uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Laurentiu Checiu

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

A New XML Schema Mapping Architecture

TITRE DE LA THÈSE / TITLE OF THESIS

Prof. Groza

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Prof. Ionescu

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Prof. A. El-Saddik

Prof. D. Petriu

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

A New XML Schema Mapping Architecture

By

Laurentiu Checiu

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering
School of Information Technology and Engineering
University of Ottawa
May 2008

Your file Votre référence
ISBN: 978-0-494-46469-4
Our file Notre référence
ISBN: 978-0-494-46469-4

Abstract

This thesis presents a solution for an XML Schema mapping problem. Furthermore, this thesis shows how this solution can be used in a practical way.

The research presented in this thesis is based on a novel approach for the internal representation of the XML Schema during the matching process. This internal representation uses the XML Schema Infoset Model. The majority of XML schema matching tools use a graph representation for XML schemas. This representation leads to limitations regarding XML Schema documents complexity and scalability of the matching tool.

An important feature that was also implemented is the mapping refinement by a user at the end of the automatic matching process. Two XML schemas can be mapped in different ways and these mappings can be stored in a repository. A domain expert can then decide which mapping is best suited for the two XML schemas.

Acknowledgments

I wish to extend my heartfelt gratitude to my supervisors, Dr. Voicu Groza and Dr. Dan Ionescu. Dr. Voicu Groza was my mentor during my studies because of his thoughtfulness, patience, insight and continuous support. He guided me with warmth on the path of shaping myself as a researcher. Dr. Dan Ionescu provided me with tremendous guidance and support over the past two years during my work on this thesis. His optimism, ability to set ambitious goals, insightful suggestions and encouragement were the cardinal pillars of my research.

I also wish to thank my good friend, David MacDonald, for his sincere and supportive advice as a proofreader.

Table of Contents

Abstract i

Acknowledgments ii

Table of Contents iii

List of figures vi

List of tables ix

Glossary of terms x

1 Problem definition 1

1.1 Introduction 1

1.2 General description 1

1.3 Applications 2

1.4 Schema matching techniques 4

1.5 Matching XML Schema documents 6

1.5.1 Content types 6

1.5.2 Referenced XML Schema components 7

1.5.3 Distributed schemas 8

1.6 Schema matching evaluations 8

1.7 Organization of the thesis and contributions 9

2 Literature review and current developments 11

2.1 Introduction 11

2.2 Correctness for schema matching systems 11

2.3 Direct and Indirect Matches 12

2.4 EXSMAL – Semi-automatic Schema Matching Algorithm 13

2.5 Quality of Matching 14

2.6 QMatch 20

2.7 Efficiency of XML Schema Matching 24

2.8 XML Schema Extraction from XML Data Instance 26

2.9 Rank Aggregation for Automatic Schema Matching 28

2.10 Similarity Flooding 30

2.11 Schema Matching Software Tools 34

2.11.1 COMA 34
2.11.2 COMA++ 36
2.11.3 Clio – IBM and University of Toronto 37
2.11.4 LSD – Learning Source Descriptions 39
2.11.5 Cupid 41
2.11.6 Corpus-based Schema Matching - MKB 43
2.11.7 SMART 44
2.11.8 iMAP 46
2.11.9 VisAXSM 48
2.11.10 TranScm 50
2.11.11 Map Force 51
2.12 Current technologies 52
2.12.1 RDF – Resource Description Framework 52
2.12.2 Sesame – RDF Query Architecture 54
2.12.3 WordNet 58
3 High Level Design 60
3.1 Introduction 60
3.2 XML Schema 61
3.2.1 XML Schema building blocks 61
3.3 Hermes – An architecture for XML Schema matching and mapping 62
3.3.1 Schema Extraction Engine 66
3.3.2 Matching Engine – High Level Architecture 69
3.3.3 Repository 71
3.3.4 Thesaurus 72
3.3.5 Mapping 74
3.3.6 Transformation Engine 74
3.3.7 Graphical User Interface 76
4 Hermes – Detailed design 79
4.1 Introduction 79
4.2 XML Schema Infoset Model 80
4.3 Matching Engine – Detailed Design 88

4.4 Graphical User Interface – Detailed design 128
4.5 Schema Extraction Engine 130
4.6 Repository and Mapping 133
4.6.1 Repository – Detailed design 134
4.6.2 XML Schema documents mapping 146
4.7 Transformation Engine - Detailed Design 148
5 Experimental Results and Evaluations 152
5.1 Introduction 152
5.2 Schema matching tools evaluations 154
5.3 Hermes – the first set of evaluations 158
5.3.1 Real estate XML Schema documents 158
5.3.2 Financial XML Schema documents 166
5.3.3 DBLP – Computer Science Bibliography 171
5.3.4 Purchase orders XML Schema documents 176
5.3.5 Summary of quality measures 180
5.4 Hermes and complex XML Schema documents 181
6 Conclusions and Future Work 190
6.1 Contributions of this Thesis 190
6.2 Future Work 191
Bibliography 193
Appendix A: XSLT Script used by the Schema Extraction Engine 203
Appendix B: XML Schema Design Guidelines 205
Appendix C: COMA++ Scalability Test 207
Appendix D: XStruct Testing 209

List of figures

Figure 2.1 QMatch – The Hybrid Match Algorithm [28] 23
Figure 2.2 K-means clustering algorithm 25
Figure 2.3 XStruct architecture [31] 27
Figure 2.4 SF algorithm [15] a.) Schema representation b.) PCG c.) Similarity propagation d.) Similarity Values 31
Figure 2.5 RDF triple (subject, predicate, object) graphical representation 53
Figure 2.6 Sesame Server [62] 55
Figure 2.7 Sesame Architecture [62] 55
Figure 3.1 High-level architecture of Generic Match [1] 63
Figure 3.2 Hermes – High-level Architecture 64
Figure 3.3 Hermes - Logical Domain Architecture 65
Figure 3.4 Hermes – Generic Component Architecture View 66
Figure 3.5 Schema Extraction Engine Data Flow 68
Figure 3.6 Schema Extraction Engine Architecture 68
Figure 3.7 Hermes – Matching Engine Architecture 70
Figure 3.8 Transformation Engine Architecture 75
Figure 3.9 Graphical User Interface Architecture 77
Figure 3.10 Finite State Machine for user interaction with Hermes GUI 78
Figure 4.1 Schema Infoset Model Class Hierarchy [41] 80
Figure 4.2 XML Schema Infoset Model – Data types hierarchy and relationships [41] 82
Figure 4.3 XML Schema Infoset Model containment relationships [41] 84
Figure 4.4 XML Schema component relationships in the XML Schema Infoset Model [41] 86
Figure 4.5 XML Schema Components: UML Relationship Diagram [35] 87
Figure 4.6 Matching Engine UML Class Diagram – Hierarchy 89
Figure 4.7 XML Schema matching algorithm 90
Figure 4.8 XML Schema element declaration matching algorithm 93
Figure 4.9 Element declaration matching algorithm UML sequence diagram 94
Figure 4.10 XML Schema attribute declaration matching algorithm 96

Figure 4.11 Attribute declaration matching algorithm UML sequence diagram 97
Figure 4.12 XML Schema simple type definitions matching algorithm 98
Figure 4.13 Example of a simple type definition with an enumeration constraint 101
Figure 4.14 Simple type definition matching algorithm UML sequence diagram 103
Figure 4.15 XML Schema complex type definition matching algorithm 104
Figure 4.16 Complex type definitions matching algorithm UML sequence diagram 106
Figure 4.17 XML Schema Attribute Use components matching algorithm 107
Figure 4.18 Attribute use components matching algorithm UML sequence diagram 109
Figure 4.19 XML Schema Model Group Definition components matching algorithm .. 110
Figure 4.20 Model group definitions matching algorithm UML sequence diagram 111
Figure 4.21 XML Schema Model Group component matching algorithm 112
Figure 4.22 Model group matching algorithm UML sequence diagram 114
Figure 4.23 XML Schema Particle component matching algorithm 115
Figure 4.24 Particle matching algorithm UML sequence diagram 117
Figure 4.25 XML Schema Wildcard component matching algorithm 118
Figure 4.26 XML Schema Identity Constraint Definition matching algorithm 120
Figure 4.27 XML Schema Notation Declaration components matching algorithm 123
Figure 4.28 Semantic Name Matching Algorithm 124
Figure 4.29 Matching Engine – Class relationships UML diagram 125
Figure 4.30 Match Composite Object Structure 126
Figure 4.31 Hermes – Graphical User Interface (GUI) 129
Figure 4.32 Hermes screenshot – A mapping between two XML Schema documents .. 130
Figure 4.33 Repository and Mapping – UML Class diagram 134
Figure 4.34 An Element Declaration and its RDF representation in the repository 136
Figure 4.35 Two element declarations that match 137
Figure 4.36 RDF representation of two element declarations that match 138
Figure 4.37 getMatchedElements method 139
Figure 4.38 getMatchedAttributes method 140
Figure 4.39 Extract un-matched element declarations 141
Figure 4.40 Extract attributes of un-matched element declarations 142
Figure 4.41 Retrieve parent elements 143

Figure 4.42 Retrieve child elements of un-matched elements 144
Figure 4.43 SeRQL statement for finding root element 145
Figure 4.44 Show Connections algorithm 147
Figure 4.45 Transformation Engine – algorithm for generating XSLT documents 149
Figure 5.1 Real matches and automatically derived matches [21] 155
Figure 5.2 "author" and "year" element declarations from *DBLP.xsd* 172
Figure 5.3 "author" and "year" element declarations from *targetDBLP.xsd* 172
Figure 5.4 Quality Measures – Precision, Recall, Accuracy and Quality of Match 181
Figure 5.5 XML Schema documents: a.) source.xsd b.) target.xsd 182
Figure 5.6 Additional XML Schema documents: a.) dateTypes.xsd b.)nameTypes.xsd c.) records.xsd 183
Figure 5.7 Hermes screenshot - mapping between *source.xsd* and *target.xsd* 184
Figure 5.8 Data instances: a.) source.xml b.) target.xml 185
Figure 5.9 XSLT Transformation script: *src_2_trg.xsl* 186
Figure 5.10 XSLT Transformation script: *trg_2_src.xsl* 187
Figure 5.11 *source.xml* transformed to *source2Target.xml* 188
Figure 5.12 *target.xml* transformed to *target2Source.xml* 188

List of tables

Table 5-1 Manual match of RE_I and RE_II XML Schema documents 159
Table 5-2 Automatic match of RE_I and RE_II XML Schema documents 160
Table 5-3 Manual match of statisticsDB and expensesDB 166
Table 5-4 Automatic match of statisticsDB and expensesDB 167
Table 5-5 Manual match of DBLP.xsd and targetDBLP.xsd 171
Table 5-6 Automatic match of *DBLP.xsd* and *targetDBLP.xsd* 173
Table 5-7 Manual match of *TPC-H.xsd* and *TPC-H-nested.xsd* 177
Table 5-8 Automatic match of *TPC-H.xsd* and *TPC-H-nested.xsd* 177
Table 5-9 Summary of quality measures 180
Table 5-10 Automatic match of *source.xsd* and *target.xsd* 184

Glossary of terms

DOM	Document Object Model
EDI	Electronic Data Intechange
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
JWNL	Java WordNet Library
PCG	Pair-wise Connectivity Graph
QoM	Quality of Match
RDBMS	Relational Database Management System
RDF	Resource Description Framework
SAIL	Storage And Inference Layer
SAX	Simple API for XML
SeRQL	Sesame RDF Query Language
SF	Similarity Flooding
SGML	Standard Generalized Markup Language
SWT	Standard Widget Toolkit
UML	Unified Modeling Language
UN/EDIFACT	United Nations/Electronic Data Interchange For Administration, Commerce, and Transport
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XPath	XML Path Language
XML	eXtensible Markup Language
XSD	XML Schema Definition Language
XSL	eXtensible Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformation

1 Problem definition

This thesis presents a novel XML Schema mapping architecture based on an innovative automatic matching engine and a semantic repository. Furthermore, this thesis presents how a mapping between two XML Schema documents can be used in a practical way.

1.1 Introduction

Current development of information technology led to an explosion of data which needs to be stored, retrieved, analyzed, transformed and eventually presented to a user. Data integration, data translation and data warehousing are the first domains to use schemas extensively and they rely on data exchange. A proper and meaningful data exchange depends on a perfect knowledge of the nature of data.

1.2 General description

A schema represents a human description of data. Due to the human factor, a set of data can be – and actually always is – described in several different ways. Given two descriptions of data a few questions which are obvious are asked. How can it be decided whether they refer to the same data? Is the described data required to determine the answer? Why is this answer even needed?

A schema is developed in a context with a specific terminology depending on the domain. It is easier to find elements that describe same sort of data in two different schemas that belong to the same domain of activity. Alternatively, in the case of two schemas from totally different domains, elements from these schemas which may describe the same type of data may have actually no correspondence because each data instance may have a different meaning in its domain context.

Schema matching is the process of finding correspondences between components of two schemas – source and target. One element from source schema may describe same data as another element from target schema. Therefore, schema matching is the process of finding a consistency between elements of the two schemas that correspond to each other with respect to their meaning. As per [1], a schema matching process can be defined when complex relationships between schema elements are discovered, i.e. not only plain correspondences. For instance, one schema can have an element "*Name*" and the other schema can have two elements in form of "*FirstName*", "*LastName*". One match can be defined like this: "*Name*" = {"*LastName*", "*FirstName*"}.

The result of the matching process is a mapping; there can be more than one mapping for two schemas. The matching process may not be based only on the two input schemas "but also on auxiliary information, such as dictionaries, global schemas, previous matching decisions, instance data, and user input" – [1].

A complete schema matching is done by human action; there is no complete automated system for this process to date. Therefore, this process is error-prone – "*Errare humanum est*" *(lat.)* and time consuming. It requires domain experts to find or validate some of the correspondences between schema elements.

As stated before, the result of a schema matching process is a mapping. This mapping is then used by computer "engines" to facilitate and perform data exchange. The whole purpose of schema matching and mapping is data exchange.

1.3 Applications

In the following research and development domains a schema matching application is useful and in cases where schemas are very big, it can be mandatory [1]:

- *Schema integration*: having a set of disparate schemas, there is a need to develop a single global schema which contains all the semantics and data definitions in the other schemas. This process of integration is difficult, because schemas may describe different domains of activity, and even for schemas from same domain the difficulty remains because those schemas may have been developed independently – in space and time by different experts. Therefore, the first step in schema integration is the finding of what is common and what is different between all schemas and this action represents schema matching.
- *Data warehouses*: at the enterprise level, different and independent data sources must be integrated in a data warehouse. A data warehouse is the enterprise "memory", it contains data regarding financial activity, employees, etc. all data that is used by the enterprise during the time. Therefore, a matching process is mandatory during the data transfer from source to the warehouse, in order to have a correct and consistent transfer.
- *E-commerce*: all e-commerce applications rely on message exchange. Every application may have its own message structure, so all messages need to be translated. Messages can be described by some schemas, and matching those schemas makes possible message translations.

There is an ongoing effort for the e-commerce standardization by different bodies such as the W3C, the United Nations and the European Committee for Standardization. The W3C focuses on core technologies for e-commerce and identifies the required common infrastructure [85]. The UN/EDIFACT is the United Nation set of rules which consists of a set of internationally agreed standards, directories and guidelines for the electronic interchange of structured data [86]. The European Committee for Standardization established the CEN/ISSS (Information Society Standardization System) to provide businesses with comprehensive standardization-oriented products and services. One of its domains is the eBusiness and eCommerce technologies standardization, focusing on electronic business data exchange [87].

1.4 Schema matching techniques

Schema matching has been a very active research field over the past decades and various approaches for its implementation have been developed. The main approaches are as follows [1]:

- *Schema and its instance*: some of schema matching applications may consider only source and target schema [3, 4, 7, 10, 11, 12, 13, 14, 15, 17, 18, 26, 27, 28, 29, 30, 32] and other may also consider their instances [7, 10, 11, 14]. Schema instances are used by applications that employ machine learning algorithms.
- *Element and structure matching*: schema elements can be matched individually [3, 4, 10, 11, 12, 13, 14, 15, 17, 18, 25, 26, 27, 28, 29, 30, 32] and considering the schema structure as an important resource of schema semantics [3, 4, 10, 11, 12, 14, 15, 26, 27, 28, 29, 30, 32].
- *Language and/or constraint:* schema element names are matched semantically [4, 27, 32] using a dictionary of synonyms such as WordNet [64]. The schema matching process can also consider keys and relationships [10, 11] – so called identity constraints in XML Schema definition language.
- *Match cardinality:* the result of a schema matching process may show that one element from the source schema matches with two or more elements from the target schema. In a generic way, the result may show one to one (1:1) [3, 7, 10, 11, 14, 17, 25], one to many (1:n) [4], many to one (n:1) and many to many (n:m) [7] cardinalities. In addition, instances of the schemas may have different cardinalities than the ones found while matching the two schemas.
- *Additional information:* consists of dictionaries of synonyms and acronyms, expert user pre- and post- match input and information regarding previously matched similar schemas

Schema elements are matched by considering their names as a first step. Matching schema element names employs several practical methods described in [1] such as:

- *Character strings equality*: the two schema element names are compared from character strings point of view. Character case (minor or major) can be ignored, so "Name" and "name" are equal [3, 4, 7, 10, 11, 12, 13, 14, 18, 25].
- *Synonyms equality*: In this case, one schema element name can be a synonym of the other schema element name (e.g. "freight" and "cargo") [3, 4, 7, 10, 11, 12, 25, 26].
- *Hypernyms equality*: one schema element name semantic includes the other schema element semantic – vehicle and car: a car is a vehicle, for instance [3, 4, 26].
- *Similarity based on common sub-strings:* examples of this case can be *ProjectId* and *ProjectNo*. Similarity can also be considered based on soundex expressions like deliver2 and deliverTo. Combining with synonyms equality technique we can match strings like *deliverTo* and *ship2* - ship and deliver are synonyms.
- *Acronyms equality:* At a first sight, a schema element named DOB can be matched with the other schema element named DOB. However, the first element could represent a date type – date of birth – and the other element could be a string type – division of banks. In this case acronym meanings highly depend on the activity domain of the two schemas and also on the schema structure; this means that this approach of matching names should be backed up with structure based matching techniques.

Constraints are always used in a schema to define data types, value ranges, and cardinality of elements. If constraints in two schemas are alike, then all elements from those constraints are likely to be similar and match. Some constraints are described in a hierarchical manner, so their main matching approach should include structure based matching practical methods.

1.5 Matching XML Schema documents

XML (eXtensible Markup Language) is a standard for data representation and exchange for documents containing structured information. There is a need to enforce rules regarding document content to define permissible structure and types in XML format. These rules can be described using a DTD (Document Type Definition) or an XML Schema. DTD has its own definition language whereas XML Schema uses XML formats for describing XML content.

XML Schema definition language defines very complex rules regarding information in an XML document. These rules include XML element name definition, XML element content definition and complex relations between XML elements themselves, to name a few.

All schema matching applications rely on relatively simple and small schemas with few tens or maybe hundred of elements, and a small number of nesting levels. Advanced modeling techniques such as aggregation and generalization can be used recursively to define nested types in XML Schema development. Because of this, XML Schemas can represent more complex data structures than database schemas. Matching XML Schemas becomes much more difficult than matching DTD or database (SQL) schemas.

Matching large and complex XML Schemas brings up several different issues than in the case of matching database schemas. These issues are discussed in the following sections [2].

1.5.1 Content types

Using XML Schema definition language, an XML element content type can be defined as simple or complex. XML elements with simple type content do not have attributes and further child elements; XML elements defined as having complex type content, always

contain attributes and/or child elements [36]. Simple and complex types defined in an XML Schema can be referenced by other schema elements and can also be extended or restricted within a new type.

Simple types include the following built-in types: string, date, float, Boolean and integer. Complex types are always user defined and their complexity is almost unlimited. Therefore, complex types matching techniques must include structural match approaches along with other techniques such as element based and linguistic. On the other hand, complex type matching process must take into consideration compositors like *all*, *choice* and *sequence* and cardinality restrictions – *minOccurs* and *maxOccurs*.

All other integrity constraints such as identity constraints and primary keys must have their own matching process.

1.5.2 Referenced XML Schema components

Schema components can use other schema components for aggregation or specialization. These other components may already be defined in the current XML schema, or in another XML Schema that is included or imported in the current schema. Therefore, referenced schema components indicated with the *ref* attribute instead of the *name* attribute must be resolved before the whole matching process takes place. Only globally declared schema components (element and attribute declarations with simple and complex type definitions) can be referenced; a globally declared or defined schema component has the schema root element as its parent (*<xsd:schema>*).

Current developments [3, 4, 7, 8, 10, 11, 12, 13, 14, 15, 17, 18, 26, 27, 28, 29, 30, 32] do not scale in the case of referenced (shared) schema components [2], because they use a tree or graph schema representation. Performance can be improved by caching referenced schema components results, so they can be used each time their containers are subject to a matching process. For instance, "*PersonType*", a complex type definition from a source

schema contains a simple element called "*Name*" and "*BeingType*" a complex type from target schema contains as well a simple element called "*Name*". The matching process of the two complex types can be sped up by using the result of the matching of the two simple elements called "*Name*".

Another issue that may arise would be the one regarding recursive definitions. Recursive definition occurs when a schema component (element and attribute declarations, simple and complex type definitions) references one of its containers. This recursive definition must be properly handled to avoid infinite loops.

1.5.3 Distributed schemas

XML Schema definition language provides a way to define an XML Schema using several XML Schema documents (*.xsd). This approach can be implemented using *import*, *redefine* or *include* directives. Each sub-schema defines global simple or complex types, elements or attributes that are referenced in the main XML Schema. Therefore, an XML schema matching application must properly handle all those sub schemas.

1.6 Schema matching evaluations

A complete schema matching and mapping includes a manual process even though there are many software tools that may ease this process through an automated matching and mapping. It requires users with expert knowledge of the domain to find the right and complete set of similarities between any given schemas from that specific domain. In order to evaluate a semi-automatic schema matching process one should define proper quality measures i.e. establish a well defined metric. Match quality measures are described in [21] and the main measures are precision and recall. These main measures have been defined in the information retrieval research field [22].

To assess the quality of a semi-automatic schema matching process between two schemas, an expert of the domain of given schemas must find the correct and complete set of similarities – schema components that match – and this result is used as a "gold standard" in the evaluation. Quality measures for automated schema matching are defined as a comparison between the outcome of schema matching performed by a domain expert and the outcome of the automatic schema matching process.

1.7 Organization of the thesis and contributions

This thesis proposes a new XML Schema mapping architecture and shows a practical use of a mapping between two XML Schema documents.

Chapter 2 presents a literature review on the subject of the schema matching and mapping. Furthermore, schema matching and mapping software tools from academic and commercial sources are analyzed.

Chapter 3 presents the high level architecture design of Hermes – a semi automatic matching and mapping and transformation tool. The starting point of the high level design of Hermes is the high level design of a generic matcher presented in [1].

Chapter 4 describes the detailed design of Hermes. In this chapter the new XML Schema matching algorithms are presented along with the detailed architecture of the semantic repository. Based on this repository, a Transformation Engine is designed.

Chapter 5 presents the experimental results and performance evaluations of Hermes. Performance evaluations consist of quality measures (precision, recall and accuracy) described in information retrieval theory [22]. The practical use of a mapping between two XML Schema documents is also presented and consists of automatic generation of XSL transformation scripts that are applied to XML data instances.

Finally, in Chapter 6 the conclusions and future work are presented.

This thesis contains the following research contributions:

- A new XML Schema document internal representation within the software XML Schema matching and mapping tool. This representation is based on XML Schema Infoset Model API [50] from the Eclipse Modeling Framework package.
- A new Matching Engine architecture. This architecture follows the XML Schema standard [34, 36 and 36] and has been designed using structural and beahvioral design patterns [56].
- A new way of storing/retrieving information to/from a semantic repository.
- A new Transformation Engine: two XSLT scripts are generated and these scripts are used to transform (1) a data instance validated by the source XML schema into a data format that is validated by the target XML schema and (2) a data instance validated by the target XML schema into a data format validated by the source XML schema.

2 Literature review and current developments

A review of current developments, research papers and studies is presented in the following sections of this chapter. Schema matching and mapping software tools from academic and commercial sources are analyzed. As a general conclusion, there is no completely automated schema matching and mapping software tool and always there is a need for a user validation and refinement of the final result.

2.1 Introduction

The schema matching problem has been actively addressed by many researchers in either generic or very specific ways. Finding a correct and complete set of correspondences between two schemas is a very difficult task and many tools have been developed to accomplish this task. Having these tools, the next natural question is whether the result is the right and expected one.

2.2 Correctness for schema matching systems

Authors of [23] investigate the notion of correctness of current schema matching methods and they use semantic methods to find relations between schema elements base on the meaning of those elements. The schema matching methods are classified in syntactic, semantic and pragmatic categories, the main focus being on semantic methods. Schema matching process is presented as a three layer model as follows:

- *Language layer*: this layer is represented by expressions based on an alphabet and a grammar. These expressions are meaningful – i.e. schema element names – and their analysis is mandatory in a schema matching process;
- *Concept layer*: this layer shows the overall meaning of schema, i.e. its domain;
- *Object layer*: this layer describes in detail each schema element and relations between all schema elements treated as a whole.

Based on these layers, schema matching methods are classified as follows:

- *Syntactic methods*: methods that work on language layer;
- *Semantic methods:* their domain is the conceptual layer;
- *Pragmatic methods:* information used by those methods comes from the object layer.

Any semantic method must operate on information that comes from the language level, because this level contains most of the meaning of the schema.

Authors of [23] formalize the schema matching problem very well by providing formal definitions based on Description Logic theory [88] for topic hierarchy, classification and retrieval functions, hierarchical classification and mapping and schema matching methods. From this paper, a very important conclusion regarding schema matching process is drawn: language based and structure based schema matching methods must be used together in order to get a correct and complete mapping of two schemas.

2.3 Direct and Indirect Matches

Another contribution on the use of semantics for schema matching is brought by the authors of [26]. This contribution consists of providing a way to find direct and indirect semantic correspondences between two schemas elements. In their schema matching approach, schemas are represented as rooted conceptual-model graphs where a node of the graph stands for an object and value of a schema element and an edge represents the relationship between two schema elements. A match between a source schema element and a target schema element is said to be *direct* if both schema elements describe the same data. However, as per [26], some of target schema elements can be derived from some other source schema elements, and these are *indirect* matches. During the search for indirect matches, the following problems are investigated:

- *Generalization and specialization*: if the source schema contains an element "Name" and the target schema contains two elements "First_Name" and "Last_Name", the latter are a specialization of the former. In this case, the "Name" value in source schema should be split. If the mapping were the source schema contained two elements "First_Name" and "Last_Name" and the target schema contained "Name", then values from the source would need to be merged to obtain data consistent with the target schema.
- *Schema element as value*: in case of Boolean schema elements, their name holds information and can be used during schema matching process.

Authors of [26] use five operations over the source schema and these operations are: selection, union, composition, decomposition and Boolean. Schema matching techniques described are: terminological relationships (for name matching, synonyms and hypernyms) and data value characteristics such as string lengths and numeric ratios. These characteristics can be interpreted as identity constraints, domain specific regular expressions and structure based matching techniques. Based on these matching techniques, an overall schema matching algorithm is developed in [26] and this algorithm performs very well according to the authors "yielding over 90% in both recall and precision." [26]. The result of the match is not used for data translation, there is no repository so mappings could not be reused and refined. The authors of [26] do not make an analysis of their approach from a scalability perspective.

2.4 EXSMAL – Semi-automatic Schema Matching Algorithm

A pragmatic approach is presented in [27] for schema matching process. EXSMAL is a schema matching algorithm suitable for finding semantic matching between EDI messages. A schema for an EDI message is represented using XML Schema definition language. Element descriptions, the overall structure and data types are subject to the matching process; the result of this matching process is obtained based on the similarity value. The similarity value has three facets: basic similarity, structural similarity and pair-

wise element similarity. Basic similarity is a weighted sum of data type similarity and textual description similarity. It is computed between an element from the source schema and an element from target schema. Textual description similarity is calculated based on information retrieval theory [22], whereas data type similarity is calculated in an empirical way, based on XML Schema primitive data type. The second facet of the similarity value is the structural similarity, which is obtained based on element neighbours' similarities: ancestor similarity, immediate child element similarity, sibling element similarity and leaf element similarity. Each of these similarities is computed based on an aggregated value of a matrix constructed from basic similarities. Pair-wise element similarity, the third facet of similarity value, is a weighted sum of basic similarity and structural similarity between two schema elements (one from source schema and the other from target schema). The main contributions of [27] consist of textual element descriptions analysis during the matching process using information retrieval techniques and structure based matching algorithms by analyzing all ancestors, siblings, children and leaves elements of a given element. However, EXSMAL lacks user schema matching refinement, has no repository for schema mappings and no data transformation based on the result of schema matching is presented. Schema element names are matched based on only character strings equality and there is no dictionary or lexicon application (e.g. WordNet [64]) involved to find synonyms among names during the matching process. The overall performance of EXSMAL is not discussed in terms of precision and recall measurement parameters, those parameters being defined in information retrieval theory [22].

2.5 Quality of Matching

An important research paper that defined quality of matching is [28]. QoM (Quality of Matching) is evaluated for two schemas that are matched and is independent of the matching algorithm used. In addition, the QoM metric is very well formalized and it is based on UML. A schema is defined as a finite set of classes, where a class is 2-tuple

consisting of a finite set of attributes and finite set of methods. A match is classified from a qualitative point of view as follows:

- *Micro match*: this is a match between attributes/methods, defined as a match of all their properties. A match between attributes is *exact* when attribute labels are exactly the same or synonyms to consider them as a match. A match between attributes is *relaxed* if the source attribute label is a generalization or specialization of the target attribute label: this means that one is a hypernym or a hyponym to the other. A match between methods can also be *exact* or *relaxed*. An *exact* match between methods is when method signatures are identical and their pre and post conditions are equivalent. A *relaxed* match is when a method is a generalization/specialization of the other and their pre and post conditions imply each other.
- *Sub-macro match:* this is a match between two classes, where a class is 2-tuple consisting of a finite set of attributes and finite set of methods. The sub-macro match is performed based on the number of attribute/methods matches and the quality of these micro matches. A sub macro match can be *total* or *partial*. A *total* match is considered when all attributes from a class in the source schema match some attributes in a target schema class. A *partial* match occurs when not all attributes of source class match some attributes in target class. A sub-macro match can be defined as total exact, total relaxed, partial exact and partial relaxed. A **total exact** match has all attributes and methods of a source class in exact match relation with some or all attributes and methods in target class; a **total relaxed** sub-macro match is when all attributes methods from a source class have at least one *relaxed* micro match with attributes and methods in target class; a **partial exact** sub macro match is when not all attributes and methods from source class are in exact match relation with some attributes and methods in target class; a **partial relaxed** sub macro match occurs when some of attributes and/or methods have a relaxed micro match with some attributes and/or methods from target class.
- *Macro match:* represents the match between two schemas and is based on the number of matched classes and the quality of sub macro matches. In the same

way as in sub macro match, a macro match can be total exact, total relaxed, partial exact or partial relaxed. A **total exact** macro match is when all source classes have a total exact match with target classes. A **total relaxed** macro match occurs when all source classes have a total exact match and at least a total partial match with target classes. A **partial exact** macro match is when all source classes have an exact sub-macro match and at least one partial sub-macro match or some of the source classes have partial or total exact sub-macro matches in the target schema. A **partial relaxed** macro match occurs when *all* source classes have a sub-macro match in the target schema with at least one partial sub-macro match and one relaxed sub-macro match *or* some of the source classes have a sub-macro match with at least either one total relaxed sub-macro match or one partial relaxed sub-macro match.

A quantitative analysis is also presented in [28]. Exact match operators for a property receive a weight of 1.0; whereas relaxed match operators for a property have a weight of 0.5. The quality of match between two attributes – one from source and one from target) can be quantified as follows:

$$QoM(a_s, a_t) = \frac{W(L_s, L_t) + W(A_s, A_t) + W(T_s, T_t) + W(N_s, N_t) + W(I_s, I_t)}{5} \quad (1) \text{ from [28]}$$

L represents the label (a character string denoting the schema component name), A is the set of applicable modifiers (private, public and protected), T is the domain type - either primitive or user-defined, N represents cardinality and I is a set of eventual initial values (*enumeration values* in XML Schema definition language, for instance). W(Ps,Pt) represents the match weight of a property.

Similarly, quality of match between a source and a target method is represented from a quantitative point of view by the following formula:

$$QoM(m_s, m_t) = \frac{QoM_{sig}(m_s, m_t) + (2 \times QoM_{spec}(m_s, m_t))}{3}$$ (2) from [28]

Where:

$$QoM_{sig}(m_s, m_t) = \frac{W(A_s, A_t) + W(O_s, O_t) + W(I_s, I_t)}{3}$$ (3) from [28]

And:

$$QoM_{spec}(m_s, m_t) = \frac{W(pre_s, pre_t) + W(post_s, post_t)}{2}$$ (4) from [28]

A is the set of applicable modifiers (private, public and protected), O is the return data type, I is the method parameter data type, all of these three representing the method signature, *pre* and *post* representing method specifications, i.e. pre- and post- conditions. Authors of [28] consider that method specifications are more accurate measure of match between two methods than their measure of signature match.

A sub-macro match model is quantified using the following measures: *micro match weight, cardinality ratio* and *congruity ratio* and these measures are defined as follows:

$$R_W(C_s, C_t) = \frac{\sum QoM(M_s, M_t)}{|C_s|}$$ (5) from [28] is the micro match weight

M represents the set of all attributes and methods of a class C and $|C_S|$ is the source class cardinality. Therefore, the formula (5) from [28] can be written more clearly – considering (1) and (2) from [28] – as follows:

$$R_W(C_s, C_t) = \frac{\sum QoM(a_s, a_t) + \sum QoM(m_s, m_t)}{|C_s|} \quad (5')$$

Cardinality ratio is expressed in the following manner:

$$R_S(C_s, C_t) = \frac{|C_s^m|}{|C_s|} \quad (6) \text{ from [28]}$$

C_s, C_t are source class and target class, respectively; (6) from [28] represents cardinality ratio measure as a ratio between the micro matches number in the source class and the cardinality of the whole source class.

Congruity ratio is the ratio between micro matches number in the source class and the cardinality of the target class:

$$R_T(C_s, C_t) = \frac{|C_s^m|}{|C_t|} \quad (7) \text{ from [28]}$$

Using (5), (6) and (7) from [28] the quality of match (QoM) for a sub macro match can be quantified as follows:

$$QoM(C_s, C_t) = \frac{R_W(C_s, C_t) + R_S(C_s, C_t) + R_T(C_s, C_t)}{3} \quad (8) \text{ from [28]}$$

The macro match model quantification is defined in the same way as the sub-macro match model quantification. Therefore, the following measures are defined:

- *Sub-macro match weight* – defined as the normalized sum of all QoM of sub-macro matches for each class in source schema (Cs and Ct are classes from source and target schema, respectively:

$$R_W(S_s, S_t) = \frac{\sum QoM(C_s, C_t)}{|S_s|} \quad (9) \text{ from [28]}$$

- *Cardinality ratio* – defined as the ratio between the number of sub-macro matches in the source schema and the cardinality of the source schema (i.e. number of classes in the source schema):

$$R_S(S_s, S_t) = \frac{|S_s^m|}{|S_s|} \quad (10) \text{ from [28]}$$

- Congruity ratio – defined as the ratio between the number of sub-macro matches in the source schema and the cardinality of the target schema:

$$R_T(S_s, S_t) = \frac{|S_s^m|}{|S_t|} \quad (11) \text{ from [28]}$$

Considering (10), (11), (12) from [28], the formula for quantitative measure of quality of match (QoM) of macro match model is:

$$QoM(S_s, S_t) = \frac{R_W(S_s, S_t) + R_S(S_s, S_t) + R_T(S_s, S_t)}{3} \quad (12) \text{ from [28]}$$

Authors of [28] provide a qualitative and quantitative metric for a schema matching algorithm, and more important, this metric is independent of the matching algorithm used.

2.6 QMatch

Another matching algorithm for XML Schemas is QMatch [29]: this algorithm is an extension of QoM [28] and provides a framework that is used to analyze and match semantic and structural information within XML schemas. Each schema element is characterized using information in four dimensions. Authors use the term "axes" instead of dimensions, this term "axes" sometimes is used in a different context in XML world, i.e. to denote a path from an XML element down to its descendents; the term "axes" will be used with the meaning given by the authors of [29]. These four axes are: label (L), properties (P), nesting level (H) and children (C). The first three axes are considered to be atomic valued and matches within these axes can be *exact* or *relaxed*, whereas the fourth is "set-valued axis" and matches along this axis can be *total* or *partial*.

QoM match classification presented in previous paragraphs is now extended to XML Schema match model as follows:

- *Leaf match*: the nesting level of a leaf schema element is set to 0. This kind of match can be exact or relaxed: an exact match is considered when all set of properties and labels are equal or equivalent for a given source element and a target leaf element; a relaxed match means that the label or some of the properties are equal or equivalent. This kind of match is similar with micro match model described in [28];
- *Sub-tree match*: this kind of match occur between non-leaf schema elements and is described by the following features:
    - Number of children matches
    - Quality of children matches
    - Quality of individual atomic valued axis for each child (label and properties)

  Furthermore, this match can be regarded as a sub macro match model from [28];
- *Tree match*: this type of match applies to complex elements defined in an XML schema and to the root element itself of XML schema (i.e. <xsd:schema>). This is a special case of a sub-tree match, thus it can be treated in the same manner.

From a quantitative perspective, a quality of match (QoM) between two nodes can be defined as follows:

$$QoM(n_1, n_2) = W_L \times QoM_L + W_P \times QoM_P + W_H \times QoM_H + W_C \times QoM_C \quad (13) \text{ from [29]}$$

W_L, W_P, W_H and W_C are weight coefficients for each axis (label, properties, level and children), respectively; QoM_L, QoM_P, QoM_H and QoM_C are QoM along label, properties, level and children axes, respectively.

QoM quantification for a leaf match is:

$$QoM(N_z, N_t) = W_L \times QoM_L + W_P \times QoM_P + C \quad (14)$$

C is a constant because leaf elements match exactly and always on nesting level and children axes.

Weight coefficients were experimentally determined as follows (Table 2 in [29]):

- Label axis weight, W_L=0.3
- Properties axis weight W_P=0.2
- Level axis weight W_H=0.1
- Children axis weight W_C=0.4

Sub-tree QoM is quantified similarly with sub macro match model in [28]; it uses the same measures: sub-tree weight and cardinality ratio. Sub-tree weight is defined by the following formula:

$$R_W(N_s, N_t) = \frac{\sum QoM(n_s, n_t)}{|N_s|} \quad (15) \text{ from [29]}$$

Cardinality ratio is:

$$R_S(N_S, N_T) = \frac{|N_S^C|}{|N_S|} \text{ (16) from [29]}$$

Therefore the normalized QoM along the children axis is given by the following formula:

$$QoM_C(N_s, N_t) = \frac{R_W(N_s, N_t) + R_S(N_s, N_t)}{2} \text{ (17) from [29]}$$

QoM_H has only two values: 1 in the case of a match along the level axis and 0 otherwise. The QMatch algorithm consists of recursively computing the quality of match (QoM) given by the formula (1) from [29] and the overall match of two schemas is decided based on a threshold value.

```
double TreeMatch(Tree: t1, Tree: t2)
{
        Node: rs, rt
        rs = getRoot(t1)
        rt = getRoot(t2)
        STs = get all sub-trees rooted at children nodes of rs
        STt = get all sub-trees rooted at children nodes of rt
        QoMsum=0   // summation for the current node
        Ns^c = 0 // number of matching children
        for all ti in STs:
                for all tj in STt:
                        QoMc = TreeMatch(ti,tj)
                        If QoMc >= ThresholdValue
                                QoMsum  += QoMc
                                Ns^c ++
```

$$R_w = \frac{QoM_{sum}}{|t_i.getRoot()|}$$

$$R_s = \frac{|N_s^c|}{|t_i.getRoot()|}$$

$$QoM_c = \frac{R_w + R_s}{2}$$

```
        QoML = linguisticMatch(rs.label, rt.label)
        QoMP = propertyMatch(rs.properties, rt.properties)
        if (rs.getLevel() == rt.getLevel()
                QoMH = 1.0
        else
                QoMH = 0.0

        QoM=WL*QoML + WP*QoMP + WH*QoMH + WC*QoMC

        return QoM
}
```

Figure 2.1 QMatch – The Hybrid Match Algorithm [28]

The experimental results are encouraging; however, a complexity of an XML schema does not reside only in the number of elements, it is also given by schema inclusion or import directive and reuse of already defined XML Schema components such as

elements, attributes, simple and complex types. The referenced schema components need to be resolved (de-referenced) somehow and this fact is not analyzed; there is no repository available for saving the match result for refinement by an expert; furthermore, the result of the match is not used for data translation and/or transformation.

2.7 Efficiency of XML Schema Matching

Authors of [30] propose a clustered schema matching algorithm based on the K-means algorithm presented in Figure 2.2. This algorithm consists of identifying regions in target schema that contain some schema components likely to be mapped by source schema components. Furthermore, the schema matching problem is well formalized along with schema mapping. XML Schema is represented as a graph formally defined as follows:

- S = (N,E,I,H) where N is the set of nodes (n_1, n_2, n_3, ...), E is the set of edges (e_1, e_2, e_3, ...), I is the "incidence function" [30] $I : E \rightarrow N \times N$ I(e_k) = (n_i, n_j) meaning that e_k is the edge between n_i and n_j nodes (or elements); H is an association function between properties and values.
- A mapping is defined as follows: $\forall n \in N_S, \exists p \in N_T$ *with* $n \mapsto p$ and this kind of mapping works only with 1:1 cardinality. This definition shows only the syntactic aspect of a mapping. Therefore, a schema matching system needs an "objective function" [30] to express the correctness of a mapping: $\Delta(s,t) \rightarrow [0,1]$; usually this objective function computes the similarity factor between two schemas (or schema components).
- Schema matching problem is defined as a quadruple $P = (s, R, \Delta(s,t), \delta)$ where s is a "personal schema" regarded as a source schema, R is a schema from a general repository regarded as a target schema, the third parameter is the objective function and the fourth is the threshold. The solution of a schema matching problem is a list of all possible and correct mappings $s \mapsto t$ having $\Delta(s,t) \geq \delta$.

A clustered schema matching algorithm consists of identifying regions in target schema ("repository schema R" as per [30]) that contain some schema components likely to be

mapped by source schema components. This technique has been developed in order to minimize the search space during the matching process. However, there is a drawback: some of the regions may miss some good mappings, so there is a decrease in effectiveness. An improvement to this algorithm consists of running an element level matching process, then clustering the target schema and finally calling the structural matching process. The element matcher is based on a fuzzy string comparator which computes normalized string similarity using "character substitution, insertion, exclusion and transposition."[30].

The objective function is a weighted sum of objective function for element level matching and objective function for structural level matching as follows:

$$\Delta(s,t) = \alpha\Delta_{sim}(s,t) + (1-\alpha)\Delta_{path}(s,t) \quad (18)$$

The K-means clustering algorithm is illustrated in the following figure [30]:

```
1:  initialize centroids
2:  repeat
3:          for each mapping element do
4:                  for each centroid do
5:                          compute distance(mapping element,centroid)
6:                  end for
7:                  assign mapping element to nearest centroid
8:          end for
9:          compute new centroids for all clusters
10:         perform reclustering
11: until convergence criterion is met
```

Figure 2.2 K-means clustering algorithm

A centroid and a distance measured between a mapping element and the centroid is the actual representation of a cluster. Centroids are chosen from the set of mapping elements; the distance is calculated as the path length between two elements.

Performance evaluation is done only from time perspective – time elapsed for a schema matching process. No well defined information retrieval parameters and measures such as precision, recall and overall [22] are analyzed.

2.8 XML Schema Extraction from XML Data Instance

In the case of XML data exchange over the Web, the XML schema may not be available at the time of the exchange. It would be useful to generate an XML schema starting from an XML data instance. Having two or more schemas and a schema matching tool, data integration would be very easy. For instance, company A sends a purchase order (an XML data format) to company B. Company B receives the purchase order, but this purchase order does not conform to its purchase order format or schema. Therefore, a schema extraction tool would be useful to extract an XML schema for the incoming purchase order. This generated schema and company B's purchase order schema can be mapped one to each other using a schema matching tool in order to integrate incoming data into target system. The problem now is how to generate an XML schema for a given XML data instance. This kind of problem has been addressed in [31]. Authors of [31] developed a tool called XStruct which generates XML schemas for given XML data instances. The generated schema must have following properties [31]:

- *Correctness*: the generated schema must validate the XML data instance that was used as an input
- *Conciseness*: the generated schema must have minimal schema components (element and attribute declarations for instance)
- *Precision*: the generated schema should not be too generic, must validate XML data instances with the same format as the one that was used as a starting point.
- *Readability:* the generated schema must be readable and clear so a user can understand.

XStruct is a Java based application tool and its main modules are:

- SAX parser

- Model extraction module
- Data type recognition module
- Factoring module
- Attribute extraction module
- Schema printer

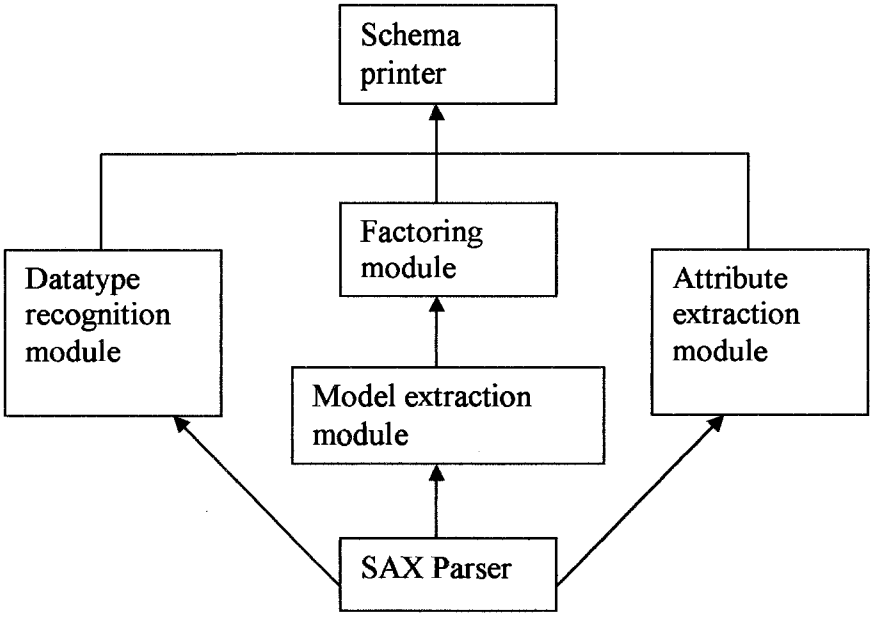


Figure 2.3 XStruct architecture [31]

Primitive data types that are handled are: string, Boolean, decimal, integer, double, date and type, which is a subset of all primitive data types defined by the XML Schema definition language specification. Attribute extraction is very simple, XStruct can determine whether an attribute is required or optional. If an attribute occurs always within all instances of an XML element then is marked as required. If an attribute is missing for some instances of an XML element then it is marked as optional. XStruct does not use DOM (Document Object Model) because it does not scale well, it uses SAX (Simple API for XML). The internal data structure for each XML data file consists of a hash table.

The limitation of XStruct is that it does not handle XML element namespaces properly. For instance, if an XML data file contains elements from different namespaces, the

generated schema does not consider those namespaces and the XML data file is not validated by the generated schema. An XML schema file can have only one target namespace, so if an XML data file contains two different namespaces then at least two XML schemas must be generated, one for each namespace in the XML data file. Details are given in Appendix D. The conclusion is that XStruct can be used only for very simple XML data files with no namespace for its elements. XML schema can be extracted using other techniques, much simpler than a Java application, by using XSLT for instance.

2.9 Rank Aggregation for Automatic Schema Matching

Rank Aggregation for Automatic Schema Matching is a schema meta-matching framework for automatic schema matching algorithms that finds the top-K mappings between two schemas, S and S' [32]. An automatic schema matching algorithm generates one or more mappings and there is a need to classify these mappings from completeness and correctness point of view, i.e. to build an ordered list of those mappings. This paper analyzes existing algorithms and presents new optimized ones in a very well formalized manner: threshold (TA), Matrix-Direct (MD), Matrix-Direct with Bounding (MDB) and CrossThreshold.

Having two schemas, S and T, each with n components, the number of 1:1 mappings is *n!*, a very large number for an algorithm measure (O(n!)). Therefore any algorithm that generates schema mappings must be optimized. An important measure used in a schema matching process is similarity, a real valued number in [0,1] interval which describes the degree of similarity between two schema components [1]. For the two schemas, S and T, a similarity matrix M can be constructed based on similarity values between each S schema component and each T schema component ($n \times n$ matrix size in this case). Based on similarity matrix local and global aggregation functions (*l-aggregator* and *g-aggregator*) have been defined in [32] as follows:

$f^{(A)}(\sigma, M^{(A)}) = f^{(A)}(M^{(A)}_{1,\sigma(1)},\ldots,M^{(A)}_{n,\sigma(n)})$ (19) from [32]

where σ is a mapping generated by the matcher A (name or linguistic matcher, structure matcher, etc. [1]), $M^{(A)}$ is the similarity matrix and $M^{(A)}_{i,\sigma(i)}$ (i=1..n) is the similarity value (in [0,1] interval) between S schema component i and its correspondent (mapped) T schema component $j = \sigma(i)$. A mapping is an element of the Σ set of all possible mappings between S and T. Usually, the *l-aggregator* (local aggregation function) is a sum or an average of similarities. A global aggregation function is defined as the aggregation of all l-aggregators for a given set of matchers A_1... A_m, and a general representation is:

$$F(f^{(1)}(\sigma,M^{(1)}),...,f^{(m)}(\sigma,M^{(m)})) = \frac{\lambda}{m}\sum_{i=1}^{m} k_i f^{(i)}(\sigma,M^{(i)}) \text{ (20) from [32]}$$

where λ is a constant and k_i weighting parameters in [0,1] interval. In the case of $\lambda = m$, the global aggregation function is the weighted sum of local aggregation functions and in the case of $\lambda = 1$, the global aggregation function is the weighted average of the local aggregation functions. Furthermore, all the mappings σ can be ordered based on their local and global aggregation functions values.

Threshold Algorithm (TA) is an efficient algorithm for optimal aggregation of several ordered lists and is intuitively very simple: it looks for mappings σ that have their aggregation values greater than a threshold value τ_{TA}. Top-K mappings means that the algorithm looks for K ≥ 1 mappings that have their global aggregation value greater than the threshold, this being also the stop condition for the entire algorithm. There is no guarantee that those K mappings are the best ones because there can be many other mappings that were not evaluated when the algorithm stops after the first K mappings. As a conclusion of TA analysis, Theorem 1 [32] states that the time complexity of TA is $O\left(\left(\frac{n}{2}\right)!\right)$.

TA is a generic algorithm; therefore exploiting some particularities of the schema matching problem leads to new algorithms that can be created based on the main idea of TA. Considering the following local and global aggregation functions:

$$f^{(i)}(\sigma, M^{(i)}) = \sum_{j=1}^{n} M^{(i)}_{j,\sigma(j)} \text{ (21) from [32]}$$

$$\left\langle \vec{f}, F \right\rangle(\sigma) = F(f^{(1)}(\sigma, M^{(1)}), \ldots, f^{(m)}(\sigma, M^{(m)})) = \sum_{i=1}^{m} k_i f^{(i)}(\sigma, M^{(i)}) \text{ (22) from [32]}$$

G-Aggregator can be written in the following manner, combining the above two formulae [32]:

$$\langle f, F \rangle(\sigma) = \sum_{j=1}^{n} \sum_{i=1}^{m} k_i M^{(i)}_{j,\sigma(j)} \text{ (23)}$$

The vector notation of l-aggregator has been replaced by the scalar notation to express an important property of the above formula (23): local and global aggregator functions commutability. With this property, TA can be improved leading to a new algorithm called Matrix Direct (MD) which has polynomial time complexity and not factorial or exponential as TA.

2.10 Similarity Flooding

A new approach in solving schema matching problem is presented in [15] and this approach is based on the Similarity Flooding (SF) algorithm. The core idea of this algorithm is that if two schema components (one from the source schema and the other from the target schema) are similar, then there are more chances to find other similar components in the neighbourhood of the original two components in each schema. Therefore, the similarity between two schema components is propagated in the semantics of the two schema, hence the name of the algorithm.

The SF algorithm is applied on schemas that are alike, i.e. two SQL schemas or two XML schemas. It does not work if one schema is an SQL schema and the other is an XML schema.

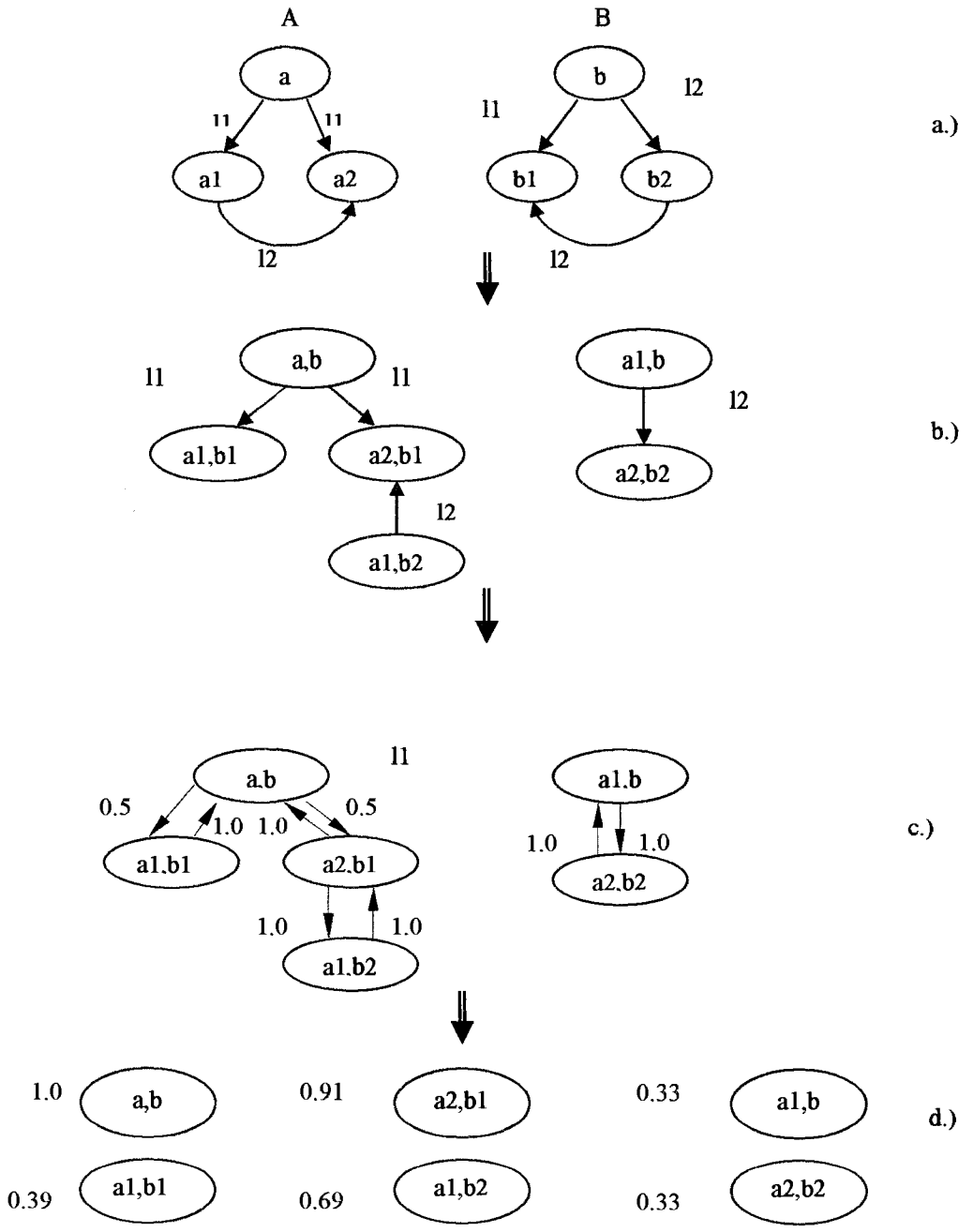


Figure 2.4 SF algorithm [15] a.) Schema representation b.) PCG c.) Similarity propagation d.) Similarity Values

Schemas that are subject to the matching process are represented internally as directed labeled graphs – Figure 2.4 a.). The next step of the algorithm is the presentation of the two schemas in the form of pair-wise connectivity graph (PCG) [15] – Figure 2.4. b.) – and it represents the similarity propagation graph. In other words, a PCG can be seen as a composition of the two (A and B) directed labeled graphs of the two schemas; this composition is similar with the composition of two Finite State Machines graphs. A PCG is formally defined in [15] as follows:

$$((x,y),p(x',y')) \in PCG(A,B) \Leftrightarrow (x,p,y) \in A \; and \; (x',p,y') \in B \quad (24)$$

Where x and y (x' and y') are source and target nodes and p is the label of the edge between x and y (x' and y') in directed labeled graph A (B).

If node "a" from A and node "b" from B are similar, then there is a high probability that a1 and b1 be similar because "a" is connected with "a1" through "l1" and "b" is connected with "b1" through the same "l1". Therefore, node "(a,b)" from PCG is connected with node "(a1,b1)" from PCG through "l1".

Figure 2.4. c.) represents the similarity propagation graph for directed labeled graphs A and B and it is obtained from pair-wise connectivity graph by adding an edge in the opposite direction for every single edge in PCG. In this new graph, all edges have different labels, named "*propagation coefficients*"– [15]. These coefficients are regarded as weights in the interval [0, 1], expressing how much is propagated to their neighbours from the similarity of a pair of nodes (e.g. "a" and "b"). For instance, there are two "*l1*" edges from (a,b) in the pair-wise connectivity graph: one towards (a1,b1) and another towards (a2,b1). This means that the similarity of "a" and "b" is propagated in two directions, therefore the edge from (a,b) towards (a1,b1) has a weight of 0.5 (i.e. w((a,b),(a1,b1)) = 0.5) and the edge from (a,b) towards (a2,b1) has the same weight of 0.5. In the case of the edge "l2", this edge always propagates similarity in one direction, so the resulting weights in the similarity propagation graphs are 1.0. All the new added

edges in the similarity propagation graph (the ones in opposite direction of each original edge in PCG) have weights of 1.0.

A mapping is represented in the Similarity Flooding algorithm as a similarity measure – a real valued number in the interval [0, 1] – between nodes from directed labeled graph A and directed labeled graph B.

$$\sigma(x,y)\in[0,1], \textit{where } x\in A \textit{ and } y\in B. \quad (25)$$

The core idea of SF algorithm is the iterative calculation of similarity values. The formula for similarity propagation is:

$$\sigma^{i+1}(x,y)=\sigma^{i}(x,y)+\sum_{(a_a,p,x)\in A,(b_a,p,y)\in B}\sigma^{i}(a_a,b_a)\cdot w((a_a,b_a),(x,y))+$$

$$\sum_{(x.p.a_d)\in A,(y,p,b_d)\in B}\sigma^{i}(a_d,b_d)\cdot w((a_d,b_d).(x,y)) \quad (26)$$

Where:

- $\sigma^{0}(x,y)$ - The similarity value obtained from matching the labels of nodes x and y.
- a_a belongs to the set of ancestors nodes of x; b_a belongs to the set of ancestors of y
- a_d belongs to the set of descendents nodes of x; b_d belongs to the set of descendents of y
- $w((a_a,b_a),(x,y))$ and $w((a_d,b_d),(x,y))$ are the propagation coefficients between (x,y) and its ancestors and descendents, described in the above paragraphs.

The iteration stops when the Euclidian distance between two consecutive mappings is less than a given threshold.

Similarity Flooding is a simple structural algorithm that can be used for matching data structures; also [15] presents some approaches in selecting the matches from the overall

matching results. The results of experiments are measured in terms of information retrieval theory [22].

2.11 Schema Matching Software Tools

Many academic and commercial software tools have been developed over the past decades. In the following sections, some of these tools are reviewed in an attempt to expose as much as possible their strengths and limitations.

2.11.1 COMA

COMA is a schema matching system that combines matchers "in a flexible way" [3]. A single match criterion such as element names correspondence is not enough to obtain a consistent mapping of two schemas. Therefore, there is a need to take into consideration other criteria such as element data types and overall structure of schemas. COMA implements different match strategies that can be used according to the concrete match problem in question, supporting different types of schemas such as XML schemas and relational schemas.

Match algorithms provided by COMA can be extended and the matching process is iterative and interactive i.e. a user can provide further information and validation of it. A very important feature of COMA is that the result of a matching process can be reused. The first step of a semi automated schema matching process is to create a schema representation. COMA represents a schema – XML or relational – as a directed acyclic graph and all COMA matching algorithms use this representation. The result of a schema mapping process is a mapping of schema elements along with a similarity value – 0 for strong dissimilarity and 1 for strong similarity. The focus is on 1:1 mapping but matching algorithms can find multiple matching candidates and then the user can keep or remove some of them.

COMA provides two major types of matchers: simple and hybrid. Simple matchers are as follows:

- Affix: a string matcher that looks for common prefixes and/or suffixes of schema element names;
- n-gram: strings are compared with regards to a sequence of *n* characters.
- EditDistance: element names similarity is computed based on Levenshtein metric.
- Soundex: string phonetic similarity is computed based on their soundex codes.
- Synonym: this matcher computes similarity of element names using an external dictionary and similarity values are 1.0 for synonymy relation and 0.8 for hypernymy relationship between element names.
- DataType: this matcher uses a table that links together similar generic data types.

Hybrid matchers use a combination of simple matchers and/or other hybrid matchers and they represent the core feature of COMA. These are the two element level hybrid matchers:

- Name: consists of tokenization, abbreviation expansion then applies Affix, Trigram (n-gram) and synonym simple matchers;
- NamePath: elements are matched based on structural aspects (hierarchy) and their names;
- TypeName: matches elements based on the above Name hybrid matcher and DataType simple matcher;
- Children: computes similarity of inner elements based on similarity of their descendents elements;
- Leaves: computes similarity of if inner elements based only on similarity of their leaf elements similarity.

One of the core features of COMA is the reuse of previous match results. Schema and Fragment are the main reuse oriented matchers. Those matchers are based on match result composition. COMA generates a mapping between two schemes Sa and Sc based on previous results of matching Sa with Sb and Sb with Sc. The two mappings are composed using a *MatchCompose* operation and the result is the matching between Sa and Sc. The

result of a schema matching process in COMA is a mapping along with similarity values for schema elements. When two matching results are composed, the similarity values of the resulting matching are computed as follows:

- Max: resulting similarity is the maximum of the two similarities;
- Weighted: similarity is a weighted sum of the two similarities:
- Average: particular case of "Weighted" strategy;
- Min: resulting similarity is the minimum of the two similarities;

For experimental results, five purchase order XML Schema documents from www.biztalk.org have been used and detailed results are presented in [3]. There is no further discussion on how the results of a schema matching process can be used in a real world data exchange application.

2.11.2 COMA++

COMA++ [4] is a significant improvement of COMA [3] and supports the following schema formats: XML Schema (XSD), XML Data Reduced, ontology (OWL) and relational schema. A MySQL relational database is used as the repository for COMA++.

COMA++ consists of the following building blocks:

- User interface: Java Swing
- Execution Engine: performs match iterations; each match iteration consists of: component identification, matcher execution using matchers available in the extensible Matcher library to obtain component similarities and similarity combination.
- Schema Pool
- Match Customizer
- Mapping Pool
- Repository

COMA++ features can be summarized as follows, using the taxonomy defined in [1]:

- Schema type –XML Schemas, XDR, OWL, database schema
- Schema internal representation: directed acyclic graph
- Match granularity: element level and structure level
- Match cardinality: 1:n
- Schema level match:
    - Name based: name equality, synonyms, soundex, edit distance (Levenshtein)
    - Structure matching
- Reuse: match composition, repository of matches
- User feedback: match refinement;
- Area of expertise: data integration, ontology integration

The main features of COMA++ are fragment based matchers to cope with large schemas and the reuse of previous matching results. Experimental results are described in [4]. An extremely large schema such as JXDM [71] cannot be loaded in COMA++. There is no further discussion on how the results of a schema matching process can be used in a real world data exchange application.

2.11.3 Clio – IBM and University of Toronto

Main features of Clio [7] are:

- Generation and management of schemas
- Schema representation in Clio: nested relational model
- Semi-automatic generation of mappings between schemas
- User can view, add, update and remove correspondences between schemas (mappings management); it supports incremental mappings

Clio's Integration Engine consists of the following three components:

- Schema Engine: governs schema loading into the application – one or more schemas; it makes possible for the user to validate and add information about the schema through a graphical user interface;
- Correspondence engine: generates and manages correspondences between schema elements
- Mapping Engine: creates and updates and maintains mappings between pairs of schemas.

Using schema matching tools taxonomy [1], Clio has the following main features:

- Schema type – generic schemas (database, XML)
- Schema internal representation: nested relational model
- Match granularity: element level and structure level
- Match cardinality: 1:1, n:m [1]
- Schema level match:
    - Name based: name equality and synonyms,
    - Structure matching
- Instance level match:
    - Constraint oriented: finding foreign keys
- Reuse: incremental mappings
- User feedback: mapping refinement
- Area of expertise: data integration and transformation

As per [7], one of the future improvements of Clio would be the ability to handle nested schemas in the matching process. Another improvement would be bi-directionality, i.e. The Clio system could be used for bi-directional data exchange by implementing the ability to invert a mapping.

2.11.4 LSD – Learning Source Descriptions

LSD [1, 10, 11] system is based on machine learning techniques, so it needs a training phase before the main matching phase. LSD works with both schemas and data. This is a major limitation fact because schema instances – data – may not be available.
The training phase consists of the following steps:

- the user must provide a set of initial mappings: this step uses only schema, it doesn't need data;
- source data is automatically added to LSD system;
- training data is created based on initial mappings and source data and it is used by training process of base learners;
- Train the base learners: each learner analyzes training data and builds an internal structure that will be used in subsequent matches. These internal structure are the output of the training phase;
- Meta-Learner training: in this step, the meta-learner computes and assigns a confidence factor for each pair of learner and label. The meta-learner knows the correct label, so the confidence factor is a measure of the accuracy of a base learner' ability to find the right label.

After the training phase, the matching phase starts and it has the following main steps:

- Load real data
- Match source tags: this step consists of using base learners and then meta learner combines the results obtain from them.
- Applies a constraint handler to refine 1:1 mappings.

Base learners implemented in LSD are:

- Name Matcher: it matches XML element tag names; it works on data, not on schema. The match process uses the nearest-neighbor classification model developed by Cohen and Hirsch (Whirl);
- Content Matcher: it matches XML elements with respect to their data content and uses the same approach as in Name Matcher;

- Naïve Bayes Learner: this learner uses Bayesian classifier approach on data instance;
- County Name Recognizer: this is a specific learner for the real estate domain. It shows that LSD can be extended with learners for each domain of applicability.
- XML learner: this learner has been added to handle nested data and correct Naïve Base Learner.

As per [1], LSD has the following main features:

- Schema type – only XML Schemas (xsd)
- Schema internal representation: XML schema trees
- Match granularity: element level and structure level
- Match cardinality: 1:1
- Schema level match:
    - Name based: name equality and synonyms,
    - Structure matching
- Instance level match:
    - Text oriented: Bayesian learners
    - Constraint oriented: list of valid names according to the domain
- Reuse: training matches are used for matching validation
- User feedback: for training data and match refinement
- Area of expertise: data integration

The main concern of LSD is matching accuracy. LSD performs a mapping between data sources using a *single* mediated schema. There is no repository for mappings, so previous mappings cannot be reused or updated. A mapping is not used to generate a transformation of source data into a data format that is validated by target schema different than the mediated schema.

2.11.5 Cupid

Cupid [12] has the following main features:

- Automated linguistic based matching;
- Simultaneous element and structure based matching;
- Its focus is on atomic elements where most of the schema semantic resides.

Internal representation of schema is a tree; like almost all rule based approaches, Cupid computes similarity coefficients between schema elements and then it builds a mapping based on those values. Similarity coefficients are obtained in two phases: linguistic match and structural match.

The first phase of the schema matching process is the linguistic matching, which works on schema element names and consist of the following steps:

- Normalization: it is actually a tokenization, then an expansion of abbreviations and then elimination of articles, prepositions, etc. This task is accomplished using a dictionary for domain specific terms.
- Categorization: classify schema elements based on their data type and name, e.g. a class of elements of date type, another class for elements describing currency or other numeric values, etc.
- Comparison: in this phase, similarity coefficients are computed based on previous steps. These coefficients are called *lsim*.

The second phase of the schema matching process is the structural matching process. The result of this phase is a set of structural similarity coefficients, *ssim* – real numbers in [0, 1] interval. The mapping is constructed using weighted similarity, which is a weighted mean between *lsim* and *ssim* using predefined constant. The core module of structural matching phase is the *TreeMatch* algorithm. This $O(n^2)$ algorithm is based on the following rules:

- Schema elements that are leaves in the schema tree are similar if they have a linguistic and data type similarity
- In the case of schema elements that are not leaves, they are considered similar if they have a linguistic similarity and sub-trees rooted from them are similar.
- In the same case of schema elements that are not leaves, they are considered *structural* similar if their leave are similar.

Cupid has the following optimizations:

- In the case of optional schema elements, their influence on structural match is reduced;
- The user can provide an initial mapping and those schema elements manually matched are assigned a maximum linguistic similarity.
- In the case of very deep schema trees, some of the leaves may not have any influence in the match of non-leaves elements, so those leaves elements are not taken into consideration for the structural match of non-leaves schema elements.

The final experiment was conducted on two XML schemas from the inventory domain – purchase orders. Linguistic matching along with the provided dictionary produced more meaningful mappings that the other schema matching tools (DIKE and MOMIS) used for comparison. All the results of the experiment are presented relative to the above mentioned tools.

Cupid features can be summarized as follows – using taxonomy defined in [1]:

- Schema type – database and XML schemas
- Schema internal representation: schema trees
- Match granularity: element level and structure level
- Match cardinality: 1:1 and n:m
- Schema level match:
    - Name based: name equality and synonyms – linguistic match
    - Structure matching – *TreeMatch* algorithm
- Reuse: mappings can be used for more complex schemas that are to be mapped

- User feedback: initial mapping
- Area of expertise: data integration, data exchange

Scalability – i.e. large and very complex schemas – is listed as an item for future work along with more elaborated test benchmarks

2.11.6 Corpus-based Schema Matching - MKB

The main purpose of this tool (Mapping Knowledge Base, [14]) is to leverage experiences from previous schema matching processes. This can be accomplished by building a knowledge base.

MKB consists of two parts:

- Schemas and Mappings: repository of schemas and mappings
- Learned Knowledge: every schema element encountered by MKB is stored and maintained in a repository. This knowledge can be used later for a probabilistic prediction of a match.

MKB employs the following base learners:

- Name Learner: works on schema element names and consists of decomposing each name in its n-gram and it is based on term frequency/inverse document frequency measure;
- Description Learner: takes into consideration schema annotations and documentation elements;
- Instance Learner: works on instances, finding specific features that can be exploited in the schema matching process;
- Data Type Learner: works on data type of schema elements;
- Structure learner: tries to find elements that appear along with another element using their neighborhoods – similar idea with Similarity Flooding algorithm [15].
- Meta-Learner: used to estimate relevance for each base learner.

Using taxonomy defined in [1], MKB has the following main features:

- Schema type – generic schemas
- Schema internal representation: unknown
- Match granularity: element level and structure level
- Match cardinality: 1:1
- Schema level match:
    - Name based: name equality
    - Structure matching
- Instance level match:
    - Text oriented: Bayesian learners
- Reuse: training matches are used for matching validation
- User feedback: for training data
- Area of expertise: data integration

The main purpose of MKB is schema matching and resulting mappings are used only as a knowledge base; they are not used to generate transformation code for data exchange. Scalability is debatable because of the maintenance knowledge base [14]. Experiments were conducted on simple schemas and not on very complex ones.

2.11.7 SMART

The main feature of SMART [13] is that schema matching is done on components of conceptual schema and not direct on schema attributes. Conceptual schemas are an internal representation of real world schemas (database or XML). The user sets inclusion and/or equivalence relationships between classes of the conceptual schema. These relationships are used by an inference engine.

The core components of SMART are:

- Reverse Engineering Engine: transforms input schemas (source and target) into conceptual schemas
- Inference Module for Class Relationships: transforms relationships give by the user into description logic. RACER is the inference engine for the description logic
- Attribute Matching Engine: takes two classes that are in an equivalence relationship and it finds the relationship between attributes of each class.

Using taxonomy defined in [1], SMART features can be summarized like this:

- Schema type –XML Schemas (xsd) and database schemas
- Schema internal representation: conceptual schema
- Match granularity: element level
- Match cardinality: 1:1
- Schema level match:
    - Name based: name equality;
- User feedback: for establishing relationships
- Area of expertise: data transformation

In conclusion, SMART uses a conceptual schema for its internal schema rendition, which is different than other schema matching tools that use tree structures or graphs. However, SMART has the following disadvantages:

- It does only element level match, there is no structure level match;
- It does not work with large and complex schemas [13]
- It does not allow user for mapping refinement
- No repository for mappings
- It generates only XQuery, it does not offer a complete solution for data exchange.

2.11.8 iMAP

The majority of schema matching research has been focused on finding simple and direct 1:1 matches. The novelty of iMAP [17] is that it finds complex matches between schema elements. These matches are expressions like "Name" <-> *concat*("FirstName", "LastName"). This problem is addressed by implementation of a match generator which consists of search modules, so called searchers. Those searchers are classified based on attribute data type, i.e. text searchers try to find complex matches between character strings attributes, numeric searchers work with numeric attributes, date searchers for date attributes. iMAP uses domain knowledge during searching of complex matches and the result is presented in the form of a ranked list. Each match candidate also has an explanation assigned to it to ease user selection and validation.

iMAP provides functionality for the following actions:

- Schema matching – finds all correspondences between schema elements.
- Generates queries that facilitate automated data exchange, based on results from previous step

iMAP architecture consists of the following core modules:

- Match generator: for each attribute from target schema, it generates simple and complex matches. This fact leads to the conclusion that iMAP perform only unidirectional matches. If data needs to be translated in the reversed way, a new entire match process needs to be performed;
- Similarity estimator: computes a similarity factor for each match found by the match generator;
- Match selector: outputs the best match for each attribute ranked by similarity factor.

A searcher has the following features:

- Search strategy: beam search – implements a scoring function which is used to evaluate each match candidate, then it keeps a pre-defined number of high scores at every level of search in the tree.
- Match evaluation: uses machine learning, statistics and heuristics methods to compute the score for each match candidate.
- Termination condition: because of the unlimited number of complex match combinations, there is a need for a termination condition. This condition is set up using maximum scores of each match, i.e. if the difference between two highest scores of match candidates in consecutive iterations is less than a pre-defined value. When this condition is met, the algorithm returns a pre-defined number of highest score match candidates. The weakness of this condition is the possibility of reaching a local maximum and the best match candidate with highest score could be missed.

iMAP implements the following searchers:

- Numeric Searcher: finds complex matches between numeric attributes using Kullback-Leibler divergence measure from statistical natural language processing.
- Category Searcher: implements complex mappings between categorical attributes. Categorical attributes have their space of values defined as an enumeration.
- Schema Mismatch Searcher: relates source schema instance to the target schema and vice-versa, trying to find links between specific properties of data and attributes in the other schema.
- Unit Conversion Searcher: converts pounds to kilograms or kilometers to miles, for instance.
- Date Searcher: looks for complex mappings between date attributes using date ontology.

Real world domains analyzed during experimentation were: real-estate, cricket, financial wizard and inventory. Complex match accuracy was 9% in the case of inventory domain and for others it was in the range of 33-55%. Other conditions were taken into

consideration such as domain constraint and data overlap and these conditions led to an improvement of accuracy of up to 64%.

Taking into consideration the taxonomy defined in [1], iMAP has the following main features:

- Schema type – database schemas
- Schema internal representation: unknown
- Match granularity: element level
- Match cardinality: 1:1 and complex matches
- Schema level match:
    - Name based: Naïve Bayes
- Instance level match:
    - Text oriented: Bayesian learners
    - Domain constraint, data overlap
- Reuse: training matches are used for matching validation
- User feedback: explanations of each complex match
- Area of expertise: data integration
- Remarks: machine learning algorithms

iMAP does not provide a repository for mappings; the user can refine those mapping only by changing some pre-defined values and thresholds, or in a more difficult way by designing new searchers. Based on a mapping, iMAP can generate an SQL query to facilitate data exchange between source and target only. On the other hand, the termination condition is debatable and there was no discussion of the case of a local maximum of highest available candidate match scores.

2.11.9 VisAXSM

The novelty of VisAXSM [18] is that source and target XML schemas are iteratively analyzed by "analysis agents". Each agent has its "own" matching strategy, being

designed to allow "plug and play" "*modus operandi*", i.e. to be added or removed by the user at any time. Also, the user has the capability to update and validate a match. The following mapping agents are described in [2]:

- Exact Name Matcher: if schema element names are the same, then there is a match.
- Partial Name Matcher: looks for common sub-strings in schema element names and, if it finds any, it reports a match. The problem is the case of PatientName and DoctorName, both may be reported as match and semantically are totally different.
- Levenshtein Name Matcher: computes Levenshtein distance between two schema element names. This distance is the number of edit operations needed to change a string in another: the lower the distance the higher probability of a match.
- Element Type Matcher: if data types of schema elements that are matched are the same, then schema elements may match. This matcher is used in conjunction with Name matcher agents and works well with small subsets of each schema in order to avoid false positive matches.
- Record Type Matcher: compares sets of schema elements with respect to their type.
- Synonym Matcher: compares schema element names and/or schema element type names to see if they are synonyms. As an observation, [2] does not provide a dictionary such as WordNET [64], nor an external table of synonyms for this agent matcher.
- Domain-specific Matcher: the same as Synonym Matcher but it takes into consideration the application domain of both schemas
- Exact Data Value Matcher: takes into consideration XML schema instance data elements rather than schema elements.
- Partial Data Value Matcher: it uses XML schema instance as well and it computes a likelihood match.

The result of an overall match can be updated and validated by the user. Another important feature of VisAXSM is the generation of XSLT scripts based on a mapping.

These scripts take as input an XML data file in the source schema format and convert it to an XML data file in the target schema format. As per [2] a very large schema is considered to have hundreds of elements. However, JXDM [71] schema has about 5000 element declarations, more than 500 global type definitions and more than 60 global attribute definitions. In addition, many of the element declarations have anonymous type definitions, so the schema can go easily over 7000 schema elements. As per [2], XSLT transformations have been generated based only on simple mappings for basic data specifications and XSLT transformations based on more complex mappings need further investigation.

2.11.10 TranScm

TranScm is a data translation tool [25, 1] and its main idea is that data needs to be translated within the same domain of activity, "much of the structure of the source data is very similar to that of the target translated data" [25]. This tool uses the rule based approach by finding similarities and differences between source and target schemas and using them later in data translation.

Using schema matching tools taxonomy [1], TranScm has the following main features:

- Schema type – SGML (DTD) and OODB
- Schema internal representation: labeled graph
- Match granularity: element level
- Match cardinality: 1:1
- Schema level match:
    - Name based: name equality and synonyms,
- User feedback: mapping refinement and new matching rules can be defined
- Area of expertise: data transformation, especially for Web documents.

In the case of multiple match cardinality (n:m), data cannot be translated and the user must update the mapping or define new rules in order to obtain a 1:1 match cardinality.

The application does not consider previous matching, i.e. there is no reuse feature available. Authors of [25] don't present an evaluation of their application in terms of precision and recall and don't make a scalability analysis of TranScm.

2.11.11 Map Force

This is a commercial tool which is included in XMLSpy Studio Professional from Altova [8]. MapForce is described as follows on its web site:

- Altova MapForce® provides a visual XML-to-XML mapper
- Mapping any combination of XML, database, flat file, and EDI data via a graphical interface
- Generates XSLT 1.0, XSLT 2.0, XQuery, Java, C++, or C# to implement data mappings (XSLT and XQuery for XML-to-XML mapping only)
- Data mapping transformations are executed and the result can be saved
- Process and filter data with an extensive library of built-in and user-defined functions
- Creates complex data processing functions

The (semi) automatic matching process in MapForce is very limited due to the following factors:

- There is no dictionary and string decomposition engine available; matching element names is only done on a character strings match basis, so an element called "Surname" will never be matched with another element called "LastName";
- The matching process is based on a correspondence performed initially by the user, i.e. the user selects 2 elements that match – or he/she assumes that match – and then the (semi) automatic matching will try to find matches between children of those selected elements. The whole structure of a schema needs to be

considered when a matching process is performed because that structure contains almost the whole meaning of the schema;

- A major limitation is that the types of elements (simple type or complex type) are not evaluated and taken into consideration properly;
- Generated XSLT scripts do not transform data in an expected result, i.e. the transformed source data is not validated by the target schema.
- MapForce does not scale very well, it fails to load very large schemas (thousands of elements).

2.12 Current technologies

This section contains an overview of some technologies employed in solving XML Schema matching problem used in this thesis. As such, RDF, Sesame (RDF query architecture) and WordNet are presented in the following sections.

2.12.1 RDF – Resource Description Framework

The Resource Description Framework is a World Wide Web Consortium (W3C) standard for describing web resources; "The Resource Description Framework (RDF) is a general-purpose language for representing information in the Web" [39]. This description of web resources should be domain independent, suitable for any domain of activity available on the World Wide Web.

As per [48], RDF has the following motivations:

- to provide information about Web resources
- to ease data processing: data from one domain should be available to other domains of activity
- to provide a universal language for data processing by automated systems – software agents

- to present information in a flexible way

Any expression in RDF is described by a set of triples consisting of a subject, a predicate and an object. The predicate denotes a sort of relationship between the subject and the object. Any RDF triple can be represented in a graphical form as an an arc between two nodes: the arc represents the predicate and nodes are the representation of the subject and the object. The direction of the arc is always from the subject towards the object, as depicted in the following figure:

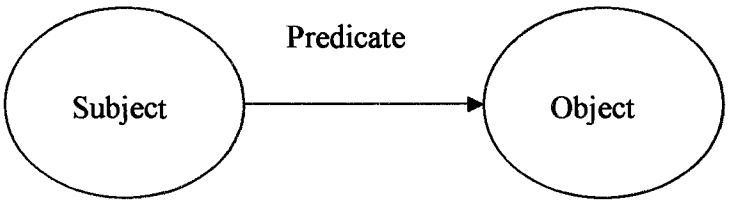


Figure 2.5 RDF triple (subject, predicate, object) graphical representation

As stated before, the three components that belong to an RDF triple are:

- Subject: this component can be an RDF URI reference of a blank node. An RDF URI reference (Uniform Resource Identifier) is a UNICODE string of characters that does not contain any control characters and conforms to RFC 2396 URI. "A blank node is a node that is not a URI reference or a literal" [48]
- Predicate: is always an RDF URI reference
- Object: can be one of the following: RDF URI reference, a literal or a blank node. All literals in RDF graphs are represented as Unicode strings and can be as follows:
    - Plain literals: lexical form with an optional language tag
    - Typed literals: lexical form with a data type URI

One of the most important features of RDF is *reification*, which means that any RDF statement can be the subject or the object of another RDF triple.

As per [63] RDF provides:

- a Standard Representation Language for metadata, relying on directed labeled graphs;
- a Schema Definition Language (RDFS) which consists of a set of labels for subjects and objects – which are defined as classes – and for predicates – which are defined as property types;
- an XML syntax for representing metadata (RDF) and schemas (RDFS).

The next step is to find a way to query the information stored in RDF(S) format.

2.12.2 Sesame – RDF Query Architecture

Aduna [61] developed the Sesame application for On-To-Knowledge European IST project as a tool for storage and retrieval middleware for ontologies and metadata described with RDF and RDF Schema. Sesame resides on a Tomcat application server and communicates with client applications over HTTP. It is an open source framework storing, querying and reasoning with RDF and RDF Schema. Sesame can parse, interpret, query and store all RDF information.

Sesame works as a server in the following manner presented in Figure 2.6:

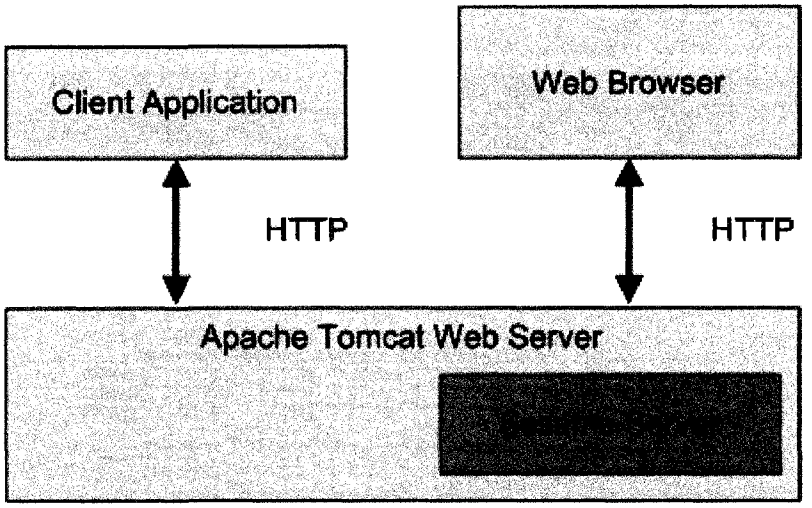


Figure 2.6 Sesame Server [62]

The Sesame architecture is presented in Figure 2.7:

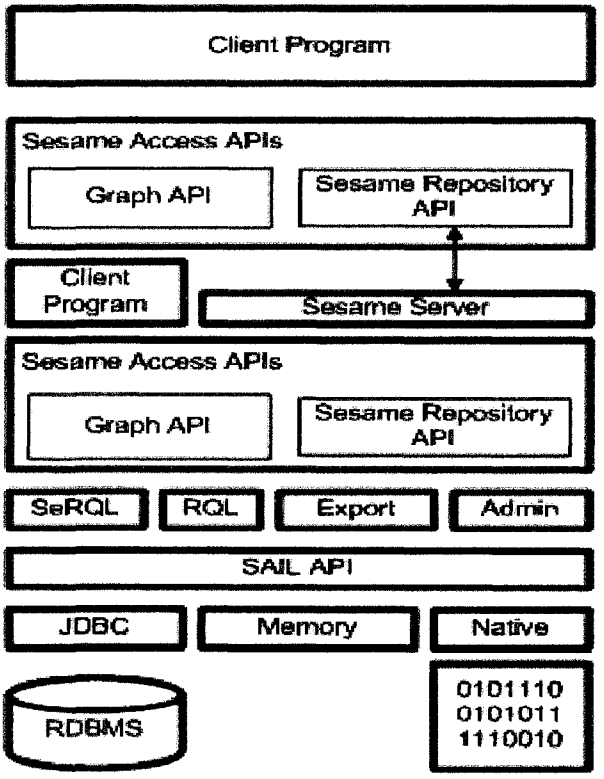


Figure 2.7 Sesame Architecture [62]

The Storage And Inference Layer (SAIL API) is the link between higher layers and data storage layer and the Sesame inference engine. The data storage layer can be represented by relational databases (RDBMS), memory or files. This layer between data storage and other Sesame components makes possible to implement Sesame on top of different RDBMS such as MySQL, PostgreSQL and Oracle [62].

Sesame functional modules are the clients of SAIL API module and consist of the following:

- SeRQL and RQL query engines;
- the RDF export module;
- the admin module

SeRQL (Seame RDF Query Language) has the following features [47]:

- *Compositionality* – this means that complex queries can be composed of simpler queries, i.e. the output of a query can be used in another query. The result of a SeRQL query is either a set of RDF statements (using CONSTRUCT statement) or a set of variable bindings (using SELECT statement).
- *Schema awareness* – a query language should be able to properly handle the data structure it works with, from the point of view of type checking and optimization.
- *Optional path expressions* – RDF is a framework for semi structured data; therefore any RDF query language must deal with this data that is not as structured as in a relational database system. For instance, a property may or may not have a value in a given instance; therefore an RDF query engine must be able to handle an optional match for this case. Optional path expressions are handled in SeRQL by placing them between square brackets.
- *Datatyping* – RDF uses XML Schema data types [36] therefore any RDF query language must support these data types. SeRQL handles data types explicitly – with the help of datatype() operator – and implicitly – related to the use of comparison operations. The current implementation of SeRQL supports only the following XML Schema built-in data types: <xsd:boolean>, <xsd:float>, <xsd:double>, <xsd:decimal> along with all sub-types of <xsd:decimal>.

RQL (RDF Query Language) is a predecessor of SeRQL and was proposed in [63]. RQL is a typed query language and consists of basic queries and iterators. Complex queries can be constructed using these basic queries and iterators, *compositionality* being a main feature of RQL. Basic RQL queries can be used to retrieve RDF descriptions, with a minimal knowledge of RDFS. This means that RQL has a schema awareness feature. Class and property hierarchies are traversed using *subClassOf* and *subPropertyOf*. In addition, RQL supports *generalized path expressions* using variables on labels for both nodes (subjects/objects/classes) and edges (predicates/properties) of the RDF graph. Another important feature of RQL is *datatyping*, where Boolean predicates such as "=", ">", "<" and "*like*" can be used. In conclusion, RQL has the same features as SeRQL.

The RDF export module exports the content of a repository formatted in XML serialized RDF.

The admin module is responsible for inserting RDF data and schema information into a repository, which can be either a relational database, memory or a file. RDF data and schema information is parsed – using an RDF parser – and then, the admin module communicates with the repository through SAIL. This information is added incrementally to the repository, i.e. if a statement is already present that statement is not added once more.

The Sesame Access API module is the link between a client program and Sesame functional modules and consists of two sub-modules:

- Graph API – provides fine-grained manipulation methods of the repository, i.e. individual RDF statements can be added or removed from the repository.
- Sesame Repository API – responsible for high level manipulation of the repository, such as querying and extracting RDF data and schema information.

A client program can also connect to a remote Sesame architecture using the Sesame Access API through a Sesame server over an HTTP connection.

The most important features of Sesame are:

- It can work in either local or remote configuration
- It can use a wide variety of data storage systems for its repository

2.12.3 WordNet

WordNet [64] is a lexical database for English language that can be used by software applications. The vocabulary of a language is defined as a set of pairs consisting of a *form* and a *sense*. A form is a string of characters and a sense is an element from a set of meanings. A word consists of a form with a sense and a dictionary is the ordered list of all the words of a language. A word can have more than one sense and two words that have one or more senses in common are called *synonyms*.

WordNet handles a set of semantic relations, such as:

- Synonymy – word senses are represented by sets of synonyms. Naturally, this relation between word forms is symmetric.
- Antonymy – the opposed sense of a form
- Hyponymy – "car" is a hyponym of "vehicle"
- Hypernymy – "vehicle" is a hypernym of "car"

WordNet is one of the best and most used machine readable dictionaries and it is the most suitable tool for linguistic matches within the wider domain of schema matching.

JWNL [65] is a Java library designed for the WordNet lexical database and provides relationship discovery and morphological processing. The WordNet database is accessed in the following ways:

- Standard file distribution;
- Database;
- In-memory map

A Java application can use JWNL to retrieve information form WordNet such as a word set of synonyms (senses) or other complex relationships between words (hypernymy, antonymy, etc.)

3 High Level Design

As this thesis is about matching and mapping two XML schemas, before proceeding to the description of the core of this thesis, the XML Schema concepts and language structures will be introduced.

3.1 Introduction

A markup language provides a mean to express additional information about a printed text and its origin is the publishing industry. A manuscript needs to be corrected and formed before being printed, i.e. information about typeface, style and size is added as an annotation to it by proofreaders who mark up the original manuscript. Nowadays information is exchanged at an unprecedented level in the history of mankind, especially in the information technology industry. Therefore, there was a need for a standard markup language and this is SGML.

This standard was too complex and too difficult for exchanging information over the World Wide Web so Hyper Text Markup Language (HTML) came onto the scene. The next markup language more elaborate and addressing major issues for documents on the internet major issues – is the eXtensible Markup Language (XML). It is extensible because it allows the user to define tags that are suitable with the information contained in that document. This fact is impossible in HTML because HTML has a fixed number of tags that cannot cover the wide variety of information available nowadays.

Another fact in XML is that the document must be well formed, i.e. there must be only one root for the entire document and every tag must have its corresponding closing tag; HTML does not enforce this kind of correctness within a document. Having the full freedom of defining of tags, the next problem that arises is how to determine whether the defined tags are the right ones and that they conform to a specific set of rules. These rules

can be specified using a Document Type Definition (DTD) or XML Schema Definition languages.

3.2 XML Schema

"XML Schemas express shared vocabularies and allow machines to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents." [33]. The sole purpose of an XML Schema is to define a class of XML documents; these documents are also called "instance documents" of a specific schema and conform to the rules defined in it, i.e. are validated by that schema. An XML document consists of elements; each element is delimited by a start and an end tag and can also contain attributes. There should be one and only one root element in an XML document, all the other elements must be descendents (in a direct or indirect way) of this root element. The XML Schema definition language has the capability to express this relationship between elements, also describing the type of elements and attributes that appear in that XML document.

3.2.1 XML Schema building blocks

The building block of an XML Schema is called schema component. There are three major groups of schema components. The first group consists of the following components, called primary components:

- Simple type definitions
- Complex type definitions
- Attribute declarations
- Element declarations

The first two primary components may have names. In the case where they don't have names, they are called anonymous type definitions. The last two must have names because they define elements and attributes that appear in an XML document.

The secondary components are:

- Attribute group definitions
- Identity constraint definitions
- Model group definition
- Notation declarations

The third group of schema components consists of "helper" components which are always part of other schema components:

- Annotations
- Model groups
- Particles
- Wildcards
- Attribute uses

3.3 Hermes – An architecture for XML Schema matching and mapping

Hermes is a novel architecture for XML Schema matching and mapping designed in this thesis. Hermes attempts to solve issues that have not been solved yet by the schema matching software tools presented in chapter 2 of this thesis.

A high level architecture of a generic match has been presented in [1] as illustrated in Figure 3.1:

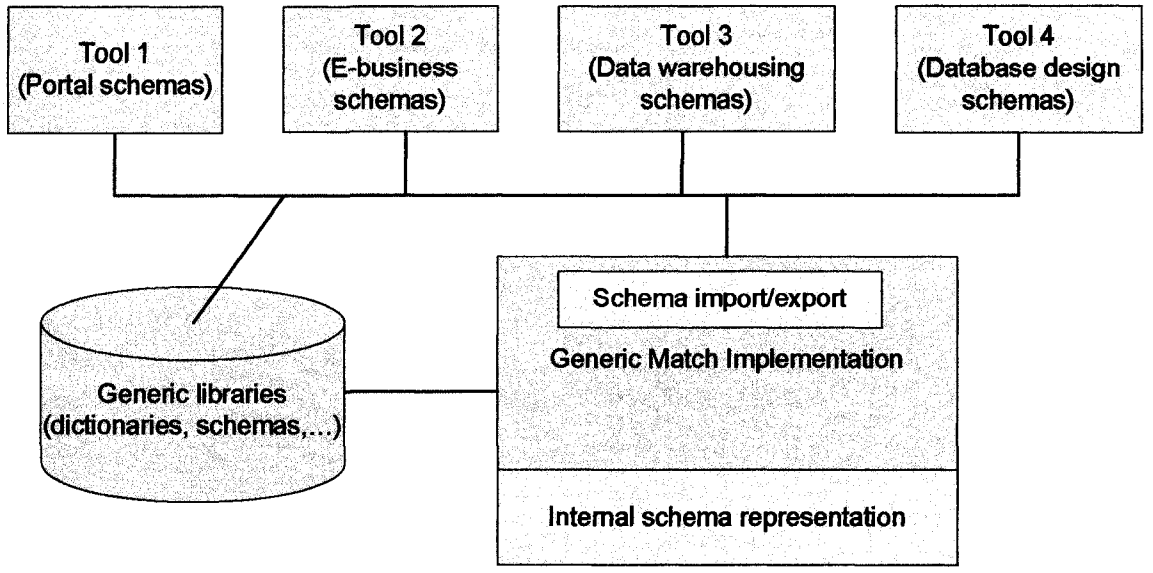


Figure 3.1 High-level architecture of Generic Match [1]

The four tools represent the main clients of the Generic Match implementation, like generic applications that use database schemas, E-business schemas (usually XML Schema documents), data warehousing schemas, etc. The generic match implementation must rely on a consistent internal schema representation, i.e. both schemas that are to be matched must have the same internal representation.

Figure 3.2 presents the high level architecture of Hermes – a (semi) automatic XML Schema matching/mapping and transformation tool.

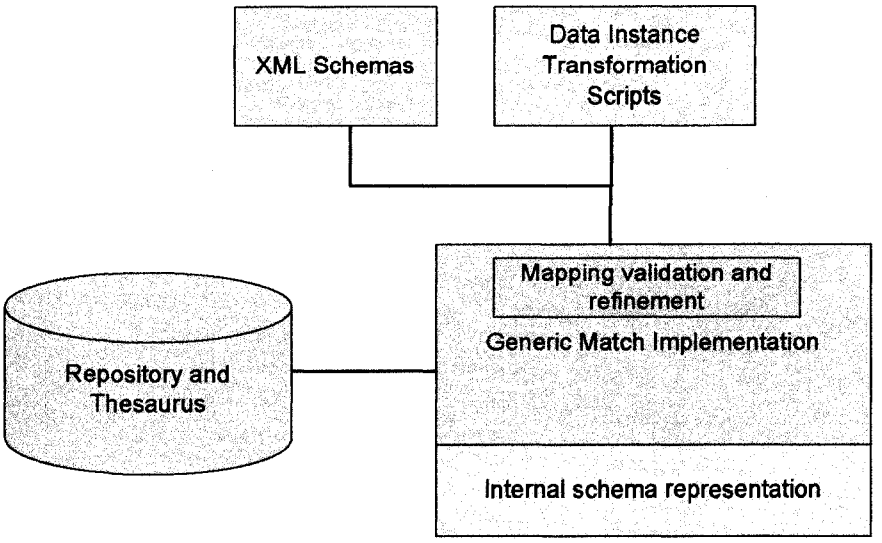


Figure 3.2 Hermes – High-level Architecture

This system accepts source and target XML Schemas as input, represented by "XML Schemas". The system produces two transformations: one from source data instance to target data instance, and the other, from target data instance to source data instance, represented by the block labeled "Data Instance Transformation Scripts". Source and target data instances are data instances that are validated by source and target XML schemas, respectively. A repository is used to store information discovered during the matching process and a mapping is constructed based on this information. A user with expert knowledge of the domain can then update and refine that mapping, and more importantly, can save and retrieve a mapping of two given XML schemas. This feature is represented by the block labeled "Mapping import/export". The "Thesaurus" block represents a generic dictionary of terms used during the XML schema components matching process. This dictionary of terms is actually used during the XML Schema components name matching process.

The following diagram presents the logical architecture of Hermes – a (semi) Automatic XML Schema Matching/Mapping and Transformation tool:

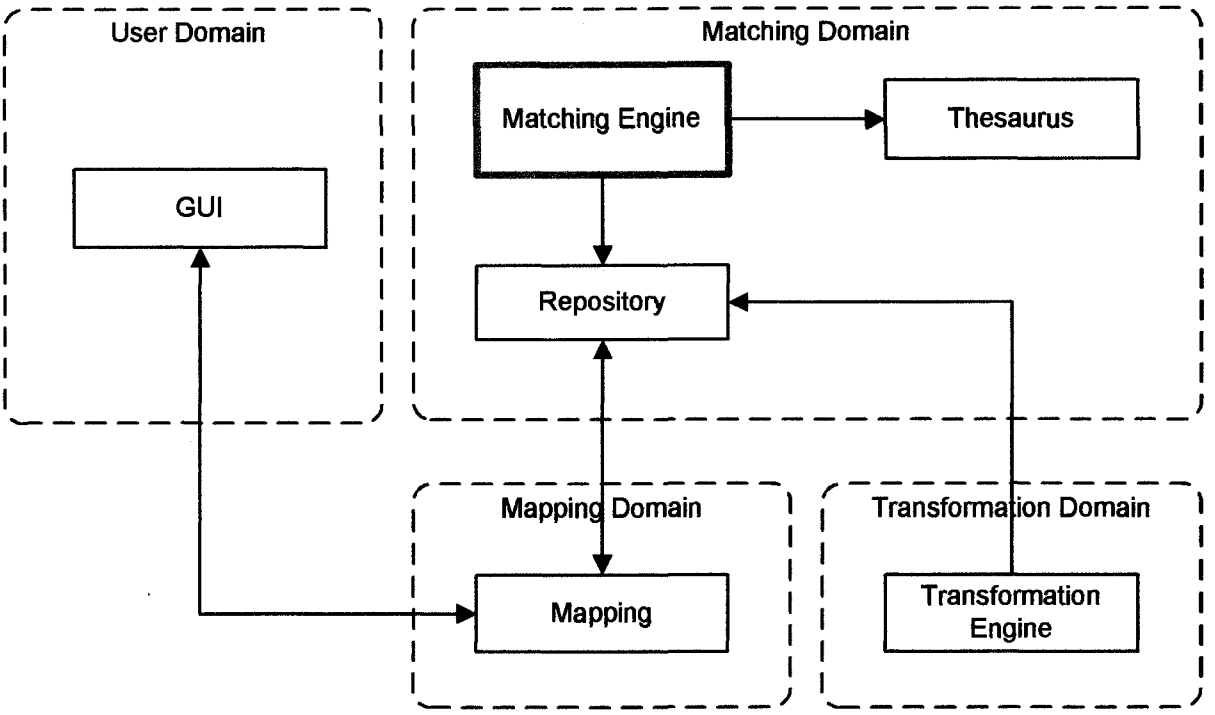


Figure 3.3 Hermes - Logical Domain Architecture

The Matching Engine within the Mapping Domain is the core component of Hermes and it performs the semi automatic matching of the source and target XML schemas. In addition, the Matching Engine uses a Thesaurus, a generic dictionary, during the matching process for names matching. The information acquired during the matching process is stored in the Repository. Based on the information available in the Repository a mapping is automatically constructed. This mapping can be refined by an expert user then the Repository is updated with the new configuration of the mapping. The Transformation Engine generates scripts that are used to transform (1) source data instances into data instances that are validated by the target XML schema and (2) target data instances to data instances that are validated by the source XML schema. GUI component within the user domain represents the graphical user interface and it shows the result of the schema matching process, i.e. a mapping. A user can update this mapping to obtain a more accurate final result of the overall schema matching process.

Figure 3.4 illustrates the generic component architecture view:

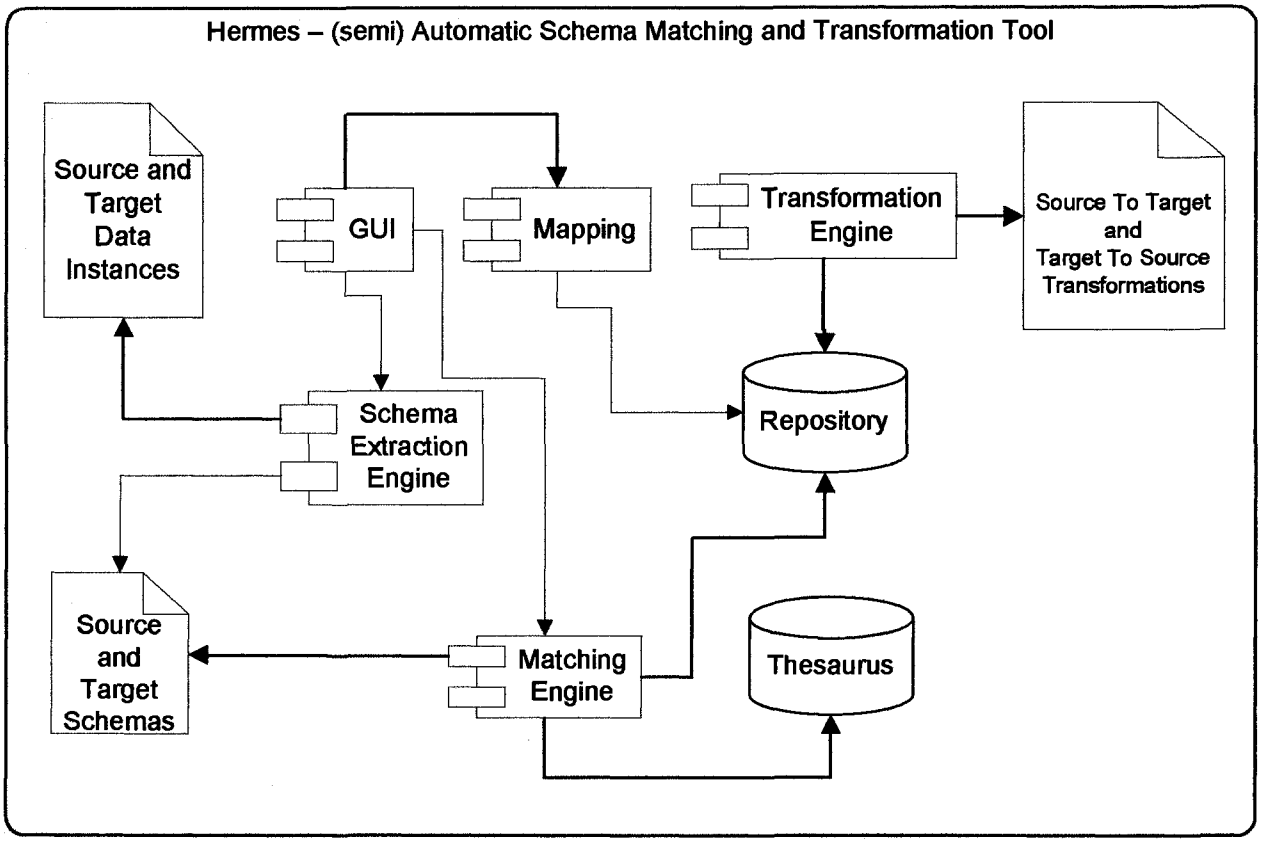


Figure 3.4 Hermes – Generic Component Architecture View

Hermes accepts two XML Schemas (source and target) or two XML documents as input. The output consists of two XSLT documents which can then be used to transform source and target data instances into data instances that are validated by target and source XML schemas.

3.3.1 Schema Extraction Engine

Extensible Markup Language (XML) is a very well established standard for data exchange over the World Wide Web. An enormous quantity of information is exchanged in this format even between simple computer networks not necessarily part of the World Wide Web. XML has also become a standard for configuration files for many software applications.

A major problem that arises regarding configuration files written in XML format is that from one release of an enterprise software application to a subsequent one, these configuration files can change. This fact leads to a very significant amount of work to be done for the deployment of new releases with regards to these configuration files. A common solution is a software tool that can populate the new configuration files with data from the old configuration file. One of the means of transferring data from one XML document to another with a similar structure is eXtensible Stylesheet Language (XSL) transformation scripts.

Consider the following situation: Company A submits a purchase order in an XML data format to Company B; however Company B may have a different data structure defined in its catalogs. Therefore, there is a need for a mapping process between the incoming purchase order data structure and the data structure of the supplier. A purchase order does not come with its XML Schema document, so a minimal XML Schema must be extracted from the incoming purchase order.

Would it be possible to generate an XSL transformation for two given XML documents with similar structure to transfer data from one XML document to another? The answer is affirmative and this task can be accomplished with the help of Hermes – a semi Automatic XML Schema Mapping and Transformation Tool.

If the user provides two XML documents (data instances instead of meta-data) as input, then the mapping tool needs to extract their schemas. The schema extraction problem has been presented in [31]. However, the software tool presented in [31] has some limitations. The Schema Extraction Engine designed in this thesis uses a different approach in solving this problem. This approach consists of an XSLT script that takes an XML document as input and generates an XML schema (XSD) that validates that XML document as output. Certainly, the generated XML schema is not unique, but it has the properties described in [31], such as correctness, conciseness, precision and readability. The resulting XML schemas are then passed to the Matching Engine. The Schema Extraction Engine has the following functionality:

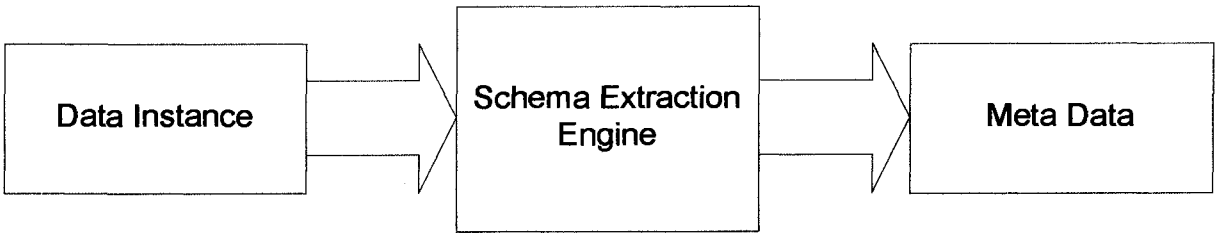


Figure 3.5 Schema Extraction Engine Data Flow

The architecture of the Schema Extraction Engine is presented in the following diagram (Figure 3.6):

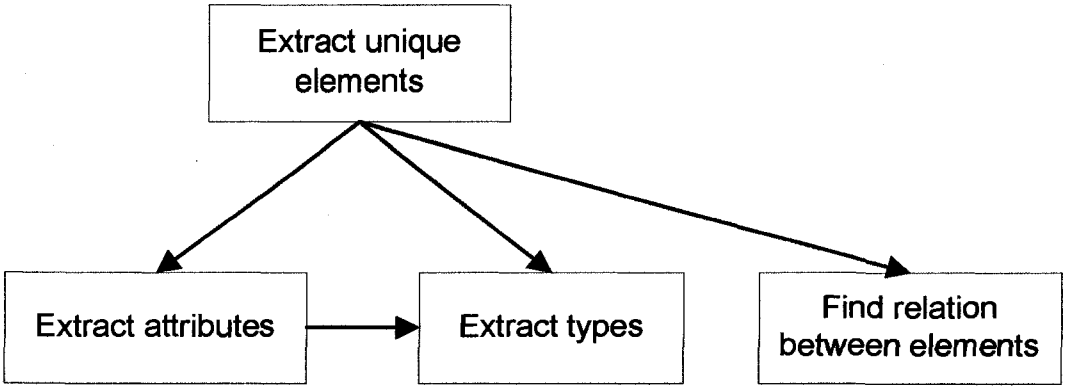


Figure 3.6 Schema Extraction Engine Architecture

The components of Schema Extraction Engine are:

- *Extract unique elements*: this system finds all XML elements and builds a list with all unique XML elements found;
- *Extract attributes*: for every single element in that list, its attributes are analyzed and a new list containing attributes is constructed;
- *Find relation between elements*: this system establishes parent, child and sibling relationships between unique elements found by the first system;
- *Extract types*: for every single element and attribute an XML Schema component type definition is constructed; for instance, all attributes found receive a simple

type definition. In case of XML elements, each element will have a simple or complex type definition, considering relationships between XML elements discovered by the previous block.

The XML Schema document that has been generated from an XML document (data instance) is not unique. There exist other XML Schema documents that would validate the XML document. These schemas may have some optional components which are not present in that data instance but still validate it.

3.3.2 Matching Engine – High Level Architecture

The Matching Engine (ME) is the core component of Hermes. This engine can be also used without a GUI component, from command line or as a component within another Java application. It takes two XML schemas and a Thesaurus as input. The Thesaurus is actually a dictionary, such as WordNet [64] and the result of the matching process is stored in the Repository.

The Matching Engine implements element and structural matching paradigms. These techniques were described in the literature review chapter of this thesis. The Matching Engine has been designed the following structural and behavioural design patterns [56] and relies on XML Schema Infoset Model [41, 42, 43 and 50].

The Matching Engine Architecture follows the XML Schema Standard [34, 35, and 36] and is presented in the following figure:

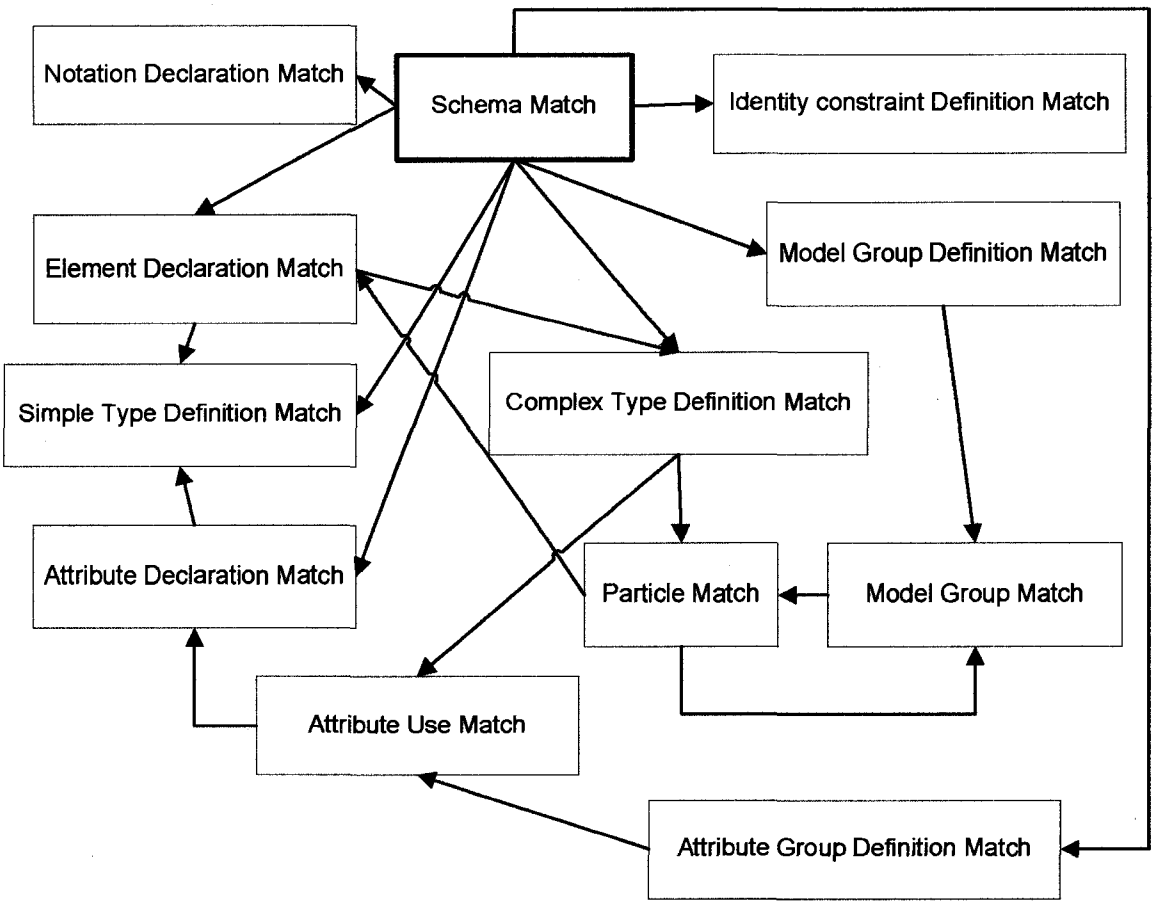


Figure 3.7 Hermes – Matching Engine Architecture

The two XML Schema documents (source and target) subject to the schema matching process are decomposed in their constitutive components and then these components are matched one by one. The "Schema Match" system takes as input the two XML Schemas. The "Element Declaration Match" system takes as input two XML Schema element declaration components. In fact, the "Element Declaration Match" system matches every element declaration component in each XML Schema one by one. An element declaration component consists of either a simple type definition component or a complex definition component; therefore the element declaration matching system will contain a reference to a simple type definition matching system or a reference to a complex type definition matching system. Alternatively, an element declaration system

can be referenced by the schema matching system or referenced by the particle matching system.

The particle matching system can be referenced or refer a model group matching system. The model group matching system is referenced either by the particle matching system or by the model group definition matching system and contains references to the particle matching system. Attribute matching system refers only a simple type definition matching system for the reason that an XML Schema attribute declaration always has a simple type definition. This attribute matching system can be referred by an attribute use matching system or by the schema match system.

The core idea of this thesis is to present a novel approach for XML Schema mapping problem. The step ahead presented in this thesis consists in the fact that the abstract **XML Schema Architecture** is resembled in the **Matching Engine Architecture**.

3.3.3 Repository

The Repository is the second important component of Hermes – after the Matching Engine – and this component stores all the information discovered by the Matching Engine during the matching process. Based on this information stored in the Repository, the following tasks are accomplished:

- *Mapping* – this is the result of the XML Schema matching process and it shows correspondences between source XML Schema components and target XML Schema components. The Repository must provide capabilities for saving and retrieving a mapping for a later use, for the reason that a mapping between two XML Schema documents is not always unique. Only a user with expert knowledge can decide which mapping is the most suitable for the case of two XML Schema documents.
- *Generate transformation scripts* – the Repository must have the capability of providing information in such way that it can be used to generate complete and

correct transformation scripts between data instances validated by the two XML Schema documents subject of the matching process.

The Repository must also work in a distributed environment and provide its functionality to more than one group of Matching Engine, Mapping and Transformation Engine systems. This feature is very important to solve scalability issues that occur in the case of very large and complex XML Schema documents.

3.3.4 Thesaurus

One of the most important parts of the XML Schema matching process is the named schema components match. Named schema components are:

- *Element declarations*
- *Attribute declarations*
- *Attribute Group Definition*
- *Simple Type Definition*
- *Complex Type Definition*
- *Identity Constraint Definition*
- *Model Group Definition*
- *Notation Declaration*

Element declarations and attribute declarations are XML Schema components that define all XML elements with their attributes that may appear in an XML document. As stated in the "Problem Definition" chapter of this thesis, schema matching is the process of finding a consistency between elements of the two schemas that correspond to each other with respect to their meaning. Therefore, names of the above mentioned schema components are crucial in finding their meaning. For instance, two XML schema documents can have two element definitions with the name of "Customer". The easiest name matching approach is a string comparison of the two names. However, XML

schemas are developed by different experts in different contexts. One expert may name an element "Customer" and another expert may name an element "Client". In this case, a name matcher based only on string comparison will fail, even though the two experts defined the same data. A name matcher must be more elaborate in order to find most correct and complete matches between names of primary schema components. This elaborate name matcher can only be implemented by using a thesaurus (a complete dictionary of terms). In this case, a name matcher functionality can be described as follows: for an XML schema component name from a given source XML schema (e.g., "Customer"), the name matcher builds a list of synonyms for this name; for a corresponding XML schema component name (e.g., "Client") from the target XML schema, the name matcher builds a list of synonyms based on the same thesaurus. Then the two lists are compared using string comparison and if at least one element form the first list matches an element from the second list, then a name match between the two schema elements is reported.

The above described functionality is still simplistic and may not work (i.e. it does not report a match when that match between those names actually exists). In cases where XML schema component names are "LastName" and "Surname" there is a need for an intermediate step, before building the synonym lists. This step is string tokenization, using simple regular expressions. Usually, a properly and professionally developed XML schema uses capitalized words for XML schema component names, so the string tokenization in this case is very simple. However, in case where capitalized words were not used in XML schema component names, the string tokenization is more difficult (but not impossible) using a technique called q-gram (described in "Literature review" chapter of this thesis), This technique consists in the breaking of respective XML schema component name in groups of words having two, three or four letters (q = 2, 3, 4). For instance, the XML schema component name "lastname" is broken into "last" and "name" using 4-gram technique. The thesaurus must provide synonyms for composed terms like "last name", i.e. it must not concatenate synonym list of "last" and synonym list of "name", and it should yield a list of synonyms that contains "surname".

3.3.5 Mapping

The result of the XML schema matching process is a *mapping*. A mapping represents correspondences between elements of two XML Schema documents; therefore it needs to be linked to the application's user interface. An expert user must be able to validate and refine the resulting mapping; hence a mapping should comply with the following rules:

- Must have save and restore (or import/export) features;
- Must have a human readable representation.

All the information discovered during the matching process is stored in the Repository. This information is represented by schema components that match and schema components relationships (parent, child and sibling relationships).

3.3.6 Transformation Engine

Two XSLT scripts are generated based on the results of semi-automatic XML Schema matching and the mapping refinement performed by user. The first XSLT script is used to transform the source data instance (a data instance validated against the source XML schema) into a data format that validated against the target XML schema. The second XSLT script is used to transform a target data instance (a data instance validated against the target XML schema) into a data format validated against the source XML schema.

The high level architecture diagram of the Transformation Engine is presented in Figure 3.8:

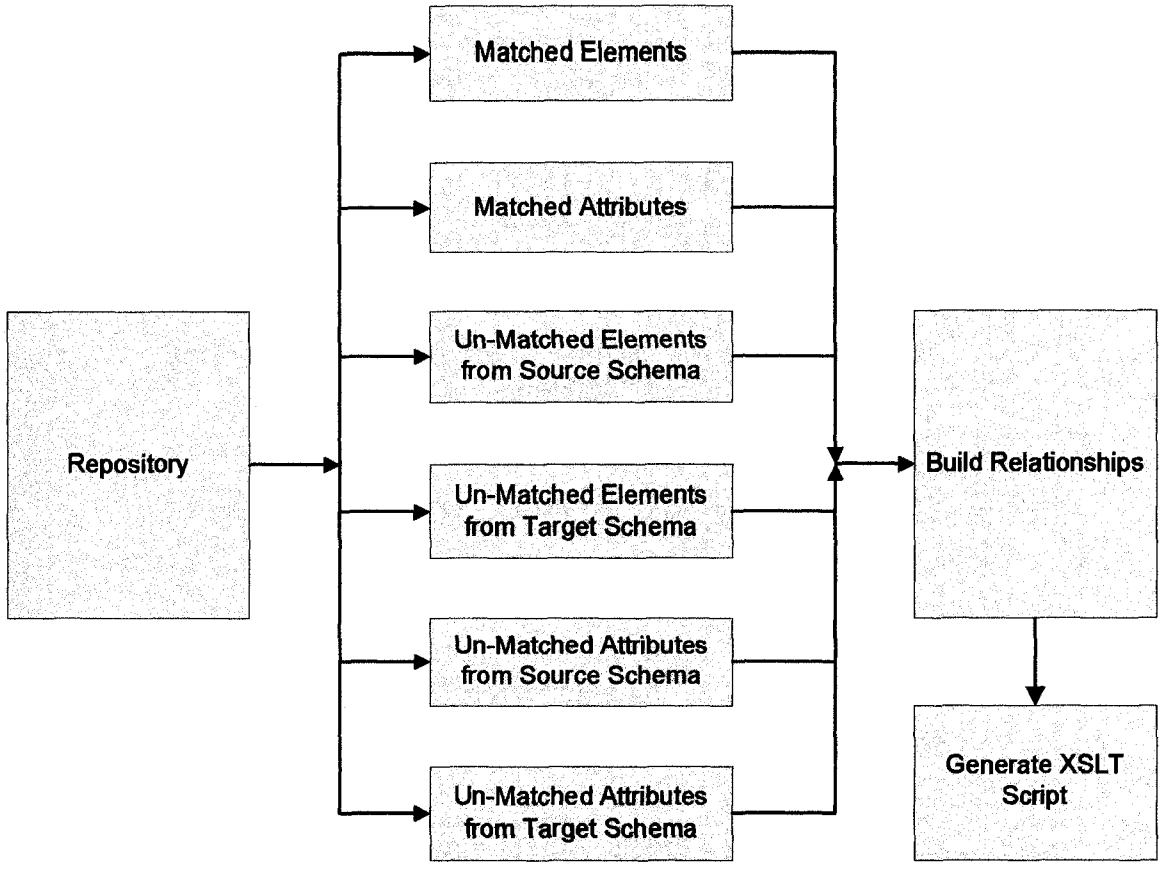


Figure 3.8 Transformation Engine Architecture

The Transformation Engine uses the Repository as follows:

- *Matched Elements*: extracts all the matched ***elements*** of the two schemas in a data format that will ease the subsequent phases of the XSLT generation process. This data structure contains all *element declarations* from one schema that have a correspondence in the other XML Schema.
- *Matched Attributes*: extracts all matched ***attributes*** of the two schemas in a data format that will ease the subsequent phases of the XSLT generation process. This data structure contains all *attribute declarations* from one schema that have a correspondence in the other XML Schema.
- *Un-Matched Elements from Source Schema*: retrieves all un-matched element declarations from source XML schema
- *Un-Matched Elements from Target Schema*: retrieves all un-matched element declarations from target XML schema

- *Un-Matched Attributes from Source Schema*: retrieves all un-matched attribute declarations from source XML Schema
- *Un-Matched Attributes from Target Schema*: retrieves all un-matched attribute declarations from target XML Schema
- *Build Relationships* constructs a data structure based on relationships stored in the Repository between XML Schema components
- *Generate XSLT Scripts*: generates the XSLT scripts based on information provided by *Build Relationships* block. The first XSLT script transforms one data instance validated against one XML Schema into a data format validated against the second XML schema. The second XSLT script performs the reciprocal transformation, i.e. it transforms a data instance validated against the second XML schema into a data format validated against the first XML schema.

3.3.7 Graphical User Interface

The Graphical User Interface (GUI) component handles the user interaction with the Hermes software application. The GUI should be able to present XML Schema documents to the user in a clear and scalable manner, i.e. to be able to handle extremely large XML Schema documents with thousands of components (element declarations, attribute declarations, type definitions, etc.).

The GUI must also provide information about the current state of a running process like the XML schema matching process and XSLT scripts generation process, and communicate about some issues that may occur during those processes.

The overall architecture of Graphical User Interface is described by the following diagram in Figure 3.9:

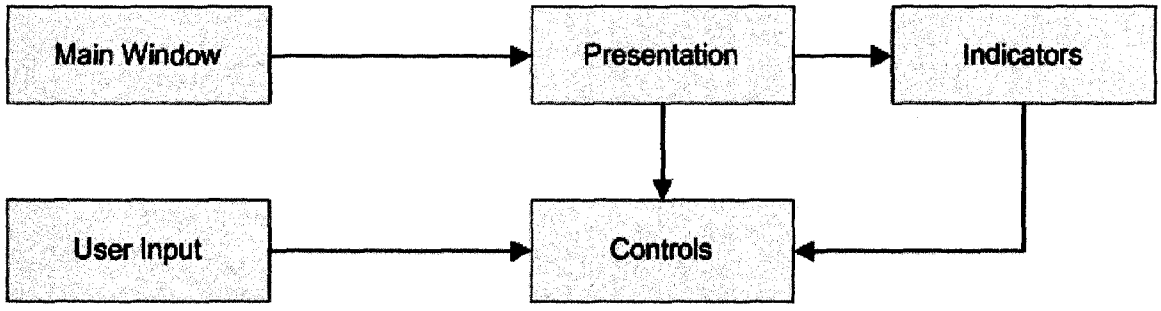


Figure 3.9 Graphical User Interface Architecture

Graphical User Interface is based on the following blocks:

- *Main Window:* represents the top level component of GUI
- *User Input*: handles all user interactions
- *Presentation*: organizes all the controls and indicators of the current state of the process in progress;
- *Indicators*: provides information to a standard control regarding the currant state of the process in progress (either XML schema matching or generation of transformation scripts;
- *Controls*: contains all standard controls for I/O representation such as menus, trees, tables, etc.

The user interaction with the GUI is presented in the following Finite State Machine diagram in Figure 3.10:

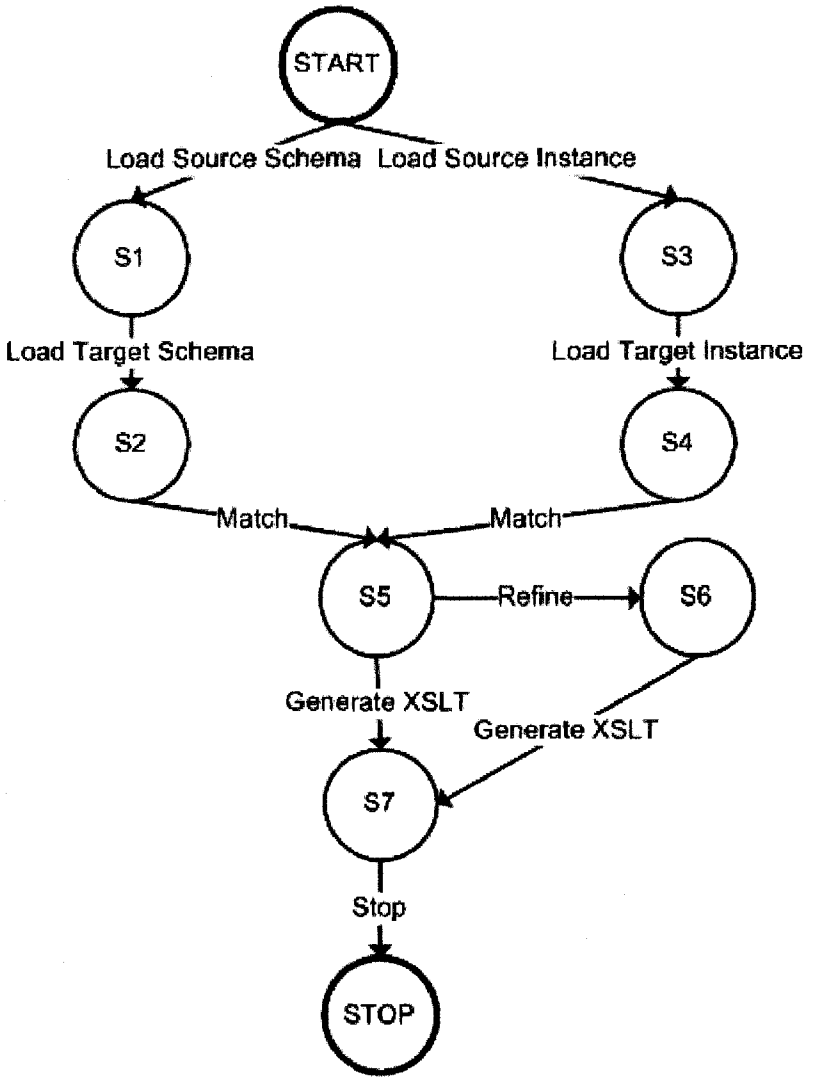


Figure 3.10 Finite State Machine for user interaction with Hermes GUI

A user can load either two XML Schema documents or two XML documents (data instances). In the latter case, two minimal XML schemas are constructed using the Schema Extraction Engine in state S4. The first minimal XML schema validates the source data instance and the second minimal XML schema validates the target data instance. These two XML schemas are presented to the user.

The two XML schemas (loaded or extracted from the data instances) are then matched in the state S5. At the end of the matching process, the user can refine the resulting mapping (state S6) and proceed to the XSLT scripts generation.

4 Hermes – Detailed design

This chapter presents the detailed design of Hermes and is structured as follows:

- A presentation of XML Schema Infoset Model [41, 42, 43 and 50], the corner stone of the Matching Engine
- The Matching Engine detailed design proposed by this thesis: its detailed architecture, functionality and novel matching algorithms for XML Schema documents.
- The Hermes GUI based on Standard Widget Toolkit [53]
- The Schema Extraction Engine design
- The design of the underlying repository system of Hermes. This repository is based on Sesame architecture [45, 46] and is used by the Matching Engine, Mapping and Transformation Engine. Design of SeRQL [47] scripts used to extract meaningful data from the repository is also presented.
- The Transformation Engine, Hermes component that generates XSLT [54, 55, 57] scripts to transform data instances validated by the matched XML Schema documents

4.1 Introduction

The majority of XML Schema matching and mapping tools presented in the "Literature Review" chapter of this thesis use a directed or labeled graph approach for internal XML Schema representation. This approach leads to scalability issues in the case of very complex XML schemas. Applications that use this approach do not perform very well. In addition, matching algorithms for directed or labeled graphs are very complex and difficult to implement. Therefore, another approach must be used for solving the XML Schema mapping problem. The new approach presented in this thesis consists of separating the XML Schema paradigm from its XML representation.

4.2 XML Schema Infoset Model

The XML Schema Infoset Model [41, 42, 43 and 50] is an API which is used by applications which handle XML Schemas. This API can also be used to access and update WSDL documents. XML Schema Infoset Model API can also be used for XML Schema documents integrity checking and validation.

The UML diagram in Figure 4.1 presents the XML Schema Infoset Model hierarchy. This diagram conforms to the XML Schema standard [35].

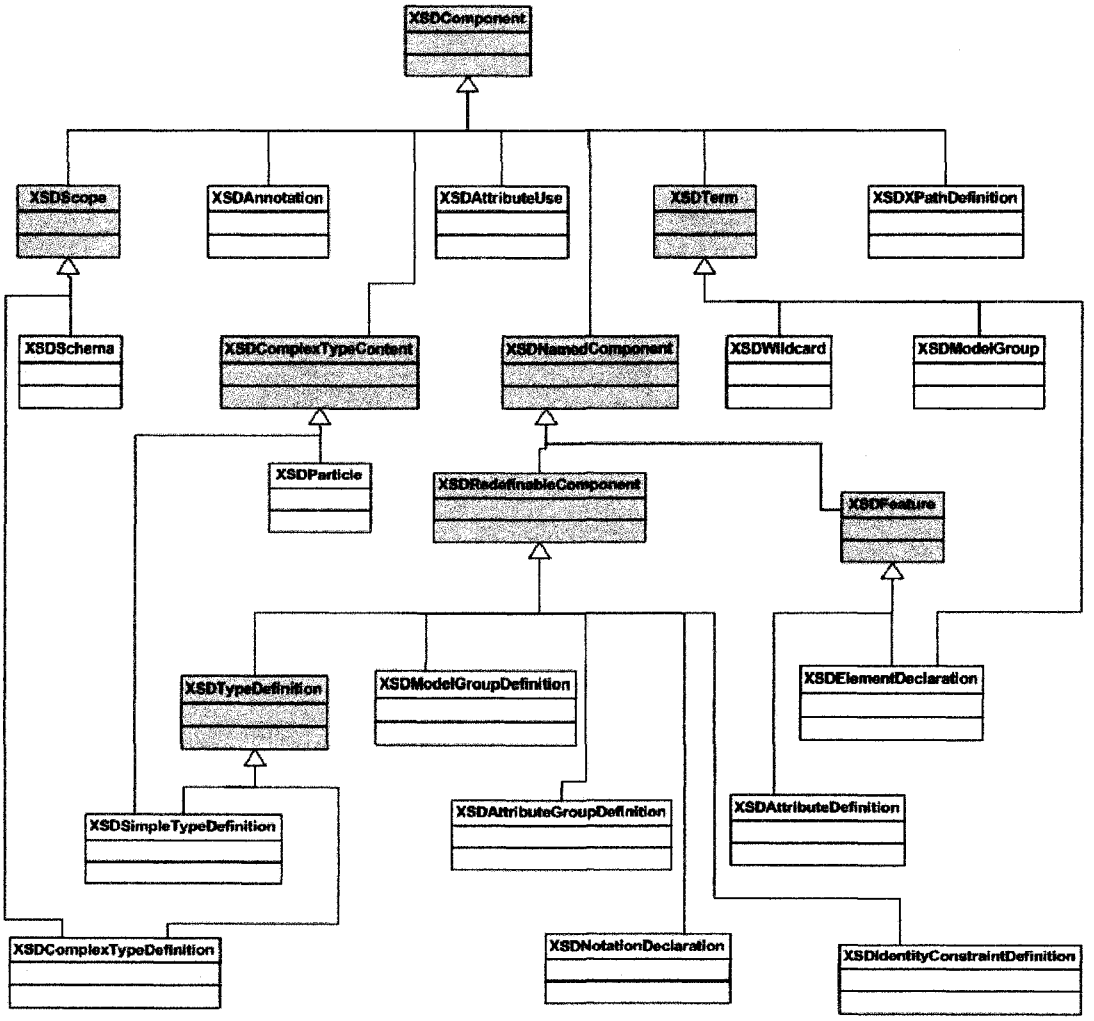


Figure 4.1 Schema Infoset Model Class Hierarchy [41]

- **XSDComponent** – an abstract class which is the root of all classes defined in the XML Schema Infoset Model (all abstract classes have been shown in blue color);
- **XSDSchema** – is a concrete class which represents the XML Schema root element (e.g., <xsd:schema>);
- **XSDAnnotation** – represents XML Schema annotations (<xsd:annotation>);
- **XSDSimpleTypeDefinition** – represents XML Schema simple type definitions (<xsd:simpleType>);
- **XSDComplexTypeDefinition** – a concrete class representing XML Schema complex type definitions (<xsd:complexType>)
- **XSDElementDeclaration** – represents XML Schema element declarations local or global or as references inside other XML Schema components constructs (<xsd:element>);
- **XSDAttributeDeclaration** – a concrete class which represents XML Schema attribute declarations in a local or global scope or as a reference in other XML Schema components (<xsd:attribute);
- **XSDAttributeGroupDefinition** – represents an attribute group definition XML Schema component (<xsd:attributeGroups>);
- **XSDModelGroupDefinition** – represents either a local or a reference XML Schema component group (<xsd:group>);
- **XSDModelGroup** – a concrete class which represents a local group (<xsd:sequence> or <xsd:choice> or <xsd:all>);
- **XSDWildcard** – represents a wild card element or an attribute such as <xsd:any> or <xsd:anyattribute>.

The following UML diagram presents the relationships and hierarchy of XML Schema data types [36] defined in the XSD Schema Infoset Model [41]. This diagram conforms to the XML Schema data types standard [36]:

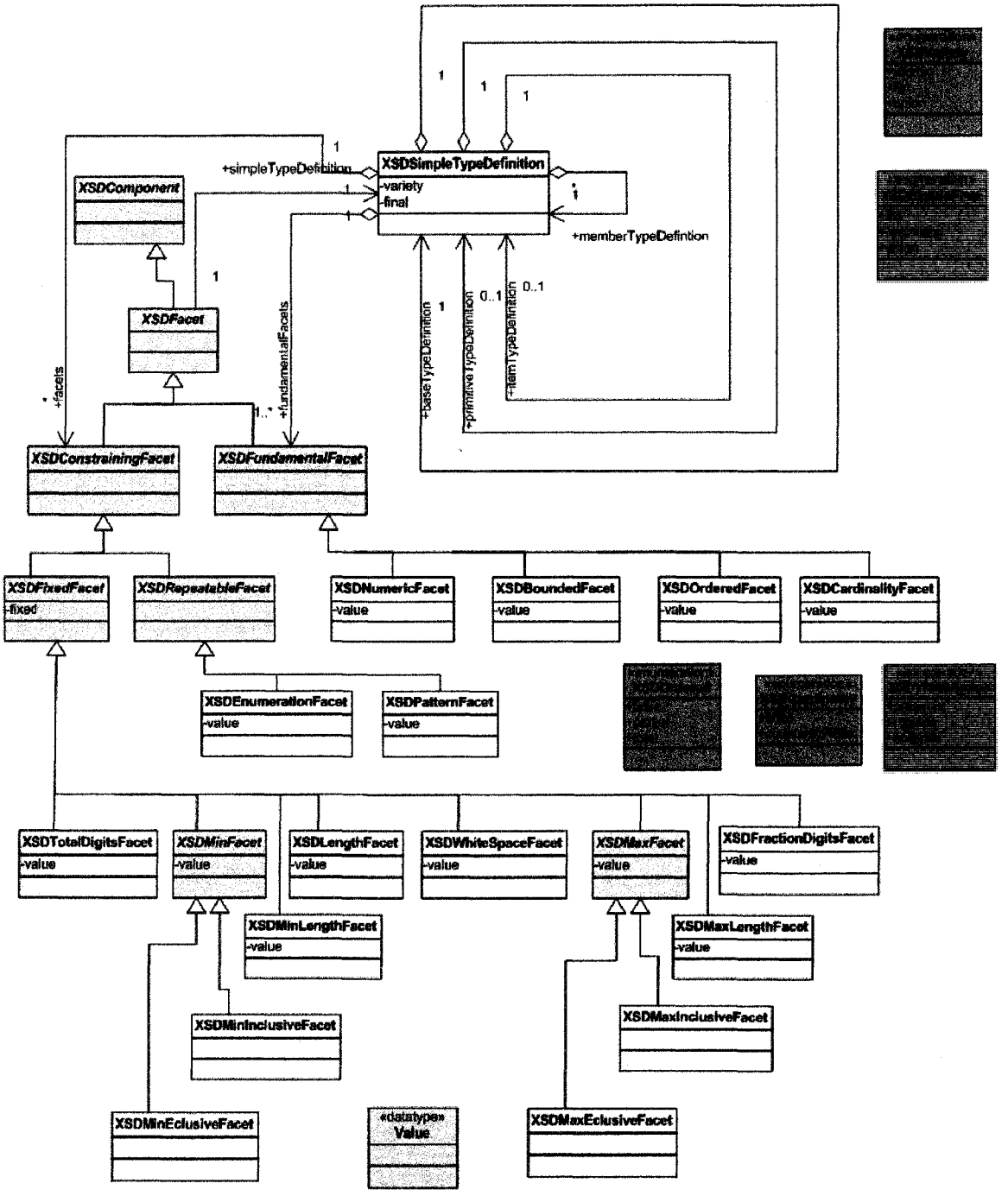


Figure 4.2 XML Schema Infoset Model – Data types hierarchy and relationships [41]

- **XSDFacet :** abstract class – the base class for all XML Schema facets;

- **XSDConstrainingFacet :** this abstract class is the base class for all XML Schema facets that define constraints for the models such as: length, inclusive, exclusive, total digits, fraction digits, enumeration, pattern and white spaces facets;
- **XSDFixedFacet:** this abstract class is the base class for all XML Schema facets that describe constraints which cannot be changed inside specialized types:
    - **XSDLengthFacet:** represents <xsd:length> XML Schema facet;
    - **XSDMaxLengthFacet**: represents XML Schema <xsd:maxLength> facet;
    - **XSDMinInclusiveFacet**: represents XML Schema <xsd:minInclusive> facet;
    - **XSDMaxInclusiveFacet**: represents XML Schema <xsd:maxInclusive> facet;
    - **XSDMinExclusiveFacet**: represents XML Schema <xsd:minExclusive> facet;
    - **XSDMaxExclusiveFacet**: represents XML Schema <xsd:maxExclusive> facet;
    - **XSDWhiteSpaceFacet**: represents XML Schema <xsd:whiteSpace> facet;
    - **XSDFractionDigitsFacet**: represents XML Schema <xsd:fractionDigits> facet;
    - **XSDTotalDigits**: represents XML Schema <xsd:totalDigits> facet;
- **XSDRepeatableFacet:** represents XML Schema facets that define repeatable constraints:
    - **XSDEnumeration**: represents XML Schema <xsd:enumeration> facet
    - **XSDPatternFacet**: represents XML Schema <xsd:pattern> facet

The XML Schema Infoset Model containment relationships [41] are described by the following UML diagram:

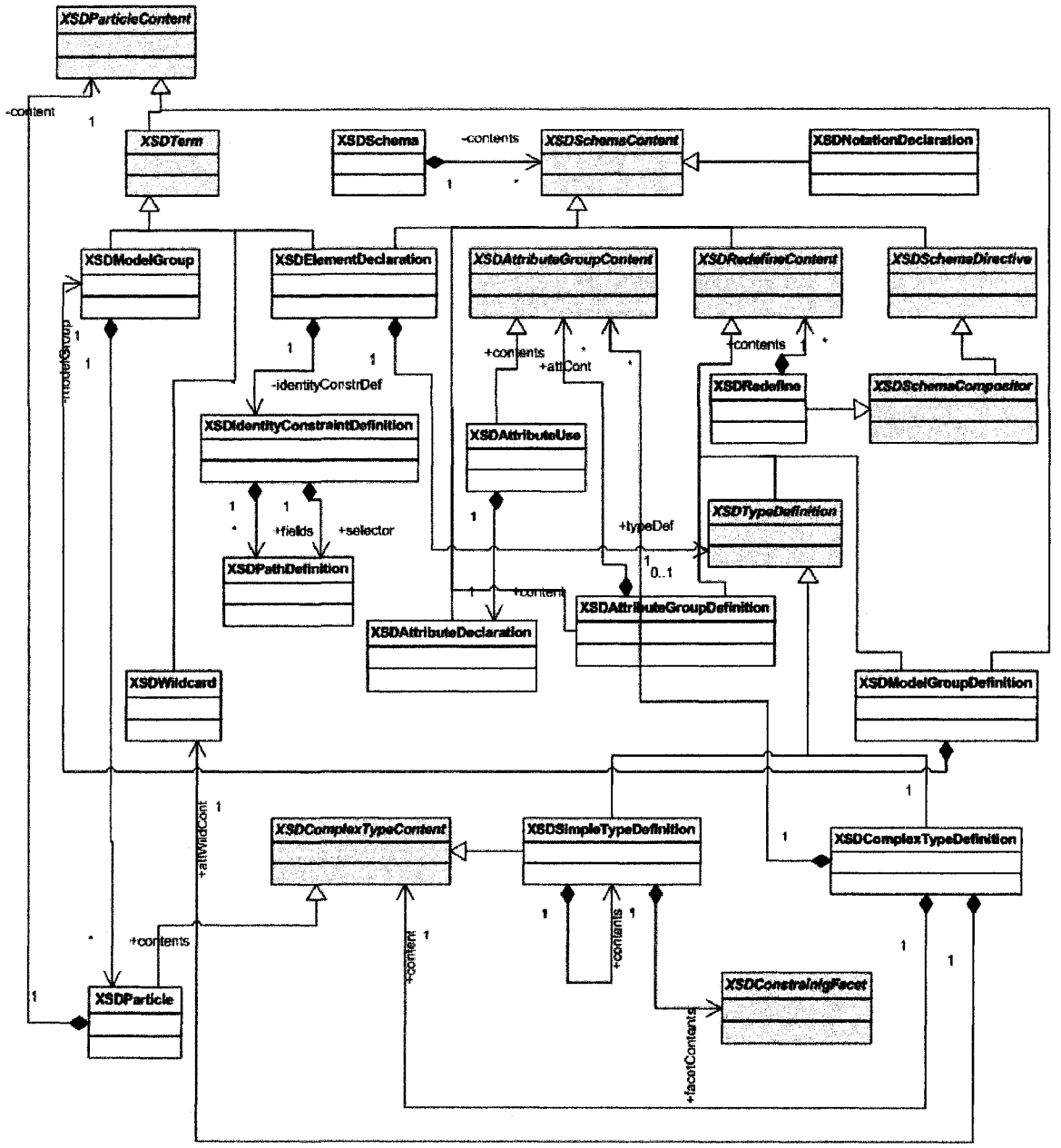


Figure 4.3 XML Schema Infoset Model containment relationships [41]

The **XSDSchema** class contains **XSDSchemaContent** objects which describe XML elements under <xsd:schema> root element. The **XSDModelGroupDefinition** class describes an XML Schema model group definition [36], where a model group definition represents an association between a name and a model group. An **XSDModelGroupDefinition** class contains only one **XSDModelGroup** object. This object can contain zero or more **XSDParticle** objects.

The particle XML Schema component [36] has the following properties:

- min occurs: a non-negative integer;
- max occurs: a non-negative integer or the word "*unbounded*";
- term: one of model group XML Schema component, a wildcard XML Schema component or an element declaration.

Therefore, an **XSDParticle** object contains only one **XSDParticleContent** object which can be one of the following objects: **XSDWildcard**, **XSDModelGroup**, **XSDModelGroupDefinition**, **XSDElementDeclaration**.

The **XSDComplexTypeDefiniton** class represents a complex type definition in an XML Schema document (<xsd:complexType>) and it can contain a single **XSDWildcard** object, zero or more **XSDAttributeGroupContent** objects and one **XSDComplexTypeContent** object. The abstract class **XSDComplexTypeContent** can be instantiated as an XSDPaticle object or an XSDSimpleTypeDefinition, therefore an **XSDComplexTypeDefinition** can contain one instance of either **XSDParticle** or **XSDSimpleTypeDefinition**.

XML Schema components modeled in the XML Schema Infoset Model have the following relationships:

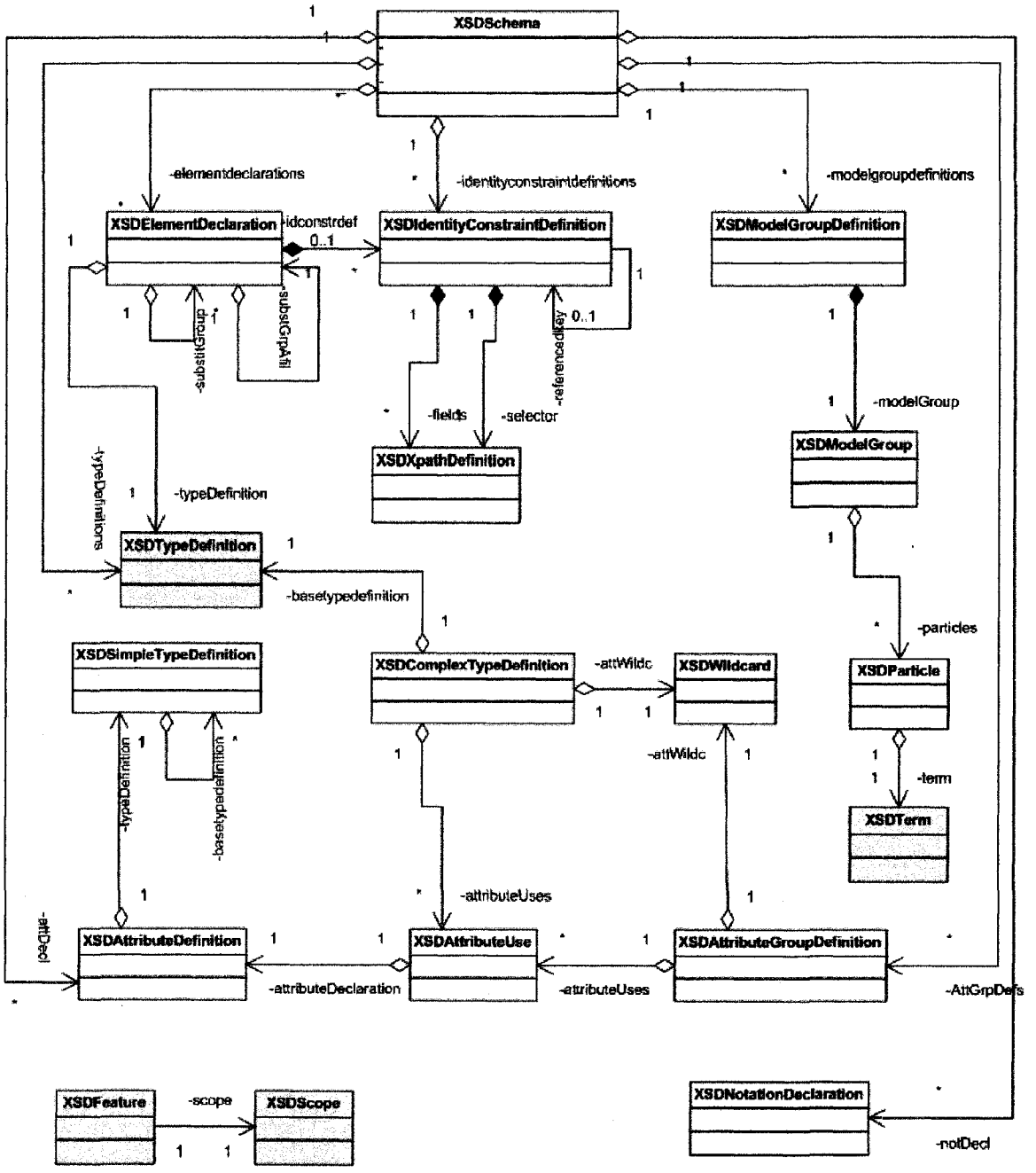


Figure 4.4 XML Schema component relationships in the XML Schema Infoset Model [41]

XML Schema Infoset Model class relationships, as shown in the above UML diagram, are very similar to the XML Schema components relationships presented in the XML Schema standard [35]:

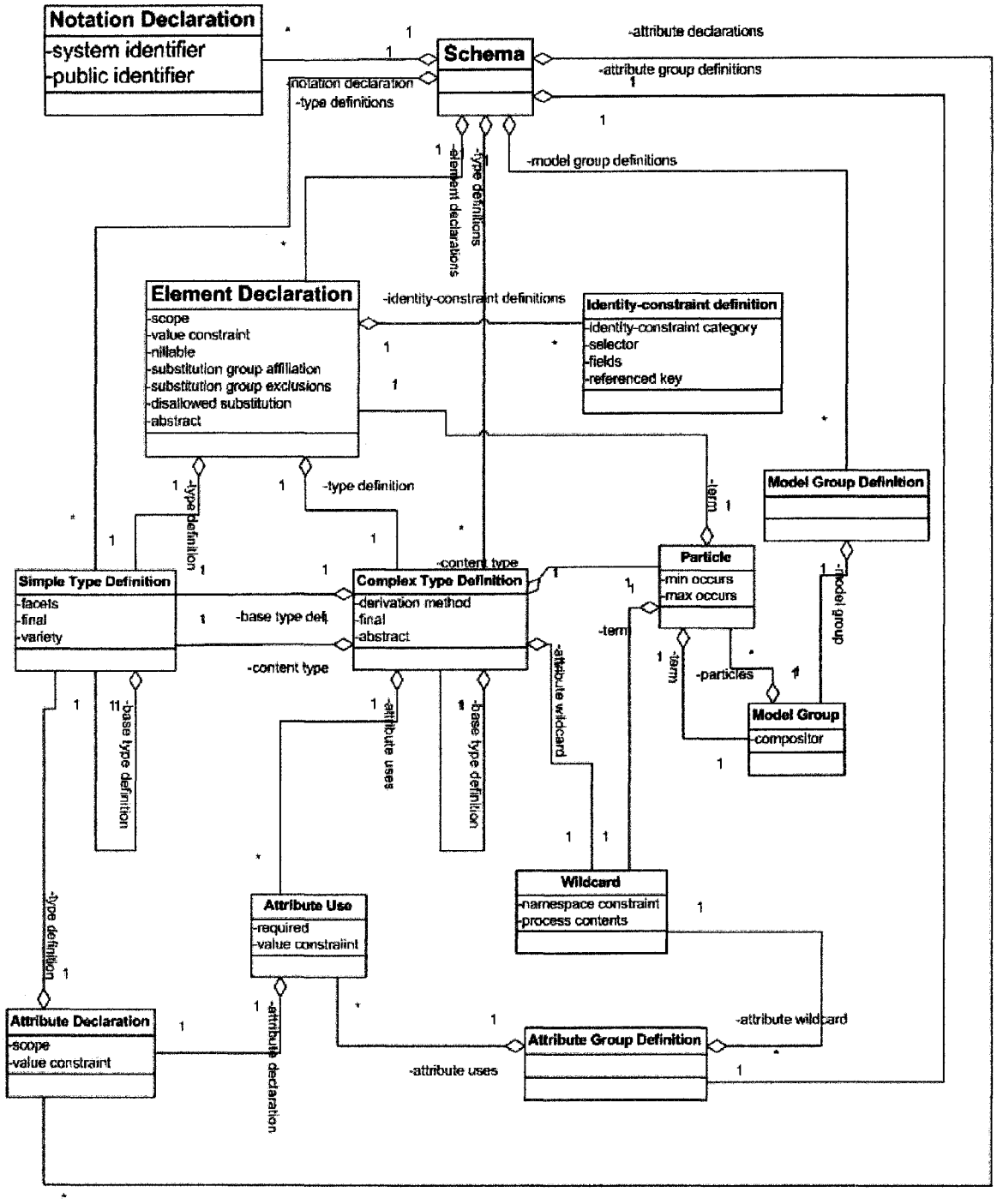


Figure 4.5 XML Schema Components: UML Relationship Diagram [35]

4.3 Matching Engine – Detailed Design

The Matching Engine has the following important features:

- Correctness:
    - Conforms to the XML Schema standard [34, 35 and 36];
    - Properly handles XML Schema components such as element declarations, attribute declarations and type definitions.
- Scalability: handles very large XML Schema documents with thousands of components.

As presented in previous sections, two XML Schema documents subject of a matching process contain element declarations, attribute declarations, type definitions, etc, and each of which are to be matched respectively. When two global component definitions are in the process of matching, other pairs of global component definitions can be matched simultaneously, therefore the multithreading paradigm can be used in the Matching Engine. When a match between two XML Schema documents is in progress, the graphical user interface should be responsive all the time, therefore the GUI and the Matching Engine should run in different threads.

The following UML Class Diagram shows the Matching Engine class hierarchy, as introduced in this thesis:

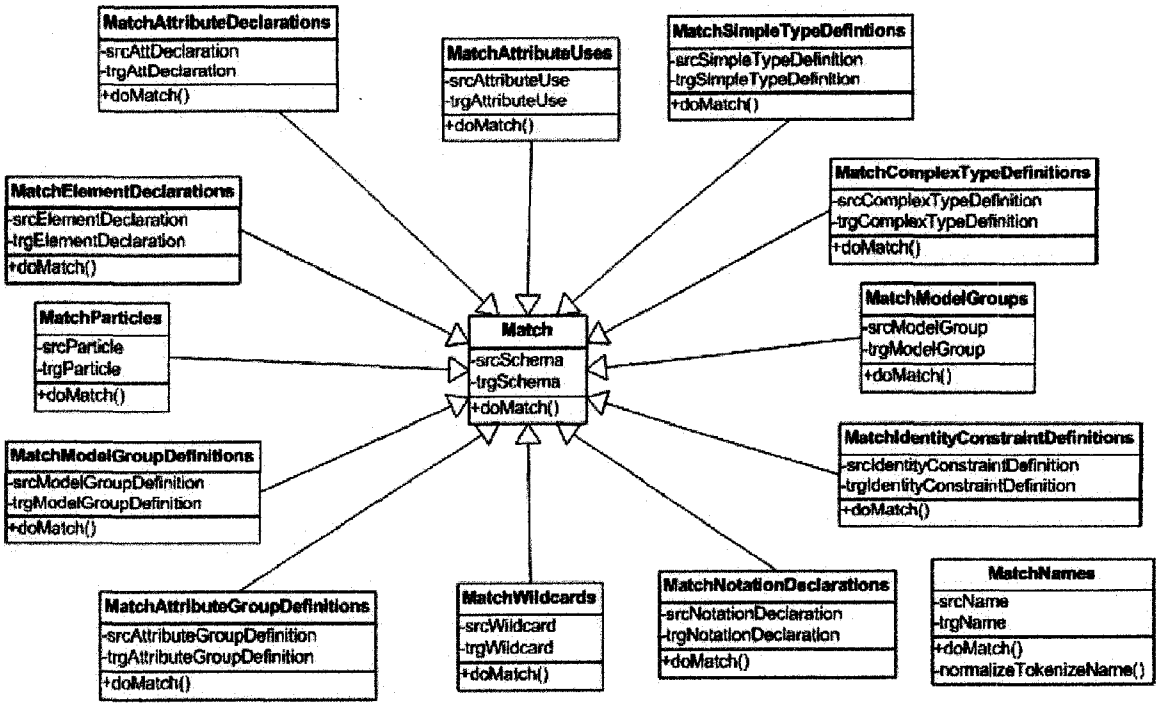


Figure 4.6 Matching Engine UML Class Diagram – Hierarchy

The base class of the Matching Engine is the **Match** class. It has two private attributes:

- *srcSchema* – an XSDSchema object instance which holds the whole source XML Schema document;
- *trgSchema* – an XSDSchema object instance which holds the whole target XML Schema document

An important note is that, once an XML Schema document is loaded and an **XSDSchema** object is instantiated, all other included or imported XML schema components from other XML Schema documents become available in the same **XSDSchema** object, i.e. these schema components are properly resolved.

The Match class has a public method – *doMatch* – which implements the new XML Schema matching algorithm. This new matching algorithm is presented in the following figure:

```
1:  Get srcGEDs from Source XML Schema
2:  Get trgGEDs from Target XML Schema
3:  For each srcED in srcGEDs:
4:          For each trgED in trgGEDs:
5:                  MatchElementDeclarations(srcED, trgED)
6:  Get srcGADs from Source XML Schema
7:  Get trgGADs from Target XML Schema
8:  For each srcAD in srcGADs:
9:          For each trgAD in trgGADs:
10:                 MatchAttributeDeclarations(srcAD, trgAD)
11: Get srcGCTDs from Source XML Schema
12: Get trgGCTDs from Target XML Schema
13: For each srcCTD in srcGCTDs:
14:         For each trgCTD in trgGCTDs:
15:                 MatchComplexTypeDefinitions(srcCTD,trgCTD)
16: Get srcGSTDs from Source XML Schema
17: Gets trgGSTDs from Target XML Schema
18: For each srcSTD in srcGSTDs:
19:         For each trgSTD in trgGSTDs:
20:                 MatchSimpleTypeDefinitions(srcSTD,trgSTD)
21: Get srcGNDs from Source XML Schema
22: Get trgGNDs from Target XML Schema
23: For each srcND:
24:         For each trgND:
25:                 MatchNotationDeclarations(srcND,trgND)
26: Get srcGAGDs from Source XML Schema
27: Get trgGAGDs from Target XML Schema
28: For each srcAGD in srcGADs:
29:         For each trgAGD in trgAGDs:
30:                 MatchAttributeGroupDefinitions(srcAGD,trgAGD)
31: Get srcGMGDs from Source XML Schema
32: Get trgGMGDs from Target XML Schema
33: For each srcMGD in srcGMGDs:
34:         For each trgMGD in trgGMGDs:
35:                 MatchModelGroupDefinitions(srcMGD,trgMGD)
36: Get srcGICDs from Source XML Schema
37: Get trgGICDs from Target XML Schema
38: For each srcICD in srcGICDs:
39:         For each trgICD in trgGICDs:
40:                 MatchIdentityConstraintDefinitions(srcICD, trgICD)
```

Figure 4.7 XML Schema matching algorithm

The algorithm structure follows the XML Schema standard structure as per [35] (see section "3.15.1 The Schema Itself").

All element declarations from both XML schema documents are extracted, as shown in lines 1 and 2 of the algorithm in the above figure (srcGEDs and trgGEDs). Then each element declaration from the list of element declarations harvested from the source XML Schema document is matched with each element declaration from the list of element declarations harvested from the target XML Schema document – lines 3, 4 and 5 in the above figure. The matching element declarations algorithm is presented later in this section as a part of **MatchElementDeclarations** class description.

The process of matching global attribute declarations is illustrated from line 6 to line 10 of the algorithm. First, all global attribute declarations are extracted from each XML Schema documents. Then each attribute declaration from the source XML schema is matched with each attribute declaration from the target XML schema. Attribute declarations matching algorithm is presented as a part of **MatchAttributeDeclarations** class description.

Each XML Schema document may contain global type definitions – simple or complex. Complex type definitions are extracted from each XML Schema document (lines 11 and 12) and then each complex type definition from the source XML Schema is matched with each complex type definition from target XML Schema. Their matching algorithm is presented in the **MatcheComplexTypeDefinitions** class description. Simple type definitions are extracted from each XML Schema document (lines 16 and 17) and then each simple type definition from source XML Schema is matched with each simple type definition from target XML Schema; their matching algorithm is presented in the MatchSimpleTypeDefinitions class description.

The notation declarations match is performed from line 21 to line 25 - all notation declarations are gathered from both XML schema documents and then matched one by

one. Their matching algorithm is presented in the **MatchNotationDeclarations** class description.

An XML Schema document may contain attribute group definitions, model group definitions and identity constraint definitions, XML Schema structures that need to be matched as well. Lines 26 to 30, 31 to 35 and 36 to 40 respectively, describe this fact. Algorithms for matching attribute group definitions, model group definitions and identity constraint definitions are presented in the **MatchAttributeGroupDefinitions, MatchModelGroupDefinitions, MatchIdentityConstraintDefinitions** class descriptions, respectively.

The **MatchElementDeclarations** class is a specialization of the main base class (**Match**) of the Matching Engine and it takes care of matching all element declarations in both XML Schema documents (source and target). This class has two private members as follows:

- *srcElementDeclaration* – an **XSDElementDeclaration** object instance which holds an element declaration that comes from the source XML schema
- *trgElementDeclaration* – an XSDElementDeclaration object instance which holds an element declaration that comes from the target XML schema

The implementation of **XSDElementDeclaration** [41, 42, 43 and 50] provides the capability of resolving referenced element declaration. If an element declaration from the source XML schema is a reference and an element declaration from the target XML Schema is not, the referenced element declaration from the first XML Schema must be resolved, so the match may be performed properly between concrete element declarations.

The public method *doMatch()* from the **MatchElementDeclarations** class implements this new XML Schema element declarations match algorithm, as illustrated in the following figure:

```
1:  Resolve(srcED)
2:  Resolve(trgED)
3:  Persist(srcED)
4:  Persist(trgED)
5:  srcTD = getTypeDefintion(srcED)
6:  trgTD = getTypeDefinition(trgED)
7:  if (srcTD is SimpleType) and (trgTD is SimpleType) then
8:          stdMatched = MatchSimpleTypeDefinitions(srcTD,trgTD)
9:  If (srcTD is ComplexType) and (trgTD is ComplexType) then
10:         ctdMatched = MatchComplexTypeDefinitions(srcTD, trgTD)
11: resultNames = MatchNames(srcEDname, trgEDname)
12: if (resultNames and stdMatched) then
13:         persist_as_matched(srcED,trgED)
14:         return true
15: if (resultNames and ctdMatched) then
16:         persist_as_matched(srcED, trgED)
17:         return true
18: return false
```

Figure 4.8 XML Schema element declaration matching algorithm

An element declaration statement in an XML Schema document can describe a global, a local or a reference element declaration. A global element declaration occurs when the actual element declaration is the child of the root element of the XML Schema document (<schema>). An element declaration which does not have XML Schema document root element as its parent constitutes a local element declaration. An element declaration reference can be recognized in an XML Schema document when the <element> tag has an attribute called "*ref*" instead of "*name*". This element declaration reference can be part of a <group> or a <complexType> and refers to a global element declaration. Every element declaration reference must be resolved in order to get the right element declaration and perform a correct match between concrete element declarations. This is what lines 1 and 2 are about in the element declarations matching algorithm.

Lines 3 and 4 of the algorithm perform the actual save into the Repository of each element declaration to be matched.

The next phase of the algorithm is to determine the type of both element declarations – srcED and trgED – as shown in lines 5 and 6. An element declaration can have either a simple type definition or a complex type definition. If both element declarations do not have same type definition then it is clear that these element declarations do not match at all. If both element declarations have the same type of definition (either simple or complex), lines 7 and 9, then the process continues with the matching of those type definitions – lines 8 and 10. The simple and complex type definitions matching algorithms are presented in the **MatchSimpleTypeDefinitions** and **MatchComplexTypeDefinitions** class descriptions, respectively.

An element declaration always has a name; therefore the next step in the algorithm (line 11) is the name matching phase. The element declarations names are represented by srcEDname and trgEDname. The names matching algorithm is presented in the **MatchNames** class description. If both element declarations have names with the same meaning and their type definitions match – lines 12 and 15 – then both element declarations are saved into the application repository with the additional information regarding their match – lines 13 and 16.

The following figure illustrates the UML sequence diagram of the matching element declarations algorithm:

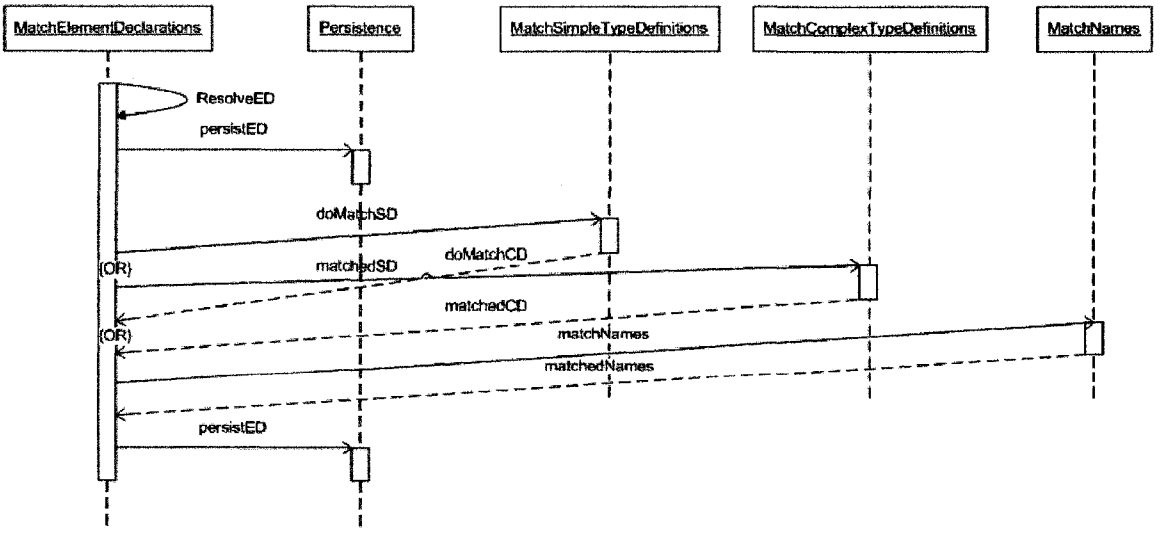


Figure 4.9 Element declaration matching algorithm UML sequence diagram

Both element declarations are first resolved (*ResolveED* message) in order to perform a match between the concrete element declarations. Then the Persistence object is activated and the concrete element declarations are saved into the repository. If both element declarations contain either simple type definitions or complex type definitions then the corresponding matching object is activated (*doMatchSD* or *doMatchCD* message). After the reply from type definitions match, the **MatchNames** object is activated through the *matchNames* message. Finally, in the case of a match of the two element declarations, these element declarations are saved into the repository as a match – persisted message.

The **MatchAttributeDeclarations** class is a specialization of the main base class **Match** and works with attribute declarations (local or global) present in both XML Schema documents. This class has the following private members:

- *srcAttributeDeclaration* – an **XSDAttributeDeclaration** object instance which contains an XML Schema attribute declaration representation from the source XML schema
- *trgAttributeDeclaration* – an **XSDAttributeDeclaration** object instance which contains an XML Schema attribute declaration representation from the target XML schema

The implementation of **XSDAttributeDeclaration** [50] provides the capability for resolving referenced XML Schema attribute declaration components as the implementation of **XSDElementDeclaration**.

The public method *doMatch()* from the **MatchAttributeDeclarations** class implements the new XML Schema attribute declarations match algorithm and this algorithm is illustrated in the following figure:

```
1:  Resolve(srcAD)
2:  Resolve(trgAD)
3:  Persist(srcAD)
4:  Persist(trgAD)
5:  srcASTD = getSimpleTypeDefintion(srcAD)
6:  trgASTD = getSimpleTypeDefinition(trgAD)
7:  stdMatched = MatchSimpleTypeDefinitions(srcASTD,trgASTD)
8:  resultNames = MatchNames(srcADname, trgADname)
9:  if (resultNames and stdMatched) then
10:         persist_as_matched(srcAD,trgAD)
11:         return true
12: return false
```

Figure 4.10 XML Schema attribute declaration matching algorithm

In the first phase, attribute declarations must be resolved in the same manner as element declarations are resolved in the element declarations matching algorithm. As per [35], when an attribute declaration occurs at the top level of the XML schema document, it represents a global attribute declaration. In addition, an attribute declaration can appear as a part of a complex type definition or as a part of an attribute group declaration. In these cases, that attribute declaration can be a local definition or a reference to a global attribute declaration. In the case of a reference to a global attribute declaration, that reference must be resolved before the actual attribute declaration matching process can proceed – lines 1 and 2 of the algorithm.

In the same manner as in element declarations matching algorithm, attribute declarations subject to the matching process are saved in the system repository – lines 3 and 4 of the algorithm. The system repository is described in the "Repository and Mapping" section.

An attribute declaration always has a simple type definition [35] and lines 5 and 6 of the algorithm show this fact. Therefore, the next line calls the simple type definition matching algorithm. Attribute declarations are named components defined by the XML Schema standard, so their names must be matched semantically, as shown in line 8 of the attribute declarations matching algorithm. In the case of a match between simple type definitions of two attribute declarations and a semantic match between their names, line 9

of the algorithm, these attribute declarations are reported as a match and stored in the system repository consequently – line 10 of the algorithm. If there is no match between attribute declarations simple type definitions and their names, then these attribute declarations do not match – line 12 of the algorithm.

The following figure illustrates the UML sequence diagram of matching attribute declarations algorithm:

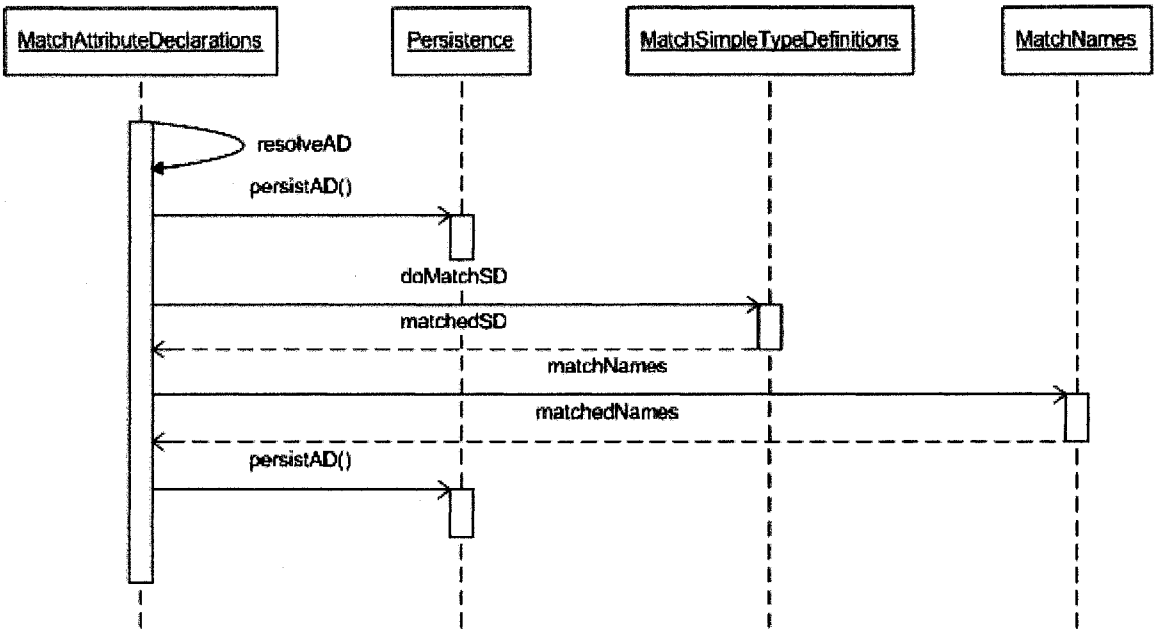


Figure 4.11 Attribute declaration matching algorithm UML sequence diagram

The two attribute declarations are first resolved in order to perform a match between concrete attribute declarations components – *resolveSD* message. Next, the two concrete attribute declarations are saved into the repository by the activating of the Persistence object through the *persistAD* message. The **MatchSimpleTypeDefinitions** object is activated by *doMatchSD* message and the simple type definitions match is performed. After the reply from the matching simple type definition method, the **MatchNames** object is activated by the *matchNames* message. If the two attribute declarations match, then the **Persistence** object is activated by the *persistAD* message and these attribute declarations are persisted into the repository as a match.

The class **MatchSimpleTypeDefinitions** is a subclass of the **Match** class and it has the following two private members:

- *srcSimpleTypeDefinition* – an XSDSimpleTypeDefinition object instance which holds a simple type definition representation from the source XML Schema
- *trgSimpleTypeDefinition* – an XSDSimpleTypeDefinition object instance which holds a simple type definition representation from the second XML Schema document (target XML Schema)

The public method *doMatch()* from the **MatchSimpleTypeDefintions** class implements the new XML Schema simple type definitions match algorithm presented in the following figure:

```
1:  Persist(srcSTD)
2:  Persist(trgSTD)
3:  srcBSTD = getBaseTypeDefinition(srcSTD)
4:  trgBSTD = getBaseTypeDefinition(trgSTD)
5:  bstdMatched = MatchSimpleTypeDefinitions(srcBSTD,trgBSTD)
6:  facetsMatched = matchFacets(srcSTD, trgSTD)
7:  if (srcSTD.hasName() and trgSTD.hasName())
8:          resultNames = MatchNames(srcSTDname, trgSTDname)
9:  if (!srcSTD.hasNmae() and !trgSTD.hasName())
10:         resultNames = true
11: if (bstdMatched and facetsMatched and resultNames)
12:         persist_as_matched(srcSTD, trgSTD)
13:         return true
14: return false
```

Figure 4.12 XML Schema simple type definitions matching algorithm

XML Schema components are always persisted in the system repository like in any analogous matching algorithm presented so far, therefore lines 1 and 2 of the algorithm illustrate this fact.

A simple type definition can be derived by either extension or restriction from another simple type definition. The latter type definition represents the base type definition, and,

as per [35], has an empty specification for description of the "{final}" attribute. This attribute is described by one of the following:

- *list* – this item represents a list of simple type definitions inside <list> XML tags;
- *restriction* – this item may contain a simple type definition and one or more of the following attributes: minExclusive, minInclusive, maxExclusive, maxInclusive, totalDigits, fractionDigits, length, minLength, maxLength, enumeration, whiteSpace, pattern inside <restriction> XML tag.
- *union* – this item is an union of simple type definitions as described in section "3.14.2 (non-normative) XML Representation of Simple Type Definition Schema Components" of [35].

Lines 3 and 4 of the algorithm show the retrieval of base type definitions, and next, in line 5, those base type definitions are matched using a recursive call of simple type definitions matching algorithm.

The sixth line of the algorithm illustrates the facets matching stage of the process. A facet is modeled by XSDFacet object within XML Schema Infoset Model API Reference [50]. Therefore, every facet of each simple type definition is evaluated and compared with the corresponding one in the other simple type definition. The *facetsMatched* Boolean variable is true when all the facets from both simple type definitions match.

As per [36]:

"[Definition:] A facet is a single defining aspect of a ·value space·. Generally speaking, each facet characterizes a ·value space· along independent axes or dimensions."

There are two types of facets: fundamental or constraining (non-fundamental). A fundamental facet is defined in [36] as follows:

"[Definition:] A fundamental facet is an abstract property which serves to semantically characterize the values in a <<value space>>"

The fundamental facets are:

- *equal* – describes the equality relation over a value space for a data type described by an XML Schema simple type definition. As a note, there is no schema component that can be characterized by this facet [36].
- *ordered* – describes the order relation over a value space for a data type described by an XML Schema simple type definition. Allowed values for this facet are: *false*, *partial* and *total*.
- *bounded* – specifies if a value space is bounded or not, being a Boolean variable.
- *cardinality* – describes a value space as being "*finite*" or "*countably infinite*" and these are the values of this facet.
- *numeric* – describes a data type as being a numeric one. This facet is a Boolean variable.

A constraining facet is defined in [36] as follows:

"*[Definition:] A constraining facet is an optional property that can be applied to a datatype to constrain its ·value space·.*"

Constraining facets are:

- *length* – represents the number (a non-negative integer) of units of length for the respective data type defined by the XML Schema simple type definition. For instance, in the case of character string data type, the length represents the number of characters allowed for that data type. In the case of *hexBinary* and *base64Binary* data types, the length attribute represents the number of bytes (octets) of respective data type.
- *minLength* – describes the minimum length expressed by a non-negative integer that is allowed for the respective data type.
- *maxLength* – describes the maximum length expressed by a non-negative integer that is allowed for the respective data type.

- *pattern* – describes a regular expression pattern (regex) that constrains a character string data type. For instance, the following *ProductIDType* simple type definition allows only digits in the actual character string data type:

```
<simpleType name='ProductIDType'>
 <restriction base='string'>
  <pattern value='[0-9]*'/>
 </restriction>
</simpleType>
```

- *enumeration* – represents a set of values that are allowed for respective data type defined by the XML Schema simple type definition. For instance, a WeekDayType simple type definition may look as follows:

```
<simpleType name='WeekDayType'>
        <restriction base='string'>
                <enumeration value='Monday'/>
                <enumeration value='Tuesday'/>
                <enumeration value='Wednesday'/>
                <enumeration value='Thursday'/>
                <enumeration value='Friday'/>
                <enumeration value='Saturday'/>
                <enumeration value='Sunday'/>
        </restriction>
</simpleType>
```

Figure 4.13 Example of a simple type definition with an enumeration constraint

- *whiteSpace* – this facet can have one of the following values: *preserve, collapse, replace*.
- *maxInclusive* – denotes the "inclusive upper bound" for the space of values for a data type defined by an XML Schema simple type definition – less or equal relation.

- *maxExclusive* – denotes the "exclusive upper bound" for the space of values for a data type defined by an XML Schema simple type definition – strictly less relation.
- *minExclusive* – denotes the "exclusive lower bound" for the space of values for a data type defined by a simple type definition – strictly greater relation.
- *minInclusive* – denotes the "inclusive lower bound" for the space of values for a data type defined by a simple type definition – greater or equal relation.
- *totalDigits* – this facet expresses the maximum size of digits allowed for a data type derived from the decimal primitive type.
- *fractionDigits* – denotes the number of digits allowed after the decimal point for a data type derived from the decimal primitive type.

A simple type definition may or may not have a name. If it does not have a name, it is called an anonymous type definition. If both simple type definitions subject of the matching process have names, then their names must be matched – line 8 of the algorithm.

If the following conditions are met:

- base type definitions of source and target simple type definitions are a match
- all facets of both simple type definitions match
- in the case of named simple type definitions, their names are semantically identical

then the two simple type definitions are a match and both of them are saved in the system repository accordingly – lines 11 and 12 of the algorithm.

The following figure illustrates the UML sequence diagram of matching simple type definitions algorithm:

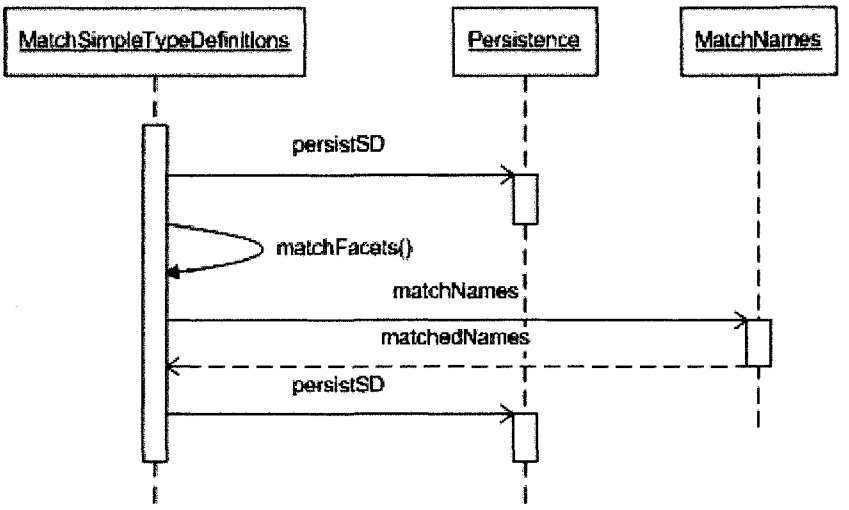


Figure 4.14 Simple type definition matching algorithm UML sequence diagram

The two simple type definitions are persisted into the repository through the activation of the **Persistence** object by the *persistSD* message. The facets of the two simple type definitions are then matched – *matchFacets* message. The **MatchNames** object is activated by the *matchNames* message. In the case of a match between the two simple type definitions, these type definitions are stored as a match in the repository by the activating of the **Persistence** object through the *persistSD* message.

The class **MatchComplexTypeDefinitions** is a subclass of the **Match** class and it has the following two private members:

- *srcComplexTypeDefinition* – an **XSDComplexTypeDefinition** object instance which holds a complex type definition representation from the source XML Schema
- *trgComplexTypeDefinition* – an **XSDComplexTypeDefinition** object instance which holds a complex type definition representation from the target XML Schema

The public method *doMatch()* in the **MatchComplexTypeDefintions** class implements the new XML Schema complex type definitions match algorithm which is illustrated in Figure 4.15:

```
1:  Persist(srcCTD)
2:  Persist(trgCTD)
3:  srcBSTD = getBaseTypeDefinition(srcSTD)
4:  trgBSTD = getBaseTypeDefinition(trgSTD)
5:  if (srcBSTD is Simple_Type) and (trgBSTD is Simple_Type) then
6:          bctdSimpleMtch = MatchSimpleTypeDefinitions(srcBSTD,trgBSTD)
7:  if (srcBSTD is Complex_Type) and (trgBSTD is Complex_Type) then
8:          bctdComplMtch = MatchComplexTypeDefinitions(srcBSTD,trgBSTD)
9:  srcAttribList = getAttributes(srcCTD)
10: trgAttribList = getAttributes(trgCTD)
11: for each srcAttribUse in srcAttribList
12:         for each trgAttribUse in trgAttribList
13:                 bAttMatch = MatchAttributeUses(srcAttribUse, trgAttribUse)
14:                 bAttListMatch.add(bAtMatch)
15: srcCTDcontent = srcCTD.getContent
16: trgCTDcontent = trgCTD.getContent
17: if (srcCTDcontent is Simple_Type) and (trgCTDcontent is Simple_Type) then
18:         bctdSimpleContentMatch = MatchSimpleTypeDefinitions(srcCTDContent,
    trgCTDContent)
19: if (srcCTDcontent is Particle) and (trgCTDcontent is Particle) then
20:         bctdParticleMatch = MatchParticles(srcCTDContent, trgCTDContent)
21: if (bctdSimpleMatch and bAttListMatch and bctdSimpleContent) then
22:         persist_as_matched(srcCTD, trgCTD)
23:         return true
24: if (bctdCmplMatch and bAttListMatch and bctdSimpleContent) then
25:         persist_as_matched(srcCTD, trgCTD)
26:         return true
27: if (bctdSimpleMatch and bAttListMatch and bctdParticleMatch) then
28:         persist_as_matched(srcCTD, trgCTD)
29:         return true
30: if (bctdCmplMatch and bAttListMatch and bctdParticleMatch) then
31:         persist_as_matched(srcCTD, trgCTD)
32:         return true
33: return false
```

Figure 4.15 XML Schema complex type definition matching algorithm

Both complex type definitions are persisted in the system repository – lines 1 and 2 of the algorithm. The next two steps of the algorithm – lines 3 and 4 – find the base type definitions of the two complex type definitions being compared by the matching process. Like simple type definitions, these complex type definitions may be a derivation by extension or restriction of other previously defined simple or complex type definitions. If

the two base type definitions are extensions of other simple type definitions then those two are matched using the simple type definitions matching algorithm – lines 5 and 6. Otherwise, there will be a recursive call of the complex type definitions matching algorithm for the two base type definitions – lines 7 and 8.

Every complex type definition – according to [35] – may contain attribute declarations. These attribute declarations can be *local* declarations of attributes, i.e. complete attribute declarations inside that complex type definition, or *reference* declarations to global attribute declarations. To be more precise, these attribute declarations from a complex type definition are contained by an attribute use XML Schema component. Therefore, the next steps of the algorithm retrieves a list of attribute use XML Schema components for each complex type definition (lines 9 and 10). Then each attribute use XML Schema component from the source complex type definition is matched with each attribute use XML Schema component from the target complex type definition. The result of each attribute use XML Schema component matching process is then stored in a Boolean array *bAttListMatch* (lines 11 to 14). The attribute uses matching algorithm is presented in the **MatchAttributeUses** class description.

The next phase of the algorithm drills further down in the complex type definition structure (lines 15 and 16). A complex type definition may contain either a simple type definition or a particle. Lines 17 and 18 of the algorithm show the simple type definitions matching process. If both complex type definitions contain particles, then those particles will be matched using the particle matching algorithm illustrated in the **MatchParticles** class description (lines 19 and 20).

The last phase of the algorithm draws a conclusion from previous phases. Because of the fact that a complex type definition can have (1) either a simple type definition or a complex type definition as base type and (2) either a simple type definition or a particle as content, there are four *"if"* branches (lines 21, 24, 27 and 30). In the case of one being true, both complex type definitions are a match and both of them are saved in the system

repository accordingly (lines 22, 25, 28 and 31). If none of the match condition is met, then the algorithm returns false, i.e. the two complex type definitions do not match.

The following figure illustrates the UML sequence diagram of the algorithm for matching complex type definitions:

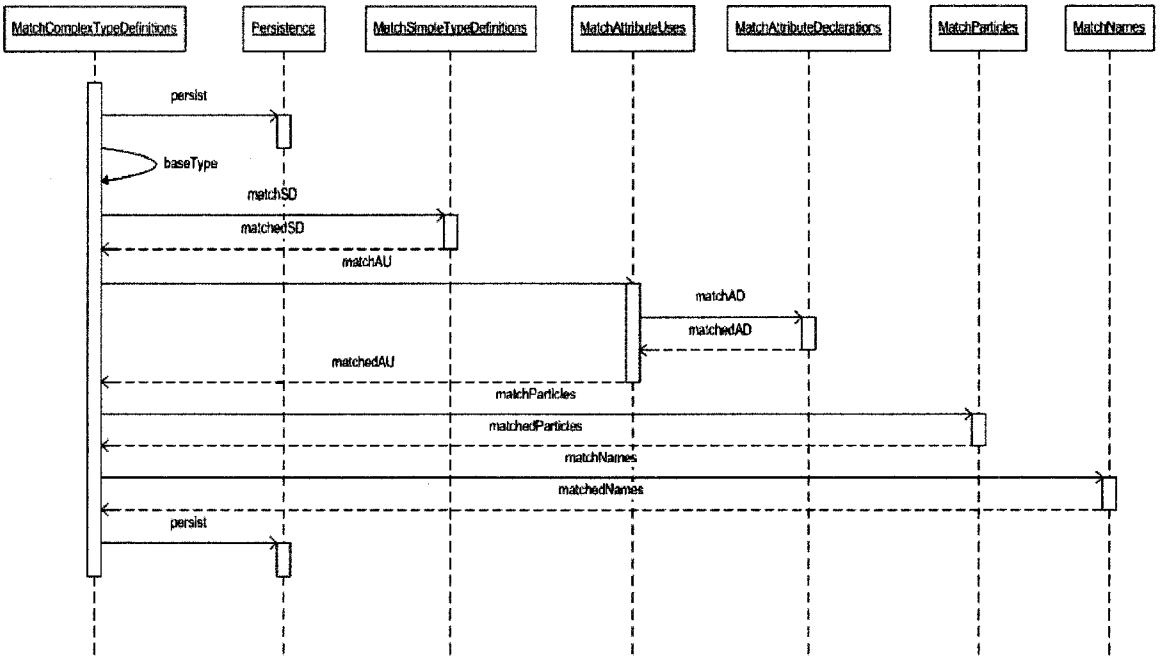


Figure 4.16 Complex type definitions matching algorithm UML sequence diagram

The two complex type definitions are stored into the repository by the activating of the **Persistence** object through the *persist* message. The a base type match is performed – *baseType* message. The **MatchSimpleTypeDefinitions** object is activated by the *matchSD* message. If the two complex type definitions contain attribute declarations then the **MatchAttributeUse** and **MatchAttributeDeclaretions** objects are activated through *matchAU* and *matchAD* messages. If the two complex type definitions contain element declarations then the **MatchParticles** object is activated by the *matchParticles* message. After the reply from the **MatchParticles** object, the **MatchNames** object is activated by the *matchNames* message. In the case of a match between the two complex type

definitions, these complex type definitions are stored as a match in the repository by the activating of the **Persistence** object through the *persist* message.

The class **MatchAttributeUses** is a subclass of the **Match** class and it has the following two private members:

- *srcAttributeUse* – an **XSDAttributeUse** object instance which holds an attribute use XML Schema component representation from the source XML Schema. As per [35]:
  "[Definition:] An attribute use is a utility component which controls the occurrence and defaulting behaviour of attribute declarations"
- *trgAttributeUse* – an **XSDAttributeUse** object instance which holds an attribute use XML Schema component representation pf an attribute use from the target XML Schema.

The public method *doMatch()* from the **MatchAttributeUses** class implements the new XML Schema attribute use matching algorithm, presented in the following figure:

```
1:  Persist(srcAttUse)
2:  Persist(trgAttUse)
3:  if (srcAttUse.isRequired() and trgAttUse.isRequired()) then
4:          matchRequired = true
5:  if (!srcAttUse.isRequired() and !trgAttUse.isRequired()) then
6:          matchRequired = true
7:  if (srcAttUse.getUseCategory() == trgAttUse.getUseCategory) then
8:          matchAttUseCategory = true
9:  if (srcAttUse.getLexical() == trgAttUse.getLexical()) then
10:         matchLexicalValue = true
11: srcAttDecl = srcAttUse.getAttributeDeclaration()
12: trgAttDecl = trgAttUse.getAttributeDeclaration()
13: matchAttribs = MatchAttributeDeclarations(srcAttDecl, trgAttDecl)
14: if (matchRequired and matchAttUseCategory and matchLexicalValue and
    matchAttribs) then
15:         persist_as_matched(srcAttUse, trgAttUse)
16:         return true
17: return false
```

Figure 4.17 XML Schema Attribute Use components matching algorithm

As per [35]:

> *"[Definition:] An attribute use is a utility component which controls the occurrence and defaulting behavior of attribute declarations. It plays the same role for attribute declarations in complex types that particles play for element declarations."*

In other words, an attribute use XML Schema component can be regarded as a wrapper for an attribute declaration XML Schema component, having the following properties:

- *required* – a Boolean attribute which specifies whether the actual attribute declaration describes an attribute that is mandatory or not within an XML element
- *attribute declaration* – the actual attribute declaration XML Schema component
- *value constraint* – a pair consisting of a value and one of *default* or *fixed*.

The attribute use XML Schema component is modeled by the **XSDAttributeUse** object in the XML Schema Infoset Model [50].

Both attribute use XML Schema utility components are stored into the system repository – lines 1 and 2 of the algorithm. The attribute use properties are analyzed from line 3 to line 10 of the algorithm. If both components have the same value for the "*required*" attribute, then *matchRequired* takes value of "true". Furthermore, if both attribute use XML Schema components have the same value for category attribute and for lexical value attribute, then the *matchAttUseCategory* and the *matchLexicalValue* Boolean variables, are set to "true".

The next phase of the algorithm consists in the matching process of the actual attribute declarations that are included in each attribute use XML Schema component (lines 11 to 13).

Line 14 of the algorithm illustrates the verification of the conditions that must be met in order to report a match between the two attribute use components. If this is the case, then

both attribute use components are stored in the system repository as being a match and the algorithm returns the value of "*true*". If there is no match, then the algorithm returns the value of "*false*".

The following figure illustrates the UML sequence diagram of attribute use matching algorithm:

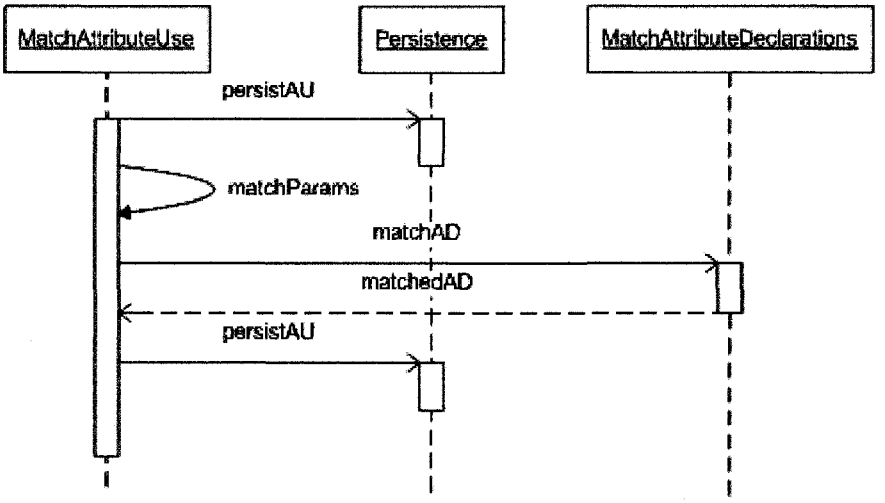


Figure 4.18 Attribute use components matching algorithm UML sequence diagram

The two attribute use XML Schema components are saved into the repository by the activating of the **Persistence** object through the *persistAU* message. The **MatchAttributeDeclarations** object is activated by the *matchAD* message. If the two attribute use XML Schema components match, then they are saved into the repository as a match – *persistAU* message.

The class **MatchModelGroupDefinitions** is a subclass of the **Match** class and has the following two private members:

- *srcModelGroupDefinition* – an **XSDModelGroupDefinition** object instance which holds a model group definition XML Schema component representation from the source XML Schema.

- *trgModelGroupDefinition* – an **XSDModelGroupDefinition** object instance which holds a model group definition XML Schema component representation from the target XML Schema

The public method *doMatch()* from the **MatchModelGroupDefintions** class implements the new XML Schema model group definitions match algorithm presented in the following figure:

```
1:  Resolve(srcMGD)
2:  Resolve(trgMGD)
3:  Persist(srcMGD)
4:  Persist(trgMGD)
5:  srcMG = srcMGD.getModelGroup()
6:  trgMG = trgMGD.getModelGroup()
7:  matchedMG = MatchModelGroups(srcMGD, trgMGD)
8:  if (matchedMG) then
9:          persist_as_matched(srcMGD, trgMGD)
10:         return true
11: return false
```

Figure 4.19 XML Schema Model Group Definition components matching algorithm

The first two lines of the algorithm resolve model group definitions. One of the model group definitions or both can be a reference to another model group definition, so these steps are mandatory in order to get consistent results at the end of matching process. The next phase consists in saving these XML Schema components into the system repository (lines 3 and 4).

As per [35], a model group definition is defined as follows:

> *"[Definition:] A model group definition associates a name and optional annotations with a Model Group (§2.2.3.1). By reference to the name, the entire model group can be incorporated by reference into a {term}."*

A model group definition is like a wrapper for a model group, being similar to an attribute use XML schema component which is a wrapper for an attribute declaration. The main difference is that the sole purpose of a model group definition is to provide a reference to a model group, whereas an attribute use XML Schema component should be included as a whole (not as a reference) in the complex type definition where is needed. A model group definition contains one and only one model group XML Schema component (lines 5 and 6).

The next phase of the algorithm consists in matching each model group XML Schema component contained in the correspondent model group definition component – line 7 of the algorithm. The model groups matching algorithm is presented within the **MatchModelGroups** class description. If the two model groups match, then the two model group definitions match as well so they are saved into the system repository accordingly and the algorithm returns value of true – lines 9 and 10. Otherwise, the model group definitions matching algorithm returns value of false, as per line 11.

The following figure illustrates the UML sequence diagram of matching model group definitions algorithm:

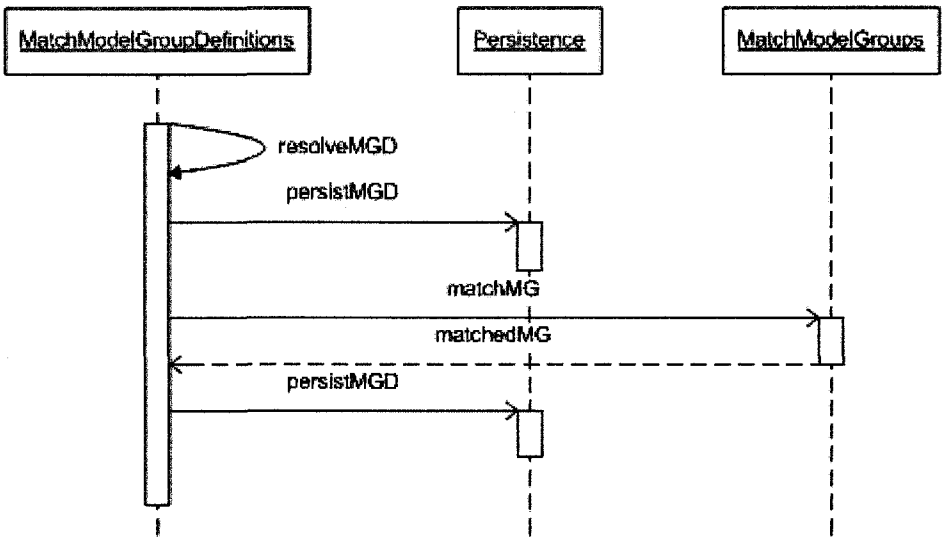


Figure 4.20 Model group definitions matching algorithm UML sequence diagram

The two model group definitions are first resolved in order to perform a match between concrete model group definitions (*resolveMGD* message). Then the two model group definitions are stored into the repository (*persistMGD* message). The **MatchModelGroups** object is activated by the *matchMG* message. In the case of a match between the two model groups definitions, these model group definitions are saved into the repository as a match by the activating of the **Persistence** object through the *persistMGD* message.

The class **MatchModelGroups** is a subclass of **Match** and it has the following two private members:

- *srcModelGroup* – an **XSDModelGroup** object instance which holds a model group XML Schema component from the source XML Schema
- *trgModelGroup* – an **XSDModelGroup** object instance which holds a model group XML Schema component from the target XML Schema

The public method *doMatch()* in the class **MatchModelGroups** implements the new XML Schema model group definitions match algorithm, illustrated in Figure 4.21:

```
1:  Persist(srcMG)
2:  Persist(trgMG)
3:  srcMGComp = srcMG.getCompositor()
4:  trgMGComp = trgMG.getCompositor()
5:  if (srcMGComp == trgMGComp) then
6:          matchedCompositor = true
7:  foreach srcParticle in srcMG.getParticles()
8:          foreach trgParticle in trgMG.getParticles()
9:                  matchedParticle = MatchParticles(srcParticle, trgParticle)
10:                 resultParticles.add(matchedParticles)
11: if (matchedCompositor and resultParticles) then
12:         persist_as_matched(srcMG, trgMG)
13:         return true
14: return false
```

Figure 4.21 XML Schema Model Group component matching algorithm

A model group XML Schema component consists of the following [35]:

- *compositor* – one of *all*, *sequence* or *choice* XML element tags (i.e. <all>, <sequence> or <choice>)
- *particles* – a list of particles
- *annotation* – a comment describing its purpose

The *<all>* model group compositor specifies that all elements contained in the list of particles can appear in the part of the data instance validated by this model group with the additional condition that – for each element - its number of instances is zero or one. Furthermore, their order of instances does not matter. The *<sequence>* model group compositor means that the element information items validated by this model group must correspond to the order of the model group's particles. The *<choice>* model group compositor denotes that only one element information item of the specified particles may be present in the data instance document.

Lines 1 and 2 of the algorithm save that the two model group XML Schema components being matched into the system main repository. Next, their compositor properties are compared (lines 3 to 5). If they are the same, the two components are a match (line 6).

The next phase of the algorithm consists in matching one by one all particles that come from both model groups. The matching algorithm for particles XML Schema components is illustrated in the **MatchParticles** class description –lines 7, 8 and 9. A Boolean array – *resultParticles* – is then constructed with all the results of matching particles algorithms (line 10).

The final phase of the algorithm consists in drawing a conclusion about the two model group XML Schema components: match or not match. In case of a match, the two model groups are persisted as a match in the system repository and the algorithm returns value of true, otherwise it returns false – lines 11 to 14.

The following figure illustrates the UML sequence diagram of matching model groups algorithm:

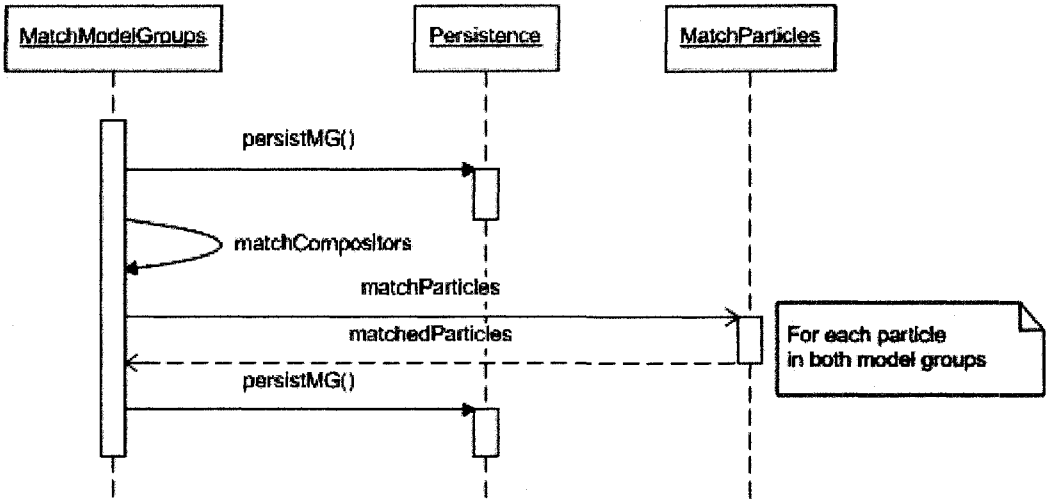


Figure 4.22 Model group matching algorithm UML sequence diagram

The two model groups XML Schema components are saved into the repository (the *persistMG* meesage) and thei compositors (*all*, *sequence* or *choice*) are then matched – *matchCompositors* message. A model group may contain more than one particle, therefore for each particle in each model group the **MatchParticles** object is activated by the *matchParticles* message. In the case of a match between the two model groups, these model groups are saved into the repository – the *persistMG* message.

The class **MatchParticles** is a subclass of the **Match** class and has the following two private members:

- *srcParticle* – an **XSDParticle** object instance which holds a particle XML Schema component from thesource XML Schema
- *trgParticle* – an **XSDParticle** object instance which holds a particle XML Schema component from the target XML Schema

The public method *doMatch()* in the **MatchParticles** class implements the new XML Schema particles match algorithm, illustrated in Figure 4.23:

```
1:  Persist(srcParticle)
2:  Persist(trgParticle)
3:  srcMinOccurs = srcParticle.getMinOccurs()
4:  trgMinOccurs = trgParticle.getMinOccurs()
5:  srcMaxOccurs = srcParticle.getMaxOccurs()
6:  trgMaxOccurs = trgParticle.getMaxOccurs()
7:  if (srcMinOccurs == trgMinOccurs) then
8:          matchedMinOccurs = true
9:  if (srcMaxOccurs == trgMaxOccurs) then
10:         matchedMaxOccurs = true
11: srcPartCont = srcParticle.getContent()
12: trgPartCont = trgParticle.getContent()
13: if ((srcPartCont is ElementDeclaration) and (trgPartCont is
    ElementDeclaration)) then
14:         matchedElementDecls = MatchElementDeclarations(srcPartCont,
    trgPartCont)
15: if ((srcPartContent is ModelGroup) and (trgPartContent is ModelGroup))
    then
16:         matchedModelGroups = MatchModelGroups(srcPartCont,
    trgPartCont)
17: if ((srcPartContent is Wildcard) and (trgPartContent is Wildcard)) then
18:         matchedWildcards = MatchWildcards(srcPartCont, trgPartCont)
19: if (matchedMinOccurs and matchedMaxOccurs and matchedElemDecls)
    then
20:         persist_as_matched(srcPart, trgPart)
21:         return true;
22: if (matchedMinOccurs and matchedMaxOccurs and matchedModelGroups)
    then
23:         persist_as_matched(srcPart, trgPart)
24:         return true;
25: if (matchedMinOccurs and matchedMaxOccurs and matchedWildcards) then
26:         persist_as_matched(srcPart, trgPart)
27:         return true;
28: return false
```

Figure 4.23 XML Schema Particle component matching algorithm

As per [35], a particle XML Schema component is an important part of content model definition and has the following properties:

- *minOccurs* – a non-negative integer
- *maxOccurs* – a non-negative integer or "*unbounded*" character string

- *term* – one of element declaration, model group or wildcard XML Schema components

As with other XML Schema component matching algorithms, the particle matching algorithm saves the two particles to be matched in the repository (lines 1 and 2). In the subsequent phase, the *minOccurs* and the *maxOccurs* particle properties are retrieved and compared. If there is a match between the *minOccurs* properties, then the *matchedMinOccurs* Boolean variable is set to true. If there is a match between the *maxOccurs* properties, then the *matchedMaxOccurs* Boolean variable is set to true.

The subsequent phase of the particle matching algorithm retrieves the particle content type (lines 11 and 12). The particle content type can be one of the following: element declaration, model group or wildcard. If both particle content types are element declarations, then they are matched using the element declaration matching algorithm presented in the **MatchElementDeclarations** class description (lines 13 and 14). In the same manner, if both particle content types are either a model group or a wildcard, then they are matched using corresponding algorithm described in the **MatchModelGroups** class description or in the **MatchWildcards** class description (lines 15 to 18).

The final stage of the algorithm draws a conclusion about particles: either they match or they do not match. In the case of a match between their content types and their *minOccurs* and *maxOccurs* properties, then both particles are saved in the system main repository and the algorithm returns value of true. If any of the conditions has not been met, then the algorithm returns the Boolean value of "false" (lines 19 to 28).

The following figure illustrates the UML sequence diagram of the particle matching algorithm:

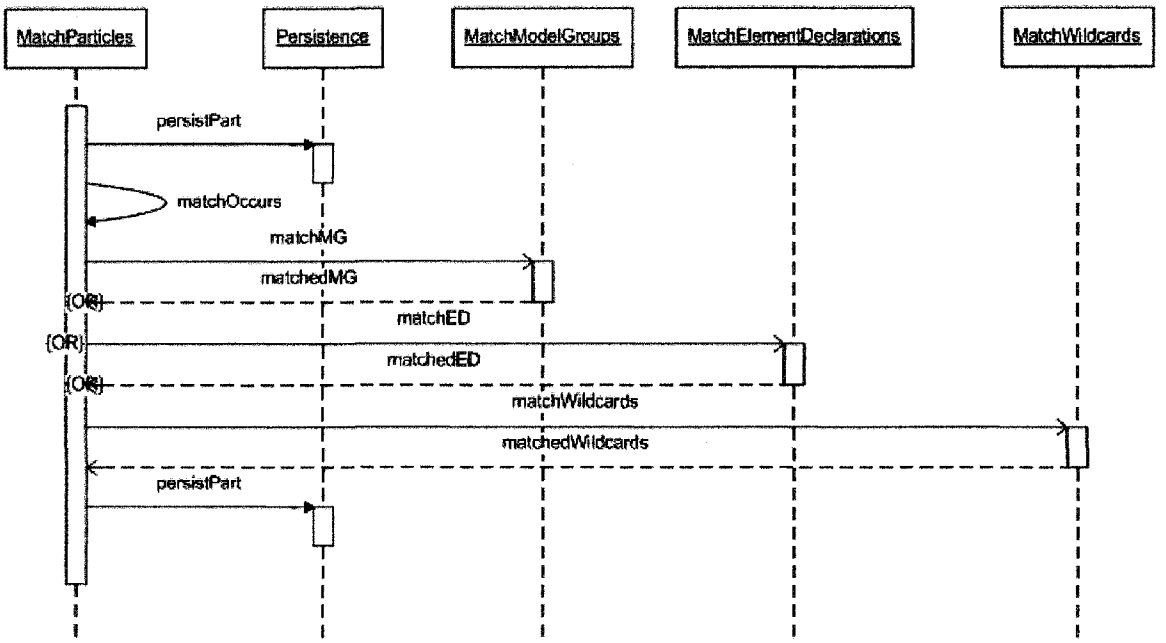


Figure 4.24 Particle matching algorithm UML sequence diagram

The two particles are saved into the repository by the activating of the **Persistence** object through the *persist* message. The min occurs and max occurs attributes of each particle are then matched (the *matchOccurs* message). The **MatchModelGroups** object is activated by the *matchMG* message and after its reply, the **MatchElementDeclarations** object is activated by the *matchED* message. The **MatchWildcards** object is activated by the *matcWildcards* message. If the two particles match, then they are saved into the repository – the *persistPart* message.

The class **MatchWildcards** is a subclass of the **Match** class and has the following two private members:

- *srcWildcard* – an **XSDWildcard** object instance which holds a wildcard XML Schema component representation from the source XML Schema
- *trgWildcard* – an **XSDWildcard** object instance which holds a wildcard XML Schema component representation from the target XML Schema

The public method *doMatch()* in the **MatchWildcards** class implements the new XML Schema wildcards match algorithm, presented in Figure 4.25

```
1:  Persist(srcWildcard)
2:  Persist(trgWildcard)
3:  srcNSConstr = srcWildcard.getNamespaceConstraint()
4:  trgNSConstr = trgWildcard.getNamespaceConstraint()
5:  if srcNSConstr == trgNSConstr then
6:          matcheNSConstr = true
7:  srcPContent = srcWildcard.getProcessContents()
8:  trgPContent = trgWildcard.getProcessContents()
9:  if srcPContent == trgPContent then
10:         matchedPContent = true
11: if (matchedNSConstr and matchedPContent) then
12:         persist_as_matched()
13:         return true
14: return false
```

Figure 4.25 XML Schema Wildcard component matching algorithm

As described in [35], a wildcard schema component has the following properties:

- *namespace constraint* – this property can be one of "*any*", or "*not*" and a namespace
- *process content* – one of *skip*, *lax* or *strict*
- *annotation* – an annotation which is a natural language description of the main purpose of respective wildcard XML Schema component

The wildcard matching algorithm compares the main properties: the namespace constraint and the process content. These properties are represented mainly as character strings in the XML Schema Infoset Model API [50], so their match is based on string comparison.

In the first phase of the algorithm, both wildcard XML Schema components are saved in the repository – lines 1 and 2. Subsequently, the namespace constraint properties of both wildcard XML Schema components are compared. The *matchedNSConstr* Boolean variable is set to "true" if they match (lines 3 to 6).

The next wildcard property to be matched is the process content. In the case of a match between the two process content properties of the source and target wildcard component, the *matchedPContent* Boolean variable is set to "true" (lines 7 to 10).

The last phase of the algorithm draws a conclusion regarding the match of both wildcard XML Schema components. If they match then they are saved in the repository and the algorithm returns a "true" value, otherwise the algorithm returns a "false" value (lines 11 to 14).

The class **MatchIdentityConstraintDefinitions** is a subclass of the **Match** class and has the following two private members:

- *srcIdentityContraintDefinition* – an **XSDIdentityConstraintDefinition** object instance which holds an identity constraint definition from the source XML Schema
- *trgIdentityConstraintDefinition* – an **XSDIdentityConstraintDefinition** object instance which holds an identity constraint definition from the target XML Schema

The public method *doMatch()* in the **MatchIdentityConstraintDefintions** class implements the new XML Schema identity constraint definitions match algorithm, shown in Figure 4.26:

```
1:  Persist(srcIdentityConstrDef);Persist(trgIdentityConstrDef)
2:  srcICDname = srcIdentityConstrDef.getName()
3:  trgICDname = trgIdentityConstrDef.getName()
4:  matchedNames = MatchNames(srcICDname, trgICDname)
5:  srcICDCat = srcIdentityConstrDef.getCategory()
6:  trgICDCat = trgIdentityConstrDef.getCategory()
7:  if ((srcICDCat == "keyref") and (trgICDCat == "keyref")) then
8:    srcReflCD = srcIdentityConstrDef.getRefKey()
9:    trgReflCD = trgIdentityConstrDef.getRefKey()
10:   matchedReflCD = MatchIdConstrDefs(srcRICD, trgRICD)
11: if ((srcICDCat == "key") and (trgICDCat == "key")) then
12:   matchedKeyCat = true
13: if ((srcICDCat == "unique") and (trgICDCat == "unique")) then
14:   matcheUniqueCat = true
15: srcSelect = srcIdentityConstrDef.getSelector()
16: trgSelect = trgIdentityConstrDef.getSelector()
17: if srcSelect == trgSelect then
18:   matchedSelector = true
19: srcFields = srcIdentityConstrDef.getFields()
20: trgFields = trgIdentityConstrDef.getFields()
21: if (srcFields.size() == trgFields.size()) then
22:   foreach srcFld in srcFields
23:     foreach trgFld in trgFields
24:       if srcFld == trgFld then
25:         matchedFieldCounter++
26: if (matchedFieldCounter == srcFields.size()) and
    (matchedFieldCounter == trgFields.size()) then
27:   matchedFields = true
28: if (matchedReflCD and matchedNames) then
29:   persist_as_matched(srcIdentityConstrDef, trgIdentityConstrDef)
30:   return true
31: if (matchedKeyCat and matchedSelector and matchedFields and
    matchedNames) then
32:   persist_as_matched(srcIdentityConstrDef, trgIdentityConstrDef)
33:   return true
34: if (matchedUniqueCat and matchedSelector and matchedFields and
    matchedNames) then
35:   persist_as_matched(srcIdentityConstrDef, trgIdentityConstrDef)
36:   return true
37: return false
```

Figure 4.26 XML Schema Identity Constraint Definition matching algorithm

As defined in [35], an identity constraint definition XML Schema component can be used to provide *uniqueness* and *reference* constraints for multiple elements and attributes that occur in an XML document. This XML Schema component has the following properties:

- *name* – a unique name
- *target namespace* – an XML namespace
- *identity-constraint category* – either one of *key*, *keyref* or *unique*
- *selector* – an XPath expression [51]
- *fields* – a non-empty list of XPath expressions
- *referenced key* – contains a *key* or *unique* property that belongs to another identity constraint definition if and only if its own identity-constraint category is *keyref*.
- *annotation* – an optional annotation which is a description (in natural language) of the main purpose of respective identity constraint definition XML Schema component.

First, the identity constraint definitions are saved in the repository – line 1. The algorithm continues with identity constraint definition name matching using the algorithm illustrated in the **MatchNames** class description (lines 2 to 4). If the categories of both identity constraint definitions are "*keyref*", then both refer to other identity constraints that need to be matched (lines 5 to 7). The referenced identity constraint definitions are matched using a recursive call of the algorithm (lines 8, 9 and 10). The next steps compare the "*key*" and "*unique*" identity constraint properties. If both identity constraint properties match, the corresponding Boolean variables are set to "true" – either *matchedKeyCat* or *matchedUniqueCat* (lines 11 to 14). The selector identity constraint definition properties are then compared, and if they match – i.e. they have the same character strings – a Boolean variable, *matchedSelector*, is set to true (line 18).

The next identity constraint definition properties that are to be compared are the "*field*" properties (lines 21 and 27). This task is accomplished by comparing all the fields from the two identity constraint definitions. This comparison takes place if both identity constraint definitions contain same number of fields. If the number of matched fields is the same as the number of fields in both identity constraint definitions, then there is a

match between these identity constraint definition properties and the *matchedFields* Boolean variable is set to value of "true".

The last phase of the algorithm draws a conclusion regarding the two identity constraint definition matching. If one of the following conditions is met, then a match between the two identity constraint definitions is reported and the algorithm returns the value of true:

- the identity constraint definition names and the referenced identity constraint definitions match (line 28)
- the identity constraint definition key categories, selectors, fields and names match (lines 31 to 33)
- the identity constraint definition unique categories, selectors, fields and names match (lines 34 to 36)

If the two identity constraint definitions do not match, then the algorithm returns value of "false" (line 37).

The class **MatchNotationDeclarations** class is a subclass of the **Match** class and has the following two private members:

- *srcNotationDeclaration* – an **XSDNotationDeclaration** object instance which holds a notation declaration XML Schema component from the source XML Schema
- *trgNotationDeclaration* – an **XSDNotationDeclaration** object instance which holds a notation declaration XML Schema component from the target XML Schema

The public method *doMatch()* in the **MatchNotationDeclarations** class implements the new XML Schema identity constraint definitions match algorithm, illustrated in the following figure:

```
1:  Persist(srcNotDecl)
2:  Persist(trgNotDecl)
3:  srcPI = srcNotDecl.getPublicIdentifier()
4:  trgPI = trgNotDecl.getPublicIdentifier()
5:  srcSI = srcNotDecl.getSystemIdentifier()
6:  trgSI = trgNotDecl.getSystemIdentifier()
7:  if (srcPI == trgPI) then
8:          matchedPublicId = true
9:  if (srcSI == trgSI) then
10:         matchedSystemId = true
11: if (matchedPublicId and matchedSystemId)
12:         persist_as_matched(srcNotDecl, trgNotDecl)
13:         return true
14: return false
```

Figure 4.27 XML Schema Notation Declaration components matching algorithm

The notation declaration XML Schema component is characterized by the following features [35]:

- *name* – a unique name
- *target namespace* – and XML namespace
- *system identifier* – an URI;
- *public identifier* – an XML public identifier
- *annotation* - a description (in natural language) of the main purpose of the notation declaration XML Schema component.

An example of a notation declaration looks like this:

<notation name="gif" public="image/gif" system="showgif.exe"/>

Like all the other XML Schema components, notation declarations from source and target XML Schema documents are saved in the repository, as a first phase in the matching algorithm.

The next phase of the algorithm retrieves the notation declaration properties for source and target components. These properties are character strings which are easy to compare. In case of a match, both notation declarations are saved in the system repository as a match and the algorithm returns a value of true, otherwise it returns a value of false.

The **MatchNames** class takes care of the XML Schema component name matching. This type of matching is also known in literature as the *linguistic matching*. This class has two private members as follows:

- *srcName* – a character string which represents a name of an XML Schema component from the source XML schema
- *trgName* – a character string which represents a name of an XML Schema component from the target XML schema

The **MatchNames** class contains the following methods:

- *normalizeTokenizeName*: a private method which takes as input a character string and returns an array of character strings. For instance, if the input character string is "LastName", the output is ["last", "name"], or in case of "unit_price", the return value is ["unit", "price"]. This method is responsible for the character string normalization and tokenization, where tokenization is performed based on the upper case characters or the underscore character ("_").
- *doMatch()* – a public method which performs the actual name matching using the Java WordNet Library (JWNL) API [65]. This method returns the Boolean value "true" if the *srcName* and the *trgName* have the same *meaning* and the Boolean value of "false", otherwise.

The actual algorithm of the *doMatch*() method is illustrated in Figure 4.28:

```
1: phraseSrc = normalizeTokenizeName(srcName)
2: phraseTrg = normalizeTokenizeName(trgName)
3: result = phrasalSemanticMatch(phraseSrc, phraseTrg)
4: return result
```

Figure 4.28 Semantic Name Matching Algorithm

The *phrasalSemanticMatch()* method (line 3) comes from the Java WordNet Library (JWNL) API [65].

The following figure illustrates the Matching Engine class relationships:

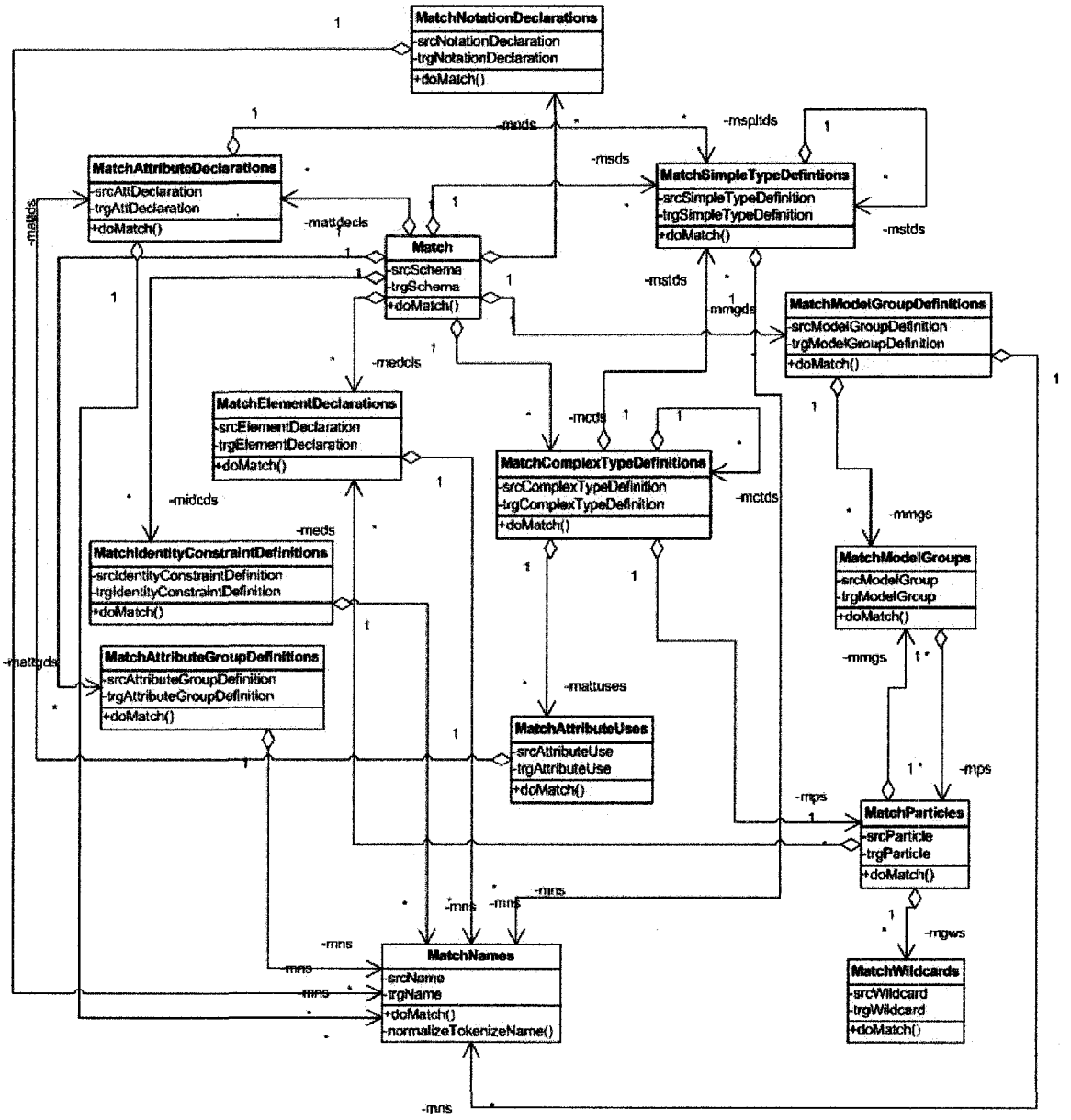


Figure 4.29 Matching Engine – Class relationships UML diagram

The whole XML Schema matching algorithm has been broken down into smaller matching algorithms for each pair of XML Schema components, following the W3C

XML Schema standard [33, 34, 35, and 36]. This approach constitutes the innovation of this thesis. The Matching Engine class relationship UML diagram resembles the XML Schema components relationship diagram. This fact leads to excellent scalability, robustness and performance for the whole application.

The main design pattern used in the Matching Engine architecture is the **Composite** structural design pattern [56]. This design pattern is suitable for objects that are composed "into tree structures to represent part-whole hierarchies" [56]. A visual description of the Composite object structure [56] is presented in Figure 4.30:

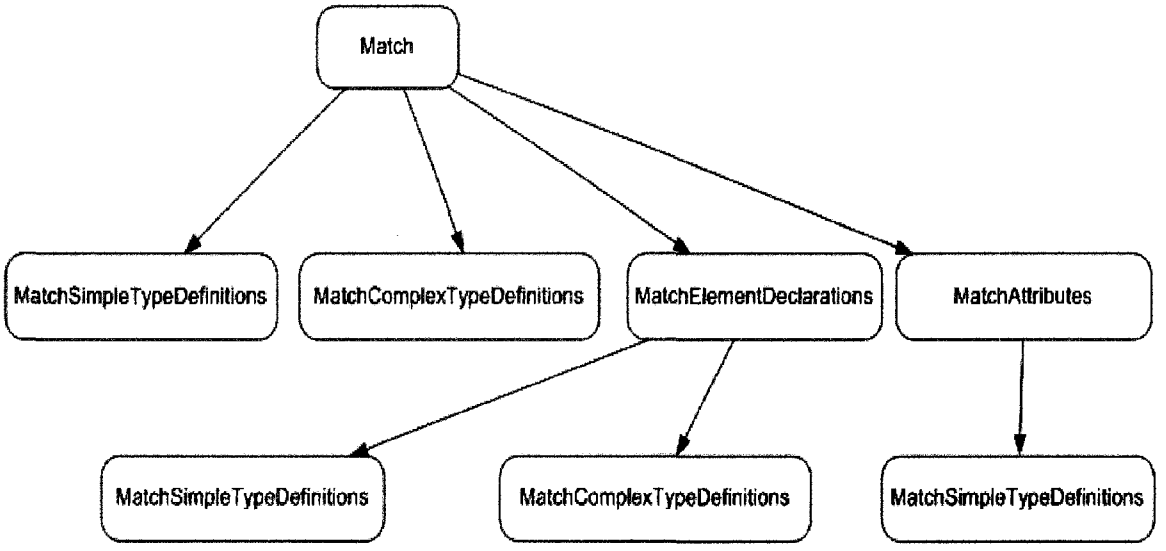


Figure 4.30 Match Composite Object Structure

Collaborations in the Matching Engine architecture follow exactly the ones described in the Composite structural design pattern. For instance, if the recipient is **MatchingElementDeclarations,** then all requests (match operations) are forwarded down to **MatchSimpleTypeDefinitions** where the result of the match is returned, or to another composite, **MatchComplexTypeDefinitions** and then further down until the result of the match is returned up to the chain.

The Composite structural design pattern [56] brings the following advantages to the Matching Engine:

- Consistency for the XML Schema components match operations: any XML Schema components are matched in the same manner and most importantly, the XML Schema structural matching paradigm is totally encapsulated by the Matching Engine.
- Any client can use the Matching Engine with no knowledge of the structure of the Matching Engine. This means that the Matching Engine can be integrated easily into other architectures.
- Should the XML Schema standard change and a new XML Schema component be introduced, then it is very easy to create and integrate a new composite or primitive object in the current Matching Engine architecture that will implement the new matching algorithm.

From an object behavioral perspective, the design pattern for the Matching Engine is **Chain of Responsibility** [56]. According to this object behavioral design pattern, a sender of a request is not coupled with the receiver that handles that request. This means that the whole matching action is not performed in the **Match** object, it is forwarded to the proper matching object according to the XML Schema components. The *doMatch()* method is the concrete handler method described in the **Chain of Responsibility** object behavioral design pattern [56].

The Chain of Responsibility behavioral design pattern brings the following advantages to the Matching Engine architecture [56]:

- *Reduced coupling* – the **MatcheElementDeclarations** object doesn't know *a priori* how far down the tree its matching request will go: it may go only one level and its request be handled by **MatchSimpleTypeDefinitions** object, or it may travel through a **MatchComplexTypeDefinitions** object, a **MatchModelGroups** object or other **MatchElementDeclarations** objects. It knows only that its request will be handled in the proper manner.

- *Flexibility* – should the XML Schema standard change and new XML Schema components be introduced, then they can be easily integrated in the current Matching Engine architecture.

The only risk is that a match request might be lost down the tree, but this risk can be easily avoided if the Matching Engine architecture follows the XML Schema standard.

In conclusion, the most important idea presented in this thesis is that an XML Schema is analyzed and handled from a **schema perspective** and not from an **XML perspective**, which actually means a higher level of abstraction. All the other XML Schema matching tools (presented in the literature review chapter of this thesis) do not use this level abstraction. Once more, an XML Schema is just written using XML and has a totally different meaning than any other XML document. Similarly, the Resource Description Framework (RDF) has several different representations and only one of them is the XML representation.

4.4 Graphical User Interface – Detailed design

The Eclipse user interface is based on the Standard Widget Toolkit (SWT) which is written in Java. However, a competitive and fast responsive user interface must rely on underlying operating system API calls. This task is accomplished in SWT by using the Java Native Interface (JNI) as presented in [53].

The first widget toolkit in Java was designed by Sun Microsystems and it was called Abstract Windowing Toolkit (AWT). This toolkit provides functionality for a minimal set of widgets like buttons, labels, menus and lists and it has a very complex interface to the native widgets – i.e. to the underlying operating system API. In order to overcome all the issues in AWT, Sun Microsystems developed another widget toolkit, Java Foundation Classes (JFC) also known as Swing™. Swing™ provides a more powerful and stable graphical user interface with a richer set of standard widgets like trees, toolbars, windows, frames, etc. Even in this case of extended set of widgets, Swing widgets do not

look the same with the native operating system widgets and their performance with respect to the response time is still far from the one of native widgets.

SWT provides graphical user interface controls with fast response time and native multi-platform implementations. These are the main reasons of choosing SWT as the foundation of Hermes GUI.

A screenshot of Hermes graphical user interface is presented in Figure 4.31:

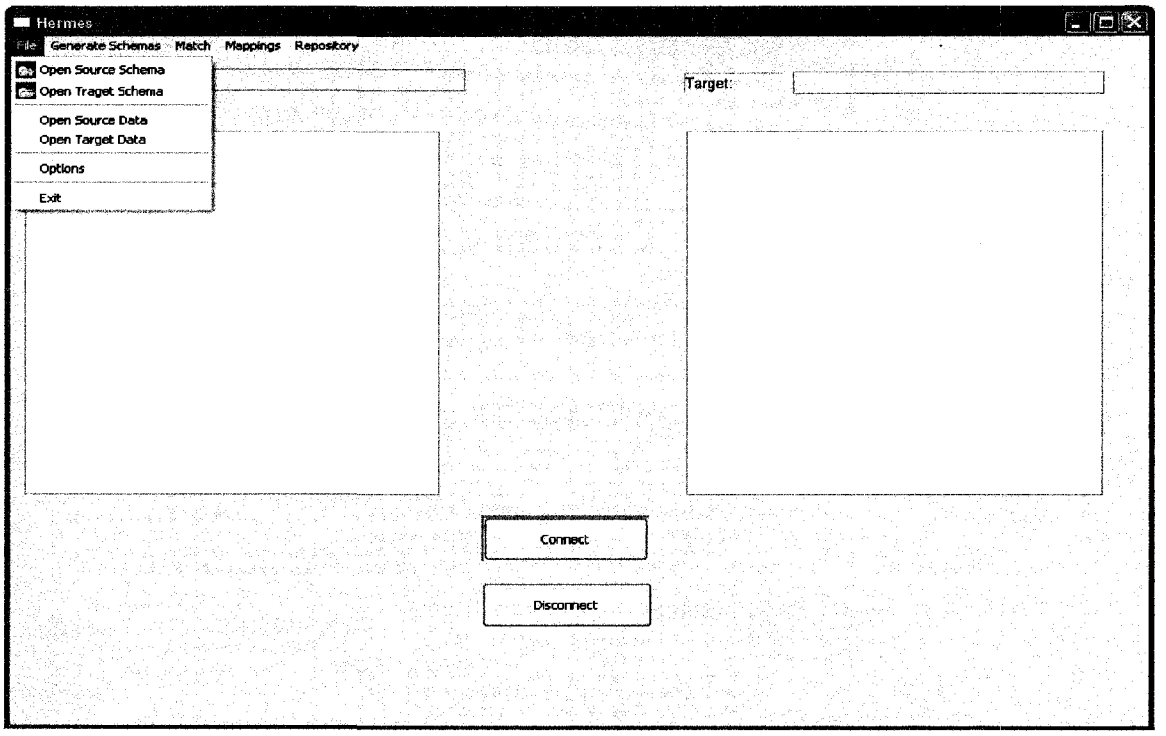


Figure 4.31 Hermes – Graphical User Interface (GUI)

Figure 4.32 illustrates a mapping between two XML Schema documents:

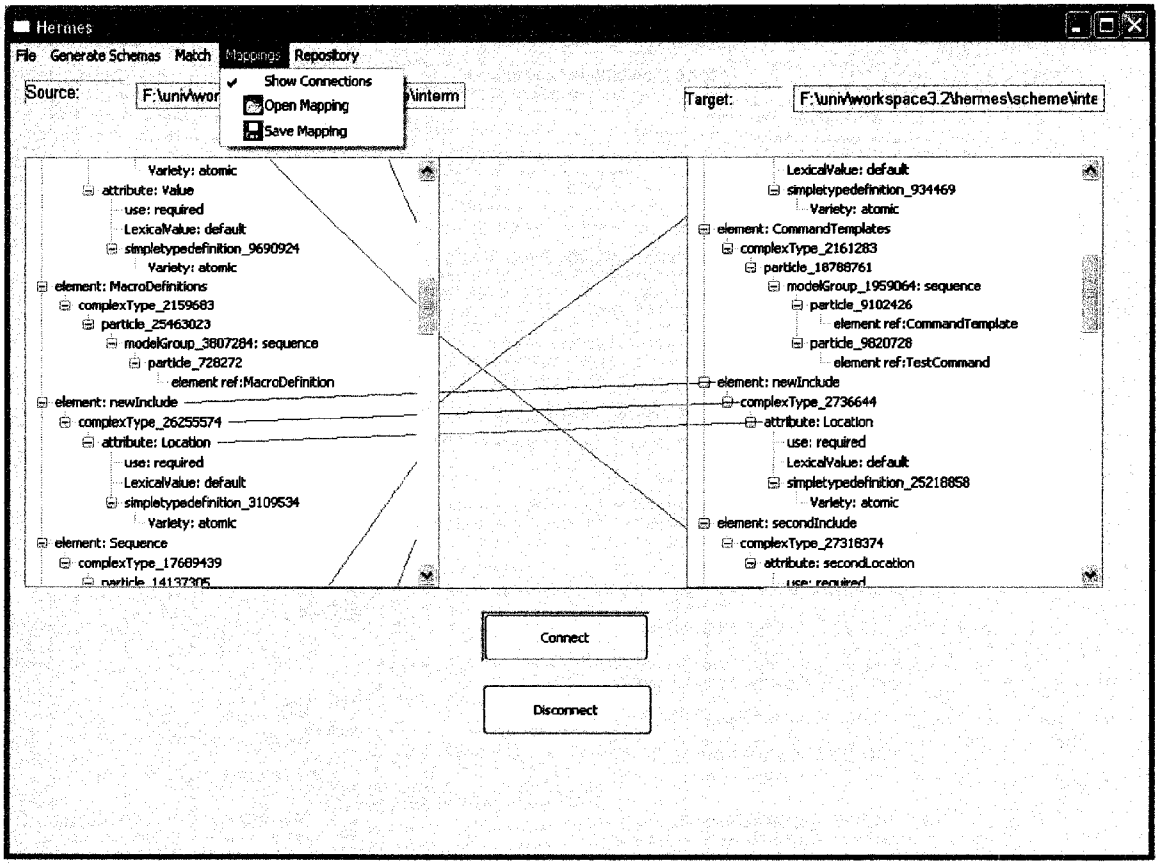


Figure 4.32 Hermes screenshot – A mapping between two XML Schema documents

A red line denotes a correspondence between two element declarations, a blue line denotes a correspondence between two type definitions and a green line denotes a correspondence between two attribute declarations.

4.5 Schema Extraction Engine

The solution proposed in this thesis for finding a minimal XML Schema for a given XML document is based on an XSLT script. Having an XML document – a data instance in XML format – it is easy to find all the elements, attributes and their relations and then format this information in a structure that conforms to the XML Schema standard.

The first step of the algorithm consists in finding all the unique XML elements [54, 55]. This task is implemented in XSLT as follows:

<xsl:if test="count(preceding-sibling::[name()=name(current())]) = 0">*

This XSLT statement checks if the current element name is different than the name of preceding sibling element [54. 55].

The type of each unique element is established as follows: if the current element has children elements and/or attributes then its type definition is a complex type definition. Otherwise, it is a simple type definition:

<xsl:if test="child:: | @*">*
<xsl:element name="xs:complexType">
...

In the case of an XML element that has children XML elements, its complex type definition must contain a model group [35]:

<xsl:if test="child::">*
<xsl:element name="xs:sequence">
...

A model group can be one of *sequence*, *all* or *choice*. The resulting XML Schema document must validate the current XML data instance, therefore the *choice* model group cannot be used, because this model group states that only one element from the list can be present and the current XML data instance contains all specified children XML elements. Consequently, only the other two model groups can be used – *sequence* and *all* – and their usage can lead to a resulting XML Schema document that validates the current XML data instance. The "*Sequence*" model group has been chosen. It provides a better description of the current XML data instance structure. The "*all*" model group may be somehow misleading because it will also validate an XML data instance with missing specified child elements of the current element analyzed.

The next phase of the transformation analyzes all the attributes for each XML data instance element. As stated in the XML Schema standard structures [35], an attribute XML Schema component is described as having a simple type definition:

```
<xsl:for-each select="@*">
        <xsl:element name="xs:attribute">
          <xsl:attribute name="name"><xsl:value-of select="name()"/></xsl:attribute>
              <xsl:attribute name="use">required</xsl:attribute>
              <xsl:element name="xs:simpleType">
                    <xsl:element name="xs:restriction">
                          <xsl:attribute name="base">xs:string</xsl:attribute>
...
```

The base data type for the simple type definition of the current analyzed attribute is considered as string. Primitive data types defined in [36] are:

- string
- boolean
- decimal
- float
- double
- duration
- dateTime
- time
- date
- gYearMonth
- gYear
- gMonthDay
- gDay
- gMonth
- hexBinary

- base64Binary
- anyURI
- QName
- NOTATION

Only a user with expert knowledge of that XML data instance may guess the most suitable base data type for the attribute under investigation. Certainly, the current XSL transformation script could be improved by adding some regular expressions which would be applied to the attribute content. In this manner. A conclusion could be then drawn with respect to the data type – string, number, date, etc. However, the problem is more complex than it seems because the base data type may be also implied by the name of attribute. In this case, a semantic analysis of the attribute name would be the most appropriate.

For the time being, the current XSL transformation generates a minimal and acceptable XML Schema that can be analyzed and compared to other XML Schema documents.

4.6 Repository and Mapping

The whole XML Schema documents matching algorithm is broken down into smaller matching algorithms between pairs XML Schema components from the two XML Schema documents, as presented in previous sections. Each XML Schema components matching algorithm contains instructions for saving every XML Schema component encountered during the matching process in the repository.

This repository is useful for the following reasons:

- It keeps an accurate record of every XML Schema component;
- It represents the starting point for the mapping process
- The Transformation Engine relies upon information from repository

- It can be backed up and restored for further refinement of an existing mapping between two given XML Schema documents

The core component of the repository is Sesame – an architecture that can store and query RDF information [45, 46, and 47].

4.6.1 Repository – Detailed design

The repository consists of the following classes illustrated in Figure 4.33:

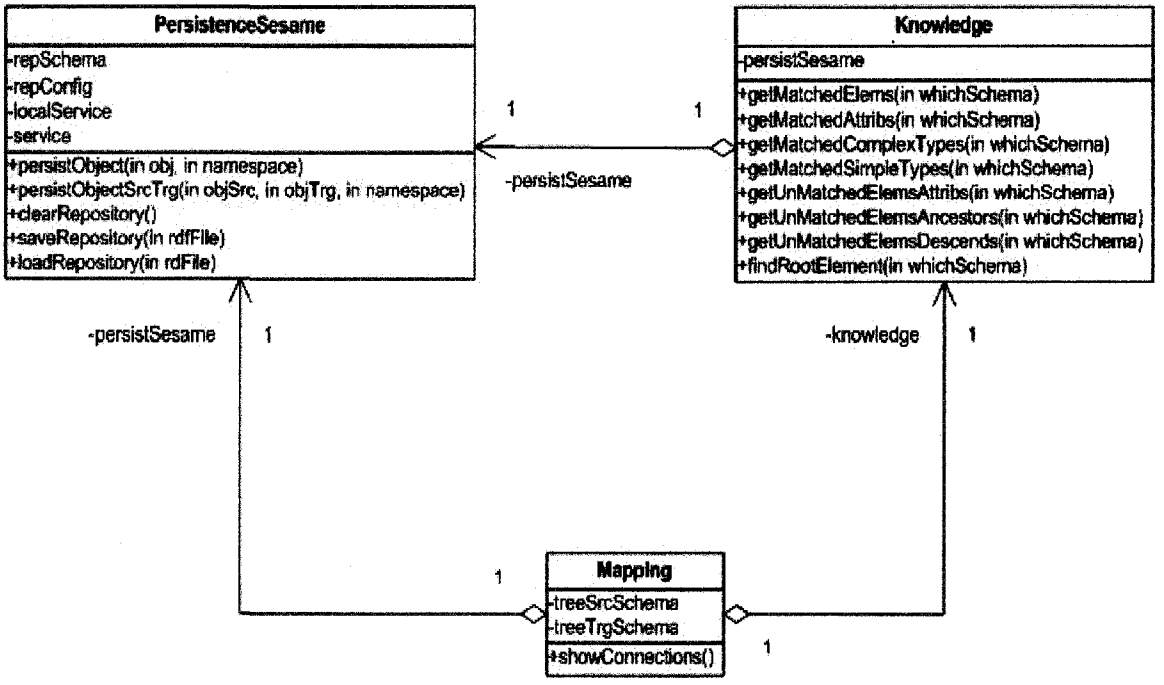


Figure 4.33 Repository and Mapping – UML Class diagram

The **PersistenceSesame** class represents the link between Hermes and the Sesame architecture [46]. Sesame can be configured to work in either a local or a remote configuration. Therefore, the **PersistenceSesame** class has two constructors: one used to configure Sesame to work in a local configuration – i.e. records are kept on the local computer – and the other configures Sesame to work in a remote configuration – i.e.

records are kept on a remote computer. The attributes of **PersistenceSesame** class are as follows:

- *repSchema* – an org.openrdf.sesame.repository.SesameRepository interface used to access a single repository
- *repConfig* – an org.openrdf.sesame.config.RepositoryConfig object which contains parameters that are used to configure a repository
- *localService* –an org.openrdf.sesame.repository.local.LocalService instance which represents a Sesame service for local repositories;
- *service* – an org.openrdf.sesame.repository.SseameService interface

The **PersistenceSesame** class has the following public methods:

- *persistObject* – this method stores an XML Schema component in an RDF format.
- *persistObjectSrcTrg* – this method stores two XML Schema components that match in an RDF format
- *clearRepository* – this method erases the current repository content so that the repository is ready for a new matching process
- *saveRepository* – this method serializes the repository content to an RDF file format
- *loadRepository* – this method loads the repository from an RDF file.

The following figure illustrates the element declaration RDF format stored in the repository – the final result of *persitObject* method call:

```
<xs:element name="FirstName">
        <xs:complexType>
                <xs:attribute name="fName" use="required">
                        <xs:simpleType>
                                <xs:restriction base="xs:string"/>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
</xs:element>
```

a.) An Element Declaration

```
1. <rdf:Description rdf:about="http://ncct.org/src#FirstName">
2. <ns1:isNillable rdf:resource="http://www.w3.org/2001/XMLSchema#false"/>
3. <rdf:type rdf:resource="http://www.w3.org/2001/XMLSchema#element"/>
4. <ns1:parentName rdf:resource="http://ncct.org/src#schema_12273995"/>
5. <ns1:hasParent rdf:resource="http://ncct.org/src#true"/>
6. <ns1:childName rdf:resource="http://ncct.org/src#ComplexType_of_FirstName"/>
7. </rdf:Description>
```

b.) Element Declaration RDF representation in the repository

Figure 4.34 An Element Declaration and its RDF representation in the repository

Figure 4.35. a.) illustrates a simple element declaration XML Schema component. Figure 4.35. b.) shows a simple RDF representation within the repository of the above element declaration. The RDF statement has the following meaning:

- rdf:about : points to the subject of the RDF statement - "*FirstName*" - which represents an XML Schema component with the name "*FirstName*"; (line 1)
- ns1:isNillable: represents a predicate (line 2)
- rdf:resource represents the object of the first sentence and its value is "false" (line 2)
- rdf:type – gives information about the type of the "*FirstName*" XML Schema component an element declaration in this case (line 3)
- ns1:parentName – gives the information about the "*FirstName*" element declaration parent. If the predicate object points to a resource that contains

"schema_" characters string, this means that the "*FirstName*" element declaration is a global declaration. An XML Schema document can import or include several different XML Schema documents; therefore it is important to know what XML Schema document contains "*FirstName*" element declaration (line 4)

- ns1:hasParent - if the "*FirstName*" element declaration is referenced in a complex type definition then the *hasParent* predicate gives that information pointing to a resource that contains "true" (line 5).
- The last predicate – *childName* – has an object that points to a resource which contains an XML Schema component that belongs to the "*FirstName*" element declaration – in the current example a complex type definition XML Schema component (line 6).

The following figure describes the final result of the *persistObjectSrcTrg* method call. This method is called in case of a match between two XML Schema components – one from the source XML Schema document and the other form the target XML Schema document.

```
<xs:element name="FirstName">
        <xs:complexType>
                <xs:attribute name="fName" use="required">
                        <xs:simpleType>
                                <xs:restriction base="xs:string"/>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
</xs:element>
```

```
<xs:element name="Forename">
        <xs:complexType>
                <xs:attribute name="fName" use="required">
                        <xs:simpleType>
                                <xs:restriction base="xs:string"/>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
</xs:element>
```

Figure 4.35 Two element declarations that match

```
<rdf:Description rdf:about="http://ncct.org/src#FirstName">
        <ns1:isNillable rdf:resource="http://www.w3.org/2001/XMLSchema#false"/>
        <ns1:hasParent rdf:resource="http://ncct.org/src#false"/>
        <rdf:type rdf:resource="http://www.w3.org/2001/XMLSchema#element"/>
        <ns1:parentName rdf:resource="http://ncct.org/src#schema_12273995"/>
        <ns1:hasParent rdf:resource="http://ncct.org/src#true"/>
        <ns1:parentName rdf:resource="http://ncct.org/src#particle_23293518"/>
        <ns1:matchedWith rdf:resource="http://ncct.org/trg#Forename"/>
        <ns1:childName
rdf:resource="http://ncct.org/src#ComplexType_of_FirstName"/>
</rdf:Description>
```

```
<rdf:Description rdf:about="http://ncct.org/trg#Forename">
        <ns4:isNillable rdf:resource="http://www.w3.org/2001/XMLSchema#false"/>
        <ns4:hasParent rdf:resource="http://ncct.org/trg#false"/>
        <rdf:type rdf:resource="http://www.w3.org/2001/XMLSchema#element"/>
        <ns4:parentName rdf:resource="http://ncct.org/trg#schema_6427893"/>
        <ns4:hasParent rdf:resource="http://ncct.org/trg#true"/>
        <ns4:parentName rdf:resource="http://ncct.org/trg#particle_28073747"/>
        <ns4:matchedWith rdf:resource="http://ncct.org/src#FirstName"/>
        <ns4:childName
rdf:resource="http://ncct.org/trg#ComplexType_of_Forename"/>
</rdf:Description>
```

Figure 4.36 RDF representation of two element declarations that match

The only difference between (1) an RDF representation of an element declaration and (2) an RDF representation of an element declaration that has a match is the additional predicate – *matchedWith*. This predicate has an object that points to a resource which contains the matching XML Schema component.

The **Knowledge** class represents the link between the repository and two other important components of Hermes: the Mapping and the Transformation Engine. This class has only one attribute *persistSesame* which is an object instance of **PersistenceSesame** class.

The Knowledge class contains the following public methods:

- **getMatchedElements** – this method returns a hash map of matched elements from source and target XML Schema documents. Its input parameter (*whichSchema*) selects the source or target XML Schema (0: source schema and 1: target schema). The SeRQL queries that extracts all matched elements from repository look like this:

```
select localName(x), localName(y) from {x} <http://ncct.org/src#matchedWith>
{y} where x in (select * from {b} <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> {<http://www.w3.org/2001/XMLSchema#element>})
```

a.) Extract matched elements from the source XML Schema

```
select localName(x), localName(y) from {x} <http://ncct.org/trg#matchedWith>
{y} where x in (select * from {b} <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> {<http://www.w3.org/2001/XMLSchema#element>})
```

b.) Extract matched elements from the target XML Schema

Figure 4.37 getMatchedElements method

The first SeRQL [47] query returns a hash map where the keys are elements from the source XML Schema and the values are elements from the target XML Schema. The second query returns a hash map where the keys are elements from the target XML Schema and values are elements from the source XML schema. This method has been implemented in this way in order to take advantage of the repository symmetry. It makes feasible the generation of both transformation scripts. One script transforms XML data that validates against source schema into XML data that validates against the target XML schema. The second script transforms XML data that validates against the target XML Schema into XML data that validates against the source XML Schema document.

- **getMatchedAttribs** – this method returns a hash map of matched attributes from source and target XML Schema documents. The SeRQL query that extracts all matched attributes from repository looks like this:

```
select localName(x), localName(y) from {x} <http://ncct.org/src#matchedWith>
{y} where x in (select * from {b} <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> {<http://www.w3.org/2001/XMLSchema#attribute>})
```

a.) Extract matched attributes from source XML Schema

```
select localName(x), localName(y) from {x} <http://ncct.org/trg#matchedWith>
{y} where x in (select * from {b} <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> {<http://www.w3.org/2001/XMLSchema#attribute>})
```

b.) Extract matched attributes from target XML Schema

Figure 4.38 getMatchedAttributes method

The SeRQL [47] statement presented in Figure 4.25 a.) extracts all attributes from source XML Schema with their matching attributes from target XML Schema in the form of a two columns array. Source XML Schema attributes are listed in the first column and target XML Schema attributes are in the second column.

Similarly, the methods **getMatchedSimpleTypes** and **getMatchedComplexTypes** extract matched simple types and matched complex types from the repository, respectively. The only difference is that the RDF type in the correspondents SeRQL statements – "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"– is queried for "<http://www.w3.org/2001/XMLSchema#**simpleType**>" and "http://www.w3.org/2001/XMLSchema#**complexType**", respectively.

- *getUnMatchedElemsAttribs* – takes as input an integer parameter (0 indicating source XML Schema and 1 for target XML Schema) and returns a hash map containing the following key – value pairs:

- keys are represented by all un-matched elements from source schema – if its input parameter is 0 – or un-matched elements from target schema if its input parameter is 1
- values are represented by lists of attributes that belong to each element

The first step of its algorithm consists in gathering all un-matched elements using the following SeRQL [47] statements:

```
select localName(b) from {b} <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> {<http://www.w3.org/2001/XMLSchema#element>} where
namespace(b) like "http://ncct.org/src#" minus select localName(x) from {x}
<http://ncct.org/src#matchedWith> {y} where x in (select * from {b}
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
{<http://www.w3.org/2001/XMLSchema#element>})
```

a.) Extract un-matched elements from source XML Schema

```
select localName(b) from {b} <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> {<http://www.w3.org/2001/XMLSchema#element>} where
namespace(b) like "http://ncct.org/trg#" minus select localName(x) from {x}
<http://ncct.org/trg#matchedWith> {y} where x in (select * from {b}
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
{<http://www.w3.org/2001/XMLSchema#element>})
```

b.) Extract un-matched elements from target XML Schema

Figure 4.39 Extract un-matched element declarations

The SeRQL statements presented in Figure 4.39 return a list of un-matched elements from either source XML Schema or target XML Schema. Then, for each XML element its attributes are retrieved in two steps. Step 1 retrieves attribute use XML Schema components. Step 2 constructs a list of attributes for each attribute use component.

```
select y1 from {x1} p1 {y1} where x1 IN (select y from {x} p {y} where
localName(x) like "elementName" and namespace(x) like "http://ncct.org/src#"
and localName(p) like "childName") and localName(y1) like "attributeuse_*"
```

a.) Extract attribute use components

```
select localName(y) from {<http://ncct.org/src#attributeuse_nnn>}
<http://ncct.org/src#childName> {y}
```

b.) Extract attribute declarations

Figure 4.40 Extract attributes of un-matched element declarations

The last step of the algorithm consists in building the above mentioned hash map data structure.

- *getUnMatchedElemsAncestors* – takes as an input an integer parameter (0 indicating source XML Schema and 1 for target XML Schema) and returns a hash map with following data:
    - keys are represented by all un-matched elements from source schema – if its input parameter is 0 – or un-matched elements from target schema if its input parameter is 1
    - values are represented by lists of all ancestor elements of each un-matched element

This method – in the first phase – retrieves all un-matched elements using the SeRQL statements described in Figure 4.39. Then for each un-matched element its list of ancestors is constructed using iterative execution of the following SeRQL statements:

*select localName(y) from {x} p {y} where localName(x) like "**elementName**" and namespace(x) like "http://ncct.org/**src**#" and localName(p) like "parentName"*

a.) Parent elements of the source XML Schema un-matched elements

*select localName(y) from {x} p {y} where localName(x) like "**elementName**" and namespace(x) like "http://ncct.org/**trg**#" and localName(p) like "parentName"*

b.) Parent elements of the target XML Schema un-matched elements

Figure 4.41 Retrieve parent elements

The "*elementName*" represents a name of an element encountered during this iterative process. This process ends when there is no element returned as a parent.

The last phase of the algorithm consists in constructing the above mentioned hash map data structure.

- *getUnMatchedElemsDescends* – takes as input an integer parameter (0 indicating source XML Schema and 1 for target XML Schema) and returns a hash map with the following data:
    - keys are represented by all un-matched elements from source schema – if its input parameter is 0 – or un-matched elements from target schema if its input parameter is 1
    - values are represented by lists of all descendent elements of each un-matched element

This method – in the first phase – retrieves all un-matched elements using SeRQL statements described in Figure 4.39. Then for each un-matched element its list of its descendents is constructed, using iterative execution of the following SeRQL statements:

*select localName(y) from {x} p {y} where localName(x) like "**elementName**" and namespace(x) like "http://ncct.org/**src**#" and localName(p) like "childName" and not localName(y) like "***elementName**" and not (localname(y) like "***attribute***")*

a.) Retrieve child elements of un-matched elements from the source XML Schema

*select localName(y) from {x} p {y} where localName(x) like "**elementName**" and namespace(x) like "http://ncct.org/**trg**#" and localName(p) like "childName" and not localName(y) like "***elementName**" and not (localname(y) like "***attribute***")*

b.) Retrieve child elements of un-matched elements from the target XML Schema

Figure 4.42 Retrieve child elements of un-matched elements

The "*elementName*" represents the name of an element encountered during this iterative process. However, an attribute of an element represents a child for that element and the purpose of this method is to retrieve only elements – i.e. similar to an XPath construct. Therefore all attributes must be eliminated from the current query as well as all anonymous type definitions like "ComplexType_of_elementName" or "SimpleType_of_elementName" – this is the explanation for "*elementName" and "*attribute*" constructs. The iterative process ends when there is no element returned as a child.

The last phase of the algorithm consists in constructing the above mentioned hash map data structure.

- findRootElement – takes as an input an integer parameter (0 for source XML Schema and 1 for target XML Schema) and it finds the element name in the corresponding XML Schema which is supposed to be the root element of the data instance that is validated by that schema.

```
select localName(x) from {x} <http://ncct.org/src#hasParent> {y} where y like
"*false" and localName(x) in (select localName(b) from {b}
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
{<http://www.w3.org/2001/XMLSchema#element>}) minus select localName(x)
from {x} <http://ncct.org/src#hasParent> {y} where y like "*true" and
localName(x) in (select localName(b) from {b} <http://www.w3.org/1999/02/22-
rdf-syntax-ns#type> {<http://www.w3.org/2001/XMLSchema#element>})
```

a.) SeRQL statement for finding root element in the source XML Schema

```
select localName(x) from {x} <http://ncct.org/trg#hasParent> {y} where y like
"*false" and localName(x) in (select localName(b) from {b}
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
{<http://www.w3.org/2001/XMLSchema#element>}) minus select localName(x)
from {x} <http://ncct.org/trg#hasParent> {y} where y like "*true" and
localName(x) in (select localName(b) from {b} <http://www.w3.org/1999/02/22-
rdf-syntax-ns#type> {<http://www.w3.org/2001/XMLSchema#element>})
```

b.) SeRQL statement for finding root element in the target XML Schema

Figure 4.43 SeRQL statement for finding root element

This method is very important for the Transformation Engine component. An XML Schema document contains element declarations, attribute declarations, type definitions and relations between them. During the matching process, valuable information (relations between XML Schema components) is extracted from both XML Schema documents subject of the matching process. Therefore, using SeRQL statements, the root element of an XML data instance validated against one of the XML Schema documents can be found, for the reason that all relations between XML Schema components are available in the repository. This method may not always find the root element. For instance, if an XML Schema document contains only global element declarations with no relations between them, then any of them could be the root element in a data instance that can be validated against that XML Schema document.

4.6.2 XML Schema documents mapping

The mapping functionality is based on the **PersistenceSesame** and **Knowledge** classes and is implemented by the **Mapping** class, as described in Figure 4.33 "Hermes Repository and Mapping – UML Class diagram".

The **Mapping** class has two main members: *treeSrcSchema* and *treeTrgSchema*. These members are the SWT tree controls and are the actual visual representation of the two XML Schema documents subject of the matching process. The public method *showConnections()* is responsible for the actual visual representation of the current mapping. The algorithm for the *showConnections()* method is illustrated in Figure 4.44:

```
1:  mtchElements = getMtachedElements()
2:  mtchAttributes = getMatchedAttributes()
3:  mtchSimpleTypes = getMatchedSimpleTypes()
4:  mtchComplexTypes = getMatchedComplexTypes()
5:  for each srcElement in mtchElements:
6:          treeItemSrc = findTreeItem(srcElement, treeSrcSchema)
7:          if not treeItemSrc.isVisible()
8:                  continue
9:          trgElement = mtchElements.get(srcElement)
10:         treeItemTrg = findTreeItem(trgElement, treeTrgSchema)
11:         if not treeItemTrg.isVisible()
12:                 continue
13:         connect(treeItemSrc, treeItemTrg)
14: for each srcAttrib in mtchAttributes:
15:         treeItemSrc = findTreeItem(srcAttrib, treeSrcSchema)
16:         if not treeItemSrc.isVisible()
17:                 continue
18:         trgAttribute = mtchAttributes.get(srcAttribute)
19:         treeItemTrg = findTreeItem(trgAttribute, treeTrgSchema)
20:         if not treeItemTrg.isVisible()
21:                 continue
22:         connect(treeItemSrc, treeItemTrg)
23: for each srcComplexType in mtchComplexTypes:
24:         treeItemSrc = findTreeItem(srcComplexType, treeSrcSchema)
25:         if not treeItemSrc.isVisible()
26:                 continue
27:         trgComplexType = mtchComplexTypes.get(srcComplexType)
28:         treeItemTrg = findTreeItem(trgComplexType, treeTrgSchema)
29:         if not treeItemTrg.isVisible()
30:                 continue
31:         connect(treeItemSrc, treeItemTrg)
32: for each srcSimpleType in mtchSimpleTypes:
33:         treeItemSrc = findTreeItem(srcSimpleType, treeSrcSchema)
34:         if not treeItemSrc.isVisible()
35:                 continue
36:         trgSimpleType = mtchAttributes.get(srcSimpleType)
37:         treeItemTrg = findTreeItem(trgSimpleType, treeTrgSchema)
38:         if not treeItemTrg.isVisible()
39:                 continue
40:         connect(treeItemSrc, treeItemTrg)
```

Figure 4.44 Show Connections algorithm

The first phase of the algorithm retrieves XML Schema components that have been found as matches (i.e. element declarations, attribute declarations, simple type definitions and complex type definitions) lines 1 to 4. Each matched XML Schema component is located in the corresponding tree control. For instance, a source element declaration is located in the source tree control using the *findTreeItem()* method (*treeSrcSchema*) and its matched element declaration from the target XML Schema is located in the target source control (*treeTrgSchema*) using the same *findTreeItem()* with the appropriate parameters. If either one of the XML Schema components (source or target) is not visible on the tree control then that pair of matched XML Schema components is ignored. Connections between matched XML Schema components are only shown for visible tree components. The *connect()* method draws a line between the two tree components that represent matched XML Schema components.

4.7 Transformation Engine - Detailed Design

The result of a matching process between two XML Schema documents is stored in the main repository as RDF statements. Then this repository can be queried using SeRQL [47] statements to extract meaningful information about the matching results of the two XML Schema documents.

The Transformation engine generates two XSLT documents (XML Stylesheet Transformation) [54, 55, and 57] as follows:

- The first XSLT document is used to transform a data instance that validates against the source XML Schema document in a data format that validates against the target XML Schema document.
- The second XSLT document is used to transform a data instance that validates against the target XML Schema document in a data format that validates against the source XML Schema document.

As stated in the "Problem Definition" chapter of this thesis, schema matching is the process of finding correspondences between components of two schemas – source and target. If a correspondence between a component A from source XML Schema document and a component B from target XML Schema document is found, then component A is mapped to component B. A simple inference is that component B from target XML Schema document is mapped to component A from source XML Schema document. This is the reason for generating two transformations: one from source to target and another from target to source. This fact is another innovation in this thesis; the majority of XML Schema matching and transformation tools generate only one transformation, the one from source to target. XML is a standard for data exchange, therefore bidirectional transformations should be implemented in any XML Schema matching and mapping and transformation tool.

The following figure illustrates the algorithm that generates such XSLT documents:

```
1:  Get matched elements data structure
2:  Get matched attributes data structure
3:  Get un-matched elements data structure
4:  Get un-matched attributes data structure
5:  rootSrc = findRootElement("src")
6:  rootTrg = findRootElement("trg")
7:  if rootSrc == null
8:          user picks one from the list of source elements
9:  if rootTrg == null
10:         user picks one from the list of target elements
11: create XSLT document header
12: add XSLT template for matched elements and attributes
13: build the tree down from matched elements
14: add un-matched elements with their dependents
15: serialize the resulting XSLT DOM
```

Figure 4.45 Transformation Engine – algorithm for generating XSLT documents

The first four steps of the algorithm retrieve meaningful information from the repository, like matched and un-matched elements and attributes along with relationships between them (lines 1 to 4). This functionality is implemented in the **Knowledge** class. Lines 5 and 6 try to find automatically if there is a root element, i.e. it can be found from the

XML Schema elements relationships available in the main repository. If a root element can not be found in the repository, then the user must select an element from a list as the root element. After a root element is provided – either automatically from the repository or selected by the user – then the *<xsl:template>* element structure can be constructed.

Line 12 of the algorithm shows that matched elements and attributes are added to the *<xsl:template>* as child elements. In this case of matched elements and attributes of the two XML Schema documents, the information from the source instance document must be propagated to the resulting XML document to be validated against the target XML Schema document. For instance, an attribute value from the source data instance is propagated using the following XSLT statement:

```
<xsl:attribute name="FilePath">
        <xsl:value-of select="./@FilePath"/>
</xsl:attribute>
```

where "FilePath" is an attribute available in the two XML Schema documents being matched.

Line 12 of the algorithm describes the fact that all child elements of matched elements are copied from the source instance to the target instance. If two XML Schema element declarations match, then all their child element declarations must match as well.

In the case of un-matched XML Schema element declarations and/or attribute declarations, there are the following situations:

- The source XML Schema document contains element declarations and/or attribute declarations that have no corresponding declarations in the target XML Schema document, therefore the information hold by these elements must not be propagated to the resulting document in order to be validated by the target XML Schema document;

- The target XML Schema document contains element declarations and/or attribute declarations that have no corresponding declarations in the source XML Schema document. These elements must be created in the resulting document in order to be validated against the target XML Schema document. These elements, created by the XSLT transformation, contain no information. This fact may lead to validation failure in case of complex restrictions for these elements defined in the target XML Schema document. These restrictions can be enumerations and/or identity constraints.

At the end of the algorithm, the resulting XSLT DOM is serialized to a document.

The information regarding the two XML Schema documents compared during the matching process is stored symmetrically in the repository. For instance, if the source XML Schema component A has a correspondence with target XML Schema component B, then the repository will contain statements like "A matchedWith B" and "B matchedWith A". This fact provides the capability of seeing a match from both ends, from the source XML Schema perspective and from the target XML Schema perspective. The next step is to extract information from the repository – lines 1 to 4 in the algorithm – from the target XML Schema perspective. Then the algorithm can continue as described in previous paragraphs. The resulting XSLT document will transform an XML document that validates against the target XML Schema document into an XML document that validates against the source XML Schema document.

5 Experimental Results and Evaluations

This chapter of this thesis presents the experimental results and their evaluations for Hermes – (semi) Automatic XML Schema Matching/Mapping and Transformation Tool along with the presentation of a practical use of a resulting mapping.

5.1 Introduction

The performance evaluation of any XML schema matching tool is based on information retrieval theory [22]. The following three questions arise with respect to a generic evaluation of an information retrieval system – as per [22]:

- Why? – The purpose of an evaluation: the answer to this question as presented by the author of [22] – has social and economic connotations. The evaluation brings to light all the benefits and disadvantages from social and economic perspectives. The social perspective of the answer is based on the willingness of a person to use the system. The economic perspective relates to the amount of work done by the user to obtain the desired results from the system.
- What? – The author of [22] presents six measurable quantities (citing another author) as follows:
    - *Coverage* – the limit to which the system gives relevant solutions
    - *Time lag* – the amount of time necessary for the system to give such relevant solutions
    - *Presentation* – how these relevant solutions are presented to the user
    - *Effort* – the amount of work to be done by a user
    - *Recall* – the ratio between relevant solutions given by the system and all relevant solutions
    - *Precision* – the ratio between all solutions given by the system and all relevant solutions

  The author focuses in [22] on the last two measures: *recall* and *precision*.

- How? – The answer to this question presented in [22] is based on the definition of relevance. All measurable quantities are computed based on relevant solutions given by respective system. "Relevance is a *subjective* notion" [22]: this means that a user with expertise on one domain of activity may consider as relevant some information whereas another user with expertise in another domain of activity may not.

Considering that:

- N – is the number of all items available: all the documents or objects that contain information
- A – is the set of relevant items: all the documents or objects that contain relevant information sought by a user
- B – is the set of retrieved items: all the documents or objects that contain information sought by a user and retrieved by the system

Precision, recall, and fallout of the information retrieval can be defined by the following ratios as per [22]:

$$PRECISION = \frac{|A \cap B|}{|B|} \quad (27)$$

$$RECALL = \frac{|A \cap B|}{|A|} \quad (28)$$

$$FALLOUT = \frac{|\overline{A} \cap B|}{|\overline{A}|} \quad (29)$$

In the above definitions (27), (26) and (29), |.| represents the cardinal number of the set, i.e. total number of elements of respective set.

In addition, author of [22] defines an effectiveness measure using precision and recall measures. This effectiveness measure is defined as follows [22]:

$$E = 1 - \frac{1}{\alpha\left(\frac{1}{P}\right) + (1-\alpha)\frac{1}{R}} \qquad (30)$$

where P is Precision (27) and R is Recall (28); $\alpha \in [0,1]$ weight coefficient which describes the importance of precision and recall measures:

- If precision and recall measures have equal importance in the evaluation then $\alpha = \frac{1}{2}$
- If precision measure has no importance in the system evaluation, then $E \to 1 - R$ with $\alpha \to 0$
- If recall measure has no importance in the system evaluation, then $E \to 1 - P$ with $\alpha \to 1$

5.2 Schema matching tools evaluations

A very good paper on schema matching evaluations is [21] which presents evaluations for several schema matching tools such as: COMA [3, 4], LSD [11], and Cupid [12]. The most important quality measures used in [21] are precision [22], recall [22] and F-measure (derived effectiveness measure) [22], along with overall – a quality measure defined in [15].

The following figure describes the relation between real matches and automatically derived matches:

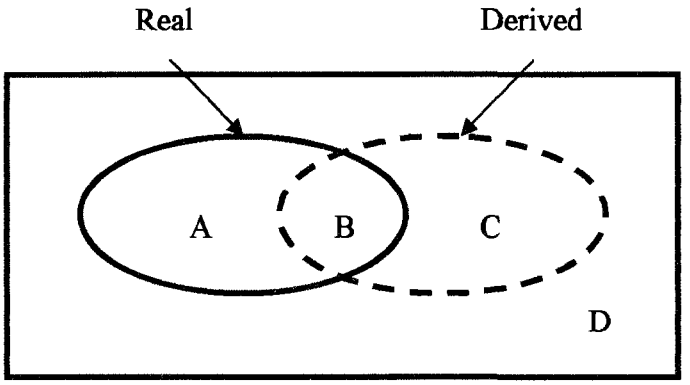


Figure 5.1 Real matches and automatically derived matches [21]

The solid line rectangle represents all source and target schema components; the solid line oval represents the set of real matches and the dashed line oval represent the set of derived matches, i.e. automatically discovered matches. Therefore, A, B, C and D have the following meaning:

- A – false negatives, set of real matches that have not been derived
- B – true positives, set of real matches that have been automatically discovered
- C – false positives, set of automatically discovered matches that are not real matches
- D – true negatives, the set of items that do not match

Using the above defined sets (A, B, C and D) precision (relation (1)) can be defined as follows:

$$precision = \frac{|B|}{|B|+|C|} \quad (31)$$

Consequently, the recall quality measure becomes (from relation (2)):

$$recall = \frac{|B|}{|A|+|B|} \quad (32)$$

F-Measure in [21] is derived from effectiveness measure [22] – i.e. 1 minus effectiveness measure – and defined by the following relation:

$$F-Measure(\alpha)=\frac{|B|}{(1-\alpha)\times|A|+|B|+\alpha\times|C|}=\frac{\Pr ecision\times \mathrm{Re} call}{(1-\alpha)\times \Pr ecision+\alpha\times \mathrm{Re} call} \quad (33)$$

Authors of [21] present the same analysis of values that α may take like the one presented in [22].

- In case of equal importance of precision and recall measures (i.e. $\alpha=\frac{1}{2}$) F-Measure becomes:

$$F-Measure=2\times\frac{\Pr ecision\times \mathrm{Re} call}{\Pr ecision+\mathrm{Re} call} \quad (34)$$

- If recall quality measure has no importance $\alpha\to 1$ then $F-Measure\to \Pr ecision$
- If precision quality measure has no importance $\alpha\to 0$ then $F-Measure\to \mathrm{Re} call$

Another important quality measure presented in [21] is Overall and defined as follows:

$$Overall=1-\frac{|A|+|C|}{|A|+|B|}=\frac{|B|-|C|}{|A|+|B|}=\mathrm{Re} call\times(2-\frac{1}{\Pr ecision}) \quad (35)$$

The overall quality measure described in [21] is called accuracy in [15] and measures the amount of work to be done after the automatic match on the resulting mapping to have a better solution.

If P = $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ is the resulting mapping ("proposed matched result" [15]) and I = $\{(a_1, b_1), \ldots, (a_m, b_m)\}$ is the real (or ideal) mapping ("intended result" [15]) then the accuracy quality measure is defined as follows:

$$Accuracy = 1 - \frac{(n-c)+(m-c)}{m} \quad (36)$$

Where:

- n = |P|
- m = |I|
- $c = |P \cap I|$

(n-c) is the number of false positives that need to be removed from the resulting mapping and (m-c) is the number of false negatives, matches that were not found by the automatic process and need to be added to the resulting mapping. Therefore, the amount of work done by the user correcting the resulting mapping is $\frac{(n-c)+(m-c)}{m}$.

In case of a manual match, the user must create m correspondences between schema components, therefore the less is the work, the higher is the accuracy.

Precision and recall can be presented using c, m and n as follows:

$$precision = \frac{c}{m} \quad (37)$$

$$recall = \frac{c}{n} \quad (38)$$

From (36), (37) and (38), accuracy can be defined using precision and recall quality measures as follows:

$$Accuracy = recall \times (2 - \frac{1}{precision}) \quad (39)$$

The overall [21] (7) and accuracy [15] (11) values describe the same quality measure for the evaluation of an automatic schema matching system as authors of [21] stated in their paper.

To compute precision and recall quality measures, real matches must be known *a priori*; therefore a manual match must be done by a user with expert knowledge of the domain of activity of the two schemas.

5.3 Hermes -- the first set of evaluations

A first set of evaluations are presented in the following sections, using XML Schema documents presented in evaluations of other XML Schema matching tools such as LSD [10, 11] and Clio [7].

5.3.1 Real estate XML Schema documents

The first XML schema documents used in quality evaluation for Hermes – (semi) Automatic XML Schema Matching and Transformation Tool – have been taken from [58] and are also used in [10, 11] for experimental results presentation. Those schemas are presented in DTD format and have been translated in XML Schema Definition Language Format for current evaluation.

The manual match of the two XML Schema documents is illustrated in the following table (matched elements are shown in bold):

RE_I				RE_II				
house_listing				house_listing				
	house_address				basic_info			
	house_description					house_location		
	price						**house_address**	
	bedrooms						neighborhood	
	bathrooms						city	
	lot_area						county	
	garage						suburb	
	school						state	
	mls_number					**house_price**		
	contact_info					contact_info		
		firm_info					agent_info	
			firm_name					**agent_name**
			firm_address					**agent_addres**
			firm_phone					**agent_phone**
		agent_info						**agent_fax**
			agent_name					agent_pager
			agent_address					agent_email
			agent_phone				firm_info	
			agent_fax					**firm_name**
								firm_address
								firm_phone
								firm_fax
								firm_voice_mail
								firm_email
						garage_info		
							garage	
							carport	
						size_info		
							building_area	
							building_dimensions	
							lot_area	
							lot_dimensions	
						rooms		
							basement	
							bath_rooms	
							bed_rooms	
							dining_room	
							living_room	
						schools		
							elementary_school	
							middle_school	
							high_school	
						house_description		
					additional_info			
						utilities		
							cooling	
							heating	
							gas	
							sewer	
							water	
							electricity	
						amenities		
							fireplace	
							patio	
							swimming_pool	
							spa	
							view	
							waterfront	
						mls_num		
						stories		
						type		
						architectural_style		
						date_built		
						age		
						availability		

Table 5-1 Manual match of RE_I and RE_II XML Schema documents

As an observation, "*agent_info*" and "*firm_info*" do not match because they have different structure, even though they have some elements in common. Actually, the content of "*agent_info*" and "*firm_info*" from source XML Schema document (RE_I) is included in the content of "*agent_info*" and "*firm_info*" from target XML Schema document (RE_II).

RE_I	RE_II
agent_address	agent_address
agent_fax	agent_fax
agent_name	agent_name
agent_phone	agent_phone
firm_name	firm_name
firm_phone	firm_phone
garage	garage
house_address	house_address
house_description	house_description
lot_area	lot_area
firm_address	firm_address
bathrooms	bath_rooms
bedrooms	bed_rooms

Table 5-2 Automatic match of RE_I and RE_II XML Schema documents

The automatic match finds thirteen pairs of matched elements and the manual match found fourteen pairs of matched elements. Therefore, precision, recall and F-Measure quality measures are in this case:

$$\text{Precision} = \frac{|B|}{|B|+|C|} = \frac{13}{13} = 1 = 100\% \quad (40)$$

The number of derived matches that are real matches is |B|=13 and the number of false positives is |C|=0 therefore precision is 100% for RE_I and RE_II XML Schema documents.

$$\text{Re}call = \frac{|B|}{|A|+|B|} = \frac{13}{2+13} = \frac{13}{15} = 0.8666 = 86.66\% \quad (41)$$

The number of real matches that were missed (false negatives) is |A| = 2 and the number of derived matches that are real matches is |B|=13. Pairs of elements that were not reported as a match are:

- "*price*" from source XML Schema document and "*house_price*" from target XML Schema document.
- "*mls_number*" from source XML Schema document RE_I and "*mls_num*" from target XML Schema document RE_II

$$Accuracy = \text{Re}call \times (2 - \frac{1}{\text{Pr}ecision}) = 86.66\% \times (2 - \frac{1}{100\%}) = 86.66\% \quad (42)$$

The accuracy obtained in [10, 58] is 77% and (42) shows an improvement of accuracy in case of Hermes using same schemas.

The quality of match between RE_I and RE_II XML Schema documents can be calculated using quality of match measurement defined in [28] and [29] – presented in chapter 2 of this thesis - as follows:

Quality of match between attributes – QoM(a_s,a_t):

QoM(agent_address,agent_address) = 1

QoM(agent_fax,agent_fax) = 1

QoM(agent_name,agent_name) = 1

QoM(agent_phone,agent_phone) = 1

QoM(firm_name,firm_name) = 1

QoM(firm_phone,firm_phone) = 1

QoM(garage,garage) = 1

QoM(house_address,house_address) = 1

QoM(house_description,house_description) = 1

QoM(lot_area,lot_area) = 1

QoM(firm_address,firm_address) = 1

QoM(bathrooms,bath_rooms) = 1

QoM(bedrooms,bed_rooms) = 1

Attributes presented above are the element declaration names.

Quality of match between methods – refers to the type of elements – QoM(m_s,m_t):

QoM(m_agent_address,m_agent_address)=1

QoM(m_agent_fax,m_agent_fax)=1

QoM(m_agent_name,m_agent_name)=1

QoM(m_agent_phone,m_agent_phone)=1

QoM(m_firm_name,m_firm_name)=1

QoM(m_firm_phone,m_firm_phone)=1

QoM(m_garage,m_garage)=1

QoM(m_house_address,m_house_address)=1

QoM(m_house_description,m_house_description)=1

QoM(m_lot_area,m_lot_area)=1

QoM(m_firm_address,m_firm_address)=1

QoM(m_bathrooms,m_bath_rooms)=1

QoM(m_bedrooms,m_bed_rooms)=1

Micro match weights, defined by the following formula

$$R_W(C_s, C_t) = \frac{\sum QoM(a_s, a_t) + \sum QoM(m_s, m_t)}{|C_s|} \quad (43)$$

are:

R_W(class_agent_address, class_agent_address)=1

R_W(class_agent_fax, class_agent_fax)=1

R_W(class_agent_name, class_agent_name)=1

R_W(class_agent_phone, class_agent_phone)=1

R_W(class_firm_name, class_firm_name)=1

R_W(class_firm_phone, class_firm_phone)=1

R_W(class_garage, class_garage)=1

R_W(class_house_address, class_house_address)=1

R_W(class_house_description, class_house_description)=1

R_W(class_lot_area, class_lot_area)=1

R_W(class_firm_address, class_firm_address)=1

R_W(class_bathrooms, class_bath_rooms)=1

R_W(class_bedrooms, class_bed_rooms)=1

Each source class has one attribute and one method, therefore its cardinality is 2.

Cardinality ratio is given by the following formula – presented in chapter 2 of this thesis:

$$R_S(C_s, C_t) = \frac{|C_s^m|}{|C_s|} \quad (44)$$

All attributes and methods of each source class have a match in a target class, therefore:

R_S(class_agent_address,class_agent_address)=1

R_S(class_agent_fax,class_agent_fax)=1

R_S(class_agent_name,class_agent_name)=1

R_S(class_agent_phone,class_agent_phone)=1

R_S(class_firm_name,class_firm_name)=1

R_S(class_firm_phone,class_firm_phone)=1

R_S(class_garage,class_garage)=1

R_S(class_house_address,class_house_address)=1

R_S(class_house_description,class_house_description)=1

R_S(class_lot_area,class_lot_area)=1

R_S(class_firm_address,class_firm_address)=1

R_S(class_bathrooms,class_bath_rooms)=1

R_S(class_bedrooms,class_bed_rooms)=1

Congruity ratio is given by the following formula presented in chapter 2 of this thesis:

$$R_T(C_s, C_t) = \frac{|C_s^m|}{|C_t|} \quad (45)$$

Target class cardinality is the same as cardinality of matched properties in the source class, therefore:

R_T(class_agent_address,class_agent_address)=1

R_T(class_agent_fax,class_agent_fax)=1

R_T(class_agent_name,class_agent_name)=1

R_T(class_agent_phone,class_agent_phone)=1

R_T(class_firm_name,class_firm_name)=1

R_T(class_firm_phone,class_firm_phone)=1

R_T(class_garage,class_garage)=1

R_T(class_house_address,class_house_address)=1

R_T(class_house_description,class_house_description)=1

R_T(class_lot_area,class_lot_area)=1

R_T(class_firm_address,class_firm_address)=1

R_T(class_bathrooms,class_bath_rooms)=1

R_T(class_bedrooms,class_bed_rooms)=1

Sub macro quality of match is quantified by the following formula:

$$QoM(C_s, C_t) = \frac{R_W(C_s, C_t) + R_S(C_s, C_t) + R_T(C_s, C_t)}{3} \quad (46)$$

QoM(class_agent_address,class_agent_address)=1

QoM(class_agent_fax,class_agent_fax)=1

QoM(class_agent_name,class_agent_name)=1

QoM(class_agent_phone,class_agent_phone)=1

QoM(class_firm_name,class_firm_name)=1

QoM(class_firm_phone,class_firm_phone)=1

QoM(class_garage,class_garage)=1

QoM(class_house_address,class_house_address)=1

QoM(class_house_description,class_house_description)=1

QoM(class_lot_area,class_lot_area)=1

QoM(class_firm_address,class_firm_address)=1

QoM(class_bathrooms,class_bath_rooms)=1

QoM(class_bedrooms,class_bed_rooms)=1

Sub macro match weight is given by the following formula:

$$R_W(S_s,S_t)=\frac{\sum QoM(C_s,C_t)}{|S_s|} \quad (47)$$

$$R_W(RE\_I,RE\_II)=\frac{13}{20}=0.65=65\% \quad (48)$$

Source schema (RE_II) has 20 classes, i.e. element declarations, and only 13 were matched in the target schema (RE_II).

Cardinality ratio is given by the following formula in chapter 2 of this thesis:

$$R_S(S_s,S_t)=\frac{|S_s^m|}{|S_s|} \quad (49)$$

$$R_S(RE\_I,RE\_II)=\frac{13}{20}=0.65=65\% \quad (50)$$

Source schema (RE_II) has 20 classes, i.e. element declarations, and only 13 sub macro matches.

Congruity ratio is given by the following formula in chapter 2 of this thesis:

$$R_T(S_s,S_t)=\frac{|S_s^m|}{|S_t|} \quad (51)$$

Therefore, congruity ratio has following value in case of RE_I and RE_II XML Schema documents:

$$R_T(RE\_I,RE\_II)=\frac{13}{66}=0.196=19.6\% \quad (52)$$

The overall quality of match of the two XML Schema documents (RE_I and RE_II) is given by the following formula in chapter 2 of this thesis:

$$QoM(S_s,S_t)=\frac{R_W(S_s,S_t)+R_S(S_s,S_t)+R_T(S_s,S_t)}{3} \quad (53)$$

$$QoM(RE\_I, RE\_II) = \frac{65\% + 65\% + 19.6\%}{3} = 49.86\% \quad (54)$$

5.3.2 Financial XML Schema documents

This evaluation is based on two XML Schema documents from [59] as a part of Clio [7] schema matching tool evaluation – i.e. Financial (Expense/Statistics DB). The following table illustrates a manual match of the two XML Schema documents, where matched elements are in bold:

statisticsDB					expenseDB		
statisticsDB					expenseDB		
	cityStatistics					company	
		city					**cid**
		organization					**cname**
			cid				**city**
			name			grant	
			funding				grantee
				pi			**pi**
				aid			**amount**
		financial					sponsor
			aid				**proj**
			amount			project	
			proj				**name**
			year				**year**

Table 5-3 Manual match of statisticsDB and expensesDB

The result of the automatic match is shown in the following table:

statisticsDB	expenseDB
proj	proj
pi	pi
year	year
cid	cid
amount	amount
name	name
city	city

Table 5-4 Automatic match of statisticsDB and expensesDB

In this case, quality measures are:

$$Precision = \frac{|B|}{|B|+|C|} = \frac{7}{7+0} = \frac{7}{7} = 1 = 100\% \quad (55)$$

There are no false positives – |C| = 0 – and true positives are |B| = 7, therefore the precision is 100%.

$$Recall = \frac{|B|}{|A|+|B|} = \frac{7}{1+7} = \frac{7}{8} = 0.875 = 87.5\% \quad (56)$$

There is only one match that have been missed, between "*name*" element declaration from "*statisticsDB*" source XML Schema document and "*cname*" element declaration from "*expensesDB*" target XML Schema document; therefore there is only one false positive |A|=1.

$$Accuracy = Recall \times (2 - \frac{1}{Precision}) = 87.5\% \times (2 - \frac{1}{100\%}) = 87.5\% \quad (57)$$

As these quality measures show, Hermes has a very good performance as an XML Schema documents matching tool.

Quality of match between attributes of source and target classes defined by [28] – $QoM(a_s,a_t)$:

QoM(proj,proj)=1

QoM(pi,pi)=1

QoM(year,year)=1

QoM(cid,cid)=1

QoM(amount,amount)=1

QoM(name,name)=1

QoM(city,city)=1

Attributes presented above are the element declaration names.

Quality of match between methods – refers to the type of elements – $QoM(m_s,m_t)$:

QoM(m_proj,m_proj)=1

QoM(m_pi,m_pi)=1

QoM(m_year,m_year)=1

QoM(m_cid,m_cid)=1

QoM(m_amount,m_amount)=1

QoM(m_name,m_name)=1

QoM(m_city,m_city)=1

Micro match weights, defined by the following formula

$$R_W(C_s,C_t)=\frac{\sum QoM(a_s,a_t)+\sum QoM(m_s,m_t)}{|C_s|} \quad (58)$$

are:

R_W(class_proj,class_proj)=1

R_W(class_pi,class_pi)=1

R_W(class_year,class_year)=1

R_W(class_cid,class_cid)=1

R_W(class_amount,class_amount)=1

R_W(class_name,class_name)=1

R_W(class_city,class_city)=1

Each source class has one attribute and one method, therefore its cardinality is 2.

Cardinality ratio is given by the following formula – presented in chapter 2 of this thesis:

$$R_S(C_s, C_t) = \frac{|C_s^m|}{|C_s|} \quad (59)$$

All attributes and methods of each source class have a match in a target class, therefore:

R_S(class_proj,class_proj)=1

R_S(class_pi,class_pi)=1

R_S(class_year,class_year)=1

R_S(class_cid,class_cid)=1

R_S(class_amount,class_amount)=1

R_S(class_name,class_name)=1

R_S(class_city,class_city)=1

Congruity ratio is given by the following formula presented in chapter 2 of this thesis:

$$R_T(C_s, C_t) = \frac{|C_s^m|}{|C_t|} \quad (60)$$

Target class cardinality is the same as cardinality of matched properties in the source class, therefore:

RT(class_proj,class_proj)=1

RT(class_pi,class_pi)=1

RT(class_year,class_year)=1

RT(class_cid,class_cid)=1

RT(class_amount,class_amount)=1

RT(class_name,class_name)=1

RT(class_city,class_city)=1

Sub macro quality of match is quantified by the following formula:

$$QoM(C_s, C_t) = \frac{R_W(C_s, C_t) + R_S(C_s, C_t) + R_T(C_s, C_t)}{3}$$

QoM(class_proj,class_proj)=1

QoM(class_pi,class_pi)=1

QoM(class_year,class_year)=1

QoM(class_cid,class_cid)=1

QoM(class_amount,class_amount)=1

QoM(class_name,class_name)=1

QoM(class_city,class_city)=1

Sub macro match weight is given by the following formula:

$$R_W(S_s, S_t) = \frac{\sum QoM(C_s, C_t)}{|S_s|} \quad (61)$$

$$R_W(statisticsDB, \exp enseDB) = \frac{7}{14} = 0.5 = 50\% \quad (62)$$

Cardinality ratio is given by the following formula in chapter 2 of this thesis:

$$R_S(S_s, S_t) = \frac{|S_s^m|}{|S_s|} \quad (63)$$

$$R_S(statisticsDB, \exp enseDB) = \frac{7}{14} = 0.5 = 50\% \quad (64)$$

Congruity ratio is given by the following formula in chapter 2 of this thesis:

$$R_T(S_s, S_t) = \frac{|S_s^m|}{|S_t|} \quad (65)$$

Therefore, congruity ratio has following value in case of *statisticsDB* and *expenseDB* XML Schema documents:

$$R_T(statisticsDB, \exp enseDB) = \frac{7}{14} = 0.5 = 50\% \quad (66)$$

The overall quality of match of the two XML Schema documents (*statisticsDB* and *expenseDB*) is given by the following formula in chapter 2 of this thesis:

$$QoM(S_s, S_t) = \frac{R_W(S_s, S_t) + R_S(S_s, S_t) + R_T(S_s, S_t)}{3} \quad (67)$$

$$QoM(statisticsDB, \exp enseDB) = \frac{50\% + 50\% + 50\%}{3} = 50\% \quad (68)$$

5.3.3 DBLP – Computer Science Bibliography

In this evaluation, two XML Schema documents describing information about publications are matched. The following table presents the manual match between *DBLP.xsd* and *targetDBLP.xsd* with matched elements in bold:

DBLP.xsd	targetDBLP.xsd
article	author
author	authorDB
book	**cdrom**
booktitle	conf_jour
cdrom	dateCreated
cite	name
dblp	**pages**
editor	pub
ee	pub_id
inproceedings	**title**
isbn	**url**
journal	year
mastersthesis	yr
month	
number	
pages	
phdthesis	
proceedings	
publisher	
school	
series	
title	
url	
volume	
www	
year	

Table 5-5 Manual match of DBLP.xsd and targetDBLP.xsd

There are two pairs of element declaration that do not match because of their structure, even though they have the same name in both XML Schema documents: "*author*" and "year":

```
<xs:element name="author" type="xs:string"/>
...
<xs:element name="year" type="xs:string"/>
```

Figure 5.2 "author" and "year" element declarations from *DBLP.xsd*

```
<xs:element name="author">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="name"/>
                        <xs:element ref="conf_jour" minOccurs="0"
maxOccurs="unbounded"/>
                        <xs:element ref="year" minOccurs="0"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:key name="tk1">
                <xs:selector xpath="./conf_jour/year/pub"/>
                <xs:field xpath="pub_id"/>
        </xs:key>
        <xs:keyref name="fk1" refer="tk1">
                <xs:selector xpath="./year/conf_jour/pub"/>
                <xs:field xpath="pub_id"/>
        </xs:keyref>
</xs:element>
...
<xs:element name="year">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="yr"/>
                        <xs:element ref="pub" minOccurs="0"
maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

Figure 5.3 "author" and "year" element declarations from *targetDBLP.xsd*

The following table illustrates the result of the automatic match of the two XML Schema documents - *DBLP.xsd* and *targetDBLP.xsd*:

DBLP.xsd	targetDBLP.xsd
cdrom	cdrom
pages	pages
title	title
url	url

Table 5-6 Automatic match of *DBLP.xsd* and *targetDBLP.xsd*

Following quality measures are obtained from this experiment using *DBLP.xsd* and *targetDBLP.xsd* XML Schema documents:

$$\text{Precision} = \frac{|B|}{|B|+|C|} = \frac{4}{4+0} = \frac{4}{4} = 1 = 100\% \quad (69)$$

There are no false positives reported by the automatic match process (|C|=0) and the number of true positives is |B|=4

$$\text{Recall} = \frac{|B|}{|A|+|B|} = \frac{4}{0+4} = \frac{4}{4} = 1 = 100\% \quad (70)$$

There are no false negatives |A|=0 – i.e. no matching element declarations have been missed – and the number of true positives is |B|=4

$$Accuracy = \text{Recall} \times (2 - \frac{1}{\text{Precision}}) = 100\% \times (2 - \frac{1}{100\%}) = 100\% \quad (71)$$

This example illustrates that Hermes can do the ideal match – precision and recall of 100% – for XML Schema documents from the same domain of activity.

Quality of match between attributes of source and target classes defined by [28] – $QoM(a_s,a_t)$:

QoM(cdrom,cdrom)=1

QoM(pages,pages)=1

QoM(title,title)=1

QoM(url,url)=1

Attributes presented above are the element declaration names.

Quality of match between methods – refers to the type of elements – $QoM(m_s,m_t)$:

QoM(m_cdrom,m_cdrom)=1

QoM(m_pages,m_pages)=1

QoM(m_title,m_title)=1

QoM(m_url,m_url)=1

Micro match weights, defined by the following formula

$$R_W(C_s,C_t)=\frac{\sum QoM(a_s,a_t)+\sum QoM(m_s,m_t)}{|C_s|} \quad (72)$$

are:

R_W(class_cdrom,class_cdrom)=1

R_W(class_pages,class_pages)=1

R_W(class_title,class_title)=1

R_W(class_url,class_url)=1

Cardinality ratio is given by the following formula – presented in chapter 2 of this thesis:

$$R_S(C_s,C_t)=\frac{|C_s^m|}{|C_s|} \quad (73)$$

All attributes and methods of each source class have a match in a target class, therefore:

R_S(class_cdrom,class_cdrom)=1

R_S(class_pages,class_pages)=1

R_S(class_title,class_title)=1

R_S(class_url,class_url)=1

Congruity ratio is given by the following formula presented in chapter 2 of this thesis:

$$R_T(C_s, C_t) = \frac{|C_s^m|}{|C_t|} \quad (74)$$

Target class cardinality is the same as cardinality of matched properties in the source class, therefore:

R_T(class_cdrom,class_cdrom)=1

R_T(class_pages,class_pages)=1

R_T(class_title,class_title)=1

R_T(class_url,class_url)=1

Sub macro quality of match is quantified by the following formula:

$$QoM(C_s, C_t) = \frac{R_W(C_s, C_t) + R_S(C_s, C_t) + R_T(C_s, C_t)}{3} \quad (75)$$

QoM(class_cdrom,class_cdrom)=1

QoM(class_pages,class_pages)=1

QoM(class_title,class_title)=1

QoM(class_url,class_url)=1

Sub macro match weight is given by the following formula:

$$R_W(S_s, S_t) = \frac{\sum QoM(C_s, C_t)}{|S_s|} \quad (76)$$

$$R_W(DBLP, target DBLP) = \frac{4}{26} = 0.153 = 15.3\% \quad (77)$$

Cardinality ratio is given by the following formula in chapter 2 of this thesis:

$$R_S(S_s, S_t) = \frac{|S_s^m|}{|S_s|} \quad (78)$$

$$R_S(DBLP, target DBLP) = \frac{4}{26} = 0.153 = 15.3\% \quad (79)$$

Congruity ratio is given by the following formula in chapter 2 of this thesis:

$$R_T(S_s, S_t) = \frac{|S_s^m|}{|S_t|} \quad (80)$$

Therefore, congruity ratio has following value in case of *DBLP.xsd* and *targetDBLP.xsd* XML Schema documents:

$$R_T(DBLP, target DBLP) = \frac{4}{20} = 0.2 = 20\% \quad (81)$$

The overall quality of match of the two XML Schema documents (*DBLP* and *targetDBLP*) is given by the following formula in chapter 2 of this thesis:

$$QoM(S_s, S_t) = \frac{R_W(S_s, S_t) + R_S(S_s, S_t) + R_T(S_s, S_t)}{3} \quad (82)$$

$$QoM(DBLP, target DBLP) = \frac{15.3\% + 15.3\% + 20\%}{3} = 16.86\% \quad (83)$$

5.3.4 Purchase orders XML Schema documents

In this evaluation, two purchase order XML Schema documents are matched. The following table presents the manual match between *TPC-H.xsd* and *TPC-H-nested.xsd* with matched elements in bold:

TPC-H.xsd	TPC-H-nested.xsd
addr	
availqty	
addr	
brand	
	cnation
	customers
	custID
customer	*customer*
custkey	
date	
lno	
lineitem	
mfgr	
name	**name**
nation	*nation*
name	
nationkey	
name	
nationkey	
orderkey	**orderkey**

order	*order*
ph	
price	
partsupp	
part	part
partkey	
prc	
partkey	
qty	
region	*region*
regionkey	
retail	
supplier	*supplier*
	suppliers
suppkey	
size	
status	**status**
suppkey	
tpcDB	
	tpcNestedDB

Table 5-7 Manual match of *TPC-H.xsd* and *TPC-H-nested.xsd*

The following elements: *customer*, *nation*, *order*, *region* and *supplier* do not match because they have different structures – i.e. different child elements.

The result of the automatic match of the two XML Schema documents – *TPC-H.xsd* and *TPC-H-nested.xsd* is illustrated in the following table:

TPC-H.xsd	**TPC-H-nested.xsd**
orderkey	orderkey
status	status
name	name

Table 5-8 Automatic match of *TPC-H.xsd* and *TPC-H-nested.xsd*

Quality measures for this evaluation are:

$$\text{Precision} = \frac{|B|}{|B|+|C|} = \frac{3}{3+0} = \frac{3}{3} = 1 = 100\% \quad (84)$$

$$\text{Re}call = \frac{|B|}{|A|+|B|} = \frac{3}{0+3} = \frac{3}{3} = 1 = 100\% \quad (85)$$

$$Accuracy = \text{Re}call \times (2 - \frac{1}{\text{Pr}ecision}) = 100\% \times (2 - \frac{1}{100\%}) = 100\% \quad (86)$$

Quality of match between attributes of source and target classes defined by [28] – $QoM(a_s,a_t)$:

QoM(orderkey,orderkey)=1

QoM(status,status)=1

QoM(name,name)=1

Quality of match between methods – refers to the type of elements – $QoM(m_s,m_t)$:

QoM(m_orderkey,m_orderkey)=1

QoM(m_status,m_status)=1

QoM(m_name,m_name)=1

Micro match weights, defined by the following formula

$$R_W(C_s,C_t) = \frac{\sum QoM(a_s,a_t) + \sum QoM(m_s,m_t)}{|C_s|} \quad (87)$$

are:

R_W(class_orderkey,class_orderkey)=1

R_W(class_status,class_status)=1

R_W(class_name,class_name)=1

Cardinality ratio is given by the following formula – presented in chapter 2 of this thesis:

$$R_S(C_s,C_t) = \frac{|C_s^m|}{|C_s|} \quad (88)$$

All attributes and methods of each source class have a match in a target class, therefore:

RS(class_orderkey,class_orderkey)=1

RS(class_status,class_status)=1

RS(class_name,class_name)=1

Congruity ratio is given by the following formula presented in chapter 2 of this thesis:

$$R_T(C_s, C_t) = \frac{|C_s^m|}{|C_t|} \quad (89)$$

Target class cardinality is the same as cardinality of matched properties in the source class, therefore:

RT(class_orderkey,class_orderkey)=1

RT(class_status,class_status)=1

RT(class_name,class_name)=1

Sub macro quality of match is quantified by the following formula:

$$QoM(C_s, C_t) = \frac{R_W(C_s, C_t) + R_S(C_s, C_t) + R_T(C_s, C_t)}{3} \quad (90)$$

QoM(class_orderkey,class_orderkey)=1

QoM(class_status,class_status)=1

QoM(class_name,class_name)=1

Sub macro match weight is given by the following formula:

$$R_W(S_s, S_t) = \frac{\sum QoM(C_s, C_t)}{|S_s|} \quad (91)$$

$$R_W(TPC-H, TPC-H-nested) = \frac{3}{43} = 0.069 = 6.9\% \quad (92)$$

Cardinality ratio is given by the following formula in chapter 2 of this thesis:

$$R_S(S_s, S_t) = \frac{|S_s^m|}{|S_s|} \quad (93)$$

$$R_S(TPC-H, TPC-H-nested) = \frac{3}{43} = 0.069 = 6.9\% \quad (94)$$

Congruity ratio is given by the following formula in chapter 2 of this thesis:

$$R_T(S_s, S_t) = \frac{|S_s^m|}{|S_t|} \quad (95)$$

Therefore, congruity ratio has following value in case of *TPC-H.xsd* and *TPC-H-nested.xsd* XML Schema documents:

$$R_T(TPC-H, TPC-H-nested) = \frac{3}{17} = 0.176 = 17.6\% \quad (96)$$

The overall quality of match of the two XML Schema documents (*TPC-H* and *TPC-H-nested*) is given by the following formula in chapter 2 of this thesis:

$$QoM(S_s, S_t) = \frac{R_W(S_s, S_t) + R_S(S_s, S_t) + R_T(S_s, S_t)}{3} \quad (97)$$

$$QoM(TPC-H, TPC-H-nested) = \frac{6.9\% + 6.9\% + 17\%}{3} = 10.26\% \quad (98)$$

5.3.5 Summary of quality measures

Results obtained from previous evaluations are summarized in following table:

XML Schema documents	Precision	Recall	Accuracy	QoM
RE_I/RE_II	100%	86.66%	86.66%	49.86%
StatisticsDB/ExpenseDB	100%	87.50%	87.50%	50.00%
DBLP.xsd/targetDBLP.xsd	100%	100%	100%	16.86%
TPC-H.xsd/TPC-H-nested.xsd	100%	100%	100%	10.26%

Table 5-9 Summary of quality measures

The QoM measure gives information about the XML Schema matching tool performance in conjunction with the XML Schema documents real similarity, i.e. if XML Schema documents have many other components that do not match then this fact is taken into consideration by QoM..

A chart with these results is illustrated in Figure 5.4:

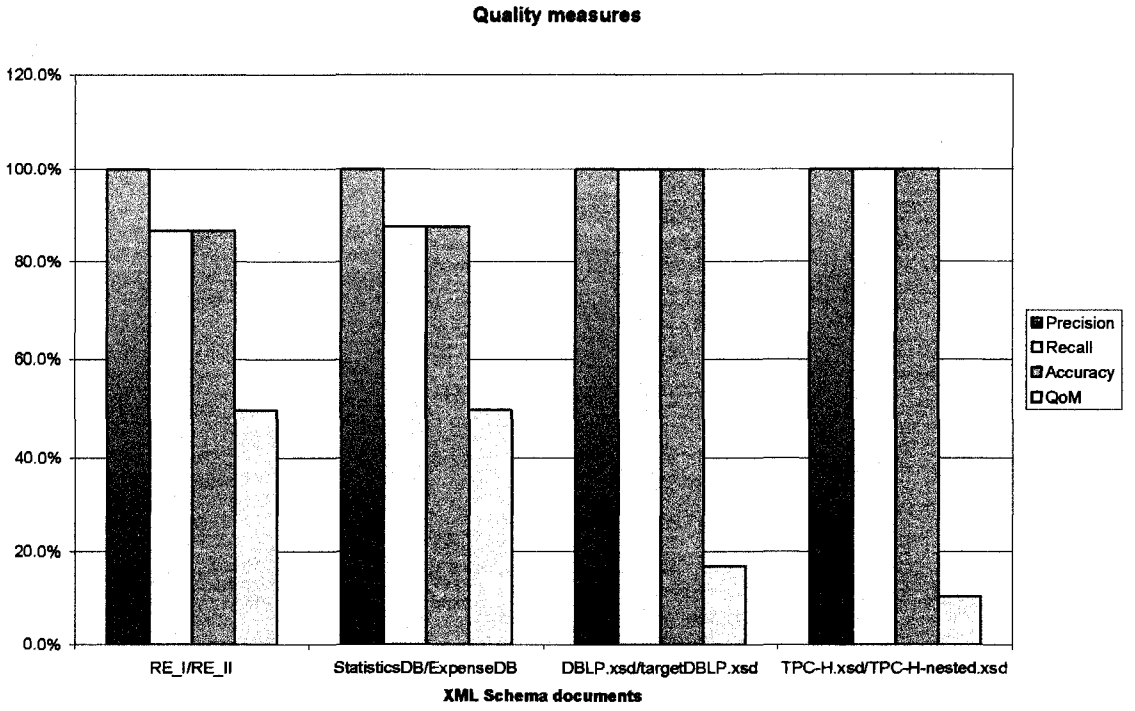


Figure 5.4 Quality Measures – Precision, Recall, Accuracy and Quality of Match

From precision quality measure point of view, Hermes has an excellent performance (100%) for all XML Schema documents, as shown in the above chart. For precision quality measure of 100%, the accuracy quality measure is given by the recall quality measure as per (11).

5.4 Hermes and complex XML Schema documents

The XML schema internal representation within Hermes – (semi) Automatic XML Schema Matching and Transformation Tool – is the main novelty of this thesis. Using this representation any XML Schema – no matter its complexity – is properly loaded and handled. For instance, COMA++ [3, 4] does not load in a proper way XML Schema documents that are very complex and include or import other XML Schema documents; these included or imported XML Schema documents are ignored. In this case, there is no doubt that a match between such complex XML Schema documents leads to unreliable

results. This evaluation takes into consideration the following two XML Schema documents:

a.)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
        <xs:include schemaLocation="dateTypes.xsd"/>
        <xs:include schemaLocation="nameTypes.xsd"/>
        <xs:include schemaLocation="records.xsd"/>
        <xs:element name="FirstName" type="NameType"/>
        <xs:element name="LastName" type="NameType"/>
        <xs:complexType name="PersonType">
                <xs:all>
                        <xs:element ref="FirstName"/>
                        <xs:element ref="LastName"/>
                        <xs:element ref="DrivingRecords"/>
                </xs:all>
                <xs:attribute ref="BirthDate" use="optional"/>
        </xs:complexType>
        <xs:attribute name="BirthDate" type="DateType"/>
        <xs:element name="Person" type="PersonType"/>
</xs:schema>
```

b.)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
        <xs:include schemaLocation="dateTypes.xsd"/>
        <xs:include schemaLocation="nameTypes.xsd"/>
        <xs:include schemaLocation="records.xsd"/>
        <xs:simpleType name="NumericType">
                <xs:restriction base="xs:integer"/>
        </xs:simpleType>
        <xs:element name="Forename" type="NameType"/>
        <xs:element name="Surname" type="NameType"/>
        <xs:complexType name="PersonType">
                <xs:all>
                        <xs:element ref="Forename"/>
                        <xs:element ref="Surname"/>
                        <xs:element ref="DrivingRecords"/>
                </xs:all>
                <xs:attribute ref="BirthDate" use="optional"/>
                <xs:attribute ref="Age" use="optional"/>
        </xs:complexType>
        <xs:attribute name="BirthDate" type="DateType"/>
        <xs:attribute name="Age" type="NumericType"/>
        <xs:element name="PersonInstance" type="PersonType"/>
</xs:schema>
```

Figure 5.5 XML Schema documents: a.) source.xsd b.) target.xsd

The above illustrated XML Schema documents (source and target) include the following additional XML Schema documents (dateTypes.xsd, nameTypes.xsd and records.xsd):

a.)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
        <xs:simpleType name="DateType">
                <xs:restriction base="xs:date"/>
        </xs:simpleType>
</xs:schema>
```

b.)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
        <xs:complexType name="NameType">
                <xs:attribute name="Name" use="required">
                        <xs:simpleType>
                                <xs:restriction base="xs:string"/>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
</xs:schema>
```

c.)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
        <xs:include schemaLocation="dateTypes.xsd"/>
        <xs:element name="DrivingRecords">
                <xs:complexType>
                        <xs:all minOccurs="0">
                                <xs:element ref="DrivingRecord"/>
                        </xs:all>
                </xs:complexType>
        </xs:element>
        <xs:element name="DrivingRecord">
                <xs:complexType>
                        <xs:attribute name="Name" use="optional"/>
                        <xs:attribute name="Date" type="DateType" use="optional"/>
                        <xs:attribute name="Location" use="optional">
                                <xs:simpleType>
                                        <xs:restriction base="xs:string"/>
                                </xs:simpleType>
                        </xs:attribute>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

Figure 5.6 Additional XML Schema documents: a.) dateTypes.xsd b.)nameTypes.xsd c.) records.xsd

Figure 5.7 illustrates the result of the matching process between *source.xsd* and *target.xsd*:

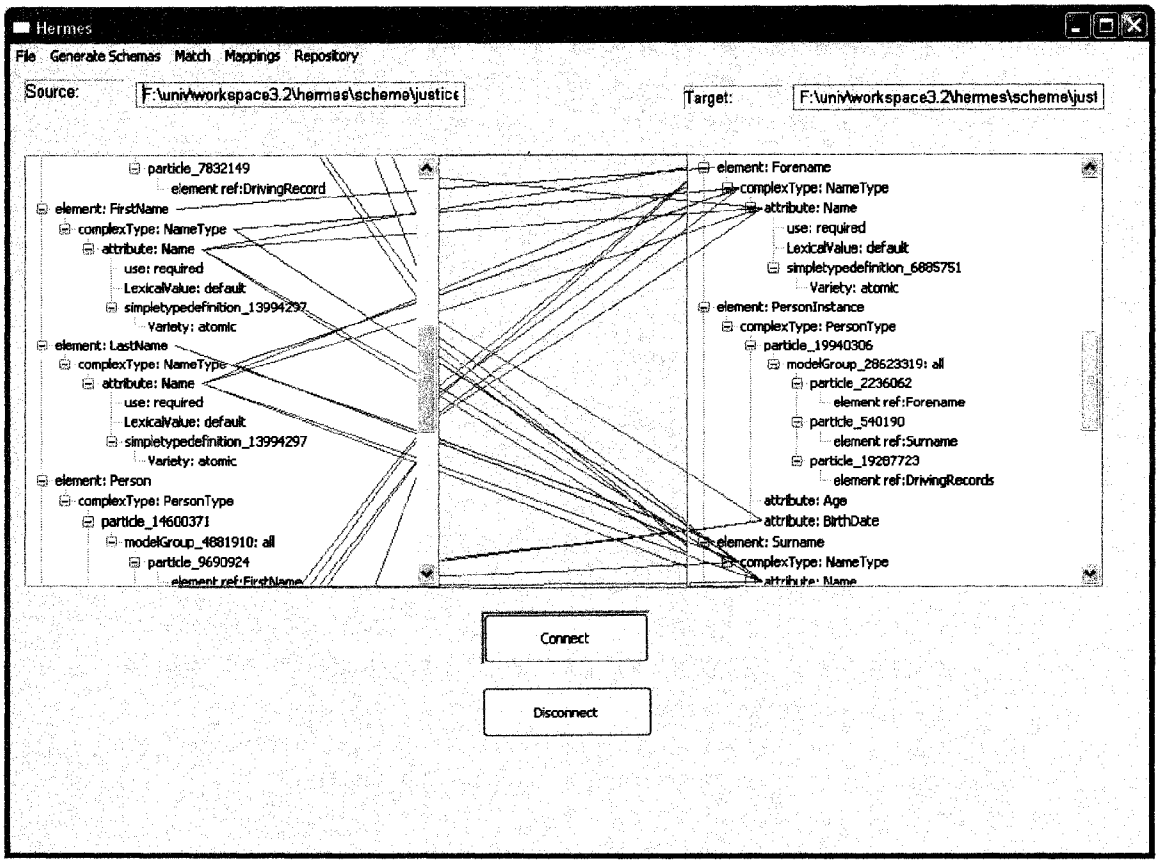


Figure 5.7 Hermes screenshot - mapping between *source.xsd* and *target.xsd*

The result of the automatic match is illustrated in the following table:

source.xsd	target.xsd
FirstName	Forename
LastName	Surname
DrivingRecord	DrivingRecord
DrivingRecords	DrivingRecords

Table 5-10 Automatic match of *source.xsd* and *target.xsd*

The next step is to generate XSLT transformation scripts that:

- Transform a source data instance validated by the source XML Schema document into an XML document that is validated by the target XML Schema document – *src_2_trg.xsl.*
- Transform a target data instance validated by the target XML Schema document into an XML document that is validated by the source XML Schema document – *trg_2_src.xsl.*

Data instances *source.xml* and *target.xml* are presented in the following figure:

```
<?xml version="1.0" encoding="UTF-8"?>
<Person xsi:noNamespaceSchemaLocation="source.xsd"                    a.)
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" BirthDate="1967-08-13">
        <FirstName Name="John"/>
        <LastName Name="Source"/>
        <DrivingRecords>
                <DrivingRecord Location="Ottawa" Name="Speeding" Date="2007-08-13"/>
        </DrivingRecords>
        <DrivingLicense>1234</DrivingLicense>
</Person>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonInstance height="6'2"" BirthDate="1982-12-01"                  b.)
xsi:noNamespaceSchemaLocation="target.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Forename Name="Jane"/>
        <Surname Name="Target"/>
        <DrivingRecords>
                <DrivingRecord Location="Toronto" Name="Parking ticket" Date="2003-10-
13"/>
        </DrivingRecords>
</PersonInstance>
```

Figure 5.8 Data instances: a.) source.xml b.) target.xml

The two XSLT transformation scripts – *src_2_trg.xsl* and *trg_2_src.xsl* – are shown in the following figures:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
        <xsl:output encoding="UTF-8" indent="yes" method="xml" version="1.0"/>
        <xsl:template match="/">
                <xsl:element name="PersonInstance">
                        <xsl:attribute name="xsi:noNamespaceSchemaLocation"
namespace="http://www.w3.org/2001/XMLSchema-instance"><xsl:value-of select="&amp;quot;
target.xsd&amp;quot;"/></xsl:attribute>
                        <xsl:attribute name="BirthDate"><xsl:value-of
select="Person/@BirthDate"/></xsl:attribute>
                        <xsl:attribute name="height"><xsl:value-of
select="PersonInstance/@height"/></xsl:attribute>
                        <xsl:for-each select="Person/child::*">
                                <xsl:if test="name(current()) = 'LastName'">
                                        <xsl:element name="Surname">
                                                <xsl:copy-of select="./child::*|@*"/>
                                        </xsl:element>
                                </xsl:if>
                        </xsl:for-each>
                        <xsl:for-each select="Person/child::*">
                                <xsl:if test="name(current()) = 'FirstName'">
                                        <xsl:element name="Forename">
                                                <xsl:copy-of select="./child::*|@*"/>
                                        </xsl:element>
                                </xsl:if>
                        </xsl:for-each>
                        <xsl:for-each select="Person/child::*">
                                <xsl:if test="name(current()) = 'DrivingRecords'">
                                        <xsl:element name="DrivingRecords">
                                                <xsl:copy-of select="./child::*|@*"/>
                                        </xsl:element>
                                </xsl:if>
                        </xsl:for-each>
                </xsl:element>
        </xsl:template>
</xsl:stylesheet>
```

Figure 5.9 XSLT Transformation script: *src_2_trg.xsl*

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
        <xsl:output encoding="UTF-8" indent="yes" method="xml" version="1.0"/>
        <xsl:template match="/">
                <xsl:element name="Person">
                        <xsl:attribute name="xsi:noNamespaceSchemaLocation"
namespace="http://www.w3.org/2001/XMLSchema-instance"><xsl:value-of select="&quot;
source.xsd&quot;"/></xsl:attribute>
                        <xsl:attribute name="BirthDate"><xsl:value-of
select="PersonInstance/@BirthDate"/></xsl:attribute>
                        <xsl:for-each select="PersonInstance/child::*">
                                <xsl:if test="name(current()) = 'Forename'">
                                        <xsl:element name="FirstName">
                                                <xsl:copy-of select="./child::*|@*"/>
                                        </xsl:element>
                                </xsl:if>
                        </xsl:for-each>
                        <xsl:for-each select="PersonInstance/child::*">
                                <xsl:if test="name(current()) = 'DrivingRecords'">
                                        <xsl:element name="DrivingRecords">
                                                <xsl:copy-of select="./child::*|@*"/>
                                        </xsl:element>
                                </xsl:if>
                        </xsl:for-each>
                        <xsl:for-each select="PersonInstance/child::*">
                                <xsl:if test="name(current()) = 'Surname'">
                                        <xsl:element name="LastName">
                                                <xsl:copy-of select="./child::*|@*"/>
                                        </xsl:element>
                                </xsl:if>
                        </xsl:for-each>
                        <xsl:element name="DrivingLicense"/>
                </xsl:element>
        </xsl:template>
</xsl:stylesheet>
```

Figure 5.10 XSLT Transformation script: *trg_2_src.xsl*

The first transformation script – *src_2_trg.xsl* – is run using *source.xml* as input and the result is an XML document *source2Target.xml* that is validated by the target XML Schema document target.xsd as shown in the following figure:

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonInstance xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="target.xsd" BirthDate="1967-08-13" height="">
        <Forename Name="John"/>
        <Surname Name="Source"/>
        <DrivingRecords>
                <DrivingRecord Location="Ottawa" Name="Speeding" Date="2007-08-13"/>
        </DrivingRecords>
</PersonInstance>
```

Figure 5.11 *source.xml* transformed to *source2Target.xml*

The above listed XML document – *source2Target.xml* – is validated by target XML Schema document *target.xsd* and it contains information from source data instance validated by the source XML Schema document *source.xsd*. For instance, "*Forname*" is an XML element defined in *target.xsd* and because it matches with "*FirstName*" defined in *source.xsd* contains the same information as in *source.xml*: "**John**". All information contained in matching XML Schema components is translated from one data instance to other data instance.

However, source XML Schema document *source.xsd* contains an element – "*DrivingLicense*" – which does not have any correspondence in target XML Schema document *target.xsd*. Therefore, the information contained by this element in *source.xml* is **not** translated into source2Target.xml, because, otherwise, *source2Target.xml* would not be validated by *target.xsd*.

```
<?xml version="1.0" encoding="UTF-8"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=" source.xsd" BirthDate="1982-12-01">
        <FirstName Name="Jane"/>
        <LastName Name="Target"/>
        <DrivingRecords>
                <DrivingRecord Location="Toronto" Name="Parking ticket" Date="2003-10-
13"/>
        </DrivingRecords>
        <DrivingLicense/>
</Person>
```

Figure 5.12 *target.xml* transformed to *target2Source.xml*

The above listed XML document – *target2Source.xml* – is validated by source XML Schema document *source.xsd* and it contains information from target data instance validated by the target XML Schema document *target.xsd*. In the same way presented in source2Target.xml, all information contained in matching XML Schema components is translated from one data instance to other data instance.

A different and more difficult problem arises in this case. "*DrivingLicense*" element is not defined in *target.xsd,* therefore there is no information in *target.xml* that can be translated under this element. In order to give a solution to this problem, the transformation script creates an empty element as shown in the above figure. However, this action may generate another problem as follows: if "*DrivingLicense*" element is defined in *source.xsd* such as it must contain specific information – controlled by some regular expression patterns in identity constraint definitions – then *target2Source.xml* cannot be validated by *source.xsd* and user intervention is mandatory to fix this element and have *target2Source.xml* data instance validated by *source.xsd*.

6 Conclusions and Future Work

The schema integration, data warehouses and e-commerce are the development domains that need XML Schema mapping software tools. These tools have not reached the expected performance and the research paths are still wide open.

6.1 Contributions of this Thesis

Overall, this thesis proposes a solution for XML Schema mapping problem and furthermore, it shows how this solution can be used in a practical way. An XML Schema document describes information that resides in an XML document, therefore and XML Schema document should be regarded from a higher perspective than XML even though it is written in XML.

The main research contributions of this thesis are summarized as follows:

- A new XML Schema document internal representation within the Matching Engine. This representation is based on XML Schema Infoset Model API [50] from Eclipse Modeling Framework package. Furthermore, XML Schema Infoset Model follows the XML Schema standard [34, 36 and 36] and can be used also to access and update WSDL documents. The most important benefit that XML Schema Infoset Model brings to the new XML Schema matching software tool proposed in this thesis is **scalability**: very large XML Schema documents such as Global Justice XML Data Model [71] and Canadian Integrated Justice Information [72] can be loaded by Hermes.
- A new Matching Engine architecture. This architecture follows the XML Schema standard [34, 36 and 36] and has been designed using "Composite" and "Chain of Responsibility" design patterns [56]. The Matching Engine has a modular design and can be integrated in other software architectures. Should the XML Schema standard change, the Matching Engine can be easily adapted to this change.

- A new way of storing/retrieving semantic information (i.e. knowledge) to/from the system repository using the RDF data representation and RDF queries.
- A new Transformation Engine: two XSLT scripts are generated and these scripts are used to transform (1) a data instance validated by the source XML schema into a data format that is validated by the target XML schema and (2) a data instance validated by the target XML schema into a data format validated by the source XML schema.

6.2 Future Work

There are two main categories for the future work:

- The improvement of the current performance of Hermes (recall and accuracy quality measures, the GUI and the transformation engine)
- The employment of new methods and technologies and the advancement of the research on the XML Schema mapping.

The current performance of Hermes can be improved as follows:

- As presented in the experimental results chapter of this thesis, Hermes missed a match between two elements with the following names: "*mls_number*" from RE_I.xsd and "*mls_num*" from RE_II.xsd. Therefore, the name matching component of the Matching Engine must be improved. There are several methods that can be used to address this issue. The first one is the q-gram method – breaking a character string in sequences of n characters, n= 2, 3, 4 – used by COMA++ [3, 4] or MKB [14]. More elaborate methods are described in [74] and these are *word-segmentation* methods. Word-segmentation methods are used in language processing research domain. Other improvements for the name matching component would be a configurable string tokenization and more elaborate regular expressions. The current version of Hermes does string tokenization only on the upper case characters and the "_" character.

- The graphical user interface should be improved such that it will scale very well for large *mappings*, in the case of XML Schema documents with thousands of elements [71, 72]. A solution to this problem is presented in [40] and is based on focusing on relevant elements. Very complex XML Schema documents contain components that refer other components; therefore the graphical user interface should have the capability of revealing the concrete definition of the referred XML Schema components.
- The Transformation Engine should be configurable and provide more alternatives for the XSLT scripts. As presented in the experimental results chapter, in the case of the un-matched XML Schema components, the transformation scripts must handle them according to the user expectations.

From the employment of new techniques perspective there are the following options:

- Hermes can be extended to use the ontology of the XML Schema documents domain during the matching process. This approach is used by iMAP [17] with the help of date ontology. For instance, in the case of two XML Schema documents from academic research domain Hermes can use the respective domain ontology – *ka.owl* [83].
- WSDL documents can be accessed and updated using the XML Schema Infoset Model API [41, 42 and 43]. The Hermes Repository is based on Sesame [45, 46 and 62], a generic architecture for storing and querying RDF(S). The specification [84] defines an RDF/OWL vocabulary to describe WSDL 2.0, therefore WSDL 2.0 documents can be translated into RDF and then stored in an RDF repository like Sesame. The idea is to generate WSDL descriptions from processed RDF data.

Bibliography

[1] Erhard Rahm, Philip A. Bernstein – "A survey of approaches to automatic schema matching" - The VLDB Journal — The International Journal on Very Large Databases, Volume 10 , Issue 4, December 2001, pages: 334 - 350 Springer-Verlag New York, Inc.

[2] Erhard Rahm, Hong-Hai Do, Sabine Maßmann – "Matching large XML schemas", ACM SIGMOD Volume 33 , Issue 4 (December 2004) ISSN:0163-5808

[3] H. H. Do and E. Rahm - "COMA - a system for flexible combination of schema matching approaches". In Proceedings of the Very Large Data Bases Conference (VLDB), pages 610-621, 2001.

[4] Aumüller, D., H.H. Do, S. Massmann, E. Rahm: Schema and Ontology Matching with COMA++ (Software Demonstration). Proc. 24. ACM SIGMOD Intl. Conf. Management of Data, 2005

[5] Huynh Quyet Thang, Vo Sy Nam - "XML Schema Automatic Matching Solution" IJCSSE VOLUME 1 NUMBER 1 2007 ISSN 1307-430X

[6] Eclipse Modeling Framework (EMF) – http://www.eclipse.org/modeling/emf/downloads/?project=emf

[7] Renee J. Miller et al – "The Clio Project - Managing Heterogeneity" March 2001 - ACM SIGMOD Record, Volume 30 Issue 1

[8] Altova XML Spy – www.altova.com

[9] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, Christoph Quix - "Industrial-Strength Schema Matching", SIGMOD Record, Vol. 33, No. 4, December 2004

[10] AnHai Doan, Pedro Domingos, Alon Y. Levy - "Learning Source Description for Data Integration", WebDB (Informal Proceedings) - pages 81-86, 2000

[11] Doan AH, Domingos P, Halevy A - "Reconciling schemas of disparate data sources: a machine-learning approach" Proc ACM SIGMOD Conf 2001, pp. 509-520

[12] J. Madhavan, P. A. Bernstein, and E. Rahm. - "Generic schema matching with Cupid" - In Proceedings of the 27th International Conference on Very Large Databases, 2001

[13] Okawara T., Tanaka J., Morishima A., Sugimoto, S. - "A Support Tool for XML Schema Matching and Its Implementation" - Data Engineering Workshops, 2005. 21st International Conference on

[14] Madhavan, J.,Bernstein, P.A., Doan, A., Halevy, A. – "Corpus-based Schema Matching", Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on, pages 57- 68, 5-8 April 2005

[15] Melnik, S. Garcia-Molina, H. Rahm, E. - "Similarity flooding: a versatile graph matching algorithm and its application to schema matching" Proceedings 18th International Conference on Data Engineering, 2002.

[16] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein - "Rondo: a programming platform for generic model management". In SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pages 193-204, New York, NY, USA, 2003, ACM Press.

[17] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Halevy, Pedro Domingos - "iMAP: Discovering Complex Semantic Matches between Database Schemas" -

International Conference on Management of Data - Proceedings of the 2004 ACM SIGMOD international conference on Management of data.

[18] Sebastian Bossung, Hermann Stoeckle, John Grundy, Robert Amor and John Hosking - "Automated Data Mapping Specification via Schema Heuristics and User Interaction", Proceedings of the 19th International Conference on Automated Software Engineering (ASE'04) 20-24 Sept. 2004;

[19] XML Beans - http://xmlbeans.apache.org/

[20] Apache Tomcat - http://tomcat.apache.org/

[21] Do, H.H., S. Melnik, E. Rahm - "Comparison of Schema Matching Evaluations" GI-Workshop Web and Databases, LNCS 2593, 2003

[22] Van Rijsbergen, C. J. – "Information Retrieval." 2nd edition, 1979, London, Butterworths

[23] M. Benerecetti, P. Bouquet, and S. Zanobini – "Soundness of schema matching methods", Springer Berlin / Heidelberg, Volume 3532/2005

[24] C. Batini, M. Lenzerini, S. B. Navathe - "A comparative analysis of methodologies for database schema integration." ACM Computing Surveys (CSUR), Volume 18 Issue 4 December 1986

[25] Tova Milo, Sagit Zohar - "Using Schema Matching to Simplify Heterogeneous Data Translation", Proceedings of the 24th VLDB Conference - New York, USA, 1998

[26] Li Xu, David W. Embley – "Discovering Direct and Indirect Matches for Schema Elements", Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03)

[27] Uddam Chukmol, Rami Rifaieh, Nabila Aicha Benharkat - "EXSMAL: EDI/XML semi-automatic Schema Matching ALgorithm", Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (CEC'05)

[28] N. Tansalarak and K. Claypool – "QoM:Qualitative and Quantitative Schema Match Measure", In Conference on Conceptual Modeling (ER), 2003.

[29] Kajal T. Claypool, Vaishali Hegde, Naiyana Tansalarak - "QMatch - A Hybrid Match Algorithm for XML Schemas", Proceedings of the 21st International Conference on Data Engineering (ICDE '05)

[30] Marko Smiljani, Maurice van Keulen, Willem Jonker – "Using Element Clustering to Increase the Efficiency of XML Schema Matching", Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)

[31] Jan Hegewald, Felix Naumann, Melanie Weis - "XStruct: Efficient Schema Extraction from Multiple and Large XML Documents", Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)

[32] Carmel Domshlak, Avigdor Gal and Haggai Roitman - "Rank Aggregation for Automatic Schema Matching", IEEE Computer Society, Jan. 2007

[33] XML Schema - http://www.w3.org/XML/Schema

[34] XML Schema Part 0: Primer Second Edition: http://www.w3.org/TR/xmlschema-0/

[35] XML Schema Part 1: Structures Second Edition: http://www.w3.org/TR/xmlschema-1/

[36] XML Schema Part 2: Datatypes Second Edition: http://www.w3.org/TR/xmlschema-2/

[37] RDF/XML Syntax Specification - http://www.w3.org/TR/rdf-syntax-grammar/

[38] Resource Description Framework (RDF): Concepts and Abstract Syntax – http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[39] RDF Vocabulary Description Language 1.0: RDF Schema – http://www.w3.org/TR/rdf-schema/

[40] George G. Robertson, Mary P. Czerwinski, John E. Churchill - "Visualization of Mappings Between Schemas", Conference on Human Factors in Computing Systems, Proceedings of the SIGCHI conference on Human factors in computing systems Portland, Oregon, USA, 2005, ISBN:1-58113-998-5

[41] Dave Spriet – "XML Schema Infoset Model, Part 1" http://www.eclipse.org/modeling/emf/docs/xsd/dW/os-schema1/os-schema1-ltr.pdf

[42] Dave Spriet – "XML Schema Infoset Model, Part 2", http://www.eclipse.org/modeling/emf/docs/xsd/dW/os-schema2/os-schema2-ltr.pdf

[43] XML Schema Infoset - http://www.eclipse.org/modeling/mdt/?project=xsd#xsd

[44] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose - "Eclipse Modeling Framework: A Developer's Guide", Addison Wesley, 2003, ISBN: 0-13-142542-0

[45] Sesame: RDF Schema Querying and Storage - http://www.openrdf.org/

[46] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen - "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema"

[47] Jeen Broekstra, Arjohn Kampman - "SeRQL: A Second Generation RDF Query Language", 2003

[48] "Resource Description Framework (RDF): Concepts and Abstract Syntax" - http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[49] RDF semantics - http://www.w3.org/TR/rdf-mt/

[50] XML Schema Infoset Model – API Reference – http://download.eclipse.org/modeling/mdt/xsd/javadoc/2.3.0/org/eclipse/xsd/package-summary.html#details

[51] XML Path Language (XPath) - http://www.w3.org/TR/1999/REC-xpath-19991116

[52] Eric Clayberg, Dan Rubel – "Eclipse: Building Commercial-quality Plug-ins", Pearson Education, March 2006.

[53] SWT: The Standard Widget Toolkit – http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html

[54] Michael Fitzgerald – "Learning XSLT", O'Reilly & Associates, 2004, ISBN 0-596-00327-7

[55] Sal Mangano – "XSLT Cookbook", O'Reilly Media, 2006, ISBN 0-596-00974-7

[56] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – "Design Patterns – Elements of Reusable Object Oriented Software", ISBN 0201633612, Addison-Wesley 1995.

[57] XSL Transformations (XSLT) – http://www.w3.org/TR/xslt

[58] AnHai Doan - "Learning to Map between Structured Representations of Data" - PhD Thesis, University of Washington, 2002

[59] Clio: A Schema Mapping Tool - Sample Schemas: http://www.cs.toronto.edu/db/clio/testSchemas.html

[60] "Global versus Local - (A Collectively Developed Set of Schema Design Guidelines)" http://www.xfront.com/GlobalVersusLocal.html

[61] Aduna - http://www.aduna-software.com/

[62] Sesame 1.2.6. User Guide - http://www.openrdf.org/documentation.jsp

[63] G. Karvounarakis, Sofia Alexaki, V. Christophides, D. Plexousakis, M. Scholl - "RQL: A Declarative Query Language for RDF", ACM 2002

[64] Miller, G.A.- "WordNet: a lexical database for English." Commun. ACM 38(11), 39-41 (1995)

[65] JWNL - Java WordNet API - http://jwordnet.sourceforge.net/

[66] Jun-Ki Min, Jae-Yong Ahn, Chin-Wan Chung - "Efficient extraction of schemas for XML documents",Information Processing Letters 85 (2003) 7-12

[67] Martin Bernauer, Gerti Kappel, Gerhard Kramler - "Approaches to Implementing Active Semantics with XML Schema", Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA'03)

[68] Michel Klein - "Interpreting XML documents via an RDF Schema ontology", Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA'02)

[69] Massimo Franceschet, Angelo Montanari, Donatella Gubiani - "Modeling and Validating Spatio-Temporal Conceptual Schemas in XML Schema", 18th International Workshop on Database and Expert Systems Applications 2007 IEEE

[70] Mohamed Boukhebouze, Rami Rifaieh, Nabila Benharkat, YoussefAmghar - "Benchmarking XML-Schema Matching Algorithms for Improving Automated Tuning" 1-4244-1031-2/07 - 2007 IEEE

[71] "Global Justice XML Data Model" - http://www.it.ojp.gov/jxdm/

[72] "Canada Public Safety Information Network (CPSIN)" - http://ww2.ps-sp.gc.ca/iji-iij/About_CPSIN_e.asp

[73] Dagmar Köhn, Lena Strömbäck - "A Schema Matching Architecture For The Bioinformatics Domain", Proceedings of the 2006 Winter Simulation Conference, L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

[74] Aminul Islam, Diana Inkpen, Iluju Kiringa - "Applications of corpus-based semantic similarity and word segmentation to database schema matching", The VLDB Journal, DOI 10.1007/s00778-007-0067-9, Springer-Verlag 2007

[75] Avigdor Gal - "Why is Schema Matching Tough and What Can We Do About It?", SIGMOD Record, Vol. 35, No. 4, December 2006

[76] Cindy X. Chen, George A. Mihaila, Sriram Padmanabhan, Isabelle M. Rouvellou - "Query Translation Scheme for Heterogeneous XML Data Sources", WIDM'05, November 5, 2005, Bremen, Germany, ACM 2005

[77] Eclipse, an open development platform - http://www.eclipse.org/

[78] Yannis Velegrakis, Renee J. Miller, Lucian Popa - "Preserving mapping consistency under schema changes", The VLDB Journal (2004) 13: 274-293

[79] Aida Boukottaya, Christine Vanoirbeek – "Schema Matching for Transforming Structured Documents", Proceedings of the 2005 ACM symposium on Document engineering

[80] J. T. Yao M. Zhang – "A Fast Tree Pattern Matching Algorithm for XML Query",Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)

[81] Doina Caragea Tanveer Syeda-Mahmood – "Semantic API Matching for Automatic Service Composition", Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, 2004

[82] Bruce Powel Douglass – "Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems", Addison Wesley, 2002 ISBN: 0-201-69956-7

[83] Protégé Ontology Library - http://protegewiki.stanford.edu/index.php/Protege_Ontology_Library

[84] Web Services Description Language (WSDL) Version 2.0: RDF Mapping - http://www.w3.org/TR/wsdl20-rdf/

[85] W3C and Electronic Commerce - http://www.w3.org/TR/EC-related-activities

[86] United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport - http://www.unece.org/trade/untdid/welcome.htm

[87] "COPRAS Reverse mapping" - www.w3.org/2004/copras/docu/D18.pdf

[88] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, Peter Patel-Schneider – "The Description Logic Handbook: Theory, implementation, and applications", ISBN-13: 9780521876254, Published August 2007

Appendix A: XSLT Script used by the Schema Extraction Engine

The XSLT script used by the Schema Extraction Engine:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
        <xsl:template match="child::*">
                <xsl:element name="xs:schema">
                        <xsl:attribute
name="elementFormDefault">qualified</xsl:attribute>
                        <xsl:attribute name="ns1"><xsl:value-of select="namespace-
uri()"/></xsl:attribute>
                        <xsl:call-template name="test1"/>
                </xsl:element>
        </xsl:template>
        <xsl:template match="node()" name="test1" mode="A">
                <!-- get unique elements-->
                <xsl:if test="count( preceding-sibling::*[name()=name(current())]) = 0">
                        <xsl:element name="xs:element">
                                <xsl:attribute name="name"><xsl:value-of
select="name()"/></xsl:attribute>
                                <!-- test for complex type and describe element -->
                                <xsl:if test="child::* | @*">
                                        <xsl:element name="xs:complexType">
                                                <xsl:if test="child::*">
                                                        <xsl:element name="xs:sequence">
                                                                <!-- how do I know it is a
sequence and not all /choice/etc ?-->
                                                                <!-- try apply template-->
                                                                <xsl:for-each
select="child::*">
                                                                        <!-- get child
elelments only once, i.e. unique -->
                                                                        <xsl:if test="count(
preceding-sibling::*[name()=name(current())]) = 0">
                                                                                <xsl:element
name="xs:element">
        <xsl:attribute name="ref"><xsl:value-of select="name()"/></xsl:attribute>
        </xsl:element>
```

```
                                                                    </xsl:if>
                                                                </xsl:for-each>
                                                            </xsl:element>
                                                        </xsl:if>
                                                        <!-- Atributes -->
                                                        <xsl:for-each select="@*">
                                                            <xsl:element name="xs:attribute">
                                                                <xsl:attribute
name="name"><xsl:value-of select="name()"/></xsl:attribute>
                                                                <xsl:attribute
name="use">required</xsl:attribute>
                                                                <xsl:element
name="xs:simpleType">
                                                                    <xsl:element
name="xs:restriction">
                                                                        <xsl:attribute
name="base">xs:string</xsl:attribute>
                                                                        <xsl:element
name="xs:enumeration">

        <xsl:attribute name="value"><xsl:value-of select="."/></xsl:attribute>

        </xsl:element>
                                                                    </xsl:element>
                                                                </xsl:element>
                                                            </xsl:element>
                                                        </xsl:for-each>
                                                        <!-- Attributes/ -->
                                                    </xsl:element>
                                                </xsl:if>
                                            </xsl:element>
                                            <xsl:apply-templates mode="A"/>
                                        </xsl:if>
        </xsl:template>
</xsl:stylesheet>
```

Appendix B: XML Schema Design Guidelines

Primary XML Schema components can be declared as global components – i.e. direct children of the XML Schema document root element *<schema>* – or as local components, as a part of another XML Schema component. Based on this fact, there are three important guidelines with respect to the XML Schema document structure as described in [60].

Russian Doll Design

Using this approach, the XML Schema document has almost the same tree structure as the XML data instance. This means that the XML Schema document root element *<schema>* has only one child and this is the element declaration of the root element of the XML data instance. All the other XML Schema components are locally declared down the tree structure. An XML Schema component contains declarations of other XML Schema components like a Russian Doll. This design can be characterized as follows:

- *Opaque content*: the content of an element declaration is opaque to other XML Schema components within the same schema or to other XML Schema documents. This fact means that an XML Schema designed using this approach cannot be reused;
- *Localized scope*: all element declarations but the element declaration of the root element of the XML data instance are declared locally, therefore these declarations cannot be used by other XML Schema components;
- *Decoupled*: each XML Schema component contain its definition, therefore a change in one of them does not impact other components;
- *Cohesive*: all related data is grouped into self contained components

Salami Slice Design

This approach is completely opposite to the Russian Doll, which means that all element declarations are global declarations – i.e. children of the XML Schema document root element *<schema>*. Then all these element declarations are assembled together like salami slices in a sandwich. The Salami Slice paradigm can be characterized as follows:

- *Transparent content*: all element declarations are reusable
- *Global scope*: all element declarations are global
- *Coupled*: A change in one element declaration will impact directly other related element declarations
- *Verbose*: everything is clear
- *Cohesive*: all related data is grouped into self contained components

Venetian Blind Design

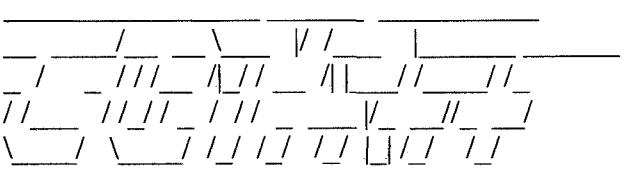
This design is a variation of the Salami Slice design, with the difference that instead of using global element declarations, it uses global type definitions (simple or complex). The Venetian Blind design has the following characteristics:

- *Maximum reuse*: only type definitions are defined globally, therefore more XML Schema components can use them
- *Maximum namespace hiding*: element declarations are nested in type definitions
- *Easy exposure switching*: elementFormDefault controls whether namespaces are hidden or exposed in the XML data instance.
- *Coupled*: components can have interdependencies between them.
- *Cohesive*: related data is grouped together within components

Appendix C: COMA++ Scalability Test

Coma++ [4] has been evaluated on JXDM [71] and CPSIN [72] XML Schema documents:

D:\work\lcheciu\SM_applications\coma++_2005>java **-Xmx1024M** comagui/Main



SourceManager(): Initializing ...
SourceManager(): Initialization DONE
Load 4 sources and 12 source relationships
Loading source paths(4) . Done!
Updating path matrices(4) . Done!
Load 40 synonyms
Load 22 abbreviations
Load 18 matcher definitions
case_xsd
controller.getSchemaManager().getSource(schemaName) done: case_xsd
controller.getSchemaManager().loadSchemaGraph(source, true): done 1.61 s
controller.loadSourceGraph(schemaGraph): done 1.063 s
jxdm_xsd
Exception in thread "Thread-3" **java.lang.OutOfMemoryError: Java heap space**
at java.util.ArrayList.<init>(Unknown Source)
at java.util.ArrayList.<init>(Unknown Source)
at match.MatcherLibrary$SimilarityCache.get(MatcherLibrary.java:37)
at match.MatcherLibrary.cacheGet(MatcherLibrary.java:998)
at match.GenericDirectMatcherCentralCache.computeConstituentSimilarity(GenericDirectMatcherCentralCache.java:38)
at match.GenericMatcher.computeConstituentSimilarityCubeComplete_1(GenericMatcher.java:1011)
at match.GenericMatcher.computeConstituentSimilarityCubeComplete_1(GenericMatcher.java:1068)
at match.GenericMatcher.matchConstituentsComplete_1(GenericMatcher.java:1107)
at match.SchemaGraph.matchIntra(SchemaGraph.java:929)
at match.SchemaGraph.simplifyGraph(SchemaGraph.java:975)
at match.SchemaGraph.preprocessGraph(SchemaGraph.java:1152)
at match.SchemaGraph.preprocessGraph(SchemaGraph.java:1165)
at app.SchemaManager.loadSchemaGraph(SchemaManager.java:931)
at comagui.LoadFromDBThread.loadTargetSchemaFromDB(LoadFromDBThread.java

:579)

at comagui.LoadFromDBThread.run(LoadFromDBThread.java:108)

Exception in thread "AWT-EventQueue-0" **java.lang.OutOfMemoryError: Java heap space**

at javax.swing.Timer$DoPostEvent.run(Unknown Source)

at java.awt.event.InvocationEvent.dispatch(Unknown Source)

at java.awt.EventQueue.dispatchEvent(Unknown Source)

at java.awt.EventDispatchThread.pumpOneEventForFilters(Unknown Source)

at java.awt.EventDispatchThread.pumpEventsForFilter(Unknown Source)

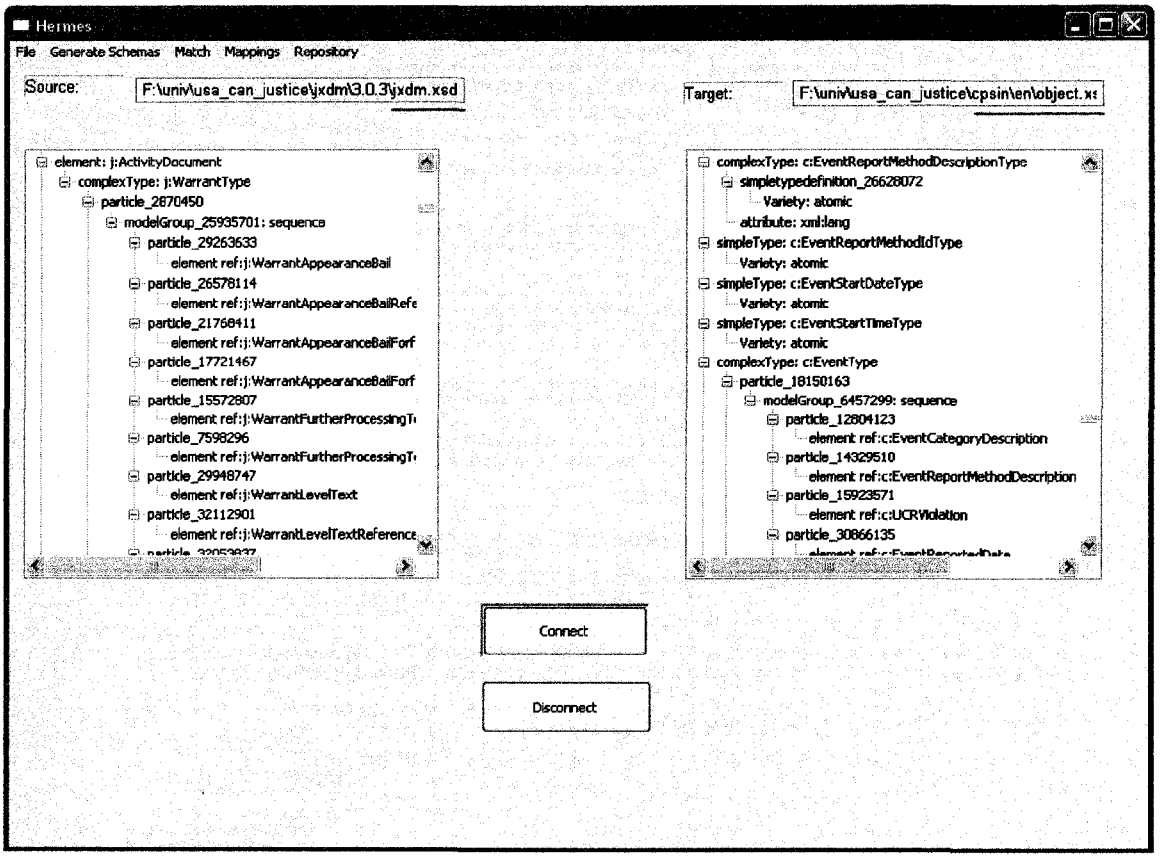
at java.awt.EventDispatchThread.pumpEventsForHierarchy(Unknown Source)

at java.awt.EventDispatchThread.pumpEvents(Unknown Source)

at java.awt.EventDispatchThread.pumpEvents(Unknown Source)

at java.awt.EventDispatchThread.run(Unknown Source)

In the case of Hermes, using same amount of RAM:



Appendix D: XStruct Testing

java -Xmx200m -Xms20m -jar XStruct.jar -SchemaNamespace xs -LimitUnboundness 20 -output result.xsd sourceINST.xml

Program output is:

XStruct Version 1.1 - Jan Hegewald 2006
Processing file F:\univ\tools\xstruct\sourceINST.xml ... Done.
Starting to factor element 'WorkflowMain'... Nothing to factor.
Starting to factor element 'WorkflowIncludes'... Nothing to factor.
Starting to factor element 'WorkflowInclude'... Nothing to factor.
Starting to factor element 'newInclude'... Nothing to factor.
Starting to factor element 'CommandTemplates'... Nothing to factor.
Starting to factor element 'CommandTemplate'... Nothing to factor.
Starting to factor element 'Sequences'... Nothing to factor.
Starting to factor element 'Sequence'... Nothing to factor.
Starting to factor element 'SequenceCommands'... Nothing to factor.
Starting to factor element 'SequenceCommand'... Nothing to factor.
Starting to factor element 'MacroDefinitions'... Nothing to factor.
Starting to factor element 'MacroDefinition'... Nothing to factor.
Done.
Elapsed time: 0 seconds.

XML data file, sourceINST.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<WorkflowMain Name="SourceName" xmlns:abc="http://abc.org"
xmlns:def="http://def.org" DefaultLogLanLoc="SourceLocation"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="F:\univ\tools\xstruct\result.xsd">
        <WorkflowIncludes>
```

```
            <abc:WorkflowInclude FilePath="SourceFileInclude">
                <newInclude Location="NewFileIncludeINSource"/>
                <newInclude Location="NewFileIncludeINSourceNewer"/>
            </abc:WorkflowInclude>
        </WorkflowIncludes>
        <CommandTemplates>
            <CommandTemplate ExePath="python.exe" Name="Command2"
Parameters="--obj"/>
            <CommandTemplate ExePath="python.exe" Name="Command3"
Parameters="--debug"/>
        </CommandTemplates>
        <def:Sequences xmlns:def="http://abc.org">
            <Sequence Name="Compile" Description="Compile Souce code">
                <SequenceCommands>
                    <SequenceCommand Command="Command1"
Name="runCommand1">
                        <MacroDefinitions>
                            <MacroDefinition Name="param"
Value="all.dsw"/>
                        </MacroDefinitions>
                    </SequenceCommand>
                </SequenceCommands>
            </Sequence>
            <Sequence Name="ListResults" Description="List the log file">
                <SequenceCommands>
                    <SequenceCommand Command="Command2"
Name="runCommand2">
                        <MacroDefinitions>
                            <MacroDefinition Name="param"
Value="ListLog.py"/>
                        </MacroDefinitions>
```

```
                    </SequenceCommand>
                </SequenceCommands>
            </Sequence>
        </def:Sequences>
</WorkflowMain>
```

Result.xsd, the XML schema generated by XStruct:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Created by XStruct, Version 1.1 -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <!-- Simple types -->
        <!-- Empty elements -->
        <!-- Complex types -->
        <xs:element name="WorkflowMain">
            <xs:complexType>
                <xs:sequence>
                    <xs:sequence>
                        <xs:element ref="WorkflowIncludes"/>
                        <xs:element ref="CommandTemplates"/>
                        <xs:element ref="Sequences"/>
                    </xs:sequence>
                </xs:sequence>
                <xs:attribute name="Name" type="xs:string"
fixed="SourceName"/>
                <xs:attribute name="DefaultLogLanLoc" type="xs:string"
fixed="SourceLocation"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="WorkflowIncludes">
            <xs:complexType>
```

```
                    <xs:sequence>
                        <xs:element ref="WorkflowInclude"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="WorkflowInclude">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="newInclude" maxOccurs="2"/>
                    </xs:sequence>
                    <xs:attribute name="FilePath" type="xs:string"
fixed="SourceFileInclude"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="newInclude">
                <xs:complexType>
                    <xs:attribute name="Location" type="xs:string" use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="CommandTemplates">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="CommandTemplate" maxOccurs="2"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="CommandTemplate">
                <xs:complexType>
                    <xs:attribute name="Name" type="xs:string" use="required"/>
                    <xs:attribute name="Parameters" type="xs:string"
use="required"/>
```

```
                    <xs:attribute name="ExePath" type="xs:string"
fixed="python.exe"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="Sequences">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="Sequence" maxOccurs="2"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="Sequence">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="SequenceCommands"/>
                    </xs:sequence>
                    <xs:attribute name="Name" type="xs:string" use="required"/>
                    <xs:attribute name="Description" type="xs:string"
use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="SequenceCommands">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element ref="SequenceCommand"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="SequenceCommand">
                <xs:complexType>
                    <xs:sequence>
```

```
                    <xs:element ref="MacroDefinitions"/>
                </xs:sequence>
                <xs:attribute name="Name" type="xs:string" use="required"/>
                <xs:attribute name="Command" type="xs:string" use="required"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="MacroDefinitions">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="MacroDefinition"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="MacroDefinition">
            <xs:complexType>
                <xs:attribute name="Name" type="xs:string" fixed="param"/>
                <xs:attribute name="Value" type="xs:string" use="required"/>
            </xs:complexType>
        </xs:element>
</xs:schema>
```

Result.xsd does not validate the XML data instance, sourceINST.xml. The right XML schema(s) for sourceINST.xml were generated using Altova XML Spy [8] (note targetNamspace attribute):

B1.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:abc="http://abc.org" xmlns:def="http://def.org"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:import namespace="http://def.org" schemaLocation="b12.xsd"/>
```

```
<xs:import namespace="http://abc.org" schemaLocation="b11.xsd"/>
<xs:complexType name="CT_newInclude">
    <xs:attribute name="Location" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration
value="NewFileIncludeINSource"/>
                <xs:enumeration
value="NewFileIncludeINSourceNewer"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="CT_WorkflowMain">
    <xs:sequence>
        <xs:element ref="def:Sequences"/>
        <xs:element ref="WorkflowIncludes"/>
        <xs:element ref="CommandTemplates"/>
    </xs:sequence>
    <xs:attribute name="Name" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="SourceName"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="DefaultLogLanLoc" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="SourceLocation"/>
            </xs:restriction>
```

```
                </xs:simpleType>
        </xs:attribute>
</xs:complexType>
<xs:complexType name="CT_WorkflowIncludes">
        <xs:sequence>
                <xs:element ref="abc:WorkflowInclude"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="CT_SequenceCommands">
        <xs:sequence>
                <xs:element ref="SequenceCommand"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="CT_SequenceCommand">
        <xs:sequence>
                <xs:element ref="MacroDefinitions"/>
        </xs:sequence>
        <xs:attribute name="Name" use="required">
                <xs:simpleType>
                        <xs:restriction base="xs:string">
                                <xs:enumeration value="runCommand1"/>
                                <xs:enumeration value="runCommand2"/>
                        </xs:restriction>
                </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="Command" use="required">
                <xs:simpleType>
                        <xs:restriction base="xs:string">
                                <xs:enumeration value="Command1"/>
                                <xs:enumeration value="Command2"/>
                        </xs:restriction>
```

```
                </xs:simpleType>
        </xs:attribute>
</xs:complexType>
<xs:complexType name="CT_Sequence">
        <xs:sequence>
                <xs:element ref="SequenceCommands"/>
        </xs:sequence>
        <xs:attribute name="Name" use="required">
                <xs:simpleType>
                        <xs:restriction base="xs:string">
                                <xs:enumeration value="Compile"/>
                                <xs:enumeration value="ListResults"/>
                        </xs:restriction>
                </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="Description" use="required">
                <xs:simpleType>
                        <xs:restriction base="xs:string">
                                <xs:enumeration value="Compile Souce code"/>
                                <xs:enumeration value="List the log file"/>
                        </xs:restriction>
                </xs:simpleType>
        </xs:attribute>
</xs:complexType>
<xs:complexType name="CT_MacroDefinitions">
        <xs:sequence>
                <xs:element ref="MacroDefinition"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="CT_MacroDefinition">
        <xs:attribute name="Value" use="required">
```

```
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="ListLog.py"/>
                            <xs:enumeration value="all.dsw"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="Name" use="required">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="param"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
            </xs:complexType>
            <xs:complexType name="CT_CommandTemplates">
                <xs:sequence>
                    <xs:element ref="CommandTemplate"
maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="CT_CommandTemplate">
                <xs:attribute name="Parameters" use="required">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="--debug"/>
                            <xs:enumeration value="--obj"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="Name" use="required">
```

```
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="Command2"/>
                            <xs:enumeration value="Command3"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="ExePath" use="required">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="python.exe"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
            </xs:complexType>
            <xs:element name="newInclude" type="CT_newInclude"/>
            <xs:element name="WorkflowMain" type="CT_WorkflowMain"/>
            <xs:element name="WorkflowIncludes" type="CT_WorkflowIncludes"/>
            <xs:element name="SequenceCommands" type="CT_SequenceCommands"/>
            <xs:element name="SequenceCommand" type="CT_SequenceCommand"/>
            <xs:element name="Sequence" type="CT_Sequence"/>
            <xs:element name="MacroDefinitions" type="CT_MacroDefinitions"/>
            <xs:element name="MacroDefinition" type="CT_MacroDefinition"/>
            <xs:element name="CommandTemplates" type="CT_CommandTemplates"/>
            <xs:element name="CommandTemplate" type="CT_CommandTemplate"/>
</xs:schema>
```

B11.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xs:schema xmlns:abc="http://abc.org"
xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://abc.org">
        <xs:import schemaLocation="b1.xsd"/>
        <xs:complexType name="CT_WorkflowInclude">
                <xs:sequence>
                        <xs:element ref="newInclude" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="FilePath" use="required">
                        <xs:simpleType>
                                <xs:restriction base="xs:string">
                                        <xs:enumeration value="SourceFileInclude"/>
                                </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
        <xs:element name="WorkflowInclude" type="abc:CT_WorkflowInclude"/>
</xs:schema>
```

B12.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:def="http://def.org"
xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://def.org">
        <xs:import schemaLocation="b1.xsd"/>
        <xs:complexType name="CT_Sequences">
                <xs:sequence>
                        <xs:element ref="Sequence" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:element name="Sequences" type="def:CT_Sequences"/>
</xs:schema>
```