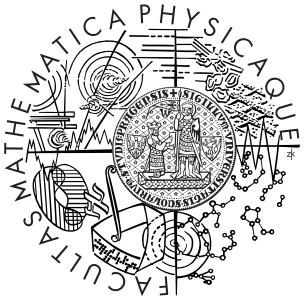
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jakub Stárka

Similarity of XML Data

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.
Studijní program: Informatika, softwarové systémy

2010

Na tomto místě bych rád poděkoval vedoucí mé práce RNDr. Ireně Mlýnkové, Ph.D. za hodnotné rady a především její nekonečnou trpělivost, kterou se mnou měla při dokončování této práce. Dále bych chtěl poděkovat své rodině a blízkým, kteří mi umožnili v klidu tuto práci dokončit.

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

V Praze dne 6.8.2010 Jakub Stárka

Contents

Název práce: Similarity of XML Data
Autor: Jakub Stárka
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Irena Mlýnková, Ph.D.
e-mail vedoucího: irena.mlynkova@mff.cuni.cz

Abstrakt: V předložené práci studujeme možnosti v oblasti reverzního inženýrství XML schémat. Práce obsahuje základní přehled jazyka XML a běžně používaných jazyků pro zápis XML schémat, přehled existujících technik pro konceptuální modelováni, reverzní inženýrství a metody pro hledání mapováni mezi XML schématy. Dále práce obsahuje návrh nové metody, založené na analýze konceptuálního modelu XSEM a následném vytvoření rozhodovacího stromu, která umožnuje efektivně najít mapováni z XML schémat na modely XSEM. V práci také ukazujeme novou techniku pro nalezení cesty mezi třídami. V neposlední řadě obsahuje práce několik experimentů, které ukazují výhody a nevýhody navrhovaného řešení.

Klíčová slova: XML, podobnost, rozhodovací stromy, reverzní inženýrství

Title: Similarity of XML Data
Author: Jakub Stárka
Department: Department of Software Engineering
Supervisor: RNDr. Irena Mlýnková, Ph.D.
Supervisor's e-mail address: irena.mlynkova@mff.cuni.cz

Abstract: In the present work we study the possibilities of reverse engineering of XML schemas. The work contains a survey of XML and commonly used languages for describing XML schemas, an overview of existing techniques for conceptual modeling, reverse engineering and methods for the mapping evaluation between XML schemas. A new method, based on analysis of the conceptual model XSEM and the subsequent creation of a decision tree, is introduced. The method allows effectively find a mapping from XML schemas to models XSEM. The work also describes a new technique for selection of the path between the mapped classes. Finally, the work contains a number of experiments that show the advantages and disadvantages of the proposed solutions.

Keywords: XML, mapping, decision tree, reverse engineering

Chapter 1

Introduction

1.1 XML

The extensible Markup Language (XML) [28] is with no doubt one of the basic instruments for data exchange. Due to its versatility and ease of use it is expanding rapidly in many areas. It becomes one of the basic formats for business-to-business applications, it finds application in Web Services, or only as a universal data format.

This causes increased interest in research of new methods for simplifying manipulation with XML data and more effective integration into the complex systems. Due to the very universal structure of XML there is a huge spread of possible definitions used. This is not caused only by the amount of existing options for definition of structure. One of the most widely used is the XML Schema [29]. This language offers versatile mechanisms for definition and allows creating dozens of specifications for same documents.

In the current complex system the related data are separated and their processing is inconsistent. Consistency in this issue brings the idea of Model Driven Architecture (MDA) [13]. The entire functionality of the system is described by the so-called Platform Independent Model (PIM). The PIM is then translated into one or more Platform Specific Models (PSMs). PIM describes the domain, i.e. one schema which contains description of whole situation and all involved subjects. On the other hand, there can be many PSMs each specific for its user group but linked to PIM. This connection between PSM and PIM allows to easy modify and manage the domain.

In real world the usage of MDA is more complicated. Companies have dozens of different sources from which they want to derive schemas and

they have to integrate these sources, respectively schemas, into their specific system, which is often a difficult and expensive activity. The result is many different schemas and many different systems to translate from one schema to another.

1.2 Motivation

We will model the whole situation, we focus on using an example with small supplier who is selling cars. The contractor will retain information on individual vehicles. We have a defined schema including manufacturer, type of car, price, performance, speed and volume of the luggage space. This supplier delivers cars to a partner shop, which has its own schema, which is indeed similar, but contains slight differences. Moreover, there are additional schemas for orders and customer records. This brings us to a relatively difficult situation where we have in a hand many different patterns, but no model of the overall situation.

We based this work on use of conceptual diagrams as described in [18]. The language, which is used here for the PSM is XML Schema. The main problem comes from the fact that it expects the creation of a diagram, and subsequently derivation of the most specific situations. If we look closer at the situation with the car trade we can see that there are more data sources available.

In ideal case the PIM is created at first and then a PSM is derived for a particular case. In real case, we come to a state where we have XML Schema definition (XSD), we can independently create a PIM, but we completely lack any association between them. Simple variant is to create completely new PSM, and along with it create the links. This possibility is, unfortunately, in most cases quite unrealistic. That leaves manual browsing of the XSDs, their transfer to the PSM and subsequent search mapping. The second option is the aim of this work.

The aim of this work is to create an adaptive system which allows semi-automatic searching for a set of mappings between XSD (but not necessarily from the same source) or its PSM representation and pre-generated PIM. This process is overseen by the domain expert who has to confirm offered mappings and resolve unexpected inputs. This work is largely based on the work of Jakub Klimek [10] and Martin Necasky [19]. They proposed basic approach for reverse engineering with fixed set of mapping methods. We improve this process by introducing more flexible strategy with usage of

decision trees. The other improvement is usage of clique search to recognize paths in a large PIM.

1.3 Content Overview

In the second chapter we describe basic information on XML and its related languages, which are needed for further understanding of the text. The conceptual modeling, basic used conceptual modeling approaches and in particular model XSEM is mentioned in the third chapter. In the fourth chapter we present representative examples of works that deal with finding similarities between XML schemas, or generally between structured data and also describe some basic methods for calculating string similarity. There are also mentioned the approaches for reverse engineering. The fifth chapter focus on describing the proposed algorithm, the individual steps, including user intervention and decision tree creation and usage. The sixth chapter provides several experimental measurements. The seventh and final chapter makes conclusions and suggestions for further work.

Chapter 2

XML Technologies

2.1 XML

XML [28] is a widely used a markup language. It is generally used for data exchange and data serialization. XML has spread rapidly over the past few years. It is also supported by more applications and programming languages which leads to greater interest in the issue of storing and processing such data.

From the beginning XML was designed with multilingual support. This is achieved by usage of Unicode [25].

2.1.1 Structure

XML uses tags for describing its content. A tag is markup construct that begins with '<' and ends with '>'. Tags form elements. An element consist of either a start and an end tag or only of an empty tag.

Elements can contain attributes. All attribute values have to be in quotes (single or double).

An important part of each document is declaration. It contains information about used encoding and version.

In Figure 2.1 we can see an example of XML document with 3 different items. At Line 1 we can see XML declaration with used encoding utf-8. At Line 2 there is a root element with defined namespace against which should be the document valid. Element *orderperson* at Line 3 contains text content and the element at Line 4 has element content. Lines 7 to 11 describe properties of an item with nested elements. The element at Line 12 contains

attribute *id* and one nested element *quantity* and an empty tag *car* only with attributes.

```
1:  <?xml version="1.0" encoding="utf-8"?>

2:  <ship xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="ship.xsd">
3:    <orderperson>John Smith</orderperson>
4:    <shipto>
5:      <name>Ola Nordmann</name>
6:    </shipto>
7:    <item id="1">
8:      <title>Empire Burlesque</title>
9:      <quantity>1</quantity>
10:     <price>10.90</price>
11:   </item>
12:   <gift price="123">
13:     <quantity>2</quantity>
14:       <car price="500" quantity="1" color="blue" />
15:   </gift>
16: </ship>
```

Figure 2.1: XML example

There are defined some basic terms like 'valid' and 'well-formed' document.

Definition 1. *A textual object is a* well-formed *XML document if it has one root element which contains all the other elements. Non-empty element has to begin and end with the same tag. All attribute values are correctly quoted. Elements can be nested but they can not overlap. The element tags are case-sensitive and match exactly.*

Definition 2. *An XML document is* valid *if it has an associated document type declaration and if the document complies with the constraints expressed in it.*

2.2 DTD

Document type declaration (DTD) [28] was created side by side with XML to allow simple description of XML. It can provide a protocol which applications will use for mutual communication.

DTD has several constructs which allow to define elements and their optionality and multiplicity. Nested elements are used to describe tree structure of document. The definition of element contains its name and its content.

```
<!ELEMENT element-name (element-content)>
```

The content can be $EMPTY$ for elements without any defined subelements or text, ANY for combination of any parsable data, $PCDATA$ for text content or the definition of subelements.

It is possible to define attributes in the same way as elements with the name of parent element, name of attribute, type and default value.

```
<!ATTLIST element-name attribute-name
                  attribute-type default-value>
```

Data types in DTD are limited to CDATA and PCDATA which means character data and parsed character data. Although DTD does not allow the type definition, there are some attributes with specific features. It is possible to define $attribute - type$ as a choice from enumerated list as $(a|b|c)$. An attribute with defined $attribute - type$ ID has unique value, the $IDREF$ is reference to ID of another element. $NMTOKEN$ is a valid XML name declared in XML document.

It is possible to define $ENTITIES$. $ENTITY$ is a variable which can be used to define special character or shortcut.

```
<!ENTITY js "Jakub Starka">
```

DTD can be inlined in an XML document or in a separate file.

In Figure 2.2 we can see the previously used XML example (see Figure 2.1) declared by DTD.

```
1:  <!ELEMENT ship (ordeperson, shipto, item+, gift?, car*)>
2:  <!ELEMENT orderperson  (#PCDATA)>
3:  <!ELEMENT shipto (name?)>
4:  <!ELEMENT item (title?, quantity?, price?)
5:  <!ELEMENT name  (#PCDATA)>
6:  <!ELEMENT title  (#PCDATA)>
7:  <!ELEMENT quantity (#PCDATA)>
8:  <!ELEMENT gift  (quantity, (car|item))>
9:  <!ELEMENT car EMPTY>

10: <!ATTLIST gift price CDATA #REQUIRED>
11: <!ATTLIST item id ID #REQUIRED>
12: <!ATTLIST car price CDATA #REQUIRED>
13: <!ATTLIST car quantity CDATA #REQUIRED>
14: <!ATTLIST car color CDATA #IMPLIED>
```

Figure 2.2: DTD example

At Line 1 is defined root element *ship*. For this element is the sequence of nested elements declared in parentheses. It has to contain elements *orderperson*, *shipto* exactly once. The plus sign after element *item* tells that this element has to be present at least once. The question mark says that element *gift* is optional and the asterisk means that there can any number of element *car*. At Line 2 there is an element with parsed character data. Element *item* at Line 4 contains three optional elements *title* and *quantity* and *price*. Lines 5 to 7 define elements with text content. At Line 8 a choice is used. The element has to contain exactly one of *car* and *item* elements and one *quantity* element. The element *car* is defined as an empty element.

At Lines 10 to 14 there are defined attributes. Four attributes contain character data and the attribute *id* contains unique identifier. The first one is defined for element *gift*. The name of the attribute is *price* and it is required. The rest of attributes is defined for element *car*. There are two required attributes *quantity* and *price* and one optional attribute *color*.

2.3 XML Schema

XML Schema [29] is a language which enables one to describe the structure of an XML document. It has many features from previously used schema lan-

guages like DTD [28] and, in addition, it brings some new features like data types, namespaces or a possibility to define only a fragment of a document.

XML Schema definition (XSD) utilizes XML format which allows its efficient processing using existing tools for XML. This feature has also its disadvantages because the resulting files are not very human-readable. Unlike DTD, XSD allows to define data types.

XSD allows the use of various constraints such as cardinality of elements or necessity of attribute. Each element can be assigned with attribute *minOccurs* or *maxOccurs*. Those attributes indicate how often the element can occur.

Other possibility which XSD offers is to use elements *choice*, *sequence* or *all*. The *choice* specifies that either one child element or another can occur. In the *sequence* all elements have to appear in the specific order. When element *all* is used all child nodes can be used in any order.

Unlike DTD, XSD supports data types. It comprises string data types like *string* or *normalizedString*, numeric data types (*integer* or *decimal*) or dates. XSD also has broad possibilities of restrictions. It is possible to set minimum or maximum value for integers or pattern and length for strings.

In the example in Figure 2.3 we can see that there is a great difference in complexity between defined structure in DTD and in XSD. There are defined 4 root elements *ship*, *shipto*, *item* and *car* at Lines 3, 25, 32 and 42. Each of them is defined as a complex type. the content of the *ship* element is defined as a sequence which means it has to contain all elements *orderperson*, *item*, $gift$ and *car* in the defined order. There is also defined the key constraint $itemKey$ at Line 4. This constraint is applied on subelements *item* and their attribute *uid*. The elements *item* and *car* have attribute ref. It is a reference to complex types defined at the Lines 32 and 42. The element $gift$ is assigned with one required attribute *price* with data type integer. Its content is defined as a choice between *item* and *car*.

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 3: <xs:element name="ship">
 4:  <xs:key name="itemKey">
 5:   <xs:selector xpath="item"/>
 6:   <xs:field xpath="@uid"/>
 7:  </xs:key>
 8:  <xs:complexType>
 9:   <xs:sequence>
10:    <xs:element name="orderperson" type="xs:string" />
11:    <xs:element ref="item" minOccurs="1" />
12:    <xs:element name="gift" type="xs:integer"
            minOccurs="0" maxOccurs="1">
13:     <xs:complexType>
14:       <xs:choice>
15:        <xs:element ref="item" />
16:        <xs:element ref="car" />
17:       </xs:choice>
18:      </xs:complexType>
19:      <xs:attribute name="price" type="xs:integer"
            use="required"/>
20:    </xs:element>
21:    <xs:element ref="car" minOccurs="0"
            maxOccurs="unbounded" />
22:   </xs:sequence>
23:  </xs:complexType>
24: </xs:element>


25: <xs:element name="shipto">
26:  <xs:complexType>
27:   <xs:sequence>
28:    <xs:element name="name" type="xs:string"
            minOccurs="0" />
29:   </xs:sequence>
30:  </xs:complexType>
31: </xs:element>
```

```
32: <xs:element name="item">
33:  <xs:complexType>
34:   <xs:sequence>
35:    <xs:element name="title" type="xs:string"
            minOccurs="0" />
36:    <xs:element name="quantity" type="xs:integer"
            minOccurs="0" />
37:    <xs:element name="price" type="xs:integer"
            minOccurs="0" />
38:   </xs:sequence>
39:   <xs:attribute name="id" type="xs:string" />
40:  </xs:complexType>
41: </xs:element>

42: <xs:element name="car">
43:  <xs:attribute name="price" type="xs:integer"
            use="required"/>
44:  <xs:attribute name="quantity" type="xs:integer"
            use="required"/>
45:  <xs:attribute name="color" type="xs:string""/>
46: </xs:element>
47:</xs:schema>
```

Figure 2.3: XSD example

Chapter 3

Conceptual Modeling

Conceptual modeling allows to model data regardless of their representation in target platform. We use it to create model of domain and then we find correspondences between model and concrete instances.

This section contains basic information about conceptual model XSEM for XML as described in [18]. This approach is based on MDA – Model Driven Architecture [13]. It is a generally used method for software engineering which we utilize in this thesis. We will recognize two types of models. Platform independent model (PIM) describes domain independently on data representation. And platform specific model (PSM) represents data modeled by the PIM diagram. The aim of this work is to create a connection between conceptual model represented by PIM and data represented by XSD.

Beside this method, we will describe other commonly used approaches for conceptual modeling like Entity-Relationship (ER) Model [5] and Unified Modeling Language (UML) [24].

3.1 Related Conceptual Models

3.1.1 ER Model

The Entity-Relationship model [5] is a simple conceptual model used primarily for database modeling. It provides two modeling constructs – *entity type* and *relation type* that allows to simply model the domain.

Entity types, or shortly *entities*, are used for modeling of real-world objects. Each entity has zero or more attributes with properties of the modeled object and a *name* for its identification in ER model.

Relation types are used to specify the relations between entities. Similarly to entities, each relation type has a *name* and attributes. Moreover, each of them is assigned with a set of entities that participate on this relation.

The work [3] extends ER Model by adding constructs for modeling of DTD – *optional*, *required* and *choice* attributes. Similarly, XER [22] is an extension which enriches the model with constructs more suitable for XML Schema modeling.

3.1.2 UML

UML [24] class diagrams are mainly used to describe software systems but it is possible to use them for data modeling as well. UML provides constructs for different types of relations (for example association, inheritance or composition). The approaches based on UML usually apply MDA. There are several approaches allowing to model XML data such as [4][17][21]. For PIM there are usually used extensions of UML constructs for specific XML schema language.

3.2 XSEM

This thesis uses conceptual model called XSEM [18]. XSEM is a model based on MDA approach which extends the ER model. It contains two subparts XSEM-ER and XSEM-H. The first part XSEM-ER is PIM used for modeling at conceptual level without any connection to XML schema languages. The second part XSEM-H is PSM designed to allow representation of different XML schemas.

3.2.1 Platform Independent Model

Platform independent model is conceptual level with no relation to specific representation. We can see an example of PIM as a UML [24] class diagram in Figure 3.1. UML offers more constructs but we consider only classes with attributes having names and data types and binary associations.

We define a *path* between two PIM classes c_1 and c_n as an expression $c_1 - ... - c_n$ where c_1,...,c_n are PIM classes and for each $i \in 1, ..., n$, there is a PIM association connecting c_i and c_{i+1}.

The diagram contains 8 PIM classes: *Address*, *Customer*, *Purchase*, *Car*, *Supplier*, *Reseler*, *E-Shop* and *Car Shop*. Two classes *Address* and *Supplier*

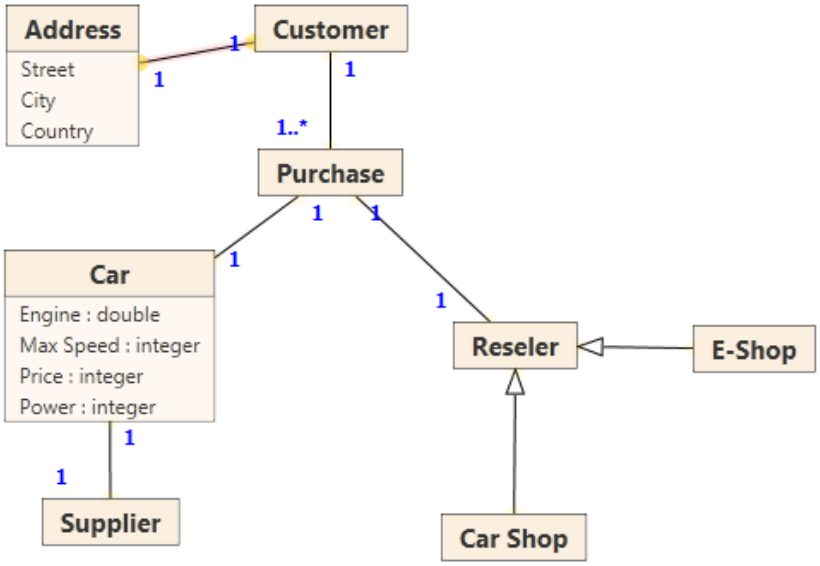


Figure 3.1: PIM diagram

contains PIM attributes. There are attributes *Street*, *City* and *Country* without defined data type and attributes *Engine*, *Max Speed*, *Price* and *Power* with defined data type.

In the example are used two types of relations: generalization (*Car Shop* to *Reseler* and *E-Shop* to *Reseler*) and association (*Address* to *Customer*, *Customer* to *Purchase*, *Purchase* to *Care*, *Purchase* to *Reseler* and *Car* to *Supplier*). The associations are assigned with multiplicities.

3.2.2 Platform Specific Model

A PSM diagram is also a UML class diagram (see Figure 3.2) extended with specific constructs for XML modeling. However, a PSM diagram is a tree which can be translated to a representation in an XML schema. In addition, multiple PSM diagrams can be derived from a single PIM diagram (see Figure 3.2).

A PSM class c_{PSM} represents a PIM class c_{PIM} and specifies how instances of c_{PIM} are represented in the modeled XML format. A PSM association goes from parent PSM class c_{PSM1} to child PSM class c_{PSM2} and

represents a path between PIM classes represented by c_{PSM1} and c_{PSM2}.

c_{PSM} can be assigned with a *label*. If c_{PSM} has label l, it is translated to element with name l. c_{PSM} can be assigned with a content as a *choice*. In that case it specifies that for all classes listed in the choice only one association to these classes can be instantiated. Another special construct called structural representative is a PSM class that inherits attributes and content from other PSM class. The choice between *NewCustomer* and *ExistingCustomer* can be seen on right diagram on Figure 3.2. This diagram also shows the structural representative *NewCustomer* which inherits from *ExistingCustomer*. The generated XSD from this PSM diagram is shown in Figure B.2.

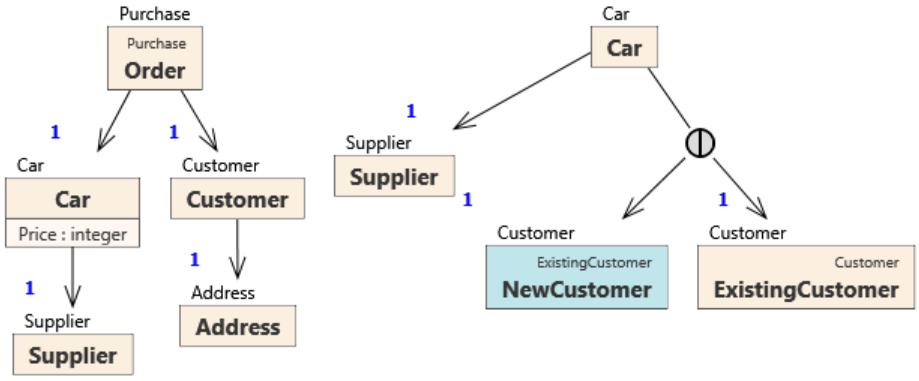


Figure 3.2: PSM diagram

In the left diagram in Figure 3.2 there are 5 PSM classes: *Address*, *Customer*, *Order*, *Car* and *Supplier*. The class *Order* represents the PIM class (see Figure 3.1) *Purchase*. The rest of the classes represent PIM classes defined in PIM diagram according to their name.

XSD generated from the left PSM diagram in Figure 3.2 is shown in Figure B.1.

The associations between PSM classes represent the associations between PIM classes except the association a_{PSM} between *Car* and *Customer* in the right PSM diagram. This association represents the path $Car - Purchase - Customer$.

Chapter 4

Related Work

4.1 Reverse Engineering

In paper [19] the process of reverse engineering for conceptual modeling is proposed. It describes a situation when an XSD and a PIM diagram is given as an input and the algorithm searches for optimal matching between them. In Figure 4.1 [10] the XSD represents *Schema Level* and the PIM *platform independent level*. The levels in this architecture are connected and it allows to propagate the changes through these connections. The aim of this work is to create connections between them. In particular we improve the solution proposed in [19] to increase precision of matching and the efficiency of this process.

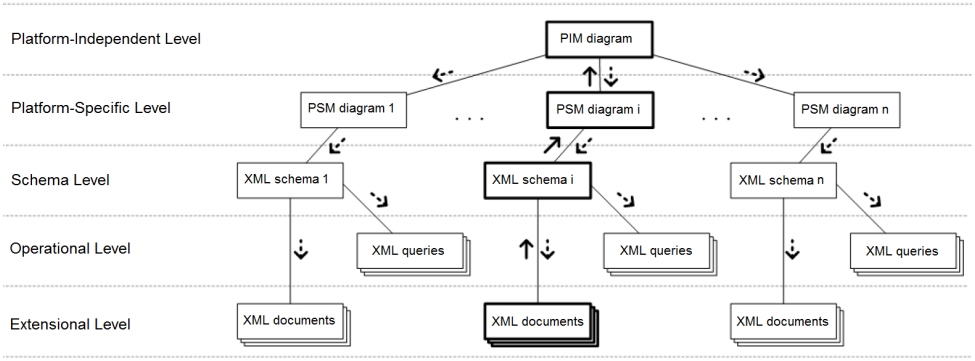


Figure 4.1: Five level XML evolution architecture

The first part of this algorithm computes an initial PSM diagram from the XSD. This is done by converting all element definitions with complex type into classes and element definitions with simple type into attributes. Note that only sequences of elements or choices are considered. The attribute declaration with name n is converted to a PSM attribute of class c_{PSM} with name n.

The second phase consists of four parts of semi-automatic conversion.

- Class Mapping Estimation – Each class c_{PSM} is compared with each class c_{PIM} from PIM diagram. The similarity of their names and names of their attributes is computed. Structural comparison between neighbors of c_{PIM} and children of c_{PSM} is exploited as well.

- Class Mapping – Mappings ordered by estimated similarity are offered to a user who selects the optimal ones.

- Association Mapping - In this step a mapping between associations in PIM diagram and PSM diagram has to be determined. The list of available paths is again provided to a user for final decision. The paths are ordered by their weights.

- Subtree Mapping – In the last step element attributes and subtree elements are recursively mapped.

This work proposed basic algorithm to find correspondences between XSD and PIM. In this thesis we will try to improve this algorithm by flexible selection of the methods used for class mapping estimation and association mapping.

4.2 XCase

XCase [27] implements XML conceptual model XSEM. The work is organized to *Projects* which contain PIM diagrams and one or more PSM diagrams. Applying XSEM we first model a schema in a PIM where we provide an XML-independent conceptual description of a given problem domain. On the base of the PIM schema we model one or more schemas in PSM where we specify how the data is to be represented in concrete types of XML documents.

UML class diagrams are used as PIM and a small augmentation of this model as PSM. Applying MDA in an advanced way, XCase greatly facilitates not only the design but also the maintenance of the XML schema.

The main goal of XCase project is to examine possibilities of XSEM and conceptual modeling for XML in general. The tool is original, because neither a conceptual model for XML nor a tool for conceptual modeling of XML has been developed neither in commercial nor academic conceptual modeling community to our best knowledge. However, the other works which provide constructs to model XML data are limited as discussed in [18].

We wil use XCase as a platform for implementation of the algorithm introduced in this thesis.

4.3 Schema Matching

This section is based on technical report [16] which describes many techniques for XML similarity measure and matching problem. This report offers an overview for three variants based on input data.

- Comparison of two XML documents – Among others this part contains the most commonly used algorithms for tree edit distance and usage of root paths.
- Comparison of an XML document and an XML schema – The least explored variant.
- Comparison of two XML schemas – Because of the nature of this work we will focus on the third variant. There are described many schema integration systems and techniques for clustering.

We consider a schema matching problem as a problem of similarity of two sets of regular expressions.

The general idea of schema matching is to create various *matchers*. A *matcher* is a function which evaluates similarity of a particular feature. The similarity sim is defined as a value from interval $[0; 1]$ where $sim = 0$ means no similarity and $sim = 1$ is maximal similarity. The results of all matchers are aggregated to get the overall, so-called composite, similarity.

4.3.1 COMA

COMA system [8] has been designed as a system for investigating of composite match approaches. It supports many input schemas and has a large library of match algorithms. The examined data can differ and the selection of specific matchers and the strategy of order of their usage can affect the final result. For such purposes COMA allows to combine those algorithms or create different matching strategies.

A user can specify the match strategy, i.e. select the matchers and their combination and accept or refuse match candidates.

COMA works with directed acyclic graphs S_1 and S_2. The match operation takes two graphs and computes which elements corresponds to each other. From different *matchers* a similarity cube is computed. A similarity cube consists of 3-tuples (two fragments $f_1 \in S_1, f_2 \in S_2$ and a result of a j-th matcher $sim_j(f_1, f_2)$. The gathered results are finally aggregated and the matching candidates with the highest similarity value are selected.

COMA supports many different techniques for measuring similarity and it is easily extensible. Some of these matchers are listed in Section 4.4. The matchers are in COMA divided into three categories:

- Simple matchers – The matchers are oriented on elements. This category includes element, attribute or data type names. Representatives of this group are n-grams, affixes, suffixes, edit distance, synonyms, data types or user feedback.

- Hybrid matchers – The matchers are used as a combination of simple matchers and/or other hybrid matchers to get more accurate results. For example *element name* hybrid matcher is proposed as a simple name matcher combined with tokenization and expanded abbreviations and acronyms. The group also includes structural matchers. For example *NamePath* matcher uses concatenated element names from all nodes on the path from the root node.

- Reuse-oriented – The matchers are motivated by situation when the currently analyzed schema is the same or at least similar as the previously analyzed schema.

COMA++ [2] is new version of COMA. This version improves system architecture and brings some new features. Main improvements lies in more effective implementation of matchers, uniform support for differen languages

(including SQL, OWL and XSD), repositories of schemas and a fragment-based approach.

The idea of schema matching with various matchers which are combined into the composite similarity is widely used and there are many other approaches based on this idea (e.g. [12]).

4.3.2 Similarity Flooding

Similarity flooding [14] is an iterative algorithm to compute graph matching. Unlike the many approaches based on matchers and aggregation it results from an assumption that if an element is similar, then its adjacent elements are also similar and their similarity value is increased.

The algorithm works in the following steps:

1. Create directed labeled graphs from given XML Schemas. Each edge represents triple: a source node, a target node and a label of the edge.

2. Between these graphs the initial mapping is computed using a string matcher. This matcher is based on comparison of common prefixes and suffixes.

3. The next step is the iterative part of the method. When two nodes are similar the similarity of adjacent elements increases. This step is repeated until all model elements stabilize which means that addition is bellow a fixed value.

4. The last operation of the algorithm selects a subset of node pairs with 'most plausible' matching entries.

4.3.3 Decision Trees

In [6] a flexible way of similarity measure is proposed as a possible improvement for COMA. The presented matching strategy is based on three aspects:

1. Performance – there are many similarity measures which can be used on different data. If we consider that we want to use k measures on m source nodes and n target nodes, we will have to compute $m \times n \times k$ similarity values. From the values not all are necessary. So the authors want to minimize performance needed for the computations.

2. Quality – as mentioned above, some computations are useless. They can even degrade right results. For instance if we use average as the aggregation function, we can find out that after one highly ranked measure and one low ranked measure from same group we get relatively small similarity value. So it is a good task to improve quality of final values.

3. Flexibility – finally it is important to say that data are not same. We should analyze the data from different domains with different demands. Selection of proper metrics is crucial.

For the reasons mentioned above, the authors proposed to exchange the common aggregation functions for decision trees.

Definition 3. *A* decision tree *is a tree whose internal nodes are the similarity measures, and the edges represent conditions on the result of the similarity measure.*

The trees contain instructions and the order for a matching system. Another selected matcher is based on the result of a previous matcher.

Value v, specified by each node, selects which children *matcher* will be called. The children count is not limited, so it is possible to set match for $v > 0.8$, mismatch for $v < 0.2$ and call other matcher for $v \int (0.2; 0.8)$.

The usage of tree can result in distribution of similar metrics (like Levenshtein and 3-grams) into distinct parts of the decision tree and thus they will not interfere as we can see in Figure 4.2 [6]. There is one precision based decision tree. It contains several matchers for string similarity measure and one matcher for context measure. The string *matchers* are used only if the parent matcher does not find match.

The algorithm begins with two input elements with assigned name and context in the node with *Equality* matcher. It computes the similarity value of the names and on the basis of the result it decides for a child node. If the names are equal there is no need for more string similarity measures and the *Context* matcher is selected, otherwise the algorithm continues in the *Label Size Sum* node. The algorithm continues until it reaches the leaf node and decides whether the elements match or not.

The main part of this approach lies in generation or modification of the decision trees to get optimal results for different domains. This can be done partially by a domain expert who modifies the decision tree by requirements of specific domain or by machine learning with user feedback.

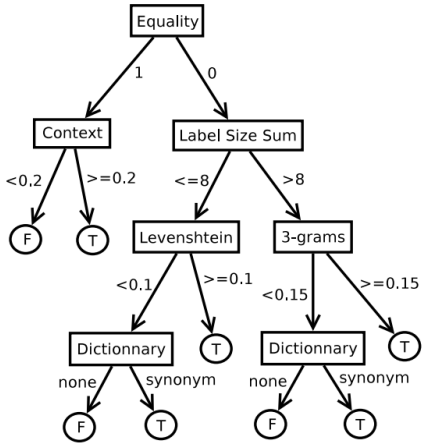


Figure 4.2: Decision tree example

4.3.4 Usage Driven Similarity

Usage driven similarity [7] represents a different approach. Instead of analyzing only data, the analysis focuses on queries, i.e. operations with the data.

The algorithm works in two phases: feature extraction and matching. It is based on so-called *structure level features* (SLUB) and *element level features* (ELUB).

Structure level features represents relationships between attributes used in queries in same schema. An attribute can be used as a result if used in SELECT clause, can filter data in WHERE or HAVING clause, can be used as aggregator in GROUP BY clause or can be used as order attribute in ORDER clause.

A usage relationship is also defined if two distinct attributes are used in the same role. Thus 4 roles are defined and we get total of 16 possible defined combinations of roles.

From these relationships a graph is created with attributes as nodes and relationships as edges. A weight of the edges is defined as a frequency of their respective relationship.

Three different query types are excepted:

- SPJGO – Single-block Select-Project-Join queries with optional grouping and ordering. This query is parsed as it comes.

- SPJGO-UEI – SPJGO query with Union, Except and/or Intersect are parsed for each sub-query separately.
- SPJGO-N – SPJGO queries with Nested sub-queries is at first parsed as SPJGO and then new relationships are found between outer attributes and inner attributes on specific positions.

Except *structure-level features*, there are also proposed so-called *element level features*. These features comprise a specific usage of each single attribute. For example, if the attribute is often used in join clause, we can expect it should be defined as a key in schema (see Section 2.3).

Beside feature extraction, matching and scoring functions are proposed as a way to select from given graph the best matching candidate.

4.3.5 Domain – Schema Similarity

Paper [11] is aimed at a different situation than the previous ones. Previous works always compare two semantically equal schemas. This work covers a situation when we have one domain XML schema and a large number of source XML schemas.

The similarity measure contains three important factors.

1. Ratio of interesting objects (RIO) – It is a ratio between nodes which are both in domain and input schema.
2. Cardinality similarity of node pairs (CSNP) – It is set to 1 when the two node pairs have consistent relative cardinality; otherwise it is set to ω. ω represents the degree that user can tolerate with the cardinality difference.
3. Similarity of node pairs (SNP) – This value is based on relationship of the examined pairs. If both pairs are in relation child-parent or descendant-ancestor, the value is set to CSNP. If one pair is in relationship child-parent and one descendant-ancestor, then the value of CSNP is multiplied by user-defined constant. If both pairs are siblings, the value of SNP is set to 1.

This approach provides some additional improvements that can increase the overall system performance. Consider domain schema with set of nodes V and input schema with set of nodes V_i. It detects unimportant nodes $v \in V \setminus V_i$ and removes them by usage of three simple rules:

1. If node v is root node, then all the child edges need to be deleted.

2. If node v is a leaf node, then the parent edge needs to be deleted.

3. If node v is an internal node, then: (1) for all child nodes v_i, the cardinality of v_i is updated as maximum of cardinality between v and v_i; (2) all edges relating to node v are deleted; (3) new edges linking the parent node and all the child nodes are inserted and re-labeled.

After application of these rules, the result becomes a schema tree or a forest of schema trees. A new coding schema is proposed to speedup the computation. With these prepared trees the SNP value is computed.

4.4 Similarity Matchers

In this section we will list some common similarity *matchers* [23].

4.4.1 Element Level

String Based

This group of similarity metrics works with words as sequences of letters. These metrics are often used to match names or descriptions. They are simple but, in many cases, give very good estimation. These methods works in a very similar way. From two strings they produce a similarity value.

- Prefixes and suffixes – The first method takes two strings s_1 and s_2 and checks whether the s_1 starts with s_2. Suffixes solve the same situation only check whether s_1 ends with s_2.

- Edit distance – This method takes two strings s_1 and s_2, compares them and finds the shortest edit distance. Edit distance represents the number of operations which have to be done to transform s_1 into s_2. Particular methods can differ in operations used. Commonly used operations are insert character, delete character and substitute character. It is possible to set the different weight for each operation and use this to penalize them.

  Example 1. *The similarity value between words* horse *and* mouses *with weights 1 for deletion and insertion and 2 for substitution is 5.*

If we lower the substitution weight to 1 (for example because we expect typing errors) the similarity value lewers 3.

- N-grams – This method takes two strings s_1 and s_2 and computes the number of the same n-grams. An *n-gram* is a sequence of n characters in a string. The n-grams from s_1 and s_2 are created and stored in set S_1 (S_2 respectively). These sets are compared and the final value is ratio between matching n-grams and all distinct n-grams.

  Example 2. *The 3-grams for word horse are hor, ors and rse.*

Language Based

The previous methods works well but they do not work with natural language wheras the names are usually defined by human. The methods understand meaning of input strings and transform them to forms more suitable for further processing.

- Tokenization – This technique is usually used when multiple words are used in a single element or attribute name. The words can be recognized by blank characters, punctuation or cases. This method splits the words into independent tokens. For each of them other listed methods can be then used.

  Example 3. *The sentence* The brown dogs sleep. *is recognized as following tokens:* the, brown, dogs, sleep.

- Lemmatization – The input strings are lemmatized to get a basic form which is send to another processing.

  Example 4. *The word* dogs *is recognized as* dog.

- Elimination – Eliminates a word or words which do not have semantical meaning (e.g. prepositions or conjunctions) which can highly boosts total efficiency.

  Example 5. *The word* The *would be removed by elimination.*

Thesauri

This group is focused on the meaning of the examined strings.

- Common knowledge thesauri – This is an important linguistic method. It uses a lexical database of a specific language and can be used for synonym or homonym search.

  Example 6. *The word* brown *is assigned following synonyms in the WordNet [26] database:* brownish, chocolate-brown, dark-brown.

- Domain specific thesauri – Apart from dictionaries based on natural language dictionaries bound to a particular domain can be used.

Constraint Based

Constraint based techniques are methods which consider internal limitations of the given environment.

- Datatypes – Comparison of other attributes can help to understand more aspects of the compared classes. Data type recognition can be done not only at the semantic level, but it is also possible to compare hierarchical level and inheritance of the data types.

  Example 7. *The* integer *data type is similar to* short *data type.*

- Multiplicity – Another constraint-based technique which compares cardinality of the given objects.

4.4.2 Structure Level Techniques

The techniques work with labeled graphs. The similarity value is based on similarity value of child or parent nodes.

- Children - This method compares two nodes and computes the similarity on the basis of similarity of their children sets.

- Leaves - Unlike the previous method this method compares the leaf sets. Two nodes are similar if their leaf sets are similar.

Chapter 5

Algorithm

The main goal of this thesis is to design an appropriate algorithm which will be able to find mappings between PSM and PIM diagrams. We use the XML Schema as a platform for PSM. The algorithm is designed with great emphasis on maximum efficiency. The result should be checked by a domain expert who will control the found mappings and will be able to simply repair or decline wrong decisions.

For maximum effectiveness, we designed a solution which uses the data from the previous calculations and tries according to the results of new improved calculations. These calculations include both analysis of the PIM diagram in terms of individual classes and the structural analysis. The algorithm also aims at data preparation for comparison algorithms and seeking for the shortest path.

In this section we will first describe the process of analysis and preparation of the PIM diagram data suitable for subsequent calculations. We will show the creation and usage of decision tree for algorithm control. Next, we will focus on comparing the similarities between PIM and PSM classes. We will introduce new method for path searching. And, finally, we will describe the system for obtaining the input from a user.

5.1 Overview

In Algorithm 1 we can see the simplified process of PIM to PSM mapping over a set of XSD schemas F_{XSD} and a user defined PIM diagram d_{PIM}. By PIM diagram we mean a set of PIM classes C_{PIM} and relations between them. The relations can be associations, aggregations or generalizations. On

the other hand, the set of XSD schemas is expected, which are matched during the computation process to PIM. We also expect a set of methods M to be used for the initial analysis.

The first step of the algorithm is obtaining *analysis* data from PIM diagram d_{PIM}. This process is shown in Section 5.2. The result of this analysis is used by *buildTree* method for creation of decision tree T as we will describe in Section 5.3. Each XSD schema $f_{XSD} \in F_{XSD}$ is then converted into PSM diagram d_{PSM} by *createPSM* method. We use the *depth first search* (dfs) to get the classes in PSM diagram, i.e. these classes are analyzed from leaf nodes to root nodes. Each PSM class $c_{PSM} \in dfs(d_{PSM})$ is matched against d_{PIM} by *computeSimlaries* method with usage of decision tree T. And the user then selects the optimal one in *classSelection*. In a similar way the paths are computed and offered to user. In last step the *pathSelection* is used to get the right path from user.

Algorithm 1: XSD to PIM mapping

Input: a set of XSD schemas F_{XSD} and a PIM diagram d_{PIM}
Output: mapped PSM diagrams to PIM diagram

```
1:  begin
2:      analysis = analyze(d_PIM)
3:      T = buildTree(analysis)
4:      foreach XSD schema f_XSD ∈ F_XSD do
5:          d_PSM = createPSM(f_XSD)
6:          foreach PSM class c_PSM ∈ dfs(d_PSM) do
7:              similarities = computeSimilarities(c_PSM, d_PIM, T)
8:              match[c_PSM] = classSelection(similarities)
9:              paths = computePaths(c_PSM, d_PSM, d_PIM)
10:             path[c_PSM] = pathSelection(paths)
11:         end
12:     end
13:     return (match[c_PSM], path[c_PSM])
14: end
```

5.2 PIM Analysis

The first part of the algorithm is the analysis of PIM diagram. This analysis is one of the most important parts. The shape of the decision tree and the procedures and methods, which are then used for further calculations depend on this analysis. The analytic methods, so-called *analyzers*, work with *features*.

Definition 4. *A* feature *is a measurable characteristic common for PIM and PSM.*

Definition 5. *An* analyzer *is a method m_{ana} which takes as input a PIM class c_{PIM} and analyzes a feature. Each analyzer is assigned with its feature.*

One feature can be analyzed by more analyzers. The result of the analyzer is a number, called threshold *which is used to determine whether feature specific matchers will be used. The analyzer can be also used to gather some method specific data for matchers. Both return values are optional.*

Then we compute and aggregate the similarity of features to get the overall similarity. In this text we use the following features of PIM diagram and PIM classes:

- Class and attribute name
- Attribute data type
- Class and attribute occurence
- Descendants similarity

For the analysis of PIM we differentiate two main types of analyzers.

- Element-based analysis – It focuses primarily on data types or classes of names and it computes statistics over all classes contained in a diagram.
- Structure-based analysis – It focuses on the structure of a PIM diagram and prepare data for structural similarity matchers and path selection.

Beside these methods we use data-preparation methods for preparation of PIM classes. These will help with further computation as described in the following section.

Our proposed algorithm has three main parts as we can see in Algorithm 2. The algorithm gets as input PIM diagram d_{PIM}. For each PIM class $c_{PIM} \in d_{PIM}$ we first call each of the data-preparation methods m_{pre}. Then, the element-based methods m_{ana} are called to get the analytic data from each c_{PIM}. The last step of analysis are structure-based methods m_{str}. These methods work with d_{PIM} and are able to get structural information. The algorithm returns computed *thresholds* and method specific data for similarity computation.

Algorithm 2: PIM analysis

Input: a PIM diagram d_{PIM}, m_{pre}, m_{ana}, m_{str}
Output: key-value pairs *analysis* with analyzed results

```
1:  begin
2:      foreach PIM class c_PIM ∈ d_PIM do
3:          foreach m ∈ m_pre do
4:              analysis[m] = m(c_PIM)
5:          end
6:          foreach m ∈ m_ana do
7:              analysis[m] = m(c_PIM)
8:          end
9:      end
10:     foreach m ∈ m_str do
11:         analysis[m] = m(d_PIM)
12:     end
13:     return analysis
14: end
```

5.2.1 Data Preparation

Data-preparation methods take PIM classes and expand them to structures which can be used in further analysis. They do not analyze the data itself but create specialized data for analytic methods and can influence the final result.

Bellow is a list of data preparation methods which we implement in this work.

- Tokenization – This method takes all names (class, attribute, data

type) and splits them accordingly into basic recognition symbols. We search for capital letters, underscores and other punctuation symbols. The split names are then stored beside original names and are used in all other methods.

- User-specific thesauri – Another generally used method to improve efficiency is to create a specific limited thesauri we allow the user to include a domain specific thesauri. In many cases the domain specific thesauri differs in many ways from widely used ones. This extension allows the domain expert to set his own thesaurus for better understanding of the domain.

5.2.2 Element-Based Analysis

The element-based analysis gathers important information from PIM classes. This comprises different attributes like class name or specified data types. This process is designed to be maximally transparent so new analytic methods can be simply added which allows to get more information for further computations.

Beside this analysis, we also prepare data structures which will be used in next steps and, in general, they have to be calculated. The data depend on the specific analytic method but, in general, we mean data which can influence whether the method should be used in computation or how it will work with additional comparisons. If the analysis found that the comparison method would have bad or none results, it is not used during decision tree creation.

Each method will analyze each of the PIM classes and it depends entirely on it what information it uses and how. It can also return the threshold value. It is a value that influence the selection of methods used in decision tree and during comparison. Threshold is a method-specific value which can be modified by the user.

Below is a list of basic analyzing methods that are implemented directly in the experimental application.

- Length – There are several properties we can observe on class names. String length is one of the properties. We count the number of letters used in names and on the basis of average length we can adjust used methods for the best performance ratio. This metric is very important if we want to compare used abbreviations between PIM and PSM.

- Data Types – Another important aspect to get is data types usage. On the basis of this analysis, data type comparison is included in the final decision tree. This method takes all PIM classes and attributes and gathers information about their defined data types.
- Occurrences – The last but not least analysis focuses on usage of distinct occurrences defined on PIM attributes and on relations between PIM classes.

5.2.3 Structure-Based Analysis

The structure-based analysis considers the whole PIM diagram and suggest the best strategy for path searching.

Floyd-Warshal Algorithm When we prepare PIM diagram, we primarily count distances between all vertices. We use the Floyd-Warshall algorithm [9] to compute the shortest paths between PIM classes. However, we make a small modification to be able to gain the whole path itself not only path length.

The Floyd-Warshall is an algorithm which compares all possible paths between each pair of vertices u and v in a weighted graph. The algorithm is based on typical usage of a dynamic programming method.

In this thesis we use this algorithm on PIM diagram. The PIM classes are used as vertices and PIM associations as edges with weight 1.

Consider a weighted graph G, vertices V numbered 1 through N and method $dist(u, v, w)$. The method $dist(u, v, w)$ returns the length of the shortest path between u and v with usage of vertices $\{1...w\}$ and it is computed as a minimum of $dist(u, v, w-1)$ and sum of lengths of paths which contain w.

$$dist(u, v, w) = min\{dist(u, v, w-1), dist(u, w, w-1) + dist(w, v, w-1)\}$$

Dense Subgraph Searching

Definition 6. Density d *of a PIM diagram with PIM classes* C_{PIM} *and PIM associations* A_{PIM} *is defined as* $d = \frac{|A_{PIM}|}{\binom{|C_{PIM}|}{2}}$.

For large PIM diagrams with high *density* d of associations the selection of the paths becomes very complicated. Hence, we propose a method based

on [1] that will lower the number of nodes in graph and prepare data for path selection.

This method works with PIM diagram as a graph and it is based on dense subgraph detection. It finds a dense subgraph and merges its nodes to a single node so-called *cloud*. We call this method repeatedly to decrease the number of nodes in graph under the user defined value. This modified graph is then used for path selection and user involvement (we will describe path selection in Section 5.8). The merged graph allows to use bfs algorithm to find all paths between mapped PIM paths. We do not use this method for small graphs and graphs with small density of edges.

Definition 7. *A* cloud *with defined density*d *is a set of PIM classes with the density of edges between them is higher or equal as* d.

The size of *cloud* depends on density value. The value should not be too high to find small number of small subgraphs but, on the other hand, it should not be too low to find only one large subgraph. We found out that the optimal value is 0.7.

In Algorithm 3 we have a PIM diagram and a density value d on input. We begin with empty node set S and define another density value d_0 for current density of S. In each step we compare whether the set still meets the requirements on density. Then the *selectGamma* method is called. This method tries to get new node which satisfies the current density. If no such node is found another node from C_{PIM} is selected and d_0 is recomputed. When there are no nodes or the density is too small S is returned subgraph.

Example 8. *There is a PIM diagram in Figure 5.1(a). We can see its cloud representation in Figure 5.1(b) where* $A = a, d, g, h, i$, $B = j, k, l, m$ *and* $C = b, c, e, f$.

Example 9. *In this example we will describe the situation of PIM analysis on the diagram in Figure 5.2. We will use the* Tokenization *as method for preparation. For element-based analysis we will use* Length, Data Type *and* Occurence *analyzers (see Section 5.2.2) and for structure-based analysis we will use the Floyd-Warshall algorithm as described in Section 5.2.3.*

First Tokenization *will split the class, attribute and data type names into separate tokens as shown in Table 5.1.*

The Length *analyzer will compute the average length of used names, the* Data Type *analyzer will investigate the usage of distinct data types and the* Occurrence *analyzer will compute number of distinct occurences defined on PIM classes and attributes. The whole results are described in Table 5.2.*

Algorithm 3: Dense subgraph search

Input: A PIM diagram d_{PIM} with classes C_{PIM}, density d
Output: Dense subgraph

```
1:  begin
2:      d_0 = 1
3:      x = select(C_PIM)
4:      S = {x}
5:      while d_0 ≥ d do
6:          x = selectGamma(C_PIM, d_0)
7:          if x = 0 then
8:              return S
9:          end
10:         S = S ∪ {x}
11:         d_0 = |E(S)| / (|S| choose 2)
12:     end
13:     return S
14: end
```

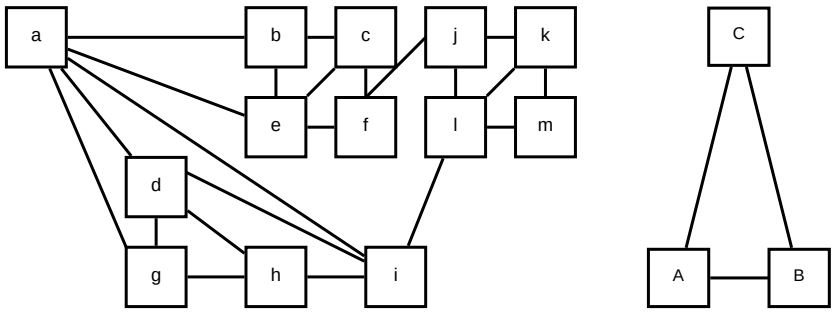
Name	Tokens
SiteOwner	Site, Owner
MediaAgenture	Media, Agenture
GraphicDesigner	Graphic, Designer

Table 5.1: Tokenization

5.3 Decision Tree

The decision tree creation is based on the PIM analysis in the previous step. The algorithm works with matchers as defined in Section 4.4. They are separated into groups by the feature they compare. Beside this division, each method is assigned with a priority which is used to get the efficient methods on top, wheras the time-consumpting methods are used only if others fail. The priority also selects the order of examined features. Each method has a default priority and the user can modify this value to get different tree.

As we can see, Algorithm 4 works with methods grouped by feature and sorted by priority in $methods_{feat}$. As the root method is selected the method



(a) Classes (b) Clouds

Figure 5.1: Cloud creation

Method Name	Threshold	Data
Length	–	Average length of name: 6.8
Data Types	50%	Distinct data types: 6
Occurrence	7.89%	Distinct occurrences: 3

Table 5.2: Used analyzers

with the highest priority. The rest of methods from the same feature group are added by *addMethodToTree(node, method)* method. This method takes node n_s in decision tree and tries to add new method m_n. We recognize the following possible situations.

- n_s has no child – m_n is added as a child of *node* (see Figure 5.3(a))
- n_s has children $c_1, ..., c_n$ with the same feature as the m_n – If the method has the same priority, it is added as a child of n_s (see Figure 5.3(b)), otherwise $addMethodToTree(c, m_n)$ is called for each $c \in \{c_1, ..., c_n\}$
- The n_s has children $c_1, ..., c_n$ with different feature than m_n – $addMethodToTree(c, m_n)$ is called for each $c \in \{c_1, ..., c_n\}$ and if feature of $c_1, ..., c_n$ is same as feature of n_s, the m_n is added as a child of n_s (see Figure 5.3(c))

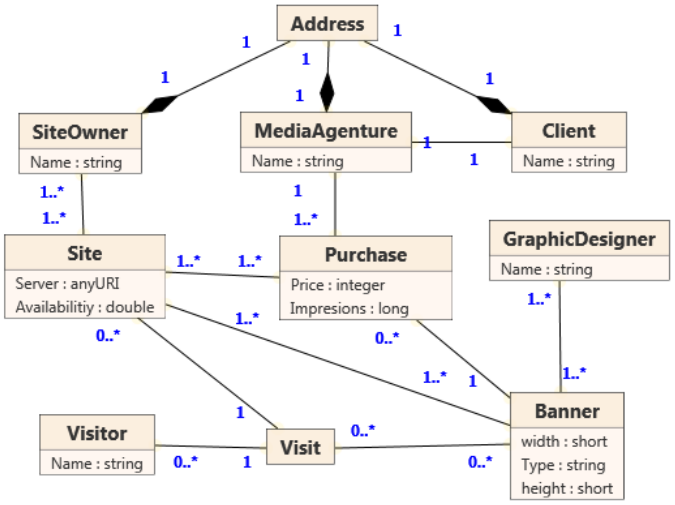


Figure 5.2: Analyzed PIM diagram

5.3.1 Matchers

We use the following methods for node comparison:

Matched Thesauri This method works with results of previous computations. The thesauri contains matched pairs. str_{PIM} is the name of a PIM class and str_{PSM} the name of a PSM class. It checks whether the pair (str_{PIM}, str_{PSM}) was a user-confirmed match. The return value of this method is 0 when no match is found or 1 if the thesauri contains the pair. We used this value because the mapping war already confirmed, so the algorithm expects there is no need for further comparision.

Prefix This method takes two strings str_{PIM} and str_{PSM}. It checks whether the shorter string is the abbreviation of . The strings str_{PIM} and str_{PSM} are separated to the tokens a_1 a_2 ... a_n and b_1 b_2... b_m and the value of similarity is computed as a ratio between the size of $S = \{(a_i; b_j) |\ a_i = abbreviation(b_j) || b_i = abbreviation(a_j)\}$ and maximum number of tokens in str_{PIM} and str_{PSM}.

Algorithm 4: Decision Tree Creation

Input: *methods* sorted by priority
Output: decision tree T

```
1:  begin
2:      root = nil
3:      foreach methods_feat in methods do
4:          if root = nil then
5:              root = methods_feat[0]
6:          end
7:          foreach method_f in methods_feat do
8:              addMethodToTree(root, method_f)
9:          end
10:     end
11:     return root;
12: end
```

$$E_{pref}(a,b) = \frac{size(S)}{max(n,m)}$$

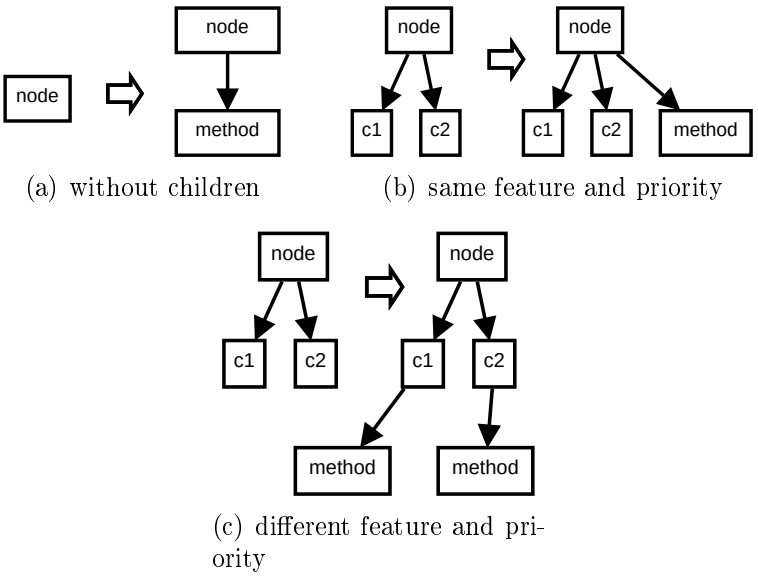
Levenshtein This algorithm computes the shortest edit distance for operations insert, update and delete a character from str_{PIM} to str_{PSM}. The value of similarity is computed as a ratio between the number of same characters and the length of compared strings.

$$E_{lev}(str_{PIM}, str_{PSM}) = \frac{2\times(max(str_{PIM},str_{PSM})-levenshtein(str_{PIM},str_{PSM}))}{len(str_{PIM})+len(str_{PSM})}$$

Length Ratio This method computes length ratio between string str_{PIM} and str_{PSM}. The similarity value is computed as a ratio between length of these strings.

$$E_{len}(str_{PIM}, str_{PSM}) = \frac{min(len(str_{PIM}),len(str_{PSM}))}{max(len(str_{PIM}),len(str_{PSM})}$$

Note that we decreased the importance of this matcher, because it is not precise method. We use it as a method which determines whether to use Prefix matcher for strings with high length ratio or Levenshtein matcher for strings of similar length.



(a) without children (b) same feature and priority

(c) different feature and priority

Figure 5.3: The decision tree creation

Data Type In this method we compare the types for basic similar types like numbers – doubles and floats. The similarity value is in this case composed of the value of data type name and the data type relation. The values are multiplied. The value of data type name is computed in the same way as in case of class and attribute names and the similarity of data type depends on their relation. For similarity of basic and derived data types we used the modified Type Tree as proposed in [15]. In particular is is extended with more abstract data types (*text*, *calendar*, *numeric*, *logical* and *other*) to create more coherent hierarchy (see Figure 5.4 [15]). The similarity value of two data types d_1 and d_2 is computed as a function of depth and the shortest path length.

It is defined as follows:

$$E_{dt}(d_1, d_2) = \begin{cases} e^{-\beta l} \times \frac{e^{\alpha h} - e^{-\alpha h}}{e^{\alpha h} + e^{-\alpha h}}, d_1 \neq d_2 \\ 1, d_1 = d_2 \end{cases}$$

where d is depth of the node which subsumes nodes d_1 and d_2, l is the length of path between them and α and β are the correction values. The optimal value, presented in [15], is recommended $\alpha = \beta = 0.3057$. The values for selected types are shown in Table 5.3.

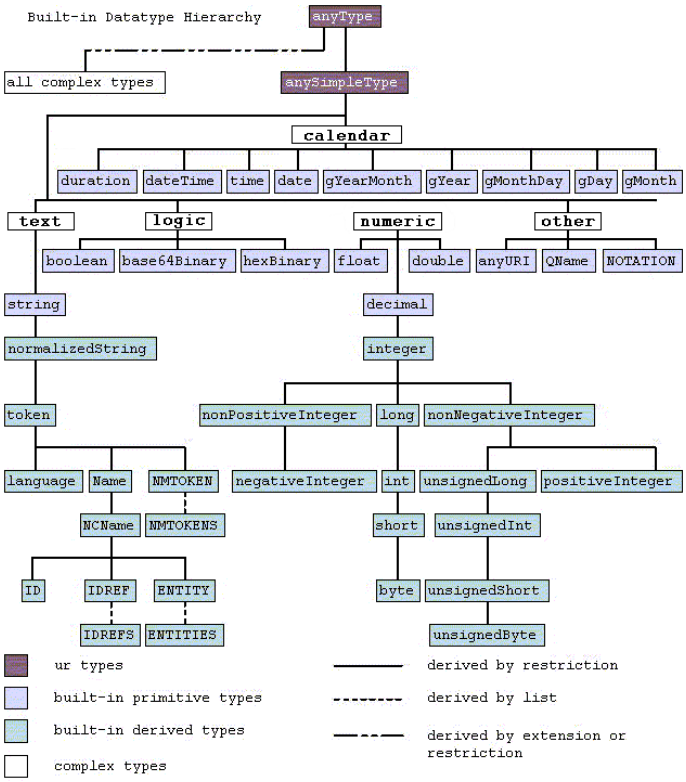


Figure 5.4: XSD Type Hierarchy

Thesauri In this method we make comparison of the name of a class against all items in user thesauri. In this case the value of similarity for synonyms is fixed, we use same value as suggested in [8] and it is 1.0.

Trigrams This method is an alternative to Levenshtein. The strings str_{PIM} and str_{PSM} are transformed to sets of trigrams T_{PIM} and T_{PSM} and the value of similarity is the ratio between $T_s = T_{PSM} \cap T_{PIM}$ and $T_a = T_{PSM} \cup T_{PIM}$.

$$E_{sim} = \frac{size(T_s)}{size(T_a)}$$

Children This method is a structural comparison method based on similarity of child nodes. We will compute the similarity between PSM node c_{PSM} and PIM node c_{PIM} by comparing their child nodes in PSM diagram ch_{PSM}

d_1	d_2	Depth	Path length	Similarity
int	long	5	1	0.67
double	float	2	2	0.30
string	unsignedLong	1	7	0.03

Table 5.3: Example of data type similarity

and neighbor nodes n_{PIM}. We compute the total distance $d(c_{PSM}, ch_{PSM})$ from c_{PSM} to all its children ch_{PSM} and the total distance $d(c_{PIM}, ch_{PSM})$ from c_{PIM} to all representatives of ch_{PSM}. The distance between two classes $dist(c_{PSM}, ch_{PSM})$ is defined as shortest path between them.

$$d(c_{PSM}) = \sum dist(c_{PSM}, ch_{PSM})$$
$$d(c_{PIM}) = \sum dist(c_{PIM}, representative(ch_{PSM}))$$

The final similarity value is the ratio between these distances, i.e.

$$E_{sim} = \frac{min(d(c_{PSM}), d(c_{PIM}))}{max(d(c_{PSM}), d(c_{PIM}))}$$

Example 10. *Considering Example 9 for the data analysis and the methods described in Section 5.3.1 in Table 5.4, we will get the tree in Figure 5.5.*

Feature	Method Name	Priority
class name	Matched Thesauri	99
	Length Ratio	80
	Levenshtein	70
	Prefix	70
data type	Data Type	60
structural similarity	Children	20

Table 5.4: Used matchers

The matchers are grouped by the following features: class name, data type *and* structural similarity *and sorted by* priority. *The group* class name *has the highest priority so* Matched Thesauri *is used as the root node. The rest of class name matchers follows. Then the* Data Type *and* Children *matcher is used.*

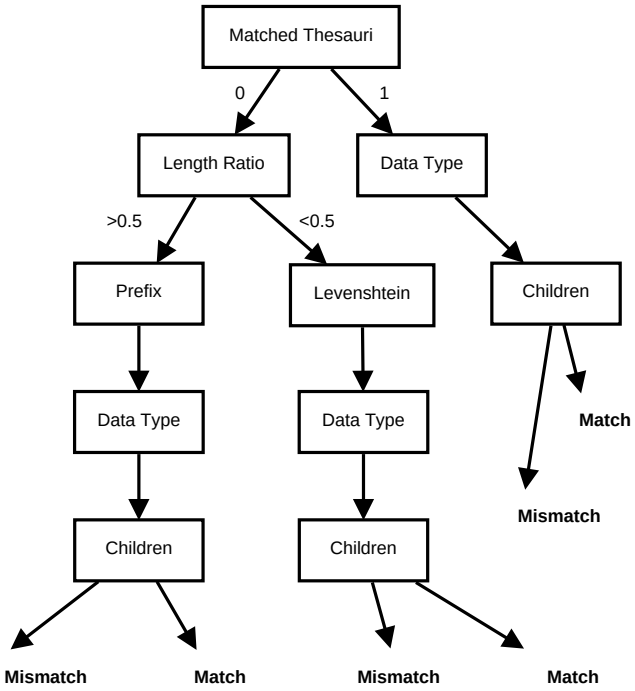


Figure 5.5: Decision tree

5.4 PSM Creation

In this section we briefly describe the mapping between XML schema and PSM diagram based on [19].

The translation of XSD schema s_{XSD} to PSM diagram d_{PSM} starts with globally defined elements. The elements with a complex data type are recursively parsed and translated to PSM classes. To simplify the algorithm we expect each complex type to be defined globally (there is a simple process to transform a locally defined type to global). The elements with simple type content are converted to attributes.

We want to transform element E with defined type T and label l to PSM class c_{PSM}. If T was not already processed, we will create c_{PSM} with the name of T and label of l. If T has been processed as PSM class c'_{PSM} and is shared with other element E', we create new structural representative of c'_{PSM} with label l.

T is translated with all attributes and content model. An attribute a

with name l and datatype t is transformed to PSM attribute a_{PSM} of class c_{PSM} with name l and type t. Note that XCase currently does not support creation of extended or restricted data types so t is transformed as a new type. The operators in content model we currently support are *sequence* or *choice*. Each of them is translated to content model of c_{PSM} and inner element declarations are translated recursively. The *sequence* is translated as a list of PSM classes with a specified order. The *choice* is translated as a PSM content choice, i.e. only one of the PSM children can be instantiated.

Example 11. *As an example we provide the translation of XML schema S_1 (Figure B.3) to PSM diagrams d_{PSM1} (Figure 5.6).*

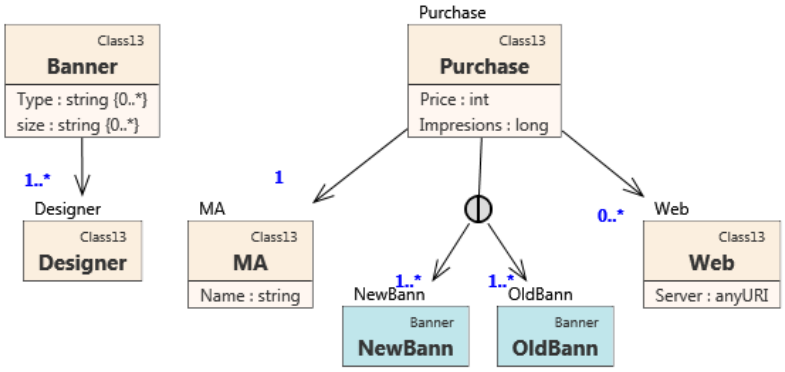


Figure 5.6: PSM diagram

5.5 Similarity Computation

Now, we have analyzed data from PIM, created a decision tree and created a PSM diagram from an XSD. In this section we will use these items to compute similarity between PSM and PIM classes.

In Algorithm 5 we can see two parts. At Line 3 the array with all similarity values is computed by method *sim*. PSM class c_{PSM} is compared with each PIM class $c_{PIM} \in d_{PIM}$ with decision tree T. When all values are computed by *sim* the results are grouped by feature and the maximum for each feature is selected. The maximums are then aggregated at Line 7, where all values are flattened as an average value, i.e.

$$sim = \frac{sim_{f1}+sim_{f2}+...+sim_{fn}}{n}$$

Algorithm 5: Similarity Computation

Input: PSM Class c_{PSM}, PIM diagram d_{PIM}, decision tree T
Output: Vector sim with similarity values for c_{PSM}

```
1:  begin
2:      foreach c_PIM ∈ d_PIM do
3:          similarityValues = sim(c_PSM, c_PIM, T)
4:          foreach Feature f_n in SimilarityValues do
5:              sim_{f_n} = max(SimValues_Feat)
6:          end
7:          sim[C_PIM] = (sim_f1+sim_f2+...+sim_fn)/n
8:      end
9:      return sim
10: end
```

Algorithm 6 works with one PSM class c_{PSM} and one PIM class c_{PIM}. The classes are compared with the usage of decision tree T. The computation starts in the root of the tree T. The method in the node is used to evaluate similarity between c_{PSM} and c_{PIM}. The result is stored to array $simValues$ associated with the used method. The next child node is selected by the result of the similarity method.

The method *getChild* at Line 9 uses the similarity $simValues[method]$ value computed by current node $curNode$. On the basis of the construction proposed int Section 5.3 we expect that the decision tree node n has one or two children n_{c1} (and n_{c2}). One child is expected if n is the last node for a feature. Two children are expected if n_{c1} compares the same feature as n and n_{c2} compares any other feature (this child is used when the match is resolved). Situation when n has two or more children occurs when its child matchers have same priority and n is used to select which will fit for concrete situation (for example *LengthRatio* matcher).

Each decision tree node is assigned with a values, so-called *borders*, which defines which child is selected. For example the decision tree node *Length Ratio* on 5.5 has border 0.5, i.e. if the result of *Length Ratio* is lower than 0.5 the *Prefix* matcher is called, for result higher than 0.5 th *Leventshtein* matcher is used.

In case of differen child features, the first child is selected when we expect the similarity value is low thus the next matcher of the same $feature$ should be selected to try to find similarity in other way. The second child means we have high similarity value and thus we do not have to compute similarity for this feature.

Algorithm 6: Decision tree similarity computation

Input: PSM Class c_{PSM}, PIM Class c_{PIM}, decision tree T
Output: Vector $simValues$ of similarity values between c_{PSM} and c_{PIM}

```
1: begin
2:     curNode = root(T)
3:     while not leaf(curNode) do
4:         method = curNode-> method
5:         simValues[method] = method(c_PSM, c_PIM)
6:         curNode = curNode-> getChild(simValues[method])
7:     end
8:     return simValues
9: end
```

5.6 Mapping Selection

Once the values of similarity between classes are computed, the selection of suitable candidates can be done. This choice is let to the user who selects from the possibilities sorted by the degree of similarity.

This mapping is also stored for later calculations in case of multiple input files from the same source which use the same names for the same classes. It can be also used by structure level *matchers* (e.g. Children matcher).

Example 12. *The similarity values for c_{PSM} NewBann from Example 9 are shown in Table 5.5.*

5.7 Path Computation

This section deals with finding paths between classes that have already been mapped and a newly mapped class. Assume that we have currently mapped

	Ling	DataType	Structural	Total
Banner	53.85%	23.11%	100.00%	58.99%
MediaAgenture	30.00%	50.00%	25.00%	35.00%
SiteOwner	12.50%	50.00%	25.00%	29.17%
Client	15.38%	50.00%	12.50%	25.96%
Visitor	0.00%	50.00%	25.00%	25.00%
GraphicDesigner	18.18%	50.00%	0.00%	22.73%
Purchase	13.33%	3.00%	50.00%	22.11%
Site	0.00%	9.00%	50.00%	19.67%
Visit	0.00%	0.00%	50.00%	16.67%
Address	0.00%	0.00%	12.50%	4.17%

Table 5.5: Simiarity values for NewBann

a PSM class c_{PSM} and its children C_{ch}. We have to map the associations between c_{PSM} and C_{ch} to *paths* between their representing PIM classes. During later schema evolution this allows us to propagate changes of selected parts in PIM into already defined PSM without involvement of a domain expert.

On the basis of the previously prepared computations we can use directly the shortest path if there is only one connection between classes. This path was computed by the Floyd-Warshall algorithm during analysis of PIM diagram (see Section 5.2.3). If there are more paths the algorithm will use the bfs algorithm to find all available paths and sort them by by length.

The input graph of bfs depends on whether we reduced PIM diagram by the dense graph reduction as described in Section 3, or we use the original PIM diagram.

5.8 Path Selection

In the final step the system offers the best found path between two PIM classes to the domain expert. The paths are sorted by the computed length and number of clouds and allows the user to accept offered possibilities.

The presented paths depend whether the detection of dense subgraphs was used. In case we did not create modified graph, all paths are considered and sorted by length and offered to a domain expert to select the optimal one. Otherwise, the paths with merged nodes are offered. The domain expert

selects one of them and then specifies a node in each *cloud* which is used again to compute by bfs all paths in each *cloud*. The computed paths are sorted by length and offered to user.

Example 13. *Consider two PIM classes c_{PIM1} and c_{PIM2} and a graph with clouds A, B, C, D and E as shown in Figure 5.7. Each of these clouds represents complete graph K_3. We look for a path between $c_{PIM1} \in A$ and $c_{PIM2} \in C$. The algorithm will offer the paths $A-B-C$, $A-E-D-C$, $A-B-D-C$ and $A-E-D-B-C$ (note that if we use bfs algorithm on the classes we get 72 possible paths). The user selects one of these possible paths and select a node from each cloud. So if we select the path $A-B-D-C$ and nodes c_{P1} and c_{P2} the algorithm will offer only four following possible paths sorted by length:*

- $c_{PIM1}, c_{P3}, c_{P4}, c_{P7}, c_{P2}, c_{P8}, c_{P1}, c_{P9}, c_{P10}, c_{P6}, c_{P5}, c_{PIM2}$
- $c_{PIM1}, c_{P4}, c_{P7}, c_{P2}, c_{P8}, c_{P1}, c_{P9}, c_{P10}, c_{P6}, c_{P5}, c_{PIM2}$
- $c_{PIM1}, c_{P3}, c_{P4}, c_{P7}, c_{P2}, c_{P8}, c_{P1}, c_{P9}, c_{P10}, c_{P6}, c_{PIM2}$
- $c_{PIM1}, c_{P4}, c_{P7}, c_{P2}, c_{P8}, c_{P1}, c_{P9}, c_{P10}, c_{P6}, c_{PIM2}$

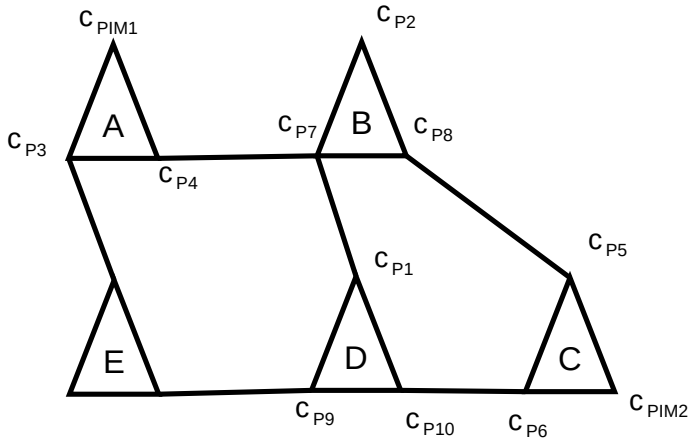


Figure 5.7: Path selection

Chapter 6

Results

6.1 Testing Implementation

We used the XCase tool as a basic platform for implementation of decision tree algorithm. For the testing purposes we implemented analysis of PIM diagram with *Occurence*, *Datatype* and *Length* analyzers. We also prepared the creation of decision trees and, last but not least, we implemented the similarity evaluation and mapping selection.

The algorithm can be started in XCase toolbar under the *Reverse* command group by *XSD->PSM* command. Note that this dialog is context based thus it is important to have opened project and PSM diagram. The application contains simple window for setting of methods priority and weight (see Figure 6.1). After the settings the dialog with file selection appears. Unlike the original implementation of reverse engineering, in XCase we allow user to select multiple files (it has no significant impact on functionality, it just simplifies transformation of more schemas).

The class selection uses the data gathered from *matchers* in a decision tree and offers the sorted results to the user. Figure 6.2 shows these results in class selection window. This window contains information about a PSM class and all PIM classes. On top there is the name of a PSM class, in this case *PickUpLocation*, followed by a list of PIM classes. Each offered mapping contains the name of a class (*Address*, *Acc*, etc.), the feature based results – *name*, *datatype* and *children*, and the final aggregated similarity. The user selects one mapping by double clicking on the list item.

The implementation is focused on decision tree and matchers experiments, thus the path selection and cloud creation is not a part of this imple-

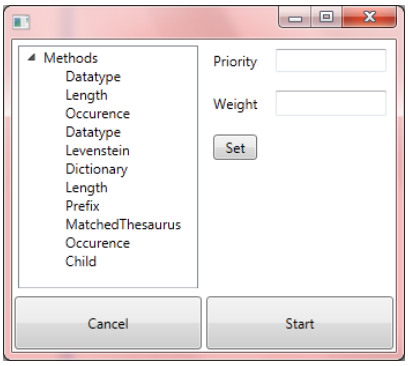


Figure 6.1: Settings window

mentation. The found mappings have no effect on XCase project and cannot be saved. We plan to add this functionality as a next step of our research.

6.2 Data sets

For a purposes of experiments we created 3 PIM diagrams describing simple situation of car rental company (see Figure 6.3 for one of diagram). These diagrams do not differ in structure but we created three different name sets to show the advantage of decision trees for string comparison.

PIM A This is the diagram in Figure 6.3. The diagram was created with an emphasis on correct full names. It was designed to be maximally helpful for mapping algorithms.

PIM B The names in this diagram were created very similarly, i.e. the classes associated with vehicle got *Vehicle* prefix.

PIM C In this case we created names with maximal focus on abbreviations and partially synonyms. The aim of this diagram is not to show that the XCase mapping algorithm does not implement these specific matchers, but to show the possibility of selection the right *matcher* during matching process.

As an input for PSM diagram we created four simple XSD schemas based on OpenTravel online XML schemas [20]. We used them as a template for final schemas. Systems like OpenTravel contain hundreds of complex types

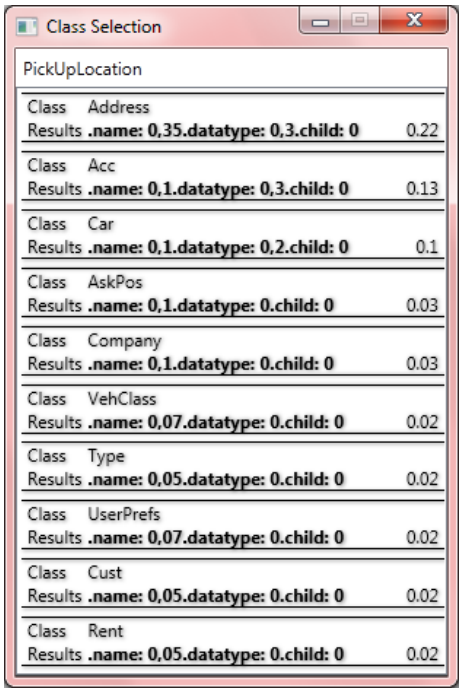


Figure 6.2: Class selection window

very closely linked together and we do not need the types which are not included in our PIM diagram, so for purposes of our experiments we removed the unnecessary classes and flattened the schemas. In particular the used schemas describe a request and a response of a car rental and location of a car vendor. All used PIM diagrams and schemas are placed on the attached DVD.

We compared our results with current implementation of reverse engineering in XCase based on [19] implemented by Jakub Klimek. This work depends on a list of fixed matchers with possibility of weight settings. For the purposes of the experiments we used the default values.

For the experiments we used *Matched Thesauri*, *Length Ratio*, *Levenshtein*, *Prefix*, *Thesauri*, *Data Type* and emphChildren matchers as shown in Figure 6.4. We set the border value on *Prefix* and *Levenshtein* to 0.1 to minimize the number of unnecessary calls, i.e. the *Thesauri* matcher is used only if the similarity on *Prefix* and *Levenshtein* matchers is lower then 0.1. For the user thesaurus we used following synonyms:

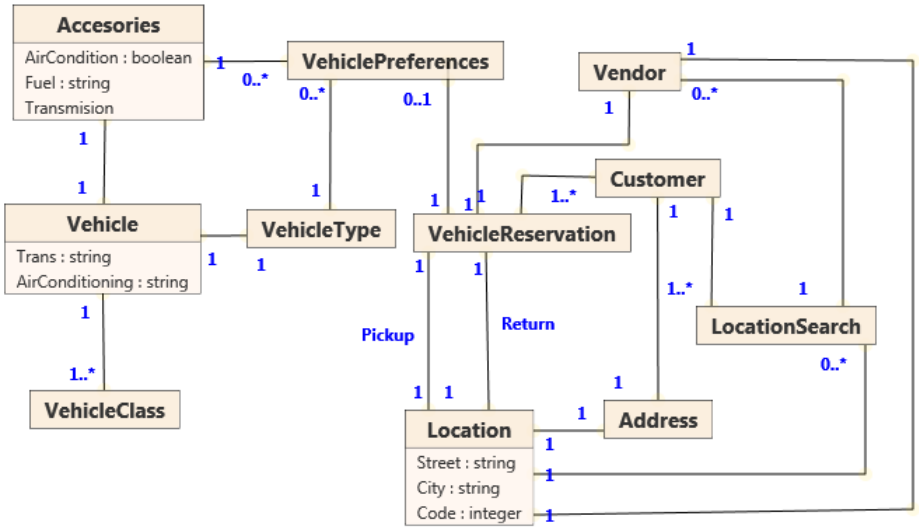


Figure 6.3: Car rental PIM diagram

- address, destination, location
- car, auto, automobile, machine, motorcar, vehicle
- vendor, seller, marketer, rental
- preference, option, alternative, choice

6.3 Metrics

We measured two main metrics for each PIM diagram. The first one p is position of correct mapping in mapping list. This is probably one of the most important variables of the whole process, because it allows the domain expert to effectively select the right mapping. We computed the final value for the PIM diagram as the average value of position of all found mappings.

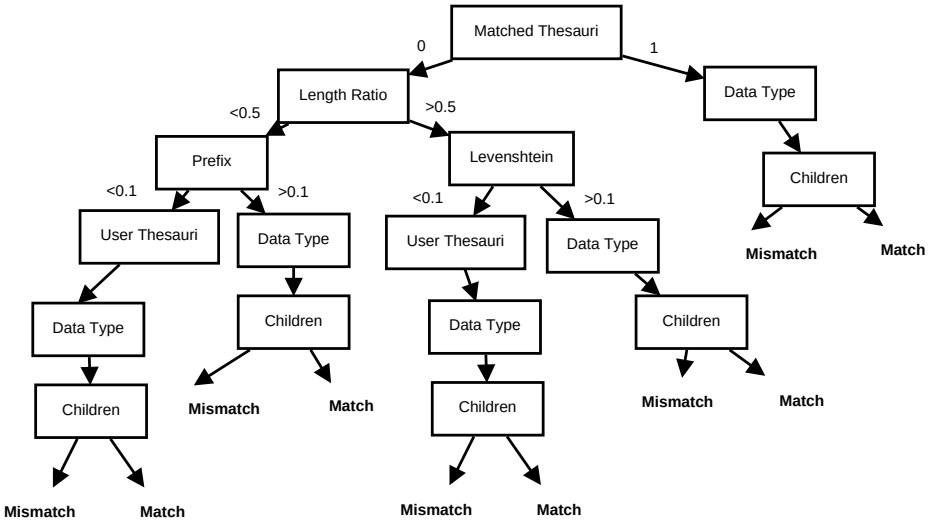


Figure 6.4: Experimental decision tree

In addition, we added one sub-metric – the worst position. It shows how is the algorithm able to provide balanced values over different inputs. The second metric d is the difference between the similarity of the correct mapping and the first mapping in list (only if the correct mapping is not the first one). This value shows the "confidence" the algorithm had in offering a wrong mapping. It distinguishes the situation when p is high but d is low and the matcher was close to select the correct one from the situation when the p and d are high and the mapping algorithm made a big mistake.

We measured mappings of PIM classes. Although PIM attributes are used in the diagrams their main purpose is to provide the information to the attribute-based matchers.

6.4 Results

We run all our experiments on standard personal computer with dual core processor AMD 2.7 GHz, with 2 GB RAM and installed Windows 7.

In general, the results fulfilled our expectations but still there were a few interesting results as we can see in Table 6.1 for the XCase mapper results and in Table 6.2 for decision tree results.

PIM diagram	XCase			
	Avg. pos.	Worst pos.	Avg. dist.	Avg. dist (%)
PIM A	1.47	5	0.06	32%
PIM B	1.8	5	0.04	20%
PIM C	2.7	12	0.04	28%

Table 6.1: Precision of XCase

Diagram PIM A was the easiest mapping example for the XCase mapper, because it contains simple and similar names to the ones used in XSD schemas. The worst position 5 of 11 is a good result. More interesting is the average distance which is the highest from all PIMs. The algorithm did not make much mistakes, but it was convinced they are right.

Surprisingly, decision tree did not make its best results on PIM A. The average depth was worse than the depth of the XCase algorithm. The cause of this result can be the selection of string matcher or used coefficients. Other possibility is that the worse value is a consequence of more matchers and their wrong selection. As we can see in Figure 6.5(a) the highest usage of *Prefix* matcher in this case was not the best selection for highly similar strings. The other measures are similar as for the XCase algorithm and the average distance is also the highest.

PIM diagram	Decision tree			
	Avg. pos.	Worst pos.	Avg. dist.	Avg. dist (%)
PIM A	1.66	6	0.05	26.2%
PIM B	1.57	4	0.03	24.7%
PIM C	1.71	5	0.04	16%

Table 6.2: Precision of decision tree

In diagram PIM B the XCase algorithm began to lose. The average position value 1.8 is still good and the worse value stays 5. The average distance lowers (the lowest from all diagrams) and shows that the correct mapping still was not far.

On the other hand, decision tree algorithm provided better results. The average position was 1.57 and it has a small distance between correct and wrong mappings. The chart in Figure 6.5(b) shows the switch between usage

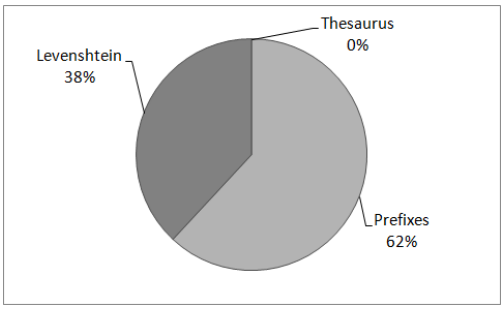
of *Prefix* matcher and *Levenshtein* matcher. The *Thesaurus* matcher was not used as in the previous case.

Finally, on the third diagram PIM C the XCase algorithm made the worst result. The main reason of the worse average position is probably inability to translate *address* to *location*.

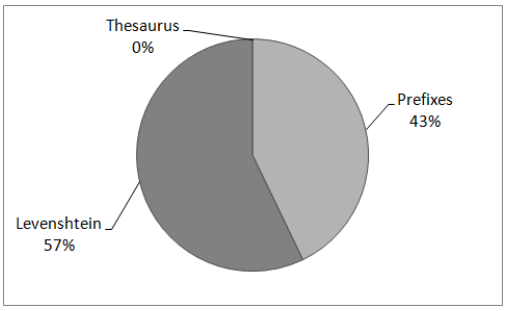
Decision tree algorithm was able to use synonym translation and thus it was capable to keep the average position under 2. As we can see in Figure 6.5(c) the usage of *Thesaurus* matcher highly increase.

In general, these experiments show that the fixed matcher can give correct mappings but due to its limited number of algorithms it is unable to react on different situations. The usage of decision trees allows to balance the found mappings with regards to input schemas.

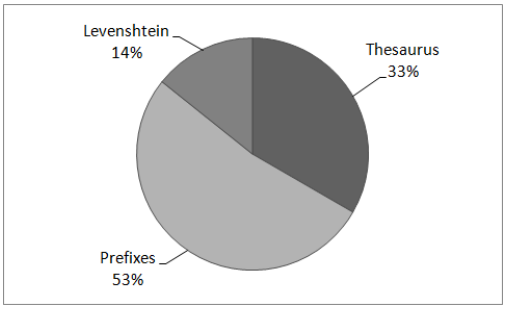
These experiments was focused only on class names, i.e. strings, the impact of the proposed method would be probably higher if we tested structure or more complex features.



(a) PIM A



(b) PIM B



(c) PIM C

Figure 6.5: Usage of String comparison methods

Chapter 7

Conclusion

In this thesis, we focused on the problem of mapping multiple schemas from different sources to one common PIM diagram. We described problems with inconsistency of XML schema usage and made a brief introduction to XML. We made a survey of current reverse engineering and XML Schema matching approaches. We presented the idea of model driven development and conceptual modeling, basic used conceptual modeling approaches and, in particular, model XSEM in Chapter 3.

In the second half of this thesis, we focused on improvement of the reverse engineering process presented in [19]. We introduced new approach based on flexible matching process of the XSEM model. The process is based on analysis of PIM diagram before the mapping starts. The data gathered by this analysis are used to build a decision tree with nodes created from different matching methods. Then we use it to find mappings between PIM and PSM classes and attributes. Unlike the brutal-force mapping systems, the trees allow to extend the number of used matchers without higher requirements on performance.

The proposed algorithm is very flexible. By the modification of tree it can adapt to different data sources and, unlike the usage of fixed matchers, our approach allows to use more different matchers without lowering overall performance, i.e. the overall similarity computation can benefit from them and return more precise results. On the other hand, the structure of the tree can be its greatest weakness. Wrong structure of matchers can provide more unbalanced results than with usage of algorithm with fixed matcher set.

We also proposed a new approach for mapping associations between PSM classes and paths between PIM classes. In Section 5.2.3 we use dense sub-

graph search to lower the number of PIM classes. Over this smaller graph we use Floyd-Warshall algorithm or breadth-first search to compute the paths between the classes and then we offer them to the user to select the optimal candidate.

7.1 Future Work

It is clear that the results of our and similar algorithms depends on selected matchers and their settings. In our future work we will focus on extending the set of used matchers and we will try to find the ideal weight and priority values automatically.

Secondly building of a decision tree is the cornerstone of our algorithm, so we will also focus on possible strategies of its creation to improve the adaptability to different inputs.

And last but not least possible direction of our research will go toward automatic modification of the decision tree during similarity computation. This could be done on the basis of statistical usage of the matchers and their repositioning in the tree, or by a quick analysis of input XML schemas, their elements and attributes. This could highly improve the precision of computation on large sets of similar data.

Bibliography

[1] Abello, J., Resende, M. G., and Sudarsky, S. (2002). Massive Quasi-Clique Detection. In Proceedings of the 5th Latin American Symposium on theoretical informatics (April 03 - 06, 2002). S. Rajsbaum, Ed. Lecture Notes In Computer Science, vol. 2286. Springer-Verlag, London, 598-612.

[2] Aumueller, D., Do, H., Massmann, S., and Rahm, E. (2005). Schema and ontology matching with COMA++. In Proceedings of the 2005 ACM SIGMOD international Conference on Management of Data (Baltimore, Maryland, June 14 - 16, 2005). SIGMOD '05. ACM, New York, NY, 906-908.

[3] Badia, A. (2002). Conceptual Modeling for Semistructured Data. In Proceedings of the Third international Conference on Web information Systems Engineering (Workshops) - (Wisew'02) (December 11 - 11, 2002). WISEW. IEEE Computer Society, Washington, DC, 170.

[4] Bernauer, M., Kappel, G., and Kramler, G. (2003). 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] Chen, P. (1976). The Entity-Relationship Model-Toward a Unified View of Data. ACM Transactions on Database Systems, 1(1):9 - 36, Mar. 1976.

[6] Duchateau, F., Bellahsene, Z., and Coletta, R. (2008). A Flexible Approach for Planning Schema Matching Algorithms. In Proceedings of the OTM 2008 Confederated international Conferences, Coopis, Doa, Gada, Is, and ODBASE 2008. Part I on on the Move To Meaningful internet Systems: (Monterrey, Mexico, November 09 - 14, 2008). R.

Meersman and Z. Tari, Eds. Lecture Notes In Computer Science, vol. 5331. Springer-Verlag, Berlin, Heidelberg, 249-264.

[7] Elmeleegy, H., Ouzzani, M., and Elmagarmid, A. (2008). Usage-Based Schema Matching. In Proceedings of the 2008 IEEE 24th international Conference on Data Engineering (April 07 - 12, 2008). ICDE. IEEE Computer Society, Washington, DC, 20-29.

[8] Do, H. and Rahm, E. (2002). COMA: a system for flexible combination of schema matching approaches. In Proceedings of the 28th international Conference on Very Large Data Bases (Hong Kong, China, August 20 - 23, 2002). Very Large Data Bases. VLDB Endowment, 610-621.

[9] Wikipedia, Floyd–Warshall algorithm — Wikipedia, The Free Encyclopedia, 2010 Online; accessed 16-July-2010

[10] Klímek, J. (2009), XML schema evolution, Master thesis, Charles University. http://www.ksi.mff.cuni.cz/ klimek/master.pdf

[11] Li, J., Liu, J., Liu, C., Wang, G., Yu, J. X., and Yangt, C. (2007). Computing structural similarity of source XML schemas against domain XML schema. In Proceedings of the Nineteenth Conference on Australasian Database - Volume 75 (Gold Coast, Australia, December 03 - 04, 2007). ACM International Conference Proceeding Series, vol. 313. Australian Computer Society, Darlinghurst, Australia, 155-164.

[12] Madhavan, J., Bernstein, P. A., and Rahm, E. (2001). Generic Schema Matching with Cupid. In Proceedings of the 27th international Conference on Very Large Data Bases (September 11 - 14, 2001). P. M. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, Eds. Very Large Data Bases. Morgan Kaufmann Publishers, San Francisco, CA, 49-58.

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

[14] Melnik, S., Garcia-Molina, H., and Rahm, E. (2002). Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In Proceedings of the 18th international Conference

on Data Engineering (February 26 - March 01, 2002). ICDE. IEEE Computer Society, Washington, DC, 117.

[15] Hong-Minh, T. and Smith, D. (2007). Hierarchical Approach for Datatype Matching in XML Schemas. In Proceedings of the 24th British National Conference on Databases (July 03 - 05, 2007). BNCOD. IEEE Computer Society, Washington, DC, 120-129. DOI= http://dx.doi.org/10.1109/BNCOD.2007.10

[16] Mlynkova, I., and Pokorny, J. (2006). Exploitation of Similarity and Pattern Matching in XML Technologies. Prague, Czech Republic.

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

[18] Nečaský, M. (2008). Conceptual Modeling for XML, PhD thesis, Charles University. IOS Press. http://kocour.ms.mff.cuni.cz/.necasky/dw/thesis.pdf

[19] Nečaský, M. (2009). Reverse Engineering of XML Schemas to Conceptual Diagrams, in Proceedings of Sixth Asia-Pacific Conference on Conceptual Modelling, Wellington, New Zealand, Australian Computer Society, ISBN: 978-1-920682-77-4, ISSN: 1445-1336, pp. 117-128, January 2009.

[20] OpenTravel Specification http://opentravel.org/Specifications/OnlineXmlSchema.aspx

[21] Routledge, N., Bird, L., and Goodchild, A. (2002). UML and XML schema. In Proceedings of the 13th Australasian Database Conference - Volume 5 (Melbourne, Victoria, Australia). ACM International Conference Proceeding Series, vol. 18. Australian Computer Society, Darlinghurst, Australia, 157-166.

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

[23] Shvaiko, P., and Euzenat, J. 2005. A survey of schema-based matching approaches, in Journal on Data Semantics IV, ser. Lecture Notes in Computer Science, 2005, ch. 5, pp. 146-171.

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

[25] The Unicode Standard 5.2.0. The Unicode Consortium, dec 2009., http://www.unicode.org/versions/Unicode5.2.0/

[26] Fellbaum, Ch. (1998) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.

[27] XCase - tool for XML data modeling. http://www.ksi.mff.cuni.cz/xcase

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

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

Appendix A

DVD Content

The DVD-ROM is a part of this thesis. It contains the text of the work, source code of the XCase tool, the source code of the implemented proposed algorithm and the executable files of the XCase application with implemented algorithm. The following files and directories are included:

- text – directory with the text of the thesis in PDF
- src – directory with source codes of XCase and the algorithm (src/XCase/Reverse2)
- bin – directory with the executables of XCase
- data – directory with files used in the experiments

Appendix B

Used XML Schemas

B.1 Figure 3.2

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="http//www.example.org/"
 elementFormDefault="qualified"
 targetNamespace="http//www.example.org/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Purchase" type="Purchase" />

  <xs:complexType name="Purchase">
    <xs:sequence>
      <xs:element name="Car" type="Car" />
      <xs:element name="Customer" type="Customer" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Car">
    <xs:sequence>
      <xs:element name="Supplier" type="Supplier" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Supplier" />

  <xs:complexType name="Customer">
```

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

  <xs:complexType name="Address" />

</xs:schema>
```

B.2 Figure 3.2.2

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="http//www.example.org/"
 elementFormDefault="qualified"
 targetNamespace="http//www.example.org/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Car" type="Car" />

  <xs:complexType name="Car">
    <xs:sequence>
      <xs:element name="Supplier" type="Supplier" />
      <xs:choice>
        <xs:element name="Customer"
            type="NewCustomer" />
        <xs:element name="Customer"
            type="ExistingCustomer" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Supplier" />

  <xs:complexType name="NewCustomer" />

  <xs:complexType name="ExistingCustomer" />

</xs:schema>
```

B.3 Figure 11

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="cz.mff.xcase.example/"
 elementFormDefault="qualified"
 targetNamespace="cz.mff.xcase.example/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Purchase" type="Purchase" />

  <xs:complexType name="Purchase">
    <xs:sequence>
      <xs:element name="MA" >
<xs:complexType>
  <xs:attribute name="Name" type="xs:string"
          use="required" />
</xs:complexType>
      </xs:element>
      <xs:choice>
        <xs:element name="NewBann" type="Banner"
            maxOccurs="unbounded" />
        <xs:element name="OldBann" type="Banner"
            maxOccurs="unbounded" />
      </xs:choice>
      <xs:element name="Web" type="Web" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="Price" type="xs:int" use="required" />
    <xs:attribute name="Impresions" type="xs:long"
        use="required" />
  </xs:complexType>

  <xs:complexType name="Banner">
    <xs:sequence>
      <xs:element name="Designer">
        <xs:complexType />
      </xs:element>
    </xs:sequence>
    <xs:attribute name="Type" type="xs:string" />
    <xs:attribute name="size" type="xs:string" />
```

```
  </xs:complexType>

  <xs:complexType name="Web">
    <xs:attribute name="Server" type="anyURI"
        use="required" />
  </xs:complexType>

</xs:schema>
```