quite straightforward [19], it can be assumed that we already have it and need to recreate the connections to the PIM model, so we can perform evolution operations (described in Chapter 6) using the XSEM approach (see Figure 3.1). The PIM and the PSM diagram can be perceived as two schemas, each of a different type, and the restoration of connections between them as a kind of a schema matching problem. There are some techniques already developed for schema matching that can be used in a slightly modified form to help with connecting PSM to PIM. In this chapter, existing approaches to schema matching and some implementations using these techniques are presented. The second part of this chapter contains a survey of existing XML schema evolution implementations.

4.1 Existing approaches to schema matching

Usually in the process of schema matching, several different techniques are combined to achieve better results. In this section there is a brief survey of these techniques, based on [25]. Only element-level and structure-level methods are listed, as only those are relevant for this thesis. For a more complete list of schema matching methods, see [10, p. 73-116].

4.1.1 Element-level techniques

The first phase of most schema matching algorithms is based on finding a similarity between elements of matched schemas while ignoring the structure of the documents. Even though this approach seems to be oversimplified, it provides a good first estimate because there is high probability, that the same concepts in two different schemas will be described in a similar manner. The resulting similarity coefficient is also a good starting point for other matching techniques described later.

String-based techniques

String-based methods work all in a similar way. From two strings on the input they produce a coefficient (usually a real number) measuring a similarity between those strings. Lets consider two XML schemas. For each element (e.g. complex type) of the first schema, a table is constructed, containing coefficients measuring the similarity between its name and the name of each complex type of the second schema. As simple as it sounds, it is not such a trivial task to choose the best fitting technique by which the coefficients are calculated. It often depends on the contents of the strings compared, or even the language used. Of course, any strings related to the elements can be used, if present (i.e. annotations, commentaries, etc.).

Prefix This technique tests whether the first string *starts* with the second one. The advantage of this method is that it is able to recognize the similarity between acronyms like *int-integer*, *str-string* etc. It can also consider the length of the prefix.

Suffix This is the same as Prefix, only testing whether the first string *ends* with the second one.

Example 4.1. *It works well for pairs like phone-telephone.*

Longest common substring Another method similar to Prefix and Suffix, but more general, is computed as the length of the longest common substring divided by the length of the longer string.

Example 4.2. *The longest common substring similarity coefficient between* DeliveryAddress *and* Address *is* $\frac{7}{15} = 0.47$.

Longest common subsequence This method is even more general than the Longest common substring. It is computed in the same way but this time we can leave out any letters we want from the strings to achieve the longest common substring.

Example 4.3. *The longest common subsequence similarity coefficient between* Msg *and* Message *is* $\frac{3}{7} = 0.43$*, because we leave out "e" and "sa" from "Message", which gives us "Msge" and "Msg" to be compared. Note that this similarity would not be detected by any other of the previous methods.*

Edit distance The edit distance method (a.k.a. *Levenshtein Distance*[1]) counts the number of insertions, deletions and substitutions required to get the second string from the first one. This number is then normalized by the length of the longer string.

Example 4.4. *The edit distance between Prg and Prague is* $\frac{3}{6} = 0.5$.

N–gram An N-gram is a sequence of N characters. This technique counts the number of common N-grams between the two strings.

Example 4.5. Trigram *for the string* color *are* col, olo *and* lor. *The coefficient between* color *and* colour *is* $\frac{2}{3} = 0.6$.

Language-based techniques

While the string-based methods are independent from the language used, the schemas are created by people and people tend to use a natural language to describe the elements of the schemas. In fact, the whole concept of XML is based on the possibility of creating custom tags to better describe the meaning of an element. With this in mind, one cannot ignore that in reality, the strings are very often words from some natural language. Given that we have the knowledge of the language used, the following techniques can greatly improve the accuracy of the resulting similarity coefficient when applied in combination with the string-based techniques.

Tokenization Names of elements are parsed into sequences of tokens by a tokenizer, which recognizes punctuation, cases, blank characters, etc.

Example 4.6. *The string* language-based techniques *is tokenized into three strings:* language, based *and* techniques.

Lemmatization Lemmatization is used in combination with the above described Tokenization. Tokens are morphologically analyzed and reduced to their basic form.

Example 4.7. techniques → technique.

Elimination An additional technique further improving the accuracy of language-based matching is called Elimination. Basically, the tokens from Tokenization are searched for articles, conjunctions, prepositions or other tokens which have no meaning in the context of matching. Those can be ignored further in the process.

[1] `http://en.wikipedia.org/wiki/Levenshtein_distance`

Constraint-based techniques

These methods are based on comparing constraints applied to the definitions of entities such as data types or cardinalities of attributes.

Data types comparison Two classes can be compared based on the data types of their attributes as these can be compared objectively. This can be done by comparing the sets of possible values or by considering an inheritance hierarchy.

Example 4.8. *The data type* day *is closer to the data type* workday *than to the data type* integer *considering both criteria: According to the sets comparison, {1, 2, 3, 4, 5} is closer to {1, 2, 3, 4, 5, 6, 7} than to the set of all integers. Using the inheritance criteria, given the inheritance hierarchy* integer → day → workday*, the* workday *data type is again closer to the* day *data type.*

Multiplicity comparison Another method that can, of course, be used together with the Data types comparison is multiplicity comparison. Many data types are in fact lists or sets on which a cardinality constraint can be applied.

Example 4.9. *A combination of Data types comparison Multiplicity comparison can be illustrated by this example: A list of names of 1 to 3 adults is closer to a list of names of 3 children than to a list of 5 to 10 company names. This is given the correct inheritance hierarchy among children, adults and people.*

Linguistic resources

Linguistic resources include common knowledge and domain-specific thesauri. Considering names of schema entities as words of a natural language, a relationship between two different words can be discovered (synonyms, hyponyms, etc.). Again, combination with some language-based techniques such as Lemmatization can help.

Common knowledge thesauri One of the used thesauri is WordNet [15]. It is a large lexical database of English, in which groups of words with similar meaning are together in a *synset*. The synsets are interconnected according to conceptual-semantic and lexical relations. There are two types of matchers exploiting common knowledge thesauri. The first ones use the semantic relations of words (synonyms, hyponyms) and the other ones use the lexical hierarchy. Specifically, they measure the number of arcs traversed between the two words in this hierarchy.

Domain-specific thesauri These thesauri contain knowledge that is not available in the common knowledge thesauri. An example of this can be proper names or technical terms. Example: A record like "NKN:Nikon = syn" can tell the matcher that an entity name *NKN* is a synonym for an entity name *Nikon*.

4.1.2 Structure-level techniques

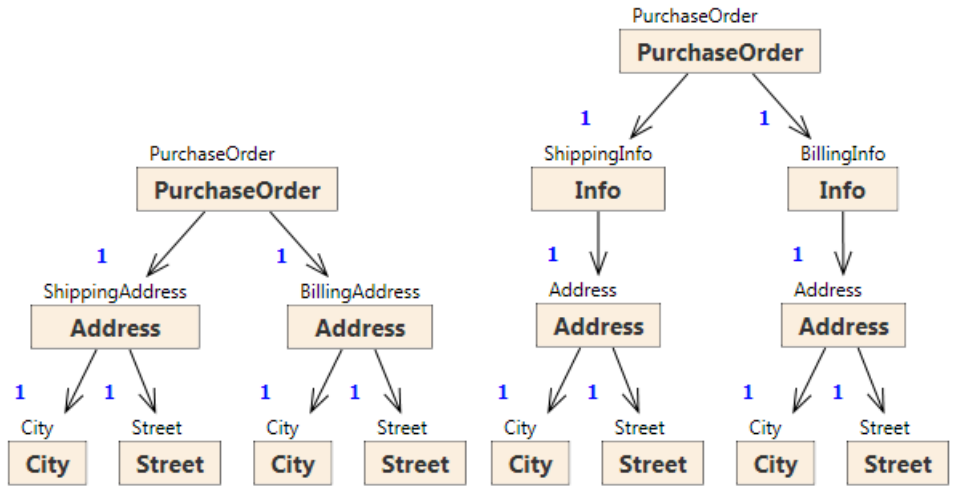
For this thesis, only graph-based techniques are considered. For more techniques see [25]. Graph–based techniques are methods that view the matched schemas as labeled graphs. The entities are compared based on the analysis of their positions in their respective graphs. The idea behind this approach is that if the two entities from the two graphs are similar, their neighbors should be somehow similar too. As always, there are some different approaches.

Graph matching This approach is based on graph matching, which has a deep theoretical background in graph theory. It is a combinatorial problem that can be computationally expensive, hence it is usually solved by approximate methods. In the case of schema matching, the method is to minimize the dissimilarity between the two schemas, which is an optimization problem.

Children This method is based on the following statement: two non–leaf schema elements are structurally similar if their immediate children sets are highly similar[2]. However, this method is not always ideal. An example of that can be Figure 4.1. The Children approach would work only if both schemas had the same structure for both billing and shipping addresses. In this case, it will fail. Clearly, the concept of shipping and billing addresses can be represented differently in different companies, but the meaning is the same and the schema matcher should align these entities correctly. To address this issue, the next approach can be used.

Leaves Instead of focusing on immediate children of an entity, this approach uses their leaf sets. It is based on the following statement: two non–leaf schema elements are structurally similar if their leaf sets are highly similar, even if their immediate children are not. This will work even with the example in Figure 4.1, because the structure between the entity and the leaf set does not matter.

[2] *Highly similar* means that their similarity exceeds a given threshold.



(a) PurchaseOrder 1 (XML Schema A.1) (b) PurchaseOrder 2 (XML Schema A.2)

Figure 4.1: Two different representations of PurchaseOrder

4.1.3 Cupid

Cupid [13] is a generic schema matcher. It combines several matching techniques described in 4.1 and was a great inspiration for the approach presented in this thesis, mainly because of its usage of element and structure-level schema matching techniques. On the input, there are two schemas. For simplicity, let them be two trees. The goal of the matching algorithm is to get a *weighted similarity* coefficient for each pair of elements from the first schema and from the second schema. This coefficient is a combination of two other coefficients described in the following paragraphs.

Linguistic similarity As a first step, Cupid calculates a *linguistic similarity* coefficient *lsim* for each pair of elements from their respective schemas. For this task it utilizes techniques such as Tokenization. It also utilizes thesauri for elimination and expansion of acronyms and abbreviations. The last, but the most interesting technique is tagging. Cupid uses the thesaurus to tag schema elements with a concept name. For example, schema elements that contain *Cost*, *Value* and *Price* tokens are all tagged by *Money* concept. These tags are then used to categorize the schema elements. This reduces the number of element-to-element comparisons, as only elements from compatible categories are compared. Two categories are compatible, if their respective sets of keywords are similar. The similarity of tokens and the similarity of categories is then combined to form the linguistic similarity coefficient.

Structural similarity The next step is the structural similarity. It is based on comparison of the elements, their neighborhoods and their leaf sets.

Specifically, the *structural similarity* coefficient *ssim* is computed for each pair of elements (from their respective schemas). First of all, the coefficient is initialized for leaves as a coefficient of their data types compatibility. For non-leaf elements, it is computed as a number of *strong links* between their leaf sets. A leaf in one schema has a strong link to a leaf in the other schema, if their weighted coefficient exceeds a given threshold. In addition, there are two thresholds, high and low. If the weighted similarity exceeds the high threshold, the structural similarity of each pair of leaves in the two subtrees is increased by a constant c_{inc}. On the other hand, if the weighted similarity drops below the low threshold, the structural similarity is decreased by another constant c_{dec}. This is because the presence of highly similar ancestors should reinforce the structural similarity of the leaves. The linguistic similarity remains unchanged.

For the description to be complete, the formula for the weighted similarity coefficient is $wsim = w_{struct} \times ssim + (1 - w_{struct}) \times lsim$, where w_{struct} is a constant from $(0, 1)$ indicating the weight of *ssim*.

With the similarity coefficients computed, a mapping generator can produce the mappings. If a mapping of non-leaf elements is required, their similarity coefficients have to be recomputed. This is because in the process, the leaf coefficients could be changed after the coefficients of ancestors are computed (by the threshold criteria adjustment).

In comparison with other matching systems (MOMIS [3] and DIKE [21]), Cupid matches more terms correctly, mainly due to the use of a thesaurus. The whole comparison is in Section 9 of [13].

4.1.4 Nečaský

This article [19] offers a complex approach to the process of reverse engineering of XML schemas to PSM diagrams. Because of this, the process of creating a PSM diagram from an XML schema is not covered by this thesis. It is described there in sufficient detail and is quite trivial, so we can assume that we already have the PSM diagram and all that is missing is the mapping. The article also contains a semi-automatic algorithm for the reconstruction of PIM-PSM mappings, but it has some severe drawbacks. The most problematic one is the computational cost which is up to m^n, where m is a maximum number of outgoing PIM associations from one PIM class and n is the number of PIM classes in the model. Therefore in practice, the algorithm will not work if the PIM diagram contains a bigger number of associations.

Definition 4.1. *A PIM path P is an expression C_1 —…— C_n where $C_1, \ldots, C_n$ are PIM classes and for each $1 \leq i < n$, there is a PIM as-*

sociation connecting C_i with C_{i+1}. If there are two or more associations connecting C_i and C_{i+1}, we need to distinguish the required association by its name l and write (l, C_{i+1}) instead of C_{i+1}. We say that P goes from C_1 to C_n. C_n is called terminal class of P.

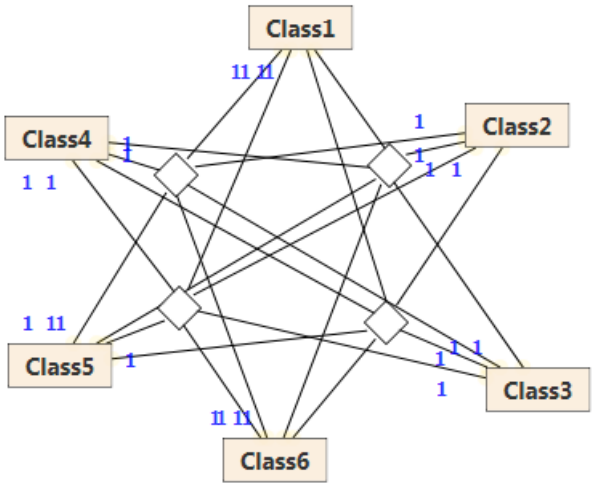


Figure 4.2: PIM that would cause trouble

Figure 4.2 shows an example of a PIM diagram that is simple enough and can cause trouble to this approach. Basically, it has 6 classes connected among each other by 4 associations, which can model that 6 entities have 4 different relations, but each one with each other. Given some restrictions like that a PIM path cannot go trough one PIM class more that once, this still gives us a total number of PIM paths between two classes in a diagram, where all n classes are connected with each other by m associations

$$\sum_{i=1}^{n} m^i \frac{(n-1)!}{(n-i)!} \tag{4.1}$$

In our case, the number would be **631 124** PIM paths between two classes, which is too much to cover for a relatively uncomplicated, six class diagram.

The algorithm

For each PSM class C_{psm}, starting with roots of a PSM diagram and descending to leaves recursively, it works like this:

Class mapping estimation performs string and structural similarity measurement (see 4.1) of C_{psm} and all PIM classes of the model.

(1) The string similarity measurement uses a maximum of two similarities. The first one is a similarity between a PIM class name and the C_{psm}

name, the second one is a similarity between the PIM class name and the element label of C_{psm}. The specific algorithm used is *longest common substring* (see Example 4.2), although any one of the algorithms for string similarity in 4.1.1 can be used simply by changing a single procedure. This coefficient is called *initial similarity*.

(2) The next step is a measurement of similarity of attributes. This is done again by using string similarity of attribute names. This time, for each attribute $Attr_{psm}$ of C_{psm}, a 2-dimensional *attribute similarity matrix* is computed. One dimension is represented by all PIM attributes of all PIM classes in the model. The second dimension contains all PIM paths connecting C, the PIM class from which C_{psm} is derived, and C', the PIM class containing the PIM attribute in question. For example, an element $x_{i,j}$ of this matrix contains a similarity coefficient computed as the similarity of names of $Attr_{psm}$ and PIM attribute i multiplied by a weight of the PIM path j. The weight of a PIM path is a similarity of labels of associations along the path with the element label of C_{psm}. In addition, it decreases as the length of the PIM path increases. For each attribute, the best coefficient from the matrix is added to the *attribute similarity* of C_{psm} and C.

(3) The final step is a measurement of similarity of the children of C_{psm}. This is done in a manner similar to how the similarity of attributes was measured. We have a 2-dimensional *child similarity matrix*, one dimension is represented by all PIM classes in the model, the second dimension contains all the PIM paths connecting C_{psm} and the PIM class in question. An element $y_{i,j}$ of this matrix contains a coefficient computed as the similarity of name and element label of C_{psm} and the name of PIM class i, multiplied by a weight of the PIM path j. Again, the best coefficient of each child's matrix is added to the *child similarity* of C_{psm} and C.

An average of the three similarities from steps described above is used as a *similarity coefficient* of C_{psm} and the PIM classes.

Class mapping specification is performed by a domain expert who is offered a list of suggested PIM classes for the mapping of each C_{psm}, ordered by the similarity coefficient.

Association mapping is a process, in which for the PSM association going to C_{psm} (if C_{psm} is not a root), the domain expert is offered a list of all possible PIM paths from C' to C (the PIM classes to which C'_{psm} - a parent of C_{psm}, and C_{psm} are mapped respectively) ordered by their weight. The selected PIM path then represents the semantics of the PSM association.

Subtree mapping is as process of mapping the attributes of C_{psm} to their counterparts in the PIM. This is done by the domain expert with

the help of the attribute similarity matrix, which was computed during the class mapping estimation. Then, the algorithm continues recursively to the children of C_{psm}.

Comments

As already demonstrated in the beginning of the description of this approach, processing all PIM paths between two PIM classes can be costly. The algorithm uses this for each PIM attribute of the model in step (2) and for each PIM class in step (3) of the class mapping estimation. Then, this approach is applied for each PSM class in the PSM diagram. It is obvious that this algorithm presents maximum comfort for the domain expert, as it compares all the possibilities, but the cost is too high as this algorithm will work only for very simple models. In Chapter 5, an approach is presented, offering less comfort for the domain expert, however, it will work for bigger models, as its computational cost is significantly lower.

4.2 Existing approaches to XML schema evolution

A lot of work has been done in the area of XML schema evolution. In this chapter, we provide a description of two representatives from the area.

4.2.1 E. Domínguez et al.

In the paper "Evolving XML Schemas and Documents Using UML Class Diagrams" [9], a method of creating and evolving XML schemas and documents using a platform-independent UML class diagram model is presented. The whole architecture consists of several components. The *conceptual* component captures the platform–independent reality using the UML class diagrams. The *logical* component captures the tool-independent knowledge describing the data structures in an abstract way. In the case of XML, it represents the XML schemas. Then there is the *extensional* component, which captures the tool–dependent knowledge, i.e. it contains textual representation of the schema using the XML Schema language. Finally, there is a *translation* component, which contains the mapping of the conceptual components to the logical ones.

Also in this paper, there is a translation algorithm described that has the UML class diagram (from the conceptual component) on the input, and creates corresponding elements within the logical component, the extensional XML schema, and a set of translation rules in a new, *intermediate* component. Every change done to the UML class diagram also affects this set of

rules in the intermediate component. This triggers an update to the logical component. The change in the logical component causes an update to the extensional component, which generates XSLT stylesheets, which modify the textual representation of the XML schema and the XML documents to conform to the new schema.

A drawback of this method is its limit to only one XML schema. It is not possible to propagate those changes to another schema, which could be connected through the platform–independent level. Another disadvantage is the lack of support for other XML schema languages.

4.2.2 Meike Klettke: Conceptual XML Schema Evolution

In this paper [11], a conceptual XML schema evolution approach is introduced. In the process described, the XML schema is normalized to the venetian blind design style. The next step is the import of the schema into the CoDEX (*Co*nceptual *D*esign and *E*volution of *X*ML schemas) tool. The CoDEX tool visualizes the schema in a way very similar to the XSEM's PSM diagram. A user can change the visualization using some basic operations such as *add*, *delete*, *change*, *move* or *rename*. These operations are logged, forming a sequence of evolution steps. This sequence is then analyzed, simplified (sequences like "new element (x), rename (x) to (y)" are substituted by "new element (y)" etc.), and finally translated into a language for XML schema updates. These updates are then applied to the XML schema.

The main disadvantage of this approach is the fact that it again considers only one XML schema. It has no connections to some platform-independent conceptual model that would enable the change to propagate to other XML schemas. In the 5-level hierarchy (Figure 3.1), its place is from the PSM level downward. It is also limited to the XML Schema language only.

Chapter 5

Mapping creation

This section contains the suggested semi-automatic algorithm for recreation of the mappings between PSM components (classes, attributes and associations) and the PIM. It helps the domain expert by offering the possibilities of mappings according to their likelihood of being correct. The main feature of this approach in contrast with the one presented in [19] and evaluated in 4.1.4 is its relatively low computational cost and therefore the ability of processing larger models. In the context of this thesis, this algorithm is used for the recreation of the mappings between a PSM diagram and a PIM, where the PSM diagram is obtained easily from an XML schema and the PIM is a conceptual model that already exists. When these mappings are in place, various evolution operations can be performed propagating the changes to all affected components in the PIM and the PSM diagrams.

Algorithm 1 Overall view of the algorithm

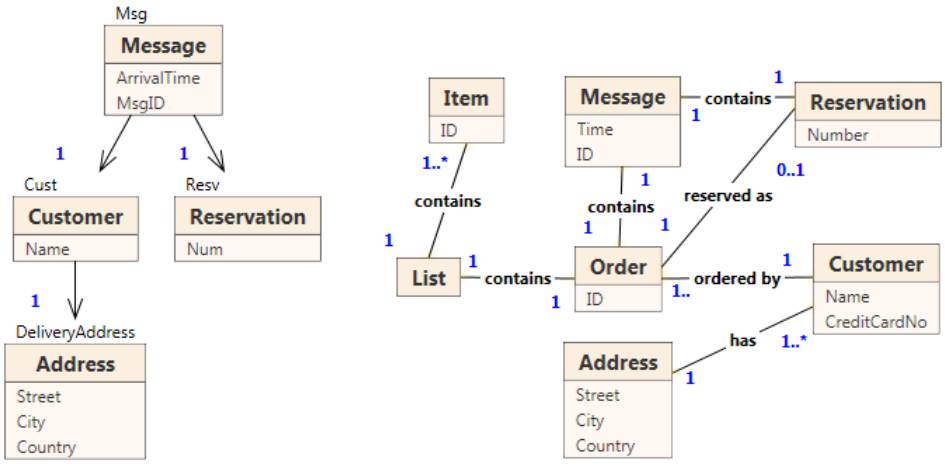
1: Compute initial attribute similarities (Algorithm 2)
2: Compute initial class similarities (Algorithm 3)
3: Map PSM classes to PIM classes (Algorithm 4)
4: Map PSM attributes to PIM attributes (Algorithm 5)
5: Map PSM associations and refine PIM paths (see 5.1.5)

5.1 Algorithm description

The algorithm described is semi-automatic and its simplest form can be seen in Algorithm 1. It has a PSM diagram and a PIM on the input. First of all, it performs some initial measurements of attribute and class similarities (lines 1 and 2) and then asks the user for a mapping of each PSM class

of the PSM diagram (line 3). The reason this is called an algorithm[1] is that the PIM classes offered to the user for mapping are sorted according to their similarity with the current PSM class. It is even possible to accept some offered mappings automatically, if the similarity is greater than a given threshold (see 5.1.3). This significantly eases the process of creation of the mappings. The next step is the mapping of attributes (line 4). The user specifies the mapping of each PSM attribute to a PIM attribute. Again, the choices are sorted by the likelihood that these attributes match. Finally, the algorithm does not guarantee a correct mapping of PIM paths. These need to be adjusted manually, but in 5.2, there is a suggested way to ease the process. This is what is happening on line 5.

Example 5.1. *As an example, we will use a modified version of the PSM diagram of Message 4 from the initial example. It has slightly modified strings for illustration of the string similarity algorithms. It can be seen in Figure 5.1(a). Our goal will be to map it to the PIM diagram in Figure 5.1(b).*



(a) PSM diagram for examples (b) PIM diagram for examples

Figure 5.1: Diagrams for examples

It is the same PIM as in Figure 1.4 and it is placed here for the convenience of the reader.

5.1.1 Attribute similarity

This task is divided into two, as attributes can be compared according to their types (including cardinality) and names. A measurement of similarity is needed for every PSM attribute from the PSM diagram processed and

[1]Some may say that if the user has to map the classes manually, what is the point in calling it an algorithm.

every PIM attribute from the model. The reason for this is described later in 5.1.4. This measurement is done by Algorithm 2.

Algorithm 2 Attributes similarity measurement

1: **for all** PSM attributes A^i_{psm} from the PSM diagram **do**
2: **for all** PIM attributes A^j_{pim} from the PIM **do**
3: $S^T_A(i,j) = T(type(A^i_{psm}), type(A^j_{pim}))$
4: $S^N_A(i,j) = StringSimilarity(A^i_{psm}, A^j_{pim})$
5: $S_A(i,j) = w * S^T_A(i,j) + (1-w) * S^N_A(i,j)$
6: **end for**
7: **end for**

Type similarity

For this step, a 2-dimensional *attribute type similarity matrix* S^T_A is computed. One dimension represents all PSM attributes from the processed PSM diagram, the second one represents all PIM attributes in the model. The coefficient $S^T_A(i,j)$ represents the type similarity of the PSM attribute A^i_{psm} and PIM attribute A^j_{pim}, which is determined using a data type comparison table T. This step is on line 3 of Algorithm 2. T can look like Table 5.1, only probably more complex as usually more data types are present in the system.

	string	*integer*	*date*	*time*
string	1	0.2	0.5	0.5
integer	0.2	1	0.1	0.1
date	0.5	0.1	1	0.7
time	0.5	0.1	0.7	1

Table 5.1: Example of a type comparison table

This table can be customized and extended with user-defined types. The specific coefficients depend on the implementation and on the user settings. At the end of this step, each pair (consisting of a PIM attribute and a PSM attribute) has a type similarity coefficient.

String similarity

The second part of the attribute similarity computation is based on string similarity. Again, there is a 2-dimensional matrix, this time called the *attribute name similarity matrix* S^N_A. The coefficients $S^N_A(i,j)$ are again computed for each pair of PIM and PSM attributes (see line 4 of Algorithm 2).

The specific method used in function *StringSimilarity* can vary. It can be any combination of string-based and language-based techniques, including linguistic resources described in 4.1.1. The choice is up to the implementation. Ideally, the user could choose which algorithm(s) to use, according to the format of attribute names used. This is because no string comparison method is universally functional and the choice of an appropriate method (and/or thesauri) can significantly improve the results.

Now, there are two matrices comparing the attributes. For further use, they can be combined into one *attribute similarity matrix* S_A, where each coefficient $S_A(i,j)$ is computed as a weighted sum of the two corresponding coefficients $S_A^T(i,j)$ and $S_A^N(i,j)$ from the previous matrices as can be seen on line 5 of Algorithm 2, where the weight w is between 0 and 1.

Example 5.2. *We will use the Longest common subsequence as a string similarity method (see Example 4.3 in Section 4.1.1). In addition, the comparison will not be case sensitive. The computed attribute similarities for the PSM attribute* num *of the PSM class* Reservation *and some of the PIM attributes are in Table 5.2. This is how the computation works for example for the PIM attribute* number *of the PIM class* Reservation *(see Algorithm 2):* $S_A^T = 1$ *(line 3) because both attributes are integers,* $S_A^N = \frac{3}{6} = 0.5$ *(line 4). Let the weight* $w = 0.5$ *meaning half of the resulting similarity will be* S_A^T *and half will be* S_A^N*. The result (line 5) is* $S_A = w * S_A^T + (1 - w) * S_A^N = 0.5 * 1 + 0.5 * 0.5 = 0.75$.

	Number	*Name*	*ID*	*Time*	*City*	*Country*	*CreditCardNo*
S_A^T	**1**	0.2	**1**	0.1	0.2	0.2	0.08
S_A^N	**0.5**	**0.5**	0	0.25	0	0.14	0.2
S_A	**0.75**	0.35	0.5	0.18	0.1	0.17	0.14

Table 5.2: Attribute similarities of *num*

5.1.2 PIM and PSM class similarity

As above with attributes, one needs to be able to compare any pair of PSM class C_{psm}^i from the processed PSM diagram and PIM class C_{pim}^j from the PIM. Again, there will be several similarity matrices which will be combined to the final *class similarity matrix* S_C, but the details are different. Classes can be compared based on their names, the element label of a PSM class, their attributes and finally, based on their neighborhood (5.1.3). This is what Algorithm 3 does.

Algorithm 3 PIM and PSM classes similarity measurement

1: **for all** PSM classes C^i_{psm} from the PSM diagram **do**
2: **for all** PIM classes C^j_{pim} from the PIM **do**
3: $S^{SN}_C(i,j) = StringSimilarity(name(C^i_{psm}), name(C^j_{pim}))$
4: $S^{SE}_C(i,j) = StringSimilarity(elementLabel(C^i_{psm}), name(C^j_{pim}))$
5: $S^N_C(i,j) = max(S^{SN}_C(i,j), S^{SE}_C(i,j))$
6: $S^A_C(i,j) = ClassAttributeSimilarity(C^i_{psm}, C^j_{pim})$ {Equation 5.1}
7: $S_C(i,j) = w * S^N_C(i,j) + (1-w) * S^A_C(i,j)$
8: **end for**
9: **end for**

String similarity

The string similarity of classes is computed in the same way as in [19]. The coefficient is a maximum of two. One is a similarity of a name of C^i_{psm} and a name of C^j_{pim} (line 3). The second one is a similarity of an element label of C^i_{psm} and the name of C^j_{pim} (line 4). After this step, each pair of PSM and PIM classes have their *string similarity coefficient* $S^N_C(i,j)$ stored in the *class name similarity matrix* S^N_C (line 5).

Attribute similarity

Here comes the part where the attribute similarity matrix S_A is used. Classes have attributes and the attributes are already compared. All that needs to be done is to combine the similarity coefficients of the attributes of C^i_{psm} to the ones of C^j_{pim} as can be seen on line 6 of Algorithm 3. There are various possibilities of how to combine them. Each PSM attribute of the PSM class has a similarity coefficient for every PIM attribute of the PIM class in question. It makes sense to only consider the maximum of these coefficients for every PSM attribute. And then these coefficients can be summed or multiplied to get the final result, which is placed to the *class attributes similarity matrix* S^A_C.

Figure 5.2: Example for combining attribute similarity coefficients

The difference between summing and multiplying is best shown on an

example. Let us have a *PIMClass* and a *PSMClass* like in Figure 5.2. Clearly, the pair of *Address* attributes will have a high similarity coefficient (e.g. 1), when the other pairs (*Address - Date*, *Address - PhoneNumber* and *Date - PhoneNumber*) will have a low similarity coefficient (e.g. 0.1). Now, what happens if the results are summed? The final similarity will be 1.1, and the semantics of that are that the classes have a high attribute similarity, because one pair of attributes is highly similar. On the other hand, when the coefficients are multiplied, the result will be 0.1, meaning that these classes have low attribute similarity, because one of the attributes of the PSM class has no similarity to any of the PIM attributes of the PIM class. Obviously, the choice is up to the implementation or, ideally, up to the user. Another way can be that when the similarity of one attribute is higher than a certain threshold, the pair of classes scores a point. As said before, there are many possibilities.

Let $A_{psm}^{C_i,k}$ be the k-th PSM attribute of C_{psm}^{i} and let n be a number of PSM attributes of C_{psm}^{i}. Then the *class attribute similarity (CAS)* from line 6 of Algorithm 3 is:

$$CAS(C_{psm}^{i}, C_{pim}^{j}) = \sum_{k=1}^{n} MAS(A_{psm}^{C_i,k}, C_{pim}^{j}) \tag{5.1}$$

Let A_{psm} be a PSM attribute, C_{pim} a PIM class, A_{pim}^{i} i-th PIM attribute of C_{pim} and n a number of PIM attributes of C_{pim}. Then *maximum of attribute similarities (MAS)* is:

$$MAS(A_{psm}, C_{pim}) = max(S_A(A_{psm}, A_{pim}^{1}), \ldots, S_A(A_{psm}, A_{pim}^{n})) \tag{5.2}$$

where $S_A(A_{psm}, A_{pim}^{l})$ means the attribute similarity coefficient of A_{psm} and A_{pim}^{l}.

Similarity matrix

Finally, a combination of the coefficients from the two matrices described above (the class name similarity matrix S_C^N and the class attributes similarity matrix S_C^A) is stored in the *class similarity matrix* S_C. The way the coefficients are combined can again be a sum or multiplication or something else. In this case, it could be useful to have a variable w (set by the user) saying how much weight is given to the name similarity and how much weight is given to the attribute similarity. The combination formula could then look like on line 7 of Algorithm 3:

$$S_C(i,j) = w * S_C^N(i,j) + (1-w) * S_C^A(i,j) \tag{5.3}$$

Example 5.3. *As an example of the PIM and PSM class similarity computation, we will compute the class similarity for the PSM class* Message *(C_{psm}) and the PIM class* Message *as in Algorithm 3. First of all, we will compute the string similarity between the names of the classes (line 3). In this case, it is clearly $S_C^{SN} = \frac{7}{7} = 1$. Next, the similarity between the name of the PIM class and the element label of C_{psm} (line 4) is $S_C^{SE} = \frac{3}{7} = 0.43$. The maximum of these two (line 5) is $S_C^N = 1$.*

Now the attribute similarity CAS (line 6) will be computed. We will start according to Equation 5.1 with PSM attribute ArrivalTime *and compute the maximum of attribute similarities (MAS) as described in Equation 5.2. $S_A(ArrivalTime, Time) = 0.68$, $S_A(ArrivalTime, ID) = 0.06$.*

Therefore $MAS(ArrivalTime, Message) = 0.68$. The same computation is done for the PSM attribute MsgID *and $MAS(MsgID, Message) = 0.7$. From this, the CAS $S_C^A(C_{psm}, Message) = 1.38$.*

Finally, let the weight $w = 0.5$. What is left to do is to combine the name and attribute similarities of the classes as on line 7. $S_C(C_{psm}, Message) = w * S_C^N(C_{psm}, Message) + (1 - w) * S_C^A(C_{psm}, Message) = 0.5 * 1 + 0.5 * 1.38 = 1.19$.

All the similarities for C_{psm} are in Table 5.3.

	Item	*List*	*Message*	*Order*	*Reserv.*	*Customer*	*Address*
S_C^{SN}	0.14	0.14	**1**	0.14	0.27	0.25	0.43
S_C^{SE}	0.25	0.25	**0.43**	0	0.09	0.13	0.14
S_C^N	0.25	0.25	**1**	0.14	0.27	0.25	0.43
S_C^A	0.8	0	**1.38**	0.8	0.72	0.59	0.54
S_C	0.53	0.13	**1.19**	0.47	0.5	0.42	0.49

Table 5.3: Class similarities of *Message*

5.1.3 Structural similarity and class mapping

Now that the initial similarities are computed, it is time to start the mapping process (Algorithm 4). In these following paragraphs, let C_{psm} be the current PSM class during the traversal of the PSM diagram tree and C_{pim} the PIM class to which C_{psm} is or is to be mapped to.

In contrast with the algorithm from [19], which chooses the top-down approach, this algorithm is bottom-up. To be exact, it uses a post-order traversal of the PSM tree. The advantage is that whenever the algorithm is in an inner node C_{psm}, all the children of C_{psm} are already mapped, and therefore it is possible to increase or decrease the class similarity by a certain factor considering the distance of C_{pim} to the PIM classes to which the children of C_{psm} are mapped.

The intuition that shorter PIM paths are better than longer PIM paths is a little bit problematic. The PIM paths used to describe the semantics of a PSM association can all be very long and in that case, the adjustment would not work very well. On the other hand, most of the PIM paths in realistic situations are in fact shorter than longer.

Algorithm 4 Class mapping algorithm

```
1:  for all roots of the PSM diagram do
2:    for all PSM classes C_psm in the post-order traversal of the PSM tree
      do
3:      if C_psm is a leaf then
4:        Sort the PIM classes according to S_C
5:        Offer them to the user as mappings for C_psm
6:      else
7:        Compute the structural similarity adjustment
8:        Sort the PIM classes according to S_C and the adjustment
9:        Offer them to the user as mappings for C_psm
10:     end if
11:   end for
12: end for
```

Leaves

As said before, the algorithm actually starts in the root. But because it traverses the tree using the post-order approach, it is in fact bottom-up. The first time the user is asked to provide a mapping is for the first leaf. The order of the offered PIM classes to map the leaf to is based solely on the class similarity matrix S_C, specifically the row corresponding to C_{psm} (the leaf), starting from the most similar classes and going to the least similar ones. This part starts on line 3 of Algorithm 4.

Inner nodes

When the algorithm reaches an inner PSM class C_{psm}, all of the children of C_{psm} are already mapped. As described before, an adjustment to the initial class similarity (in S_C) can be made to account for the structural similarities.

Let $D^1_{psm}, \ldots, D^n_{psm}$ be the children of C_{psm}. Let $D^1_{pim}, \ldots, D^n_{pim}$ be the PIM classes to which the children are mapped to respectively. The intuition behind the structural similarity adjustment is that the PIM class C_{pim} should be as close as possible to $D^1_{pim}, \ldots, D^n_{pim}$ in the PIM. Therefore, we measure the distance between each PIM class C^j_{pim} from the model and each one of $D^1_{pim}, \ldots, D^n_{pim}$. Dijkstra's algorithm can be used to find the shortest paths to every C^j_{pim} from every one of $D^1_{pim}, \ldots, D^n_{pim}$. The results are

stored in an array, representing the *final distance* D_j between C^j_{pim} and all of $D^1_{pim}, \ldots, D^n_{pim}$. That final distance is computed for each C^j_{pim} as a sum of distances d_i from each of $D^1_{pim}, \ldots, D^n_{pim}$, averaged by n, the number of children of C_{psm}. The final distance D_j of a PIM class C^j_{pim} from $D^1_{pim}, \ldots, D^n_{pim}$ is:

$$D_j = \sum_{i=1}^{n} \frac{d_i}{n} \tag{5.4}$$

Now that the distances are computed, a similarity adjustment S^j_{adj} can be added to each C^j_{pim} depending on how close to $D^1_{pim}, \ldots, D^n_{pim}$ the class is. The exact nature of this adjustment can again be set by the user, who can express his preference of the structural similarity over the string and attribute similarities by setting a higher weight for the distance coefficient. This is what is happening on lines 7-9 of Algorithm 4.

Example 5.4. *As an example of structural similarity and class mapping, we will describe the moment Algorithm 4 reaches the PSM class* Message *(C_{psm}), which is on lines 7-9.*

Structural similarity adjustment *In this case, the adjustment will be computed as $S^j_{adj} = \frac{1}{D_j}$ where D_j is the final distance of a PIM class C^j_{pim} from $D^1_{pim}, \ldots, D^n_{pim}$ as in Equation 5.4. We assume that the children of C_{psm} (*Customer *and* Reservation*) are mapped to their respective counterparts in the PIM. Let C_{pim} be the PIM class* Message*. Its distance from* Customer *is* 2 *and from* Reservation *it is* 1*. Therefore, $D_j = \frac{2}{2} + \frac{1}{2} = \frac{3}{2}$ and the structural similarity adjustment is $S^j_{adj} = \frac{1}{D_j} = \frac{2}{3} = 0.67$.*

	Message	*Order*	*Reserv.*	*Address*	*Customer*	*Item*	*List*
S_C	**1.19**	0.47	0.5	0.49	0.42	0.53	0.13
S_{adj}	0.67	**1**	0.5	0.5	0.5	0.33	0.5
c	**1.86**	1.47	1	0.99	0.92	0.86	0.63

Table 5.4: Similarities when mapping *Message*

Class mapping *With the structural similarity adjustments calculated for each PIM class, we will simply add them to the similarity coefficients already calculated in S_C. Any method of combining the coefficients can be chosen here. Addition is used for its simplicity. Table 5.4 contains the original S_C coefficients, similarity adjustments and final resulting coefficients c, by which the list offered to the user as well as the table are sorted. The user now chooses the right PIM class to which C_{psm} will be mapped to.*

Thresholds

As a step toward automation, suggestions can be automatically accepted or rejected according to the similarity coefficient. We can introduce a low threshold th_{low}. PIM classes with similarity lower than th_{low} do not need to be offered at all, or only on demand. Similarly, there can be a high threshold th_{high} for the case that there is only one PIM class with the similarity greater than th_{high}. In that case, this PIM class could be accepted automatically without asking the user. The specific threshold values should be variable and to be used by advanced users. Note that if the thresholds are present, class attribute similarity needs to be normalized. The way it is in Equation 5.1, it produces results dependent on the number of PSM attributes and therefore a universal threshold cannot be defined.

5.1.4 Attribute mapping

The last thing left to do is the mapping of PSM attributes to their PIM counterparts. There are some options of what to do with a PSM attribute A_{psm} of a PSM class C_{psm}, which is now mapped to a PIM class C_{pim}. The attribute mapping algorithm is Algorithm 5.

Algorithm 5 Attribute mapping algorithm

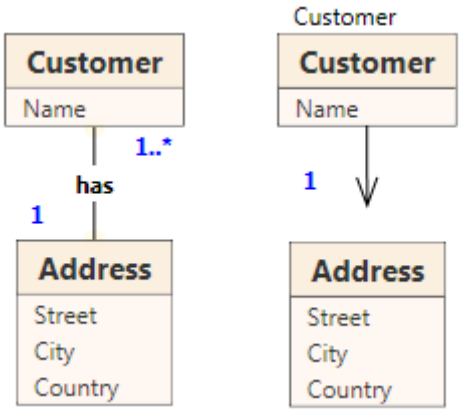
1: **for all** PSM attributes A_{psm} **do**
2: Add bonus B for the PIM attributes of C_{pim}
3: Sort the PIM attributes according to $S_A + B$
4: Offer them to the user as mappings for A_{psm}
5: **end for**

(1) The simplest option is to map A_{psm} to a corresponding PIM attribute A_{pim} of C_{pim}. The method of offering a list of possibilities sorted by the similarity between A_{psm} and the PIM attributes of C_{pim} is very similar to the one used for classes. The list is displayed in descending order of similarity. Again, there can be thresholds for auto-accepting and ignoring the suggestions.

(2) The most complicated option is to map the PSM attribute A_{psm} to a PIM attribute A'_{pim} of any PIM class C'_{pim} that is connected by a PIM path to C_{pim}. This may be needed in a case of an XML schema created from a PIM and PSM like in Figure 5.3, which can look like this:

```
<xs:complexType name="Customer">
  <xs:attribute name="Name"    type="xs:string"/>
  <xs:attribute name="Street"  type="xs:string"/>
```

```
  <xs:attribute name="City"    type="xs:string"/>
  <xs:attribute name="Country" type="xs:string"/>
</xs:complexType>
```



(a) PIM diagram (b) PSM diagram

Figure 5.3: Example for attribute similarity

This is because when a PSM class does not have an element label (like *Address* in Figure 5.3(b)), it does not form a subelement. Its attributes are inserted into the parent class (*Customer*) instead. This situation may not be detected by the algorithm that reverse engineers the XML schema to a PSM diagram. Therefore, a PSM attribute A_{psm} can appear in a PSM class C_{psm} to which it does not belong in the context of the PIM. Because the situation number 1 is more common, an appropriate bonus B to the similarity coefficient of the PIM attributes of C_{pim} is added (line 2), so that these attributes appear before all the others when the list is offered to the user (line 4).

When the user maps the A_{psm} to a PIM class other than C_{pim}, it can be solved by creating a new PSM class C'_{psm} representing the PIM class C'_{pim} containing the PIM attribute A_{pim}, to which A_{psm} is mapped. A_{psm} can then be moved to C'_{psm} and C'_{psm} can be added as a child of C_{psm}. This way, the problem is transformed to the situation number 1. Of course, when more than one PSM attribute does not belong to its PSM class C_{psm}, this approach would generate a new PSM class C'_{psm} for each one of the attributes, so a detection of this situation should be considered while implementing this approach. It is not hard as it only requires detecting whether there already is a PSM class C'_{psm} among the children of C_{psm} representing C'_{pim} containing A_{pim}, to which A_{psm} is mapped. If there is such a class, A_{psm} can be moved to that class instead of creating a new one. This is what happens on line 4 of Algorithm 5.

(3) There is yet another option, which is further described in 5.3.2.

Example 5.5. *We will describe the process of mapping the PSM attribute* MsgID *of the PSM class* Message *to its PIM counterpart as in Algorithm 5. Table 5.5 contains the similarity coefficients from* S_A[2]*, eventual bonuses* $B = 0.5$ *for being an attribute of* C_{pim} *(line 2) and the final coefficients* c*, according to which the list is sorted (line 3) before it is offered to the user (line 4). In this example, the bonus* B *is simply added to the coefficient from* S_A*.*

	Message:ID	*Order:ID*	*Item:ID*	*Time*	*Number*	*Name*
S_A	**0.7**	**0.7**	**0.7**	0.15	0.59	0.2
B	**0.5**	0	0	**0.5**	0	0
c	**1.2**	0.7	0.7	0.65	0.59	0.2

Table 5.5: Some similarities of PSM attribute *MsgID*

5.1.5 PSM associations

The mapping of the PSM associations is simple. When a non-leaf PSM class C_{psm} is mapped to C_{pim}, the PSM associations leading to its children D^j_{psm} mapped to D^j_{pim} are mapped to represent the shortest PIM paths from C_{pim} to D^j_{pim} respectively. This mapping may be inaccurate and because of that, an approach to fixing these is suggested in 5.2.

5.2 Methods for refining PIM paths

One of the problems of the algorithm described above is that the mapping of PSM associations is the shortest PIM path between the PIM classes represented by the PSM parent and PSM child classes. This mapping may be semantically wrong, even if the resulting XML schema is the same. In that case the domain expert has to check the PSM associations and their mappings and fix it according to the right semantics the PSM associations are supposed to represent. In this section, there is a proposal of how a tool for fixing the PIM paths could look like. We will demonstrate it on an example.

Let us have a PIM diagram like the one in Figure 5.4. There is a *Supplier* who offers *Parts*, which can be supplied by multiple suppliers. Multiple parts together can make a *Supply*. The *Supplier* can also supply the whole *Supply*. Now, let us have a PSM diagram like the one in Figure 5.5. The PSM asso-

[2]The PSM attributes of the PSM class *Address* are missing because their similarities are low and therefore they are not important for the example.

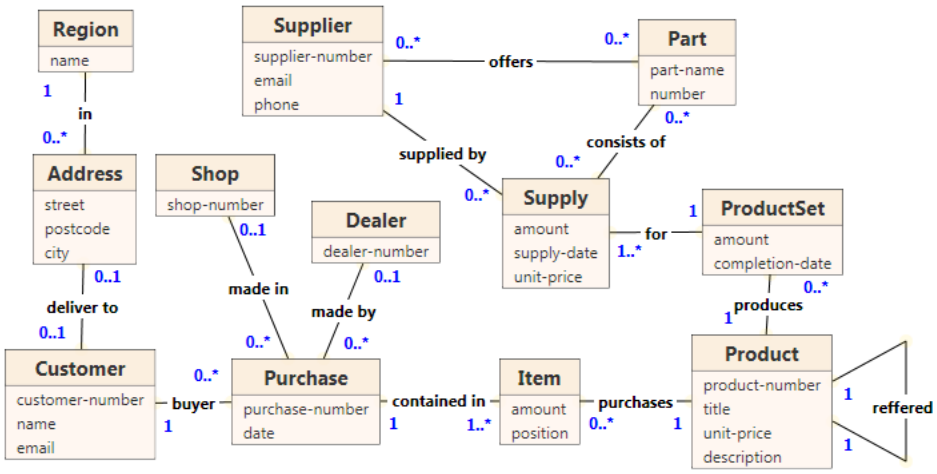


Figure 5.4: Sample PIM diagram

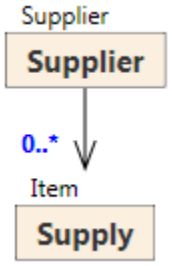
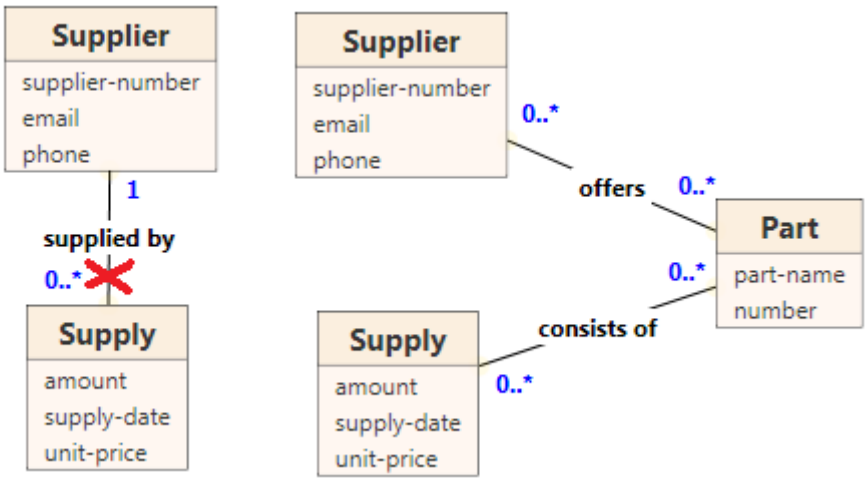


Figure 5.5: Sample PSM diagram

ciation in that diagram can either represent the PIM association *Supplier-Supply* or the PIM path *Supplier-Part-Supply*. The algorithm chooses the shortest path, which is *Supplier-Supply*. Now, if the domain expert decides that the shortest path is wrong, there are several approaches to how to ease the process of fixing the PIM path. We will explain two possible approaches in the following subsections. Note that the approaches presented here are applicable to any situation where the PIM path needs to be altered, not only at the end of the algorithm described earlier.

5.2.1 Refusing associations

One of the possibilities is to show the user the current PIM path in a graphical way. The user can then refuse single PIM associations from which the PIM path is composed of. When a PIM association is refused, it is automatically replaced by the next shortest PIM path connecting the two PIM classes. This approach can be particularly useful when dealing with more complex PIM paths as it does not require the domain expert to specify the whole



(a) Refusal of association (b) Next shortest PIM path

Figure 5.6: Sample of refusing associations method

PIM path from scratch. There can be a situation when there are two or more PIM paths of the same length. In that case, the next method (5.2.2) can be used to choose from the possibilities, or the algorithm can offer them in some arbitrary order. An example of this approach applied to the sample PIM diagram in Figure 5.4 is in Figure 5.6.

5.2.2 Manual PIM path selection

Another method more suitable for shorter PIM paths is to let the domain expert go through a tree of PIM associations from the source PIM class of the PIM path and choose exactly through which PIM associations the PIM path should go. Again, applied to our example in Figure 5.4, it can look like in Figure 5.7.

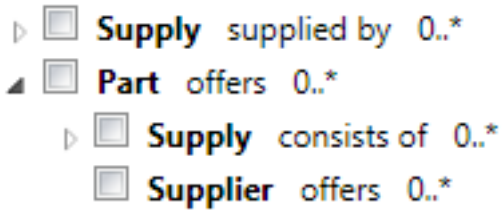


Figure 5.7: PIM path selection

5.3 Adjusting the PIM

There can be a situation when the user wants to map a PSM component to a new non-existing PIM component. That is usually when the XML schema described by the PSM diagram adds something new to the conceptual model that was not there before.

5.3.1 Missing PIM classes

In the case of mapping a PSM class C_{psm}, it is possible to add a new PIM class C_{pim} instead of mapping C_{psm} to an existing PIM class when the user is asked. PIM associations are added between C_{pim} and every PIM class C^i_{pim} represented by the children C^i_{psm} of C_{psm}. The corresponding PSM associations will then represent these new PIM associations.

Another PIM association is added when mapping a PSM class D'_{psm} to D'_{pim} and the fact that a child D_{psm} of D'_{psm} was mapped to a newly created PIM class D_{pim} is discovered. The PIM association then leads between D_{pim} and D'_{pim}. This includes the situation when D'_{psm} is also mapped to a new PIM class.

5.3.2 PIM-less attributes

Another option when mapping PSM attributes is that A_{psm} will only be present in the PSM and will have no counterpart in the PIM. In that case, the attribute can be left unmapped and is called *PIM-less*. This option can be selected by the user when a list of possible mappings is offered.

5.3.3 Missing PIM

A special case of a PIM is an empty PIM, i.e. a PIM with zero classes, associations and attributes. Therefore, it is possible to use the approach described above even if there is no PIM present. In that case, the user adds the PIM classes as described in 5.3.1 until all the PSM classes from the PSM diagram are mapped to some newly created PIM classes in the PIM. The PSM attributes can be left as *PIM-less* and then propagated to the PIM classes as necessary.

5.4 Analysis

In this section, we will determine the computational complexity of the described algorithm. Let n be the number of PIM classes in the PIM, N be the number of PSM classes in the PSM diagram, a be the maximal number of attributes in one PIM/PSM class and s be the maximal length of a string

(names of classes and attributes, element labels). Next, let us assume that $S(s)$ is a function that returns the computational complexity of the algorithm chosen to compare strings. We will follow Algorithm 1 and compute the complexity for each step.

Initial attribute similarities - Algorithm 2

On line 4 we have performed a computation of a string similarity of two attributes - $O(S(s))$. On lines 3 and 5 we have performed computations that can be done in a constant time $O(1)$. This was done for each PSM ($N \times a$) and each PIM ($n \times a$) attribute. Therefore, the computational complexity of this step is:

$$O(n \times N \times a^2 \times S(s))$$

Initial class similarities - Algorithm 3

On lines 3 and 4, we have performed computations of string similarities of a PIM and a PSM class - $O(S(s))$. On line 5 we have combined these similarities. This can be done in a constant time $O(1)$. On line 6 we compute attribute similarity of a PIM and a PSM class. The *MAS* (Equation 5.2) takes $O(a)$ as it computes a maximum of similarities between one PSM attribute and all PIM attributes of one PIM class (a). Therefore, the *CAS* (Equation 5.1) takes $O(a^2)$, because it sums the *MAS* from each PSM attribute of a PSM class (a). Finally, on line 7 a combination of coefficients is computed, which can be done in a constant time $O(1)$. This routine is done for each PIM and each PSM class, therefore the computational complexity of this step is:

$$O(n \times N \times (a^2 + S(s)))$$

Class mapping - Algorithm 4

In this step, we will use a sorting algorithm to sort the PIM classes offered to the user. Let it be *Heap sort*[3]. Its running time in the worst case is $O(n \times log(n))$.

For a computation of the structural similarity adjustment, we use *Dijkstra's algorithm*[4]. Its computational complexity can be $O(m + n \times log(n))$ when using *Fibonacci heap*[5], where m is a number of PIM associations in the PIM.

If we are in a leaf, we only sort the PIM classes and offer them to the user. This takes $O(n \times log(n))$. Otherwise, we are in an inner class and in addition to sorting the classes, we have to compute the structural similarity

[3] `http://en.wikipedia.org/wiki/Heapsort`

[4] `http://en.wikipedia.org/wiki/Dijkstra's_algorithm`

[5] `http://en.wikipedia.org/wiki/Fibonacci_heap`

coefficient. This means running the Dijkstra's algorithm in the PIM up to N times - once for each child of the current PSM class. This makes the computational complexity of this option $O(N \times n \times log(n) + n \times log(n))$.

This whole routine is done for each root of a PSM diagram (up to N) and for each PSM class of the diagram (N). For simplicity, we will take into account only the costlier of the two options described above. This makes the computational complexity of this step:

$$O(N^2 \times (N \times n \times log(n) + n \times log(n))) = O(N^3 \times n \times log(n))$$

Attribute mapping - Algorithm 5

In this step, we need to sort the PIM attributes for each PSM attribute of the PSM diagram. This takes:

$$O(N \times a \times (n \times a \times log(n \times a))) = O(N \times n \times a^2 \times log(n \times a))$$

Entire algorithm

When we sum the computational complexity of each described step, we get:

$$O(N \times n \times a^2 \times (log(n \times a) + S(s)) + N^3 \times n \times log(n))$$

Conclusion

As demonstrated above, the algorithm depends mainly on the number of PSM classes $O(N^3)$ in the PSM diagram processed. This is not a problem, because the number of PSM classes in a PSM diagram is usually considerably less than the number of PIM classes in the PIM. In relation to the PIM, the algorithm runs in $O(n \times log(n))$ which cannot be better due to the use of a sorting algorithm and an algorithm for the shortest paths in a graph.

Chapter 6

Evolution operations

In this chapter we describe a set of evolution operations that can be performed on PIM and PSM levels (see Figure 3.1) in a system containing a PIM and connected PSM diagrams representing XML schemas. We have implemented a certain subset of these operations in XCase, which is described in 3.1.6. We also lay down foundations of a formal system for operations description. The main concept of the operations is *composition*. There are a few *atomic* operations that do not use other operations. *Composite* operations can consist of these atomic ones. We will first describe the atomic operations as they work on their respective levels. The propagation of changes between the levels will be added later. Today's software tools for the modeling of XML schemas (e.g. Enterprise Architect [26]) do not focus on the propagation of changes between a PIM and a PSM, not to mention among one PIM and more PSMs. But as we have shown in 1.1, it can be very helpful. Finally, we will show some examples of composite operations. In the formal system, addition operations are denoted α, operations that change components are denoted δ and operations that delete components are denoted ρ.

Definition 6.1. *An* operation *is a function. Its input is a system of a PIM and multiple PSM diagrams and possibly other information required to fulfill its purpose. Its output is a system modified by this operation. The operation can leave the system in an inconsistent state and may therefore require to perform additional operations to put the system to a consistent state.*

6.1 Atomic operations

In this section we will describe atomic operations on both PIM and PSM levels. These operations are then used to create composite operations.

6.1.1 PIM level

The PIM level contains constructs from UML class diagrams (see 3.1.4) - Classes with attributes and associations connecting the classes.

Adding PIM components

The operations for adding PIM components are trivial. They are:

- $\alpha(C_{pim})$ - Add a PIM class C_{pim}.
- $\alpha(A_{pim}, C_{pim})$ - Add a PIM attribute A_{pim} to a PIM class C_{pim}.
- $\alpha(R_{pim}, C^1_{pim}, \ldots, C^n_{pim})$ - Add a PIM association R_{pim} (create a self-reference or connect n PIM classes C^i_{pim}).

Changing PIM components

Changing a PIM component is also quite straightforward. The atomic operations are:

- $\delta(C_{pim}, name)$ - Rename a PIM class C_{pim}.
- $\delta(A_{pim}, name)$ - Rename a PIM attribute A_{pim}.
- $\delta(R_{pim}, name)$ - Rename a PIM association R_{pim}.
- $\delta(A_{pim}, type/multiplicity/defaultvalue)$
  Change properties of a PIM attribute A_{pim} (type, multiplicity, default value).
- $\delta(R_{pim}, label/role_1, \ldots, role_n/multiplicity_1, \ldots, multiplicity_n)$
  Change properties of a PIM association A_{pim} (label, roles, multiplicities).

Deleting PIM components

As in many other problems, deleting a component is a little bit trickier than adding a component, because of its connections to other components. These atomic operations can be performed on their own safely:

- $\rho(A_{pim})$ - Delete a PIM attribute A_{pim}.
- $\rho(R_{pim})$ - Delete a PIM association R_{pim}.

Deleting PIM class $\rho(C_{pim})$ - deleting a PIM class C_{pim} - is an example of an atomic operation that will lead to an inconsistent state of the system. When a PIM class is deleted, all of its PIM attributes $A_{pim}^{i=1...n}$ also need to be deleted. In addition, all of the PIM associations $R_{pim}^{j=1...m}$ connecting this PIM class need to be deleted as well. Formally:

1. $\forall i \in \{1, \ldots, n\} \rho(A_{pim}^{i})$.
2. $\forall j \in \{1, \ldots, m\} \rho(R_{pim}^{j})$.

6.1.2 PSM level

The PSM level contains PSM classes with PSM attributes and other constructs that are not discussed in this thesis (see 3.1.5). It is important to realize that a PSM diagram has a forest structure.

Adding PSM components

There are only two addition operations on the PSM level, as PSM associations are created automatically when a PSM class is added under another PSM class (or when it is reverse-engineered from an XML schema).

- $\alpha(C_{psm}, C'_{psm})$ - Add a PSM class C_{psm} as a child of a PSM class C'_{psm}. This also creates a PSM association R_{psm} connecting them.
- $\alpha(C_{psm})$ - Add a PSM class C_{psm} as a root of a PSM diagram.
- $\alpha(A_{psm}, C_{psm})$ - Add a PSM attribute A_{psm} to a PSM class C_{psm}.

Changing PSM components

- $\delta(C_{psm}, name)$ - Rename a PSM class.
- $\delta(C_{psm}, elementlabel)$ - Change an element label of a PSM class.
- $\delta(C_{psm}, C'_{psm})$ - Change the parent of a PSM class C_{psm} to C'_{psm}. This means reconnecting the PSM association R_{psm} leading to C_{psm}.
- $\delta(A_{psm}, alias/type/multiplicity/defaultvalue)$ - Change properties of a PSM attribute (alias, type, multiplicity, default value).
- $\delta(R_{psm}, multiplicity)$ - Change multiplicity of a PSM association R_{psm}.
- $\delta(R_{psm}, PIMpath)$ - Change the PIM path represented by a PSM association R_{psm}.

Deleting PSM components

In the same way as when deleting PIM components, we will first list the operations that are safe to perform. In this case, it is only $\rho(A_{psm})$ - the *deletion of a PSM attribute* A_{psm}.

Deleting PSM association $\rho(R_{psm})$ - deleting a PSM association R_{psm} - is an atomic operation. The PSM class, C_{psm} to which this PSM association leads, becomes a new root of the PSM diagram.

Deleting root PSM class $\rho(C_{psm})$ - deleting a root PSM class C_{psm} - is also an atomic operation which leads to an inconsistent PSM diagram. For the PSM diagram to be put to a consistent state, it involves a deletion of all of C_{psm} PSM attributes $A_{psm}^{i=1...n}$ - $\forall i \in \{1, \ldots, n\}\rho(A_{psm}^{i})$. In addition, all PSM associations $R_{psm}^{j=1...m}$ leading from C_{psm} have to be deleted, making the children of C_{psm} new roots of the PSM diagram. $\forall j \in \{1, \ldots, m\}\rho(R_{psm}^{j})$.

6.2 Propagation of changes

The main advantage of the XSEM model (3.1.3) is that the components on the PIM level are connected with the corresponding PSM components, allowing the propagation of changes between the levels. We will now revisit the operations described above in 6.1.1 and 6.1.2 and extend them with the propagation of changes.

6.2.1 PIM level

It is obvious that addition operations do not require any propagation of changes, as the newly created PIM components are not connected to any of the PSM components. We can safely skip them and move straight to the operations that change and delete PIM components. The designer can, of course, use the standard XSEM model mechanisms to derive PSM components from the new PIM components, which is already supported by XSEM (see 3.1.3).

Changing PIM components

Renaming PIM class This operation has no effect on the connected PSM classes, as they use their own names. The only thing to consider is during implementation. When a PSM class has a different name than the represented PIM class, both names have do be displayed.

Renaming PIM attribute This operation has no effect on the connected PSM attributes, as they use their aliases.

Renaming PIM association This operation has no effect on the connected[1] PSM associations.

Changing properties of PIM attribute When changing properties of a PIM attribute such as the data type, multiplicity or default value, this change usually needs to be propagated to all the connected PSM attributes A^i_{psm} - $\forall i \in \{1, \ldots, n\} \delta(A_{psm}, \mathit{property})$. However, it is up to the implementation whether this change is propagated automatically or the user is asked if the propagation is required. Technically, it is possible for the PIM attribute to have different properties than the connected PSM attribute, but it is unusual.

Changing properties of PIM association A change of properties of a PIM association such as the label or the roles has no effect on the connected PSM associations. On the other hand, a change of multiplicities needs to be propagated to all the affected PSM associations $R^{i=1\ldots n}_{psm}$, as it has a direct impact on their multiplicities - $\forall i \in \{1, \ldots, n\} \delta(R_{psm}, \mathit{multiplicity})$.

Deleting PIM components

Deleting PIM attribute This operation requires either deleting all of the connected PSM attributes $A^{i=1\ldots n}_{psm}$ or converting them to *PIM-less* - $\forall i \in \{1, \ldots, n\} \rho(A^i_{psm})$.

Deleting PIM association When deleting a PIM association, all of the connected PSM associations $R^{i=1\ldots n}_{psm}$ need to be removed as well - $\forall i \in \{1, \ldots, n\} \rho(R^i_{psm})$. For details, see the description of deleting a PSM association in 6.1.2.

Deleting PIM class Deleting a PIM class requires removing all PIM associations connecting it. These are removed in a manner described in the previous paragraph. In addition, all the PIM attributes of this PIM class need to be removed, which was also described earlier. Finally, when deleting the PIM class itself, all the derived PSM classes need to be removed as well. For details, see the description of deleting a PSM class in 6.1.2.

[1] When talking about PSM associations connected to PIM associations, what is technically meant is PSM associations that are connected to PIM paths which this PIM association is part of.

6.2.2 PSM level

There is one operation listed in 6.1.2 that may need the automatic propagation of changes, and that is *changing properties of a PSM attribute*. With the introduction of the mapping algorithm in Chapter 5, it is now possible to add PSM classes and PSM attributes directly to a PSM diagram and then create the proper mappings to the PIM. This was not possible before, as all the PSM components needed to be derived from the PIM in order to create the mappings. Therefore, the addition operations on the PSM level can be propagated with the assistance of the user using a slightly modified version of the mapping algorithm described in Chapter 5. The propagation of changes for these operations is also described in this section.

Changing properties of PSM attribute

A change of properties of a PSM attribute can be propagated to the PIM attribute A_{pim} to which this PSM attribute is connected - $\delta(A_{pim}, property)$. The user should be asked if the change is to be propagated. If it is, it should initiate the propagation of a change of properties of the PIM attribute A_{pim} described earlier. This means that the change can be propagated to the PIM and then back to all the PSM diagrams in which a PSM attribute derived from A_{pim} is present.

Adding PSM class as root

When adding a new PSM class C_{psm}, which is not yet mapped to any PIM class, as a root of a PSM diagram, the user is asked to which PIM class is C_{psm} to be mapped to. If the user chooses to create a new PIM class, this change is propagated as a creation of a PIM class C_{pim}. Formally: $\alpha(C_{pim})$.

Adding PSM class as child

When adding a new PSM class C_{psm}, which has no counterpart in the PIM, as a child of a PSM class C'_{psm}, the user is asked to which PIM class is C_{psm} to be mapped to. If the user chooses to create a new PIM class, this change is propagated as a creation of a PIM class C_{pim} and a PIM association R_{pim} connecting it to C'_{pim} (the PIM class represented by C'_{psm}). Formally: $\alpha(C_{pim}), \alpha(R_{pim}, C_{pim}, C'_{pim})$.

Adding PSM attribute

When a new PSM attribute A_{psm} is added to a PSM class C_{psm} representing a PIM class C_{pim}, the user is asked to which PIM attribute is A_{psm} to be mapped. If the user chooses to add a new PIM attribute A_{pim}, this addition

can be propagated as an addition of A_{pim} to the PIM class C_{pim}. Formally: $\alpha(A_{pim}, C_{pim})$.

6.3 Composite operations

With the propagation of changes described in 6.2 in mind, it is possible to compose various more complex operations from the atomic PIM and PSM operations described in 6.1. Below, we will demonstrate complex operations on several examples. We will always consider the PIM from Figure 1.4, but for simplicity, we will only show the parts that are changed.

Moving PIM attribute

A movement of a PIM attribute A_{pim} from one PIM class to another is a composite operation. It consists of removing A_{pim} from the original PIM class C_{pim} and adding it to the target PIM class C'_{pim}. The PSM attributes A^i_{psm} derived from A_{pim} are deleted through the propagation of changes when A_{pim} is removed from C_{pim}. On the other hand, when A_{pim} is placed into C'_{pim}, no change is propagated. Therefore we must add a PSM attribute A^j_{psm} to each PSM class $C^{j=1\ldots m}_{psm}$ derived from C'_{pim}. Formally:

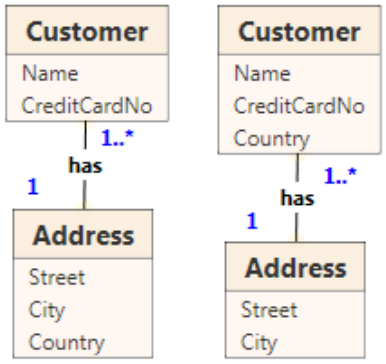
1. $\rho(A_{pim})$
2. $\alpha(A_{pim}, C'_{pim})$
3. $\forall j \in \{1, \ldots, m\} \alpha(A^j_{psm}, C^j_{psm})$

Example 6.1. *We will move the PIM attribute* Country *from the PIM class* Address *to the PIM class* Customer *as can be seen in Figures 6.1(a) and 6.1(b). The affected PSM diagrams are* Message 1 *and* Message 4*, which can be seen in Figures 6.1(c) and 6.1(d).*

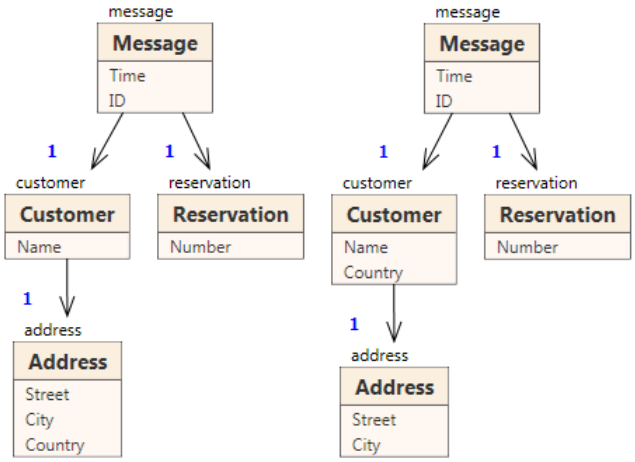
Changing PIM attribute into PIM class

It is possible to change an existing PIM attribute A_{pim} of a PIM class C_{pim} into a new PIM class. In this case, we will remove A_{pim} - all derived PSM attributes A^i_{psm} will be removed through the propagation of changes. Then we will create a new PIM class C'_{pim} with the name of A_{pim} and add a PIM association R_{pim} connecting C_{pim} and C'_{pim}. Because we want a similar effect on the PSM level, we need to add a PSM class C^i_{psm} derived from C'_{pim} as a child to each C'^i_{psm} from which A^i_{psm} was removed. Formally:

1. $\rho(A_{pim})$
2. $\alpha(C'_{pim}), \alpha(R_{pim}, C_{pim}, C'_{pim})$



(a) PIM before (b) PIM after



(c) Message 4 before (d) Message 4 after

Figure 6.1: Examples of moving a PIM attribute

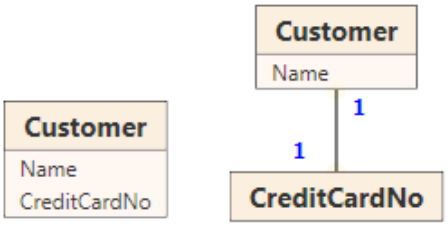
3. $\forall i \in \{1, \ldots n\} \alpha(C^i_{psm}, C'^i_{psm})$

4. $\forall i \in \{1, \ldots n\} \delta(R^i_{psm}, PIMpath(C_{pim}, C'_{pim}))$

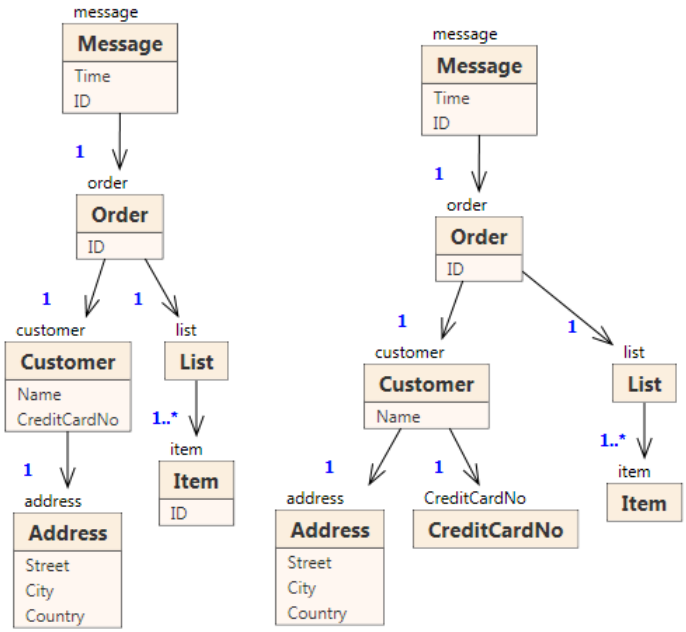
Example 6.2. *We will change the PIM attribute* CreditCardNo *of the PIM class* Customer *into a new PIM class, as can be seen in Figures 6.2(a) and 6.2(b). The affected PSM diagram is* Message 1*, which can be seen in Figures 6.2(c) and 6.2(d).*

Splitting PIM attribute

Splitting a PIM attribute A_{pim} of a PIM class C_{pim} is a composite operation. We will simply add a new PIM attribute A'_{pim} to C_{pim}. Then to each PSM class $C^{i=1\ldots n}_{psm}$ derived from C_{pim} we will add a PSM attribute A^i_{psm} derived from A'_{pim}. Formally:



(a) PIM before (b) PIM after

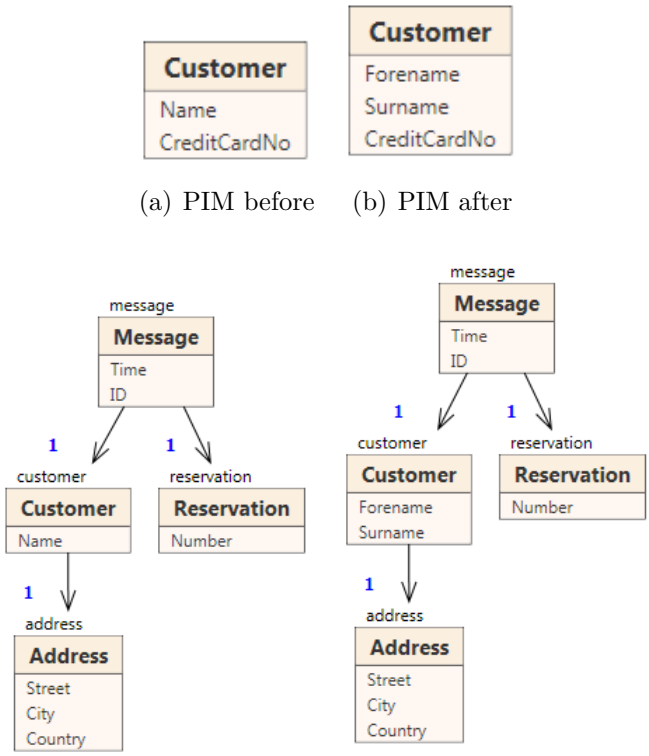


(c) Message 1 before (d) Message 1 after

Figure 6.2: Examples of changing a PIM attribute into a PIM class

1. $\alpha(A'_{pim}, C_{pim})$
2. $\forall i \in \{1, \ldots n\} \alpha(A^i_{psm}, C^i_{psm})$

Example 6.3. *This operation can be used to solve the problem described in 1.1 - a change of a representation of a customer's name from a single value* name *into two values* forename *and* surname. *The operation is performed on the PIM class* Customer *as can be seen in Figures 6.3(a) and 6.3(b). The affected PSM diagrams are* Message 1 *and* Message 4, *which can be seen in Figures 6.3(c) and 6.3(d).*



(a) PIM before (b) PIM after

(c) Message 4 before (d) Message 4 after

Figure 6.3: Examples of splitting a PIM attribute

Deleting a PSM tree

Deleting a PSM tree rooted in C_{psm} is a composite operation. We will denote it $\omega(C_{psm})$. First of all, the PSM associations $R_{psm}^{i=1\ldots n}$ leading to the children of C_{psm} need to be removed. This means that all the children $C_{psm}^{i=1\ldots n}$ of C_{psm} are now new roots. The trees rooted in them need to be deleted recursively. Formally:

1. $\forall i \in \{1, \ldots, n\} \rho(R_{psm}^{i})$
2. $\forall i \in \{1, \ldots, n\} \omega(C_{psm}^{i})$

Chapter 7

Conclusions

In this thesis, we began with an introduction to the problem of XML schema evolution and we showed that the current situation, in which large sets of XML schemas must be altered manually when the IT infrastructure evolves, is unsatisfactory (see Chapter 1). A brief introduction to XML technologies followed in Chapter 2. In Chapter 3 we introduced a *5-level XML evolution architecture* consisting of conceptual and logical levels. In this architecture, this thesis is focused on the PIM and PSM conceptual levels. In the description of the conceptual levels we provided a survey of various approaches to conceptual modeling of XML data and then we presented *XSEM*, a conceptual model for XML, including its first implementation, a tool called *XCase*. A survey of current schema matching techniques followed in Chapter 4, describing element-level techniques like string-based matching, language-based matching, constraint-based matching and linguistic resources such as thesauri and also describing structure-level technique *graph matching*. Then we surveyed some of the current work in the area of schema matching and XML schema evolution.

In the second half of this thesis, a new semi-automatic technique of mapping XML schemas to a conceptual model was introduced (Chapter 5). It incorporates element-level and structure-level methods of schema matching and is customized to fit the XSEM model. Its main advantage over the method described in 4.1.4 is its computational speed, which enables this method to process large conceptual diagrams. In addition, a set of evolution operations was presented for both PIM and PSM conceptual levels, including the propagation of changes between those levels (Chapter 6). With the mapping method present and with the set of evolution operations in mind, we can summarize the XML schema evolution process in a few simple steps:

1. Create mappings between each XML schema and the conceptual model (as in Chapter 5)

2. With these mappings present, perform evolution operations (as in

Chapter 6)

3. Generate new, evolved XML schemas

7.1 Open problems

There are a few problems not covered in this thesis that we need solved for this whole process to work in practice, but that does not seem difficult.

7.1.1 Reverse engineering of XML schemas

One of the open problems is the process of getting a PSM diagram from an XML schema. One possibility of getting a PSM diagram from a schema in XML Schema language is sketched out in [19], but it requires a specific type of schema and therefore is not universal. Further research is needed to make a method that would reverse-engineer any XML Schema and also to accept different XML schema languages like Relax NG.

7.1.2 Generating XML schemas from PSM diagrams

Another open problem is the generation of XML schemas from PSM diagrams. There is a method proposed in [18, p. 109-128] and partially implemented in *XCase*, which generates an XML Schema from a PSM diagram, but further research is needed for the method to allow different schema styles and additional languages like Relax NG.

7.2 Future work

In our future work, we will focus on extending the process of change propagation to the whole 5-level XML evolution architecture, so it will be possible to really evolve the whole XML system in an IT infrastructure from one place - the conceptual diagram.

7.2.1 Mapping creation enhancements

It is clear that while the method for creating mappings of XML schemas to the conceptual model presented in 4.1.4 is computationally too expensive, it provides maximal comfort and accuracy of suggested mappings, as it utilizes all information that the PIM and PSM diagrams contain. On the other hand, the method proposed in this thesis is fast and can process large diagrams, but utilizes a smaller amount of information, thus providing less accuracy. Our future research in this area will be aimed at providing a reasonable

compromise between these two methods: increasing accuracy while checking the computational cost.

To be specific, the method in this thesis (see Chapter 5) only considers the shortest PIM paths when adjusting for the structural similarities of classes. It would be possible to consider, for example, k shortest paths, measure similarities with PIM association labels on the paths and offer them to the user in an order influenced by these similarities, instead of relying on the user to fix eventual inaccuracies manually.

7.2.2 XSLT transformations for modifying XML documents

One of the possible ways our research could go is toward automatic generation of XSLT scripts that would modify the existing XML documents so that they stay compatible with the evolved XML schemas. This could be done either by propagating the changes directly to the XML documents by elementary XSLT transformations, or by a detection of changes between two versions of a PSM diagram and generation of a corresponding XSLT script containing all the changes between two versions at once.

Chapter 8

CD contents

The attached CD contains a PDF version of this thesis - *thesis.pdf*. In addition, it contains XCase installer in the *XCase* folder.

Bibliography

[1] Altova Inc. XML Spy 2009. http://www.altova.com.

[2] A. Badia. Conceptual Modeling for Semistructured Data. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering Workshops*, pages 170–177, Singapore, Dec. 2002. IEEE Computer Society.

[3] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Rec.*, 28(1):54–59, 1999.

[4] M. Bernauer, G. Kappel, and G. Kramler. Representing XML Schema in UML - An UML Profile for XML Schema. Technical Report November 2003, Department of Computer Science, National University of Singapore, 2003.

[5] R. Bourret. XML and Databases. http://www.rpbourret.com/xml/XMLAndDatabases.htm, September 2005.

[6] P. Chen. The Entity-Relationship Model–Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, Mar. 1976.

[7] J. Clark and M. Makoto. *RELAX NG Specification*. Oasis, December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[8] D. Booth, C. K. Liu. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. W3C, June 2007. http://www.w3.org/TR/wsdl20-primer/.

[9] E. Domínguez, J. Lloret, A. L. Rubio, and M. A. Zapata. Evolving XML Schemas and Documents Using UML Class Diagrams. In K. V. Andersen, J. K. Debenham, and R. Wagner, editors, *DEXA*, volume 3588 of *Lecture Notes in Computer Science*, pages 343–352. Springer, 2005.

[10] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[11] M. Klettke. Conceptual XML Schema Evolution - The CoDEX Approach for Design and Redesign. In M. Jarke, T. Seidl, C. Quix, D. Kensche, S. Conrad, E. Rahm, R. Klamma, H. Kosch, M. Granitzer, S. Apel, M. Rosenmüller, G. Saake, and O. Spinczyk, editors, *Workshop Proceedings Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, pages 53–63, Aachen, Germany, March 2007.

[12] B. Loscio, A. Salgado, and L. Galvao. Conceptual Modeling of XML Schemas. In *Proceedings of the Fifth ACM CIKM International Workshop on Web Information and Data Management*, pages 102–105, New Orleans, USA, Nov. 2003.

[13] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 49–58, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[14] M. Mani. Erex: A conceptual model for xml. In *Proceedings of the Second International XML Database Symposium*, pages 128–142, Toronto, Canada, Aug. 2004.

[15] G. A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[16] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.

[17] K. Narayanan and S. Ramaswamy. Specifications for Mapping UML Models to XML. In *Proceedings of the 4th Workshop in Software Model Engineering*, Montego Bay, Jamaica, 2005.

[18] M. Nečaský. *Conceptual Modeling for XML*, volume 99 of *Dissertations in Database and Information Systems Series*. IOS Press/AKA Verlag, January 2009.

[19] M. Nečaský. Reverse Engineering of XML Schemas to Conceptual Diagrams. In *Proceedings of The Sixth Asia-Pacific Conference on Conceptual Modelling*, pages 117–128, Wellington, New Zealand, January 2009. Australian Computer Society.

[20] Object Management Group. *UML Infrastructure Specification 2.1.2*, nov 2007. `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/`.

[21] L. Palopoli, G. Terracina, and D. Ursino. DIKE: a system supporting the semi-automatic construction of cooperative information systems from heterogeneous databases. *Softw. Pract. Exper.*, 33(9):847–884, 2003.

[22] G. Psaila. ERX: A Conceptual Model for XML Documents. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 898–903, Como, Italy, Mar. 2000. ACM.

[23] N. Routledge, L. Bird, and A. Goodchild. UML and XML Schema. In *Proceedings of 13th Australasian Database Conference (ADC 2002)*. ACS, 2002.

[24] A. Sengupta, S. Mohan, and R. Doshi. XER - Extensible Entity Relationship Modeling. In *Proceedings of the XML 2003 Conference*, pages 140–154, Philadelphia, USA, Dec. 2003.

[25] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, 4:146–171, 2005.

[26] Sparx Systems. Enterprise architect. `http://www.sparxsystems.com.au/products/ea/index.html`.

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

[28] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, October 2004. `http://www.w3.org/TR/xmlschema-1/`.

Appendix A

Used XML Schemas

The XML schemas listed here are used as PSM diagrams (see 3.1.5) in the thesis. They are intentionally written without the <xs:schema> head, as it is always the same.

A.1 Figure 4.1(a)

```
<xs:element name="PurchaseOrder" type="PurchaseOrder" />

<xs:complexType name="City" />
<xs:complexType name="Street" />
<xs:complexType name="Address">
  <xs:sequence>
    <xs:element name="City" type="City" />
    <xs:element name="Street" type="Street" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="PurchaseOrder">
  <xs:sequence>
    <xs:element name="ShippingAddress" type="Address" />
    <xs:element name="BillingAddress" type="Address" />
  </xs:sequence>
</xs:complexType>
```

A.2 Figure 4.1(b)

```
<xs:element name="PurchaseOrder" type="PurchaseOrder" />

<xs:complexType name="City" />
<xs:complexType name="Street" />
<xs:complexType name="Address">
  <xs:sequence>
    <xs:element name="City" type="City" />
    <xs:element name="Street" type="Street" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="PurchaseOrder">
  <xs:sequence>
    <xs:element name="ShippingInfo" type="Info" />
    <xs:element name="BillingInfo" type="Info" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Info">
  <xs:sequence>
    <xs:element name="Address" type="Address" />
    ...
  </xs:sequence>
</xs:complexType>
```

A.3 Message 1 (Figure 1.2(a))

```
<xs:element name="message" type="Message" />
<xs:complexType name="Message">
  <xs:sequence>
    <xs:element name="order" type="Order" />
  </xs:sequence>
  <xs:attribute name="Time" type="xs:time"/>
  <xs:attribute name="ID" type="xs:integer"/>
</xs:complexType>

<xs:complexType name="Order">
  <xs:sequence>
    <xs:element name="customer" type="Customer" />
    <xs:element name="list" type="List" />
  </xs:sequence>
  <xs:attribute name="ID" type="xs:integer"/>
</xs:complexType>

<xs:complexType name="Customer">
  <xs:sequence>
    <xs:element name="address" type="Address" />
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string"/>
  <xs:attribute name="CreditCardNo" type="xs:string"/>
</xs:complexType>

<xs:complexType name="Address">
  <xs:attribute name="Street" type="xs:string"/>
  <xs:attribute name="City" type="xs:string"/>
  <xs:attribute name="Country" type="xs:string"/>
</xs:complexType>

<xs:complexType name="List">
  <xs:sequence>
    <xs:element name="item" type="Item"
       maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Item">
  <xs:attribute name="ID" type="xs:integer"/>
</xs:complexType>
```

A.4 Message 2 (Figure 1.2(b))

```
<xs:element name="message" type="Message" />

<xs:complexType name="Message">
  <xs:sequence>
    <xs:element name="list" type="List" />
  </xs:sequence>
  <xs:attribute name="Time" type="xs:time"/>
  <xs:attribute name="ID" type="xs:integer"/>
</xs:complexType>

<xs:complexType name="List" />
```

A.5 Message 3 (Figure 1.2(c))

```
<xs:element name="message" type="Message" />

<xs:complexType name="Message">
  <xs:sequence>
    <xs:element name="reservation" type="Reservation"
       minOccurs="0" />
  </xs:sequence>
  <xs:attribute name="Time" type="xs:time"/>
  <xs:attribute name="ID" type="xs:integer"/>
</xs:complexType>

<xs:complexType name="Reservation">
  <xs:attribute name="Number" type="xs:integer"/>
</xs:complexType>
```

A.6 Message 4 (Figure 1.3(a))

```
<xs:element name="message" type="Message" />

<xs:complexType name="Message">
  <xs:sequence>
    <xs:element name="customer" type="Customer" />
    <xs:element name="reservation" type="Reservation" />
  </xs:sequence>
  <xs:attribute name="Time" type="xs:time"/>
  <xs:attribute name="ID" type="xs:integer"/>
</xs:complexType>

<xs:complexType name="Customer">
  <xs:sequence>
    <xs:element name="address" type="Address" />
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string"/>
</xs:complexType>

<xs:complexType name="Address">
  <xs:attribute name="Street" type="xs:string"/>
  <xs:attribute name="City" type="xs:string"/>
  <xs:attribute name="Country" type="xs:string"/>
</xs:complexType>

<xs:complexType name="Reservation">
  <xs:attribute name="Number" type="xs:integer"/>
</xs:complexType>
```

A.7 Message 5 (Figure 1.3(b))

```
<xs:element name="message" type="Message" />

<xs:complexType name="Message">
  <xs:attribute name="Time" type="xs:time"/>
  <xs:attribute name="ID" type="xs:integer"/>
</xs:complexType>
```

A.8 Message 6 (Figure 1.3(c))

```
<xs:element name="message" type="Message" />

<xs:complexType name="Message">
  <xs:sequence>
    <xs:element name="reservation" type="Reservation"
       minOccurs="0" />
  </xs:sequence>
  <xs:attribute name="Time" type="xs:time"/>
  <xs:attribute name="ID" type="xs:integer"/>
</xs:complexType>

<xs:complexType name="Reservation">
  <xs:attribute name="Number" type="xs:integer"/>
</xs:complexType>
```