TÉCNICO LISBOA

XML Documents Transformation in Large Scale

João Paulo Fonseca Sequeira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Pável Pereira Calado

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Pável Pereira Calado
Member of the Committee: Prof. Helena Isabel de Jesus Galhardas

November 2015

Dedicated to my family...

Acknowledgments

I would like to express my gratitude to my supervisors for the support and understanding. This work would not be possible without them.
I would like also to thank my colleagues for the discussions about the topic, and for pointing me the right direction so many times.
I would like to express my gratitude also to a special person who motivated and helped me so much in this task. Without you, this would not be possible.
A special thanks for my father and sister for their unconditional support.

Resumo

Extensible Markup Language (XML) é principalmente usado para disseminação de informação pela internet. No entanto, também pode ser usado com propósitos diferentes, como guardar dados produzidos por ferramentas como as do Microsoft Office. O esquema dos dados pode evoluir com o tempo, e essa evolução pode levar a que os dados fiquem incompatíveis com o esquema. Para evitar essa incompatibilidade, os dados têm que ser transformados. XSLT é uma ferramenta muito usada para fazer transformações em dados, mas para grandes quantidades de dados requer demasiada memória e tempo. Este trabalho tem como objetivo encontrar uma solução para transformar dados produzidos por uma ferramenta de planeamento de redes de fibra ótica que por vezes podem ultrapassar os 3GB. Para aplicar uma transformação num desses ficheiros, Extensible Stylesheet Language Transformations (XSLT) requer cerca de 30 minutos enquanto que, para executar 20 transformações requer cerca de 10 horas. No entanto, se as 20 modificações forem fundidas numa só, a transformação será executada em 30 minutos. Apesar da melhoria, o tempo e memória necessários não são satisfatórios. Para resolver o problema, foi criada uma nova solução baseada em fluxo de dados que permite transformar os mesmos dados em cerca de 3 minutos e a memória necessária é residual.

Palavras-chave: transformação, migração, esquema, evolução, modelo

Abstract

Although the primary use of Extensible Markup Language (XML) is the dissemination of documents in the internet, it can also be used for different purposes, like storing data produced by stand alone software programs such as the Microsoft Office suite tools. It is well known that schemas for data models evolve over the time, and that may turn old data not compliant with the schema. To avoid incompatibility between data and schema, the data needs to be transformed. A commonly used tool to transform the data is Extensible Stylesheet Language Transformations (XSLT). However, for very large amounts of data, XSLT requires too much memory and time to perform the transformation. This work aims to find a solution to transform the data produced by a stand alone software planning tool for optical networks.The data produced can reach sizes larger than 3GB, and XSLT approaches are found to take around 30 minutes to execute one modification in such a large data file. Considering 20 modifications, it would require around 10 hours. However, there is the possibility of merging all the 20 modifications in only one, reducing the time for completing the data transformation for around 30 minutes. Despite the improvement, the time and memory consumption (16GB) are still not satisfactory. To solve this problem, a new system based on stream XML parsing was created. The stream system is capable of transforming the large data in around 3 minutes and its memory consumption is residual.

Keywords: transformation, migration, schema, evolution, model

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Extensible Markup Language (XML)[1] is most commonly used for dissemination of information over the Internet. A well formed XML document needs to follow some basic rules, however those rules do not define any structure for the data content. A schema[2] is used to define the structure of data inside XML documents. Schemas express shared vocabularies and allow machines to carry out rules made by people. Schemas force the data in XML documents to have a structure, and provide automatic mechanisms the possibility of understanding the structure defined by humans. As stated by [1], the schema may evolve over the time and that evolution often leads to incompatibilities with the previous schema. In other words, XML documents that are compliant with a schema, may not be compliant with the evolution of the same schema. Figures 1.1 and 1.2 represent Schemas that define the structure of XML documents containing information about a world.

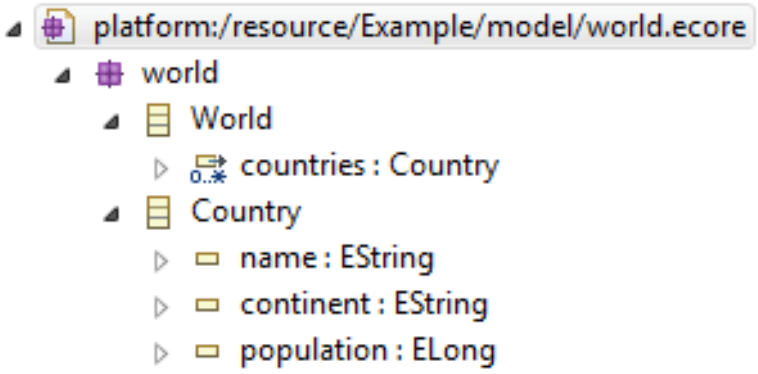

Figure 1.1: Version 1 of XML Schema to represent the world

In figure 1.1, the world has several countries. Each country is represented by its name, continent and the number of persons that live in the country (represented by the attribute population). The following XML document is an example of a world compliant with the XML Schema represented in figure 1.1.

Listing 1.1: XML document compliant with figure 1.1

```
<?xml version="1.0" encoding="UTF-8"?>
```

[1] http://www.w3.org/XML/
[2] http://www.w3.org/XML/Schema

```
<thesis:World xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:thesis="http://thesis/worldv1">
        <countries name="Portugal" continent="Europe" population="10000000"/>
        <countries name="Spain" continent="Europe" population="48000000"/>
        <countries name="Brazil" continent="America" population="200000000"/>
</thesis:World>
```

Considering now that the schema evolves to schema in figure1.2 so it allows the representation of persons. Note that population attribute is no longer required because that information can be calculated by the sum of persons in the country. Due to the disappearing of population attribute, the document compliant with figure 1.1 is not compliant with figure 1.2.

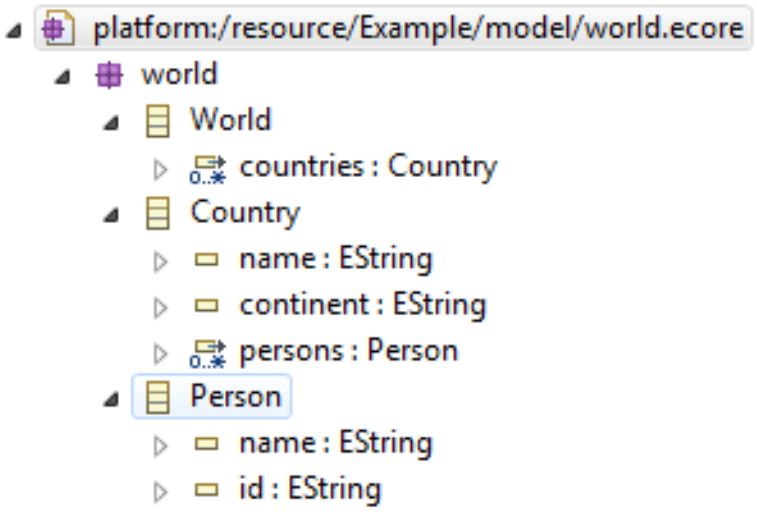

Figure 1.2: Version 2 of XML Schema to represent the world

In order to establish compatibility between XML document in listing 1.1 and schema in figure 1.2, the attribute population must be deleted from the document.
Even though the data is not lost, it may not be possible to process it with automatic mechanisms because they depend on the schema to make sense of the data. A human could read the data and make sense of it, however that is not a valid solution for software that require the data rapidly, and also because the data in XML documents can be massive and for that reason not processable by humans. Therefore, the solution for this problem relies on automatic mechanisms to transform XML documents.
There are several automatic mechanisms capable of performing transformations on XML documents, such as Extensible Stylesheet Language Transformations (XSLT)[3] that will be reviewed further ahead in section 2.1. However, with those approaches, for very large XML documents the time and memory required to perform transformations may ascend to excessively high values.

[3]http://www.w3.org/TR/xslt

1.1 Motivation

Eclipse Modeling Framework (EMF)[4] is a Domain-Specific Visual Language as described in [2]. It provides a platform to create a schema for a model, and generates the model code. It handles the writing and reading of the model as an XML document. EMF is intended to be integrated in eclipse based applications. Figures 1.1 and 1.2 are produced using EMF's user interface. As an example, the code generated by EMF for the country concept represented in the schema from figure 1.1 is in appendix A. For a more in depth explanation on how EMF is used to build applications, please refer to [3].
For a Major Telecommunications Company (referred as MTC from this point onwards), that manufactures and sells equipment for the optical fiber Long Haul and metro markets, the main business is to provide complete solutions to build optical fiber networks (hardware and software for networking management). A very important step in this business is to plan the network to meet client expectations, providing competitive solutions on price, network extensibility, customization, fault tolerance, etc. To support the network planning, the MTC develops an Optical Planning Tool (OPT). OPT is an eclipse based application and uses EMF to store the network configuration in XML format, hereinafter designated SPT files. A SPT file stays for the OPT in a manner similar to how a DOC file stays for Microsoft Word.
In order to support new features in the OPT, the schema evolves and the legacy SPT files become incompatible with the schema. The OPT uses a mechanism based on XSLT transformations to make the SPT file compatible with the new schema. With the increase in both transformations and data size, the required time and memory to open SPT files rise sharply. For some of the biggest SPT files the time reached an estimate of 15 hours and the memory escalated to 16GB. Thus, the motivation for this work is to reduce the memory and time spent by OPT to open SPT files.

1.2 Contributions

This work aims to provide a solution for OPT problem when loading large SPT files. To accomplish that objective, the following studies where performed:

- Relationship between schema elements and content in the XML documents written by EMF.
- What kind of transformation may occur in OPT (or EMF since OPT uses EMF to define the schema and deal with persistence of SPT files) and what transformations are actually required.
- What are the possible solutions for the problem.
- What are the most promising solutions to solve the problem and compare their performances with the solution originally implemented in OPT:
  - XSLT fusion - fuse the XSLT transformations in a unique transformation
  - Transformation in stream

[4] https://eclipse.org/modeling/emf/

- Build a tool for OPT based on the best performing solution.

Chapter 2

Concepts and Related Work

In this chapter the most relevant concepts will be presented and a succinct explanation about them will be provided, in order to identify what is the concept refers to and why it is important. Also in this chapter will be presented state of the art tools that propose solutions to execute similar tasks and could be used to solve the transformation of XML documents. In the end will be discussed the advantages and disadvantages of those tools and why they where not used as a solution in OPT.

2.1 Concepts

This section will introduce the concepts XML, XSLT and SAX. They are very important concepts in this work, due to the fact that XML defines the documents to be transformed and XSLT is the technology originally used in OPT to perform the transformation. XSLT is also the basis of one of the proposed solutions (XSLT fusion). In the following chapters more information about the concepts will be provided as required for exemplification or explanation purposes.

2.1.1 Extensible Markup Language (XML)

XML[1] is a simple and flexible text format used to store information. It is composed by elements, attributes and text. An element may have child elements or text, and attributes. Attributes have value. The possible combinations are described in listing 2.1.

Listing 2.1: XML language components

```
<elementName attribute1="value1" attribute2="value2">
        <childElementName1 attribute3="value3"/>
        <childElementName2 attribute4="value4">
                text1
        </childElementName2>
        <childElementName3>
```

[1] http://www.w3.org/XML/

```
            text2
        </childElementName3>
</elementName>
```

2.1.2 Extensible Stylesheet Language Transformations (XSLT)

XSLT is a language created to perform transformations in XML documents. The language is based on XML, meaning that the mapping is an XML document. It allows the definition of templates that will be applied to all nodes, attributes or text that satisfy a given match sentence. It requires access to the entire content of the XML document, since it allows the XSLT programmer to specify access to any part of the XML document, which means that the XML document has to be represented in memory. As an example, consider the listing 2.2.

Listing 2.2: XSLT example

```
<xsl:template match="elementaName[@attribute1='value']/attribute2">
        <xsl:attribute name="attribute3">
                <xsl:value-of select="." />
        </xsl:attribute>
</xsl:template>
```

The example is a transformation that would select from all elements named ”elementName” with an attribute named ”attribute1” which has value ”value”, and from those nodes it will replace the attribute named ”attribute2” by ”attribute3” with the same value as ”attribute2”.

2.1.3 Simple API for XML (SAX)

SAX[2] is a stream XML parser tool. It acts by reading the XML file and streaming XML Events that represent the content of the file. The SAX events are:

- Start Element: represents the start of a new element. This event contains information about the Element name and value plus the names and values of the attributes.
- End Element: represents that a previously started element closes.
- Characters: represents the occurrence of characters. Typically, this event occurs to represent an element value, but it may also occur just to represent a new line at the end of an element. It may also occur that more than one event of this kind is emitted in a row due to the occurrence of XML reserved characters.

[2]https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html

2.2 Related Work

The literature refers several tools and concepts to perform transformations on XML documents. In this section, two relevant approaches will be explained: Edapt and STX.

2.2.1 Edapt

Edapt is an eclipse supported tool[3] that aims to provide a solution for coupled evolution of model and metamodel (referred in this work as schema and XML documents). The main idea behind this approach is to maintain a history of changes in the schema and produce immediately the transformations required for each change. Edapt provides the functionality of implementing new schema changes and XML document transformation. In references [4] and [5], the authors present how they solved real scenarios of EMF models evolution. In those works and in [6] is possible to understand the Edapt's graphical interface and how custom migrations (XML documents transformations) are implemented if they are not offered by default.

In reference [7], the authors describe the process of migration (or XML document transformation) as illustrated in figure 2.1. Migrators are produced by the framework when changes in the schema are performed (coupled evolution). It is not explained in the literature what is XML document V1.0. However, in EclipseCon NA 2015[4] the presenter reveals while answering the first question, that the data represented in figure 2.1 as XML document V1.0 is internally loaded through an older version of the schema. This means that the schema is reverted to the XML document version, the model code for that particular version is then generated internally and the XML document is loaded to memory using the code generated. Once in memory, the transformations are performed sequentially until the final version is reached.

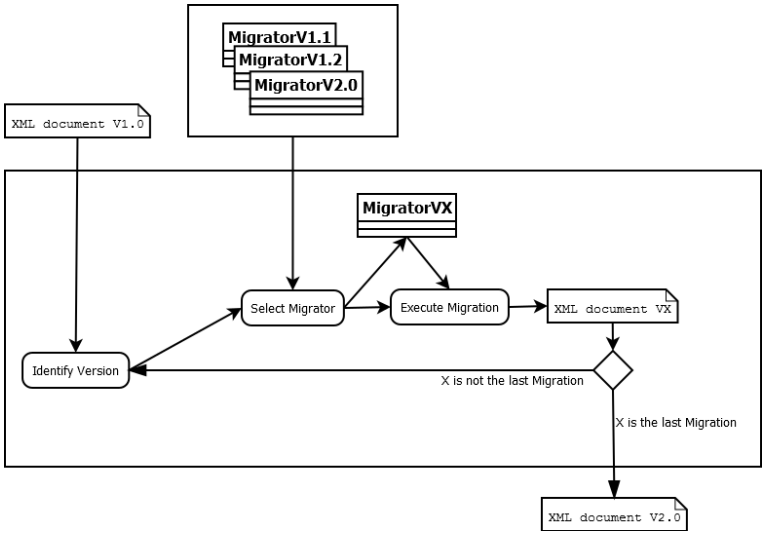

Figure 2.1: Edapt Migration Process

[3]https://www.eclipse.org/edapt/
[4]http://www.infoq.com/presentations/edapt

2.2.2 Streaming Transformations for XML (STX)

STX is an XML transformation language that operates in stream, based on SAX [8]. The goal is to provide a fast and low memory consumption tool to perform transformations on very large XML documents. The language is based on XPath [9], and supports the same operations as XSLT, since they are both turing complete languages [8].
Since STX is based on a stream of information coming from an XML document, it is not possible to have access to all elements. In fact, STX maintains in memory the current node and its attributes, and all its ancestors nodes and their attributes [8].This characteristic is a plus in terms of performance and memory consumption, however the lack of context has to be overcome. To do so, STX provides the ”buffer” feature, which allows to store SAX events to be accessed or processed later. However, maintaining information about other nodes requires memory, which may ruin the performance.

2.3 Critical Discussion

On the process of choosing a solution for the OPT transformation problem, some aspects have to be considered:

- The existing XSLT based mechanism has already a mapping that has to be implemented on the new solution.
- The risk of introducing errors on implementing the existing mapping in the new solution.
- The impact of the mechanism replacement in the application structure.

STX presents a compatible XSLT transformation solution. However, it relies on buffers to keep informations in memory that may affect negatively the system's performance. In addition, besides the current node and its attributes, the STX requires all its ancestors and respective attributes to be available in memory. In the case of OPT this may result in an elevated memory consumption, since it has a very complex schema that results in XML documents with a very deep element hierarchy, containing large pieces of data. Another STX disadvantage is the complex development language, that would require a huge effort on implementing the existing mapping and with high risk of introducing errors.
On the other hand, Edapt is a completely different approach. Instead of transforming the XML documents, it rebuilds the model to load the information to memory so it can perform transformations. Edapt's performance was not studied within the scope of this work because when the work was developed, Edapt was not being supported by Eclipse tools, and at the time implementation did't even work with the latest versions of eclipse. Therefore, it was not a viable solution to be integrated in the development process of OPT. Today, Edapt is a very promising solution for the problem presented in this work, in terms of performance. However, the fact that Edapt rebuilds the old schema and generates the code is a problem for OPT due to the fact that OPT's EMF generated code is not clean and has business logic. Therefore, a clean code generation would introduce undesired transformations in the SPT files. Due to the complexity of the OPT's EMF defined schema, code generation takes a long time, which could spoil

the performance. Another disadvantage of Edapt would be the existing mapping, that would have to be reimplemented in the completely new and different Edapt approach.

Chapter 3

The XML Transformation Solution

The research for this work follows two different paths:

- Identifying what transformations are most commonly used in the scope of EMF defined schemas and XML documents. Stating the relationship of schema modifications and the actual required transformations in the XML document.
- Building bases to test the theoretical solutions to execute performance tests whose results are presented in section 4.

3.1 Required Transformations in EMF Models

In reference [2], the authors describe the evolution of schemas as syntactic or semantic:

- Syntactic: just the required modification of XML documents that make the XML document satisfy the schema rules (in reference [2], the schema is referred as language).
- Semantic: dedicated to maintain the meaning of the XML document after the transformation.

In OPT there is a separate transformation process, executed after EMF loading the document to memory, dedicated to handle semantic transformations. For that reason, this work is dedicated to the syntactic transformations that will prevent EMF from loading the SPT files from XML to memory. The OPT is using EMF to persist data for many years. It is also using XSLT to migrate SPT files whenever it is necessary. Thus, it provides a solid source of information about the transformations that should be prioritized in the new transformation solution.
Based on OPT experience, these are the most commonly necessary transformations in EMF XML documents:

- Change Attribute Name
- Change Attribute Value
- Change Element Name

- Change Element Value
- Delete Attribute
- Delete Element

Change Attribute/Element Name:

There is one operation that requires to change an attributes name in the XML file, which is changing an attribute name in the schema. Consider the example schema in figure 1.1 and the respective compliant XML document represented in listing 1.1. Consider now that, for example, the schema attribute ”continent” is renamed to ”area” like represented in figure 3.1. In order to make the XML document in listing

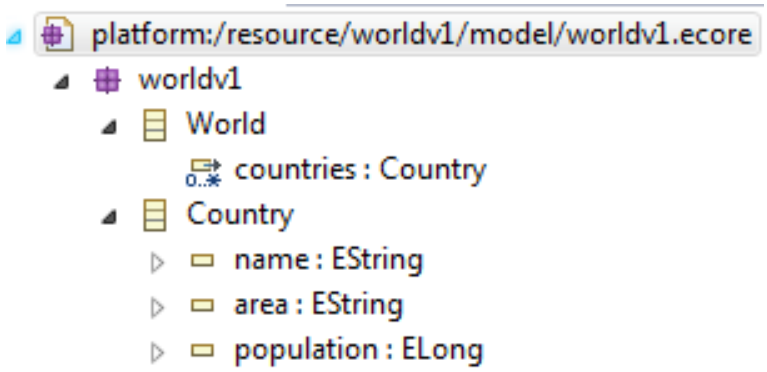

Figure 3.1: Modifications of World XML Schema V1 - changed attribute continent name to area

1.1 compatible with the schema, the XML document attribute ”continent” has to be renamed to ”area”, as represented in listing 3.1

Listing 3.1: XML document with attribute name changed

```
<?xml version="1.0" encoding="UTF-8"?>
<thesis:World xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:thesis="http://thesis/worldv1">
        <countries name="Portugal" area="Europe" population="10000000"/>
        <countries name="Spain" area="Europe" population="48000000"/>
        <countries name="Brazil" area="America" population="200000000"/>
</thesis:World>
```

In the same way, to rename the attribute ”countries” to ”listOfCountries” for example, the transformation would be on the element name, as represented in listing 3.2

Listing 3.2: XML document with element name changed

```
<?xml version="1.0" encoding="UTF-8"?>
<thesis:World xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:thesis="http://thesis/worldv1">
        <listOfCountries name="Portugal" area="Europe" population="10000000"/>
```

```
        <listOfCountries name="Spain" area="Europe" population="48000000"/>
        <listOfCountries name="Brazil" area="America" population="200000000"/>
</thesis:World>
```

Change Attribute/Element Value:

It is required to change an attribute value if the attribute references an enum. As an example, consider the schema in figure 3.2 and compliant XML document in listing 3.2. In the Ecore, enum values have two parameters, name and literal. The name is a reference to the enum in the code, and the literal is the enum representation in the XML document. For this particular case, it was considered that the continent literals are: ”EUROPE” and ”AMERICA”.

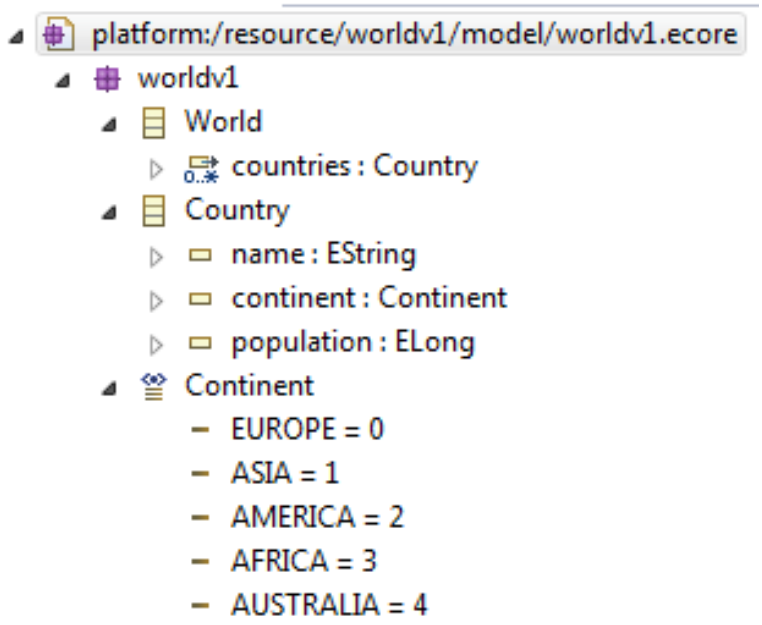

Figure 3.2: Modifications of World XML Schema V1 - Continent becomes represented by an enum

Listing 3.3: XML document compliant with schema in figure 3.2

```
<?xml version="1.0" encoding="UTF-8"?>
<thesis:World xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:thesis="http://thesis/worldv1">
        <countries name="Portugal" continent="EUROPE" population="10000000"/>
        <countries name="Spain" continent="EUROPE" population="48000000"/>
        <countries name="Brazil" continent="AMERICA" population="200000000"/>
</thesis:World>
```

Consider now that in the schema the enum literals are changed to ”Europe” and ”America”. EMF will not be able to convert those new literals to the respective enums when loading the XML document to memory. Therefore, the values of XML document attributes ”continent” have to be changed accordingly. The Change Element Value follows the same principle, but for a list of enums. Consider for example, that

the schema includes a list of continents for a world. Listing 3.4 represents an XML document with that information. In this case, the value of the elements ”continent” would have to be changed accordingly.

Listing 3.4: XML document with list of enums

```
<?xml version="1.0" encoding="UTF-8"?>
<thesis:World xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:thesis="http://thesis/worldv1">
        <continents>EUROPE</continents>
        <continents>AMERICA</continents>
        <countries name="Portugal" continent="EUROPE" population="10000000"/>
        <countries name="Spain" continent="EUROPE" population="48000000"/>
        <countries name="Brazil" continent="AMERICA" population="200000000"/>
</thesis:World>
```

Delete Attribute/Element:

Transformations to delete elements or attributes are required when attributes are removed from the schema. Considering the schema in listing 3.2, deleting the attribute ”countries” from the class world would result in removing an element from the XML document, but deleting the attribute ”name” from the class country would result in removing the attribute ”name” from each element named ”countries” in the XML document.

3.2 XSLT Fusion

The main idea behind the XSLT Fusion concept is that a mapping can be modified automatically in order to execute as many transformations at once as possible. As an example of XSLT Fusion, consider the two transformations represented in listings 3.5 and 3.6:

Listing 3.5: XSLT example transformation - transforms attribute value ”Europe” in ”EUROPE”

```
<xsl:template match="@*|node()">
        <xsl:copy>
                <xsl:apply-templates select="@*|node()" />
        </xsl:copy>
</xsl:template>

<xsl:template match="countries[@continent='Europe']/continent">
        <xsl:attribute name="continent">
                <xsl:value-of select="EUROPE" />
        </xsl:attribute>
</xsl:template>
```

Listing 3.6: XSLT example transformation - transforms attribute value ”America” in ”AMERICA”

```
<xsl:template match="@*|node()">
	<xsl:copy>
		<xsl:apply-templates select="@*|node()" />
	</xsl:copy>
</xsl:template>

<xsl:template match="countries[@continent='America']/continent">
	<xsl:attribute name="continent">
		<xsl:value-of select="AMERICA" />
	</xsl:attribute>
</xsl:template>
```

These two transformations can be executed one at a time. However, in order to improve the performance these two transformations can be replaced by one transformation that transforms the XML documents exactly the same way as exemplified in listing 3.7.

Listing 3.7: XSLT example transformation - transformations merged

```
<xsl:template match="@*|node()">
	<xsl:copy>
		<xsl:apply-templates select="@*|node()" />
	</xsl:copy>
</xsl:template>

<xsl:template match="countries[@continent='America']/continent">
	<xsl:attribute name="continent">
		<xsl:value-of select="AMERICA" />
	</xsl:attribute>
</xsl:template>

<xsl:template match="countries[@continent='America']/continent">
	<xsl:attribute name="continent">
		<xsl:value-of select="AMERICA" />
	</xsl:attribute>
</xsl:template>
```

In order to test the performance of a solution based on XSLT fusion, the XSLT OPT original solution was used. The work done was to rewrite the mapping manually to perform a series of transformations together as one, as exemplified above. This way, the XSLT Transformer reduces the amount of times the XML content is loaded, transformed and written, reducing the time spent to accomplish the task of transforming the XML document.

3.3 Stream Prototype

In the scope of this work, the term stream is used to refer that the data is read from the XML document and processed element by element. It is not a real stream of data, it is in fact a stream of XML elements. Only one XML element is read to memory, transformed, stored in the destination and then discarded from memory before the next XML element is read to memory (there is no context to perform the transformation or even recognize that a transformation is required). Child elements are read separately from its parent element.

This raises some complications, for example if some transformation action requires any data that was already processed. To overcome this problem, the notion of round was implemented in this prototype. Each round consists on executing the mapping transformations in the entire model, and then, feed the resulting transformed model as input of the next round. Each round has its own mapping. In section 3.3.1 is explained how rounds and its mapping are specified.

Another problem associated with not having the context available in memory, is performing a transformation in more than one element, for example, deleting an element and its child elements. Section 3.3.2 explains how this problem was solved.

This prototype aims to perform not only the transformations described in section 3.1, but also to provide the possibility of functionality extension. Section 3.3.2 explains how this was accomplished.

The prototype to test the performance and feasibility of this solution was built integrated in OPT, transforming SPT files.

3.3.1 Mapping Specification Language (MSL)

In this prototype, the mapping is described as an XML document. The MSL is a schema that provides a structure to describe transformation mappings. The rules defined by MSL are the following:

- The root element must be named “migrations”.
- Inside the root element only elements with the name “migration” are allowed. At least one migration must exist.
- It is mandatory that each “migration” element has an attribute named action.
- Each migration may have the following non-mandatory attributes:

  input

  inputDelimiter
- Each migration may have the following non-mandatory ”match attributes”:

  matchParentElementName

  matchElementName

  matchAttributeName

  matchAttributeValue

- Each migration may have an arbitrary number of child elements named ”round”.
- If round elements exist, then migration elements must not have ”match attributes”.
- Each round element may have the ”match attributes” indicated above for migration.

As an example, listing 3.8 shows a mapping specified as MSL that would transform the XML document in order to cope with the schema modifications described in section 3.1. The schema modifications are the following:

1. changing attribute ”continent” name to ”area”
2. changing attribute ”countries” name to ”listOfCountries”

Listing 3.8: Example of a mapping defined in MSL

```
<migrations>
        <migration matchElementName="countries" matchAttributeName="continent action="
            ChangeAttributeName" input="continent;area"/> <!-- 1 -->
        <migration matchElementName="countries" action="ChangeElementName" input="
            listOfCountries"/> <!-- 2 -->
        <migration action="MoveElement"> <!-- 3 -->
                <round matchElementName="countries" matchAttributeName="name"
                    matchAttributeValue="Brazil"/>
                <round matchElementName="World"/>
        </migration>
</migrations>
```

Each ”element migration” (migration) is a specification of a transformation. Migration number 1 specifies that, in the XML document, if an element is named ”countries” (matchElementName=”countries”) and it has an attribute named ”continent” (matchAttributeName=”continent”), then an attribute must be renamed (action=”ChangeAttributeName”), and that attribute is currently called ”continent” and must be renamed to ”area” (input=”continent;area”). There are two distinct parts in all migrations: matching clause and action.

Matching Clause

A matching clause of a migration answers the question: ”which XML nodes must be transformed by the action?”. It is composed by the MSL match attributes (matchParentElementName, matchElementName, matchAttributeName and matchAttributeValue). Those attributes are not mandatory, however at least one must be defined, otherwise the action is not applicable to any XML node. In order to have a match (an XML element that fulfills the matching clause), all match attributes must be satisfied. The absence of a match attribute means that any value is accepted.
If both ”matchAttributeName” and ”matchAttributeValue” are defined in the same matching clause, then

they have to be satisfied by the same attribute. For example, if a migration matching clause defines matchAttributeName=”a” and matchAttributeValue=”x”, then an element with attribute a=”x” is accepted, but an element with attributes a=”y” and b=”x” is not.
MSL attribute matchParentElementName accepts elements whose parent element has the name equal to its value.
MSL attribute matchElementName accepts elements whose name is equal to its value.
MSL attribute matchAttributeName accepts elements that contain an attribute whose name is equal to its value.
In listing 3.8, migration number 3 specifies two rounds. In this case, the matching clause defined in the first MSL element ”round” is used to match XML elements while executing the first migration round, the matching clause defined in the second MSL element ”round” is used to match XML elements while executing the second round and so on. For the example of migration number 3, considering the XML document described in listing 3.1, the idea is that in the first round the action would remove the element that represents Brazil and save the data in memory, whereas in the second round, it would put the element as first child of the root element ”world”.

Action

The action part is composed by the attributes ”action”, ”input” and ”inputDelimiter”. The attribute ”action” defines the transformation to be done on the match elements. The attribute ”input” defines certain parameters required to perform the ”action”. For example, in the scope of the action ”ChangeAttributeName”, input=”continent;area” means that the attribute called ”continent” must be renamed to ”area”. This is a consequence of the fact that the action part of the migration is separated from the matching clause. This means that an action can be defined to transform attributes in the matched XML element that are not referred in the matching clause. For example, consider the same schema and XML document defined in figure 3.1, and that for some arbitrary reason Portugal duplicated its population. One possible MSL mapping that performs the described transformation is represented in listing 3.9

Listing 3.9: MSL mapping matching one attribute and changing another one

```
<migrations>
        <migration matchElementName="countries" matchAttributeName="name"
            matchAttributeValue="Portugal" action="ChangeAttributeValue" input="
            population;10000000;20000000"/>
</migrations>
```

Note that in this example the modification would not require a transformation in EMF, because it would not change the schema. This serves just as an illustrative example of the ”input” attribute functionality.
The value of the MSL attribute ”input” uses the character ”;” to separate different parameters (delimiter). However, there is a chance that the delimiter exists in one of the strings that composes the parameters. If that happens, the delimiter can be changed using the MSL attribute ”inputDelimiter”. If ”inputDelimiter” is not defined, the default delimiter ”;” is used.

Stream prototype provides ”actions” to perform the transformations identified in section 3.1. As referred before, each action requires a certain input content in order to have complete information about the transformation:

- ChangeAttributeName - Receives two arguments as input: “old attribute name” and “new attribute name”. Finds in the element the attribute with name equal to “old attribute name” and replaces it by “new attribute name”.

- ChangeAttributeValue - Receives three arguments as input: “attribute name”, “old attribute value” and “new attribute value”. Finds in the element the attribute with name equal to “attribute name” and if the value of the attribute is equal to “old attribute value” replaces it by “new attribute value”. If “old attribute value” is empty, the “old attribute value” is not checked.

- ChangeElementName - Receives one argument as input: “new element name”. Changes the element name to “new element name”.

- ChangeElementValue - Receives one argument as input: “new element name”. Changes the element name to “new element name”.

- DeleteAttribute - Receives as input one argument: “attribute name”. Finds the attribute with name equal to “attribute name” and removes it from the element.

- DeleteElement - Does not receive arguments. Deletes the element.

Additionally, some extra transformations were implemented with the purpose of testing the extensibility of the prototype. They do not represent actual EMF required transformation cases:

- ChangeAttributesValues - Receives two arguments as input: “old value” and “new value”. Checks all attributes of the element and replaces all values equal to “old value” by “new value”.

- ChangeChildElementsValue - Receives three arguments as input: “name to match”, “value to match” and “new value”. When matched with an element, changes the value of every child element with name equal to “name to match” and value equal to “value to match” to “new value”. If “name to match” is empty, the “name to match” is not checked. If “value to match” is empty, the “value to match” is not checked.

- MoveAttributeToElementValue - Receives one argument as input: “attribute name”. Gets the attribute value of the attribute with name equal to “attribute name” and sets it as the current element value. If the current element has child elements, deletes them.

- TransformAttributeInChildElement - Receives one argument as input: “attribute name”. Gets the “attribute value” from the attribute with name equal to “attribute name” and creates a child element with name “attribute name” and value “attribute value”. Deletes the attribute with name equal to “attribute name” from the current element.

3.3.2 Architecture

The prototype is composed by three distinct parts with different responsibilities:

- Engine
- Manager
- Transformers

These components are represented in figure 3.3 as a class diagram.

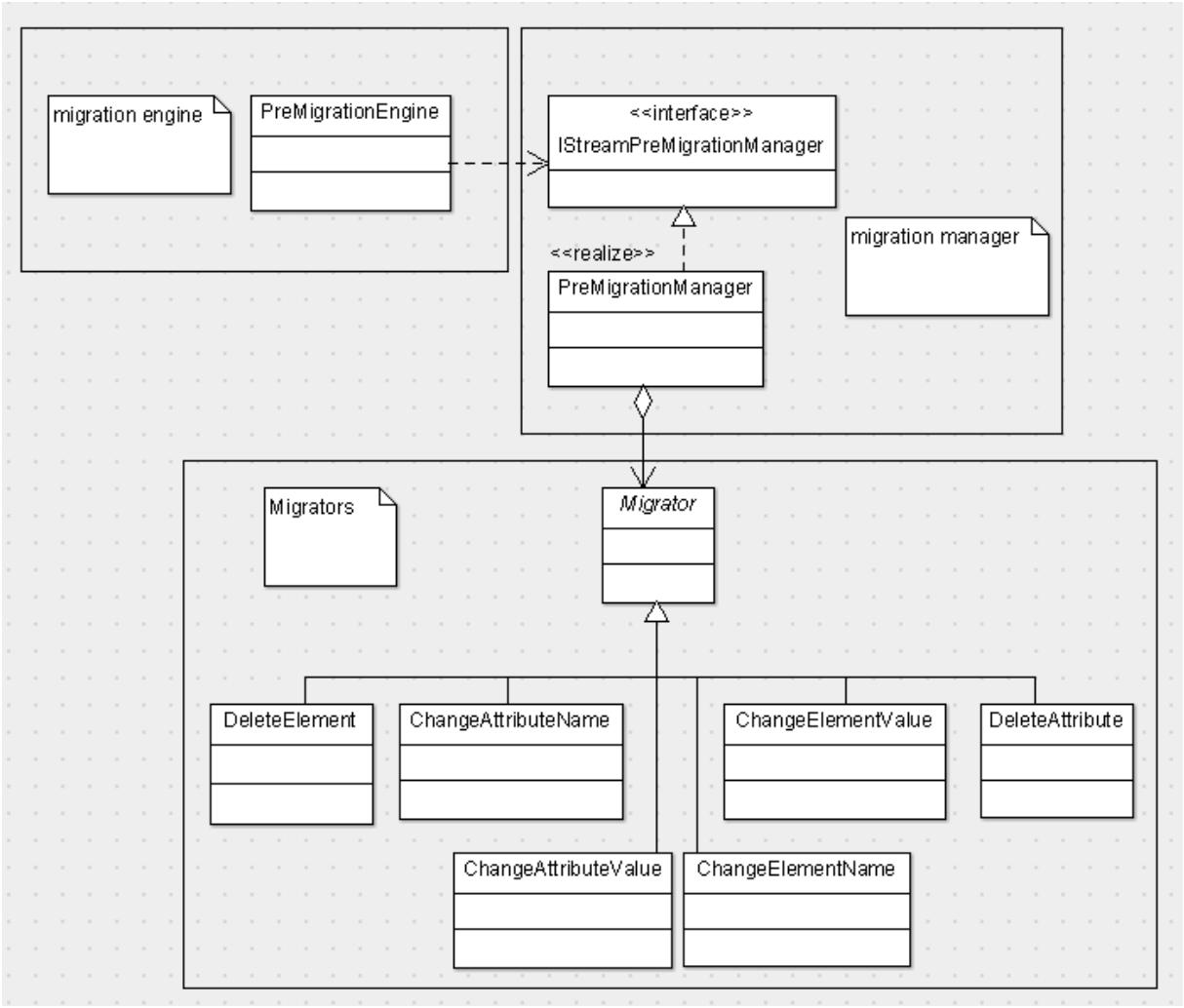

Figure 3.3: Representation of the stream prototype parts as a class diagram

The components communicate using a Data Transfer Object (DTO) named ElementDTO. It is used to represent an XML element from the XML document and contains the following information:

- Type: represents the type of the ElementDTO. “type” can have three values:

  Start Element: indicates that ElementDTO is representing an XML Element that has child elements and later on, it will exist an ElementDTO representing the end of the current XML Element. ElementDTOs of this type contain the XML attributes if they exist.

End Element: indicates that the current ElementDTO is ending a previously started XML Element. ElementDTOs of this type do not contain XML attributes. The attributes of the element ending with this ElementDTO were represented in the ElementDTO with type “start element” that represented in the past, the start of the XML element.

Element: represents an XML Element that has no child elements. This element may have a value associated and there will not exist an ElementDTO representing the end of the current XML Element. Type “Element” represents the start and the end of the XML Element. ElementDTOs of this type contain the XML attributes if they exist.

- Parent Element Name: represents the name of the parent of the current XML Element.
- Element Name: represents the name of the current XML Element.
- Element Prefix: represents the prefix of the current XML Element if it exists.
- Element Namespace: represents the name space of the XML Element if it exists.
- Text: represents the XML Element value if it exists.
- Element Namespaces: a list representing all the namespaces defined by attributes with prefix “xmlns” if they exist in the XML Element.
- Element Namespaces Prefixes: a list representing all namespaces prefixes if they exist in the current XML Element.
- Attributes Names: a list representing the names of the attributes of the current XML Element. The names are in the list by the same order they are in the XML Element.
- Attributes Values: a list with the values of the attributes of the current XML Element. The values are in the list by the same order they are in the XML Element.
- Attributes Namespaces: a list with the namespaces of the attributes of the current XML Element. There is, at most, one namespace per attribute. The namespaces are in the list by the same order they are in the XML Element.
- Attributes Prefixes: a list with the prefixes of the attributes of the current XML Element. There is, at most, one prefix per attribute. The prefixes are in the list by the same order they are in the XML Element.
- Skip: if set to “true” means that the element must not be written to the result. In other words, it will be deleted.

Engine

The migration engine is responsible for parsing the input SPT file using SAX, and creating round result files with the result of executing the transformations of each round. The result file of each round will be

the input of the next round. The result file of the last round will be delivered to the application and then EMF will read the data to memory.

While processing a round, the migration engine will gather the information from XML Events and builds an ElementDTO when it has the complete information to represent the XML element. Next, it sends the ElementDTO to the migration manager, which will return a list of ElementDTO as a result of the transformation. Then, the engine writes the element received to the round result file. Figure 3.4 illustrates the sequence of actions performed by the migration engine.

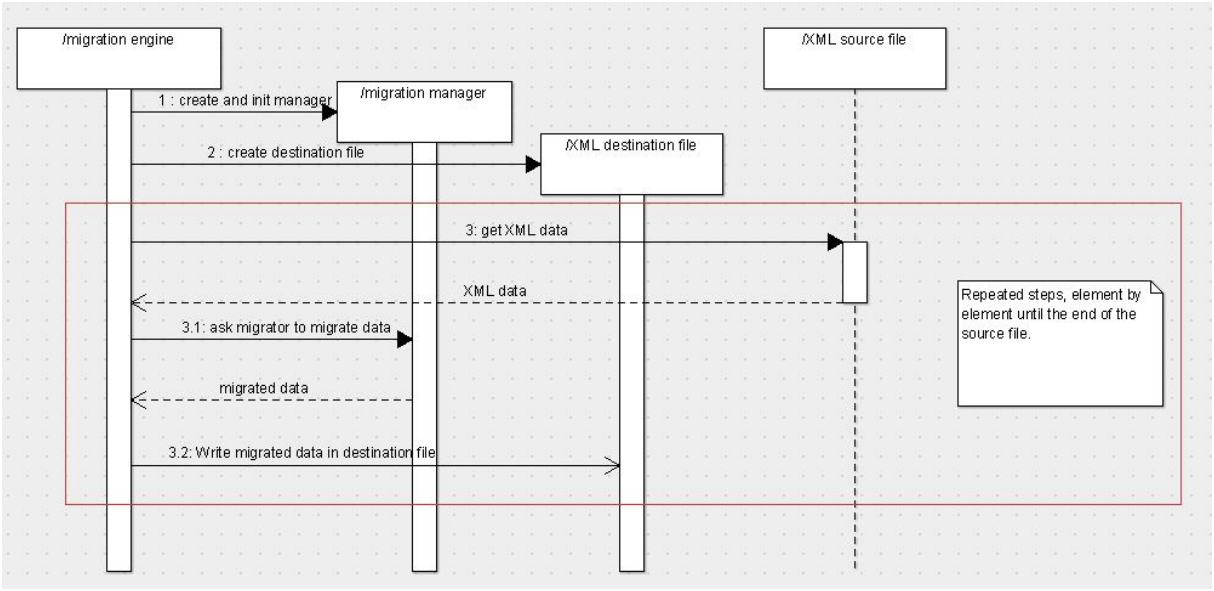

Figure 3.4: Representation of the actions performed by the migration engine

Migration Manager

The Migration Manager is the component responsible for recognizing that a transformation has to be performed in an given ElementDTO (previously called a match). Therefore, it requires the information from the MSL mapping and it needs to store that information in a structure that has to be efficient in terms of time spent to recognize a match. From now on, this structure will be called Matching Data Base (MDB) and the process of populating the Matching Data Base with the information from the MSL mapping will be called MSL Ingestion. MSL Ingestion is responsible for reading MSL matching clauses and representing them in MDB. For each round, the MDB will contain only the round respective matching clauses, which means that MSL Ingestion is executed once per round.

MDB is composed by a Java class named KeyNode and a map of String in KeyNode (Java HashMap for performance reasons). The KeyNode class contains a list of actions (classes that perform transformation on ElementDTOs) and a map from String to KeyNode. The root structure of the MDB is the map. Figure 3.5 illustrates the relation between the classes that compose the MDB structure. The algorithm of MSL Ingestion uses the classes in figure 3.5 to build a tree in which the root is the map and the leaves are KeyNodes. For demonstration, consider the MSL mapping represented in listing 3.10.

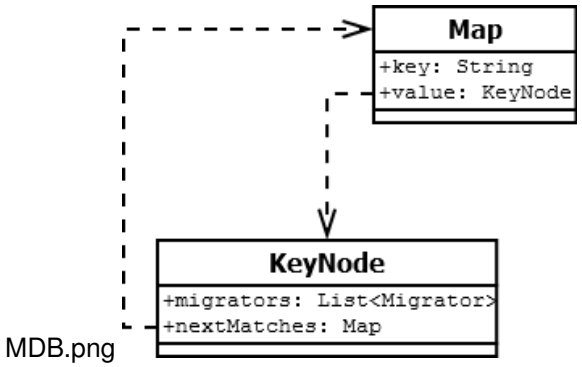

Figure 3.5: MDB Class Diagram

Listing 3.10: MSL Mapping Sample

```
<migrations>
        <migration matchElementName="countries" matchAttributeName="continent action="
            ChangeAttributeName" input="continent;area"/> <!-- 1 -->
        <migration matchElementName="countries" matchAttributeValue="Europe" action="
            ChangeAttributeValue" input="population;10000000;20000000"/> <!-- 2 -->
</migrations>
```

Figure 3.6 ilustrates the memory structure built by MSL Ingestion algorithm resultant of the mapping in listing 3.10.

XML element names, attributes names and attributes values may be equal. Therefore, there is the need to identify that the ”countries” is a match for an Element name, ”continent” is a match for an attribute name and so on. To do so, when building the MDB, the matching strings must be prefixed with one of the following values:

- PEN: stands for Parent Element Name
- EN: stands for Element Name
- EV: stands for Element Value
- AN: stands for Attribute Name
- AV: stands for Attribute Value

The order in which the match attributes are disposed in the matching clause is not important, as they must be placed in the MDB respecting the order:

1. matchParentElementName
2. matchElementName
3. matchElementValue
4. matchAttributeName

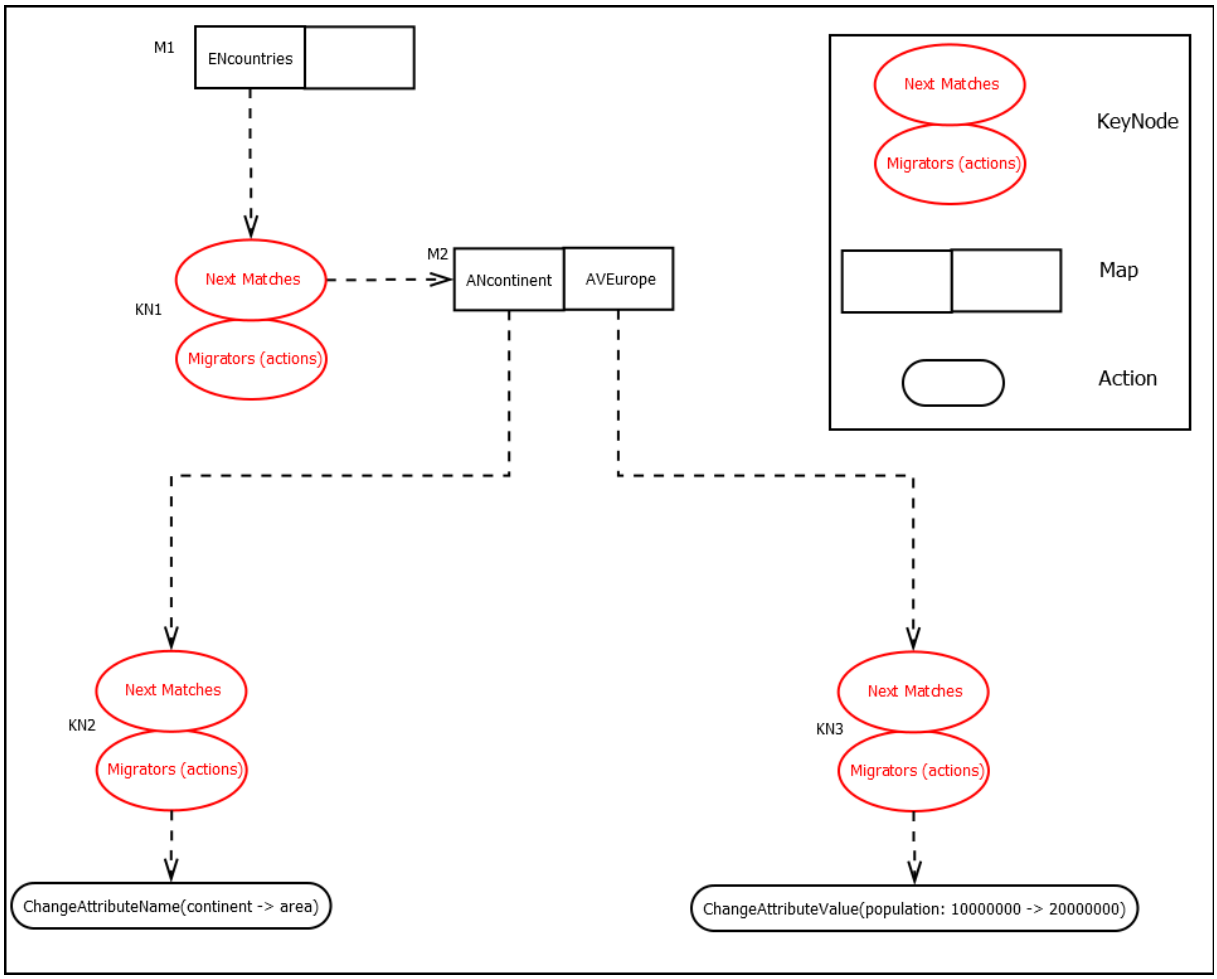

Figure 3.6: MDB in Memory Structure

5. matchAttributeValue

None of the match attributes is mandatory, so the order must be respected if all match attributes are defined, if not, the next attribute in the order should be used. For example, in figure 3.6, there is no migration defined with match attribute ”parent element name”, therefore, the first entry in the map is the next available match attribute (”element name” for both migrations).

The matching process relies on two algorithms, the MSL ingestion by which the MDB is built and populated; and the matching algorithm by which the Migration Manager identifies which are the Migrators that must be executed in each ElementDTO. The MSL ingestion algorithm is represented in the pseudo code in listing 3.11. Note that, in the first phase the algorithm searches for the right KeyNode and creates a new one if it does not exist, and in the second phase it adds the Migrator to the created KeyNode. This step is repeated for all matchings defined in the MSL mapping.

Listing 3.11: MSL ingestion algorithm

```
void ingestToKeyNodes(String parentElementName, String elementName, String
    elementValue, String attributeName, String attributeValue, Map<String, KeyNode>
    map, Migrator migrator) {

        List<String> keysToMatch = getParametersToMatch(parentElementName, elementName,
             elementValue, attributeName, attributeValue);

        String key;
        Map<String, KeyNode> currentMap = map;

        /* First Phase - search in the MDB for the new KeyNode and creates if it does
            not exist */
        for (int i = 0; i < keysToMatch.size(); i++) {
                key = keysToMatch.get(i);
                if (currentMap.containsKey(key)) {
                        KeyNode node = currentMap.get(key);
                        currentMap = node.getNextMatches();
                } else {
                        KeyNode newNode = new KeyNode();
                        currentMap.put(key, newNode);
                        node = newNode;
                        currentMap = node.getNextMatches();
                }
        }
```

```
        // Second Phase - add migrator to correct KeyNode
        node.addMigrator(migrator);
}
```

Once the MDB is populated, the migration manager will use the MSL Matching algorithm whose main goal is to identify which Migrators should be applied on the XML element represented as ”dto” in the pseudo code. The algorithm starts in the method ”getMatchingMigrators” in listing 3.12, which will consider that a match may start in any of the matching attributes (parentElementName, elementName, elementValue, attributeName or attributeValue) because in the MSL none of the matching attributes is mandatory, and adds the matching Migrators to the ”result” list.

Listing 3.12: getMatchingMigrators method pseudo code

```
List<Migrator> getMatchingMigrators(ElementDTO dto) {
        List<Migrator> result;
        // adding prefixes to identify the kind of the key.
        String parentElementName = PARENT_ELEMENT_NAME_PREFIX + dto.
            getParentElementName();
        String elementName = ELEMENT_NAME_PREFIX + dto.getElementName();
        String elementValue = ELEMENT_VALUE_PREFIX + dto.getText();
        List<String> attributesNames = dto.getAttributesNames();
        List<String> attributesValues = dto.getAttributesValues();
        // get the MDB root's map
        Map<String, KeyNode> mdbRoot = getMDBRoot();

        /* when searching for matches in the root map, all matching attributes must be
            checked */
        result.addAll(matchParentElement(parentElementName, elementName, elementValue,
            attributesNames, attributesValues, mdbRoot));
        result.addAll(matchElementName(elementName, elementValue, attributesNames,
            attributesValues, mdbRoot));
        result.addAll(matchElementValue(elementValue, attributesNames, attributesValues
            , mdbRoot));
        result.addAll(matchAttributesNames(attributesNames, attributesValues, mdbRoot))
            ;
        result.addAll(matchAttributesValues(attributesValues, mdbRoot));
        return result;
}
```

The first step executed by ”getMatchingMigrators” represented in listing 3.12 is to check if the parent element name of the ”dto” has a match in the ”mdbRoot”. To do so, it uses the method ”matchParentElement” represented in the pseudo code in listing 3.13.

Listing 3.13: matchParentElement method pseudo code

```
List<Migrator> matchParentElement(String parentElementName, String elementName, String
     elementValue, List<String> attributesNames, List<String> attributesValues, Map<
    String, KeyNode>  currentMDBMap) {
        List<Migrator> result;

        if (currentMDBMap.containsKey(parentElementName)) {
                KeyNode keyNode = currentMDBMap.get(parentElementName);
                /* when searching for matches in a map that results from the match of
                    the parent element name, only the following attributes must be
                    checked */
                result.addAll(keyNode.getMigrators());
                result.addAll(matchElementName(elementName, elementValue,
                    attributesNames, attributesValues,
                keyNode.getNextMatches()));
                result.addAll(matchElementValue(elementValue, attributesNames,
                    attributesValues, keyNode.getNextMatches()));
                result.addAll(matchAttributesNames(attributesNames, attributesValues,
                    keyNode.getNextMatches()));
                result.addAll(matchAttributesValues(attributesValues, keyNode.
                    getNextMatches()));
        }
        return result;
}
```

The method ”matchParentElement” checks if the ”mdbRoot” contains the parent element name of ”dto”, if it does not the search stops and an empty list of Migrators is returned (note that next, the algorithm will try starting the match with the other matching attributes). If it does, then it gets the KeyNode from the ”mdbRoot”, adds the Migrators in the KeyNode to the ”result” and continues the searching process for the next attributes (elementName, elementValue, attributeName and attributeValue). To do so, it calls the methods ”matchElementName”, ”matchAttributesNames”, ”matchAttributeValue” and ”matchAttributesValues” represented in listings 3.14, 3.15, 3.16 and 3.17 respectively. Those methods work in a similar way.

Listing 3.14: matchElementName method pseudo code

```
List<Migrator> matchElementName(String elementName, String elementValue, List<String>
    attributesNames, List<String> attributesValues, Map<String, KeyNode> currentMDBMap
    ) {
        List<Migrator> result;
```

```
        if (currentMDBMap.containsKey(elementName)) {
                KeyNode keyNode = currentMDBMap.get(elementName);
                /* when searching for matches in a map that results from the match of
                    the element name, only the following attributes must be checked */
                result.addAll(keyNode.getMigrators());
                result.addAll(matchElementValue(elementValue, attributesNames,
                    attributesValues, keyNode.getNextMatches()));
                result.addAll(matchAttributesNames(attributesNames, attributesValues,
                    keyNode.getNextMatches()));
                result.addAll(matchAttributesValues(attributesValues, keyNode.
                    getNextMatches()));
        }
        return result;
}
```

Listing 3.15: matchAttributesNames method pseudo code

```
List<Migrator> matchAttributesNames(List<String> attributesNames, List<String>
    attributesValues, Map<String, KeyNode> currentMDBMap) {
        List<Migrator> result;

        if (attributesNames == null) {
                return result;
        }

        for (int i = 0; i < attributesNames.size(); i++) {
                String attributeName = ATTRIBUTE_NAME_PREFIX + attributesNames.get(i);
                if (currentMDBMap.containsKey(attributeName)) {
                        KeyNode keyNode = currentMDBMap.get(attributeName);
                        /* when searching for matches in a map that results from the
                            match of an attribute name, only the following attributes
                            must be checked */
                        result.addAll(keyNode.getMigrators());
                        String attributeValue = attributesValues.get(i);
                        result.addAll(matchAttributeValue(attributeValue, keyNode.
                            getNextMatches()));
                }
        }
        return result;
}
```

Listing 3.16: matchAttributeValue method pseudo code

```
// used only when an attribute name is matched
List<Migrator> matchAttributeValue(String attributeValue, Map<String, KeyNode>
    currentMDBMap) {
        List<Migrator> result;
        String value = ATTRIBUTE_VALUE_PREFIX + attributeValue;

        if (currentMDBMap.containsKey(value)) {
                KeyNode keyNode = currentMDBMap.get(value);
                result.addAll(keyNode.getMigrators());
        }

        return result;
}
```

Listing 3.17: matchAttributesValues method pseudo code

```
List<Migrator> matchAttributesValues(List<String> attributesValues, Map<String,
    KeyNode> currentMDBMap) {
        List<Migrator> result;

        if (attributesValues == null) {
                return result;
        }

        for (int i = 0; i < attributesValues.size(); i++) {
                String attributeValue = ATTRIBUTE_VALUE_PREFIX + attributesValues.get(i)
                    ;
                if (currentMDBMap.containsKey(attributeValue)) {
                        KeyNode keyNode = currentMDBMap.get(attributeValue);
                        List<Migrator> matchedMigrators = keyNode.getMigrators();
                        result.addAll(matchedMigrators);
                }
        }

        return result;
}
```

To clarify, a practical example will be reproduced bellow:

Listing 3.18: XML document sample

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<thesis:World xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:thesis="http://thesis/worldv1">
        <countries name="Portugal" continent="Europe" population="10000000"/>
        <countries name="Spain" continent="Europe" population="48000000"/>
        <countries name="Brazil" continent="America" population="200000000"/>
</thesis:World>
```

Considering the XML document in listing 3.18 and that the Engine would send ElementDTOs representing the XML elements in listing 3.18 (only elements named ”country” will be represented to make the example more clear and understandable), the Migration Manager would execute the following list of steps:
Element country Portugal:

1. Check ParentElementName ”world” in Map M1 - does not match
2. Check ElementName ”countries” in Map M1 - matches, gathers the actions in KN1 to be executed later and moves to Map M2.
3. Check ElementValue ”empty” in M2 - does not match.
4. Check AttributeName ”name” in M2 - does not match.
5. Check AttributeValue ”Portugal” in M2 - does not match.
6. Check AttributeName ”continent” in M2 - matches, gathers the actions in KN2 to be executed later and moves to empty map.
7. Check AttributeValue ”Europe”, does not match. Moves back to Map M2 to continue the search for attributes names. This is the step that enables the rule ”*If both matchAttributeName and matchAttributeValue are defined in the same matching clause, then they have to be satisfied by the same attribute*” defined in section 3.3.1.
8. Check AttributeValue ”Europe” - matches, gathers the actions in KN3 to be executed later and moves back to M2 because it is currently checking the attribute value. Therefore, no more matchings are possible before that, as defined by the order in which the matching clause is inserted in the MDB.
9. Check AttributeName ”population” in Map M2 - does not match.
10. Check AttributeValue ”10000000” in Map M2 - does not match.

In the end, the gathered actions to be executed in the ElementDTO are: ChangeAttributeName(continent to area) and ChangeAttributeValue(10000000 to 20000000).

Element country Brazil:

1. Check ParentElementName ”world” in Map M1 - does not match

2. Check ElementName ”countries” in Map M1 - matches, gathers the actions in KN1 to be executed later and moves to Map M2.
3. Check ElementValue ”empty” in M2 - does not match.
4. Check AttributeName ”name” in M2 - does not match.
5. Check AttributeValue ”Brazil” in M2 - does not match.
6. Check AttributeName ”continent” in M2 - matches, gathers the actions in KN2 to be executed later and moves to empty map.
7. Check AttributeValue ”America”, does not match. Moves back to Map M2 to continue the search for attributes names.
8. Check AttributeName ”population” in M2 - does not match.
9. Check AttributeValue ”200000000” in M2 - does not match.

In the end, the gathered action to be executed in the ElementDTO is: ChangeAttributeName(continent to area).

Each action is in fact an instance of the class ”Migrator” represented in figure 3.3. Each ”Migrator” knows how to change the ElementDTO in order to perform the desired transformation. After executing all actions in the ElementDTO, a list of ElementDTO is returned to the engine, because the result may be more than one ElementDTO. This will be explained in section 3.3.3.

3.3.3 Actions

As illustrated in figure 3.3, actions are implemented in this prototype as classes that extend the class ”Migrator”. The functionality of the prototype can be extended simply creating a new Class, make it extend ”Migrator” and use it in the MSL mapping. When match occurs, the new Migrator will receive the matched ElementDTO that can be manipulated and then returned to the manager that will deal with it. This extensibility feature allows not only to implement new migration cases, but also to solve some very specific transformations that may occur and are not accounted for.
By default, there are some functionalities provided via the inheritance from ”Migrator” to overcome the lack of context about the XML document, due to the ”stream nature” of this solution. In order to explain those functionalities, consider the XML document represented in listing 3.19 in which the countries have a list of persons.

Listing 3.19: XML document sample

```
<?xml version="1.0" encoding="UTF-8"?>
<thesis:World xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
       xmlns:thesis="http://thesis/worldv1">
       <countries name="Portugal" continent="EUROPE" population="10000000>
```

```
            <person name="Jorge" id="1234" />
            <person name="Maria" id="4321"/>
        </countries>
        <countries name="Spain" continent="EUROPE" population="48000000">
            <person name="Pedro" id="7357" />
        </countries>
        <countries name="Brazil" continent="AMERICA" population="200000000">
            <person name="Carolina" id="5268" />
            <person name="Jorge" id="2065" />
        </countries>
</thesis:World>
```

In order to delete all information about Portugal from the XML document, a transformation needs to delete 4 ElementDTOs representing the element ”Portugal”, ”Jorge”, ”Maria” and the termination of element ”Portugal”. Specifying matching clauses to delete each of those elements is not a valid solution. Specifying a migration to delete all persons named ”Jorge” is not a solution because that way it would also delete the person named Jorge from Brazil, and defining matching clauses to match the ”id” means that all ”ids” would have to be known by the person who specifies the matching clause. So, to solve this problem Migrators have the capability of claiming the next ElementDTOs, meaning that the claimer Migrator will receive all ElementDTOs until it ”unclaimes” them, even if they are not a match. So, the ”DeleteElement” Migrator when matched with the element ”Portugal”, markes it as deleted and claimes the next ElementDTOs. Therefore, it will receive ElementDTOs ”Jorge”, ”Maria” and close ”Portugal”, marks all of them as deleted, and unclaimes the next ElementDTOs.

If instead of deleting the element ”Portugal”, the required transformation was to move it to the last position of the list, then the Migrator would have to claim the ElementDTOs, store them in memory, and in the second round, return all of them at once to the correct place. This is the reason for both Migrators and Migration Manager returning a list of ElementDTOs instead of only one ElementDTO.

Chapter 4

Experiments

OPT is the test bed for the performance comparison of the solutions presented in sections 3.2 and 3.3 with the original XSLT solution of OPT. The experiment consisted in opening SPT files and recording the required time to successfully load the SPT file. The experiment was conducted for the following parameters:

- SPT file sizes: 770MB, 1.07GB, 1.45GB, 1.86GB and 2.06GB
- Virtual machine memory sizes: 4GB, 8GB and 16 GB
- Number of transformations: 1, 5 and 20

The results are expressed in tables 4.1, 4.2, 4.3, 4.4 and 4.5. The analysis of the data in the referred tables reveals very clearly the following facts:

- The time and memory required for the XSLT original solution escalate exponentially with the growth of SPT file size and the number of transformations.
- The time required for the XSLT fusion solution is not impacted by the growth of the transformations number. The number of transformations increases, but the time required to perform the transformation remains constant. However, the growth of the SPT file size affects the time, so that with the duplication of the SPT file size, the time increases 5 times.
- The stream solution is not affected by the number of transformations or virtual memory available. However, the growth of the SPT files has a linear effect on the time required to perform the transformations, so with the duplication of SPT file size, the time also duplicates.

Table 4.1: Experimental results for a SPT file with 770MB

770MB				
	Virtual Memory	4GB	8GB	16GB
Solution	Number of Transformations	Time Spent (minutes)		
XSLT	1	4	3.75	3.75
	5	20	15	15
	20	80	75	75
XSLT Fusion	1	4	3.75	3.75
	5	4	3.75	3.75
	20	4	3.75	3.75
Stream	1	1	1	1
	5	1	1	1
	20	1	1	1

Table 4.2: Experimental results for a SPT file with 1.07GB

1.07GB				
	Virtual Memory	4GB	8GB	16GB
Solution	Number of Transformations	Time Spent (minutes)		
XSLT	1	10	7	7
	5	50	35	35
	20	200	140	140
XSLT Fusion	1	10	7	7
	5	10	7	7
	20	10	7	7
Stream	1	2	2	2
	5	2	2	2
	20	2	2	2

Table 4.3: Experimental results for a SPT file with 1.45GB

1.45GB				
	Virtual Memory	4GB	8GB	16GB
Solution	Number of Transformations	Time Spent (minutes)		
XSLT	1	>300	14.5	13
	5	>300	72.5	65
	20	>300	290	260
XSLT Fusion	1	>300	14.5	13
	5	>300	14.5	13
	20	>300	14.5	13
Stream	1	2	2	2
	5	2	2	2
	20	2	2	2

Table 4.4: Experimental results for a SPT file with 1.86GB

1.86GB				
	Virtual Memory	4GB	8GB	16GB
Solution	Number of Transformations	Time Spent (minutes)		
XSLT	1	>300	25	22
	5	>300	125	110
	20	>300	>300	>200
XSLT Fusion	1	>300	25	22
	5	>300	25	22
	20	>300	25	22
Stream	1	2.7	2.7	2.7
	5	2.7	2.7	2.7
	20	2.7	2.7	2.7

Table 4.5: Experimental results for a SPT file with 2.06GB

2.06GB				
	Virtual Memory	4GB	8GB	16GB
Solution	Number of Transformations	Time Spent (minutes)		
XSLT	1	>300	31	28
	5	>300	155	140
	20	>300	>300	>300
XSLT Fusion	1	>300	31	28
	5	>300	31	28
	20	>300	31	28
Stream	1	3	3	3
	5	3	3	3
	20	3	3	3

Chapter 5

Conclusions

The contributions described in section 1.2 were all satisfied in this work:

- The relationship between the schema transformation and the impacts on the XML document were studied and are described in section 3.1. Six transformations were identified as the most currently used in OPT and most probable to occur in EMF models (Change Attribute Name, Change Attribute Value, Change Element Name, Change Attribute Value, Delete Attribute and Delete Element).
- Two possible solutions to solve OPT problem were studied: XSLT fusion and Stream. For the XSLT fusion, there was no need to implement a prototype. Due to the fact that the OPT had already a solution based on XSLT, the mapping was changed manually so that transformations were performed all at once, emulating that way the XSLT fusion solution. For the stream, a prototype was built.
- Experiments were carried out in order to understand the performance of both solutions. The experiments consisted in measuring the time required to transform a SPT file in OPT, with different sizes, numbers of transformations and virtual memory configurations.

The final conclusion is that the best solution in terms of performance is to use stream, since it is capable of handling the required migrations described in section 3.1, and provides extensibility for possible future requisites or special transformation cases that may have not been accounted in this work. This solution represents the best choice to replace the XSLT transformation system. In fact, the stream solution is included in the last official release of OPT. It is delivering the expected performance, and both language and extensibility capabilities are adequate to the OPT requirements, providing a huge improvement for clients, planners, developers, testers and even for the future of OPT, since the transformation process is no longer a limitation for the growth of SPT files.

Chapter 6

Future Work

The Stream solution has some limitations due to its ”stream nature”, and the lack of context about the adjacent XML content may be the stream characteristic that imposes more restrictions in the solution. For future work, it would be interesting and important to consider the extensibility of the language and solution architecture to provide more information about the context. The matching clause must be re-designed, possibly to format closer to XSLT or XPath to support matches with context. This work would be a replacement for the ”claim next events” functionality described in section 3.3.3.

Another great improvement for this work would be a the addition of a mechanism to produce the respective MSL mapping transformation for a given modification in the schema. It could be integrated in the EMF User Interface and it would extend the mapping every time a modification is done in the schema, allowing the developers not to be concerned about transformations while implementing new features. This approach was designated coupled evolution in Edapt [6].

Bibliography

[1] P. Genevès, N. Layaïda, and V. Quint. Impact of xml schema evolution. *ACM Trans. Internet Technol.*, 11(1):4:1–4:27, July 2011. ISSN 1533-5399. doi: 10.1145/1993083.1993087. URL http://doi.acm.org/10.1145/1993083.1993087.

[2] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15:3–4, 2004.

[3] F. Budinsky. *Eclipse Modeling Framework: A Developer's Guide*. The eclipse series. Addison-Wesley, 2004. ISBN 9780131425422.

[4] M. Herrmannsdoerfer. The edapt solution for the gmf model migration case.

[5] M. Herrmannsdoerfer. Solving the ttc 2011 model migration case with edapt. In *TTC*, pages 27–35, 2011.

[6] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 425–439. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-45772-5. doi: 10.1007/11880240_30. URL http://dx.doi.org/10.1007/11880240_30.

[7] M. Eysholdt, S. Frey, and W. Hasselbring. Emf ecore based meta model evolution and model co-evolution. *Softwaretechnik-Trends*, 29(2):20–21, 2009.

[8] O. Becker. Transforming xml on the fly. In *XML Europe*, volume 2003. Citeseer, 2003.

[9] Xml path language (xpath). http://www.w3.org/TR/xpath/. Accessed: 2015-10-14.

Appendix A

EMF Generated Java Code

This appendix contains the Java code generated by EMF for the country concept represented in the schema from figure 1.1.

Listing A.1: Country.java

```
/**
*/
package worldv1;

import org.eclipse.emf.ecore.EObject;

/**
* <!-- begin-user-doc -->
* A representation of the model object '<em><b>Country</b></em>'.
* <!-- end-user-doc -->
*
* <p>
* The following features are supported:
* </p>
* <ul>
* <li>{@link worldv1.Country#getName <em>Name</em>}</li>
* <li>{@link worldv1.Country#getContinent <em>Continent</em>}</li>
* <li>{@link worldv1.Country#getPopulation <em>Population</em>}</li>
* </ul>
*
* @see worldv1.Worldv1Package#getCountry()
* @model
* @generated
*/
```

```
public interface Country extends EObject {
/**
* Returns the value of the '<em><b>Name</b></em>' attribute.
* <!-- begin-user-doc -->
* <p>
* If the meaning of the '<em>Name</em>' attribute isn't clear,
* there really should be more of a description here...
* </p>
* <!-- end-user-doc -->
* @return the value of the '<em>Name</em>' attribute.
* @see #setName(String)
* @see worldv1.Worldv1Package#getCountry_Name()
* @model
* @generated
*/
String getName();

/**
* Sets the value of the '{@link worldv1.Country#getName <em>Name</em>}' attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @param value the new value of the '<em>Name</em>' attribute.
* @see #getName()
* @generated
*/
void setName(String value);

/**
* Returns the value of the '<em><b>Continent</b></em>' attribute.
* <!-- begin-user-doc -->
* <p>
* If the meaning of the '<em>Continent</em>' attribute isn't clear,
* there really should be more of a description here...
* </p>
* <!-- end-user-doc -->
* @return the value of the '<em>Continent</em>' attribute.
* @see #setContinent(String)
* @see worldv1.Worldv1Package#getCountry_Continent()
* @model
```

```
* @generated
*/
String getContinent();

/**
* Sets the value of the '{@link worldv1.Country#getContinent <em>Continent</em>}'
    attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @param value the new value of the '<em>Continent</em>' attribute.
* @see #getContinent()
* @generated
*/
void setContinent(String value);

/**
* Returns the value of the '<em><b>Population</b></em>' attribute.
* <!-- begin-user-doc -->
* <p>
* If the meaning of the '<em>Population</em>' attribute isn't clear,
* there really should be more of a description here...
* </p>
* <!-- end-user-doc -->
* @return the value of the '<em>Population</em>' attribute.
* @see #setPopulation(long)
* @see worldv1.Worldv1Package#getCountry_Population()
* @model
* @generated
*/
long getPopulation();

/**
* Sets the value of the '{@link worldv1.Country#getPopulation <em>Population</em>}'
    attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @param value the new value of the '<em>Population</em>' attribute.
* @see #getPopulation()
* @generated
```

```
*/
void setPopulation(long value);

} // Country
```

Listing A.2: CountryImpl.java

```
/**
*/
package worldv1.impl;

import org.eclipse.emf.common.notify.Notification;

import org.eclipse.emf.ecore.EClass;

import org.eclipse.emf.ecore.impl.ENotificationImpl;
import org.eclipse.emf.ecore.impl.MinimalEObjectImpl;

import worldv1.Country;
import worldv1.Worldv1Package;

/**
* <!-- begin-user-doc -->
* An implementation of the model object '<em><b>Country</b></em>'.
* <!-- end-user-doc -->
* <p>
* The following features are implemented:
* </p>
* <ul>
* <li>{@link worldv1.impl.CountryImpl#getName <em>Name</em>}</li>
* <li>{@link worldv1.impl.CountryImpl#getContinent <em>Continent</em>}</li>
* <li>{@link worldv1.impl.CountryImpl#getPopulation <em>Population</em>}</li>
* </ul>
*
* @generated
*/
public class CountryImpl extends MinimalEObjectImpl.Container implements Country {
/**
* The default value of the '{@link #getName() <em>Name</em>}' attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @see #getName()
* @generated
* @ordered
*/
```

```
protected static final String NAME_EDEFAULT = null;

/**
* The cached value of the '{@link #getName() <em>Name</em>}' attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @see #getName()
* @generated
* @ordered
*/
protected String name = NAME_EDEFAULT;

/**
* The default value of the '{@link #getContinent() <em>Continent</em>}' attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @see #getContinent()
* @generated
* @ordered
*/
protected static final String CONTINENT_EDEFAULT = null;

/**
* The cached value of the '{@link #getContinent() <em>Continent</em>}' attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @see #getContinent()
* @generated
* @ordered
*/
protected String continent = CONTINENT_EDEFAULT;

/**
* The default value of the '{@link #getPopulation() <em>Population</em>}' attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @see #getPopulation()
* @generated
* @ordered
```

```
*/
protected static final long POPULATION_EDEFAULT = 0L;

/**
* The cached value of the '{@link #getPopulation() <em>Population</em>}' attribute.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @see #getPopulation()
* @generated
* @ordered
*/
protected long population = POPULATION_EDEFAULT;

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
protected CountryImpl() {
	super();
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
@Override
protected EClass eStaticClass() {
	return Worldv1Package.Literals.COUNTRY;
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public String getName() {
	return name;
```

```
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public void setName(String newName) {
        String oldName = name;
        name = newName;
        if (eNotificationRequired())
                eNotify(new ENotificationImpl(this, Notification.SET, Worldv1Package.
                    COUNTRY__NAME, oldName, name));
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public String getContinent() {
        return continent;
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public void setContinent(String newContinent) {
        String oldContinent = continent;
        continent = newContinent;
        if (eNotificationRequired())
                eNotify(new ENotificationImpl(this, Notification.SET, Worldv1Package.
                    COUNTRY__CONTINENT, oldContinent, continent));
}

/**
* <!-- begin-user-doc -->
```

```
* <!-- end-user-doc -->
* @generated
*/
public long getPopulation() {
	return population;
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public void setPopulation(long newPopulation) {
	long oldPopulation = population;
	population = newPopulation;
	if (eNotificationRequired())
		eNotify(new ENotificationImpl(this, Notification.SET, Worldv1Package.
		    COUNTRY__POPULATION, oldPopulation, population));
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
@Override
public Object eGet(int featureID, boolean resolve, boolean coreType) {
	switch (featureID) {
		case Worldv1Package.COUNTRY__NAME:
			return getName();
		case Worldv1Package.COUNTRY__CONTINENT:
			return getContinent();
		case Worldv1Package.COUNTRY__POPULATION:
		return getPopulation();
	}
	return super.eGet(featureID, resolve, coreType);
}

/**
```

```
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
@Override
public void eSet(int featureID, Object newValue) {
        switch (featureID) {
                case Worldv1Package.COUNTRY__NAME:
                        setName((String)newValue);
                        return;
                case Worldv1Package.COUNTRY__CONTINENT:
                        setContinent((String)newValue);
                        return;
                case Worldv1Package.COUNTRY__POPULATION:
                        setPopulation((Long)newValue);
                        return;
        }
        super.eSet(featureID, newValue);
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
@Override
public void eUnset(int featureID) {
        switch (featureID) {
                case Worldv1Package.COUNTRY__NAME:
                        setName(NAME_EDEFAULT);
                        return;
                case Worldv1Package.COUNTRY__CONTINENT:
                        setContinent(CONTINENT_EDEFAULT);
                        return;
                case Worldv1Package.COUNTRY__POPULATION:
                        setPopulation(POPULATION_EDEFAULT);
                        return;
        }
        super.eUnset(featureID);
```

```
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
@Override
public boolean eIsSet(int featureID) {
        switch (featureID) {
                case Worldv1Package.COUNTRY__NAME:
                        return NAME_EDEFAULT == null ? name != null : !NAME_EDEFAULT.
                            equals(name);
                case Worldv1Package.COUNTRY__CONTINENT:
                        return CONTINENT_EDEFAULT == null ? continent != null : !
                            CONTINENT_EDEFAULT.equals(continent);
                case Worldv1Package.COUNTRY__POPULATION:
                        return population != POPULATION_EDEFAULT;
        }
        return super.eIsSet(featureID);
}

/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
@Override
public String toString() {
        if (eIsProxy()) return super.toString();

        StringBuffer result = new StringBuffer(super.toString());
        result.append(" (name: ");
        result.append(name);
        result.append(", continent: ");
        result.append(continent);
        result.append(", population: ");
        result.append(population);
        result.append(')');
```

```
        return result.toString();
    }

} //CountryImpl
```