XML Documents Transformation in Large Scale

João Paulo Fonseca Sequera
joao.sequeira@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2015

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

1. Introduction

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 in [3], 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. 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 automatic mechanisms 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 XSLT [4]. However, with those approaches, for very large XML documents the time and memory required to perform transformations may rise to values ascend to excessively high values.
Eclipse Modeling Framework (EMF) [5] is a Domain-Specific Visual Language as described in [6]. 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. For a more in depth explanation on how EMF is used to build applications, please refer to [7].
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,

networks 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 STP 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 bigger 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.
This work presents a new solution to perform transformations on XML files based on a stream of XML documents requiring few memory and with fast times. The solution is compared with a normal XSLT transformation and with an ideal XSLT solution in which all transformations are performed at the same time, denominated in this work as XSLT fusion.

2. Concepts and Related Work

2.1. Concepts

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 1.

Listing 1: XML language components

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

Listing 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".

2.2. 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 [8]. The tool 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 an 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 [9] and [10], the authors present how they solved real scenarios of EMF models evolution. In those works and in reference [11] 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 [12], the authors describe the process of migration (or XML document transformation). 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 reference [13] the presenter reveals while answering to the first question that legacy XML documents are 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.

2.2.2 Streaming Transformation for XML (STX)

STX is an XML transformation language that operates in stream, based on SAX [14]. 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 [15], and supports the same operations as XSLT, since they are both turing complete languages [14].
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 [14]. 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.

3. Implementation

The proposed solution is to use a stream of data. The data is read from source to memory in small parts. 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 3.1 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 3.2 explains how this problem was solved.
The prototype to test the performance and feasibility of this solution was built integrated in OPT, transforming SPT files.

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 ”migration”.
- 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 shows a mapping specified as MSL. The corresponding schema modifications are:

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

Listing 3: 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.

3.1.1 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 which the parent element has the name equal to its value.
MSL attribute matchElementName accepts elements which name is equal to its value.
MSL attribute matchAttributeName accepts elements that contain an attribute which name is equal to its value.
In listing 3, 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.

3.1.2 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. 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. 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 elements name to "new element name".
- ChangeElementValue - Receives one argument as input: "new element name". Changes the elements 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 attributes value of the attribute with name equal to "attribute name" and sets it as the current elements 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.

3.2. Architecture

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

- Engine
- Manager
- Transformers

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.

3.2.1 Engine

The migration engine is responsible for parsing the input SPT file, and creating round result files with the result of executing the transformations of each round. Each round result file 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.
Migration engine uses SAX to parse the XML document. SAX 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.

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, sends the ElementDTO to the migration manager, which will return a list of ElementDTOs as a result of the transformation. Then, the engine writes the element received to the round result file.

3.2.2 Migration Manager

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 needs 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 rounds respective matching clauses, which means that MSL Ingestion is executed once per round.
MDB is composed by a Java class called 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 1 illustrates the relation between the classes that compose the MDB structure. The al-

MDB.png

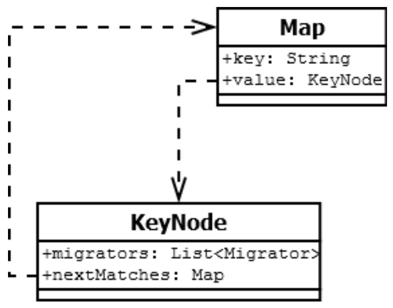

Figure 1: MDB Class Diagram

gorithm of MSL Ingestion uses the classes in 1 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 4.

Listing 4: 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 2 illustrates the memory structure built by MSL Ingestion algorithm resultant of the mapping in listing 4.

XML element names, attributes names and attributes values may be equal. Therefore, there is a need to identify that ”countries” is 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

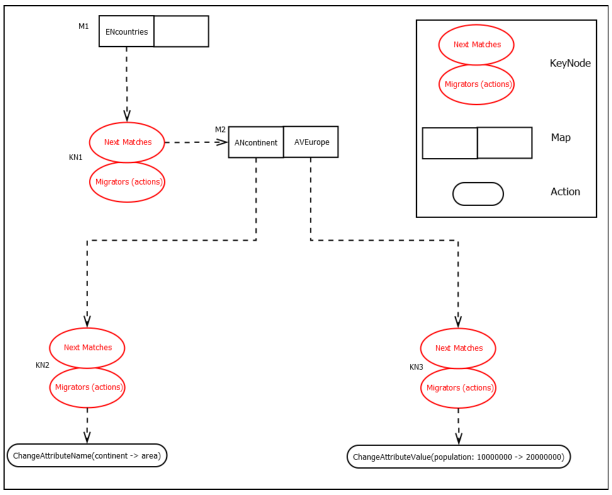

Figure 2: MDB in Memory Structure

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
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 2, 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).
Consider the XML document in listing 5.

Listing 5: XML document sample

```xml
<?xml version="1.0" encoding="UTF-8"?>
<thesis:World>
 <countries name="Portugal"continent="
     Europe" population="10000000"/>
 <countries name="Spain" continent="Europe
     "population="48000000"/>
 <countries name="Brazil" continent="
     America" population="200000000"/>
</thesis:World>
```

The algorithm used to match elements is better explained following an actual example. Considering that the Engine would send ElementDTOs representing the XML elements in listing 5 (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 3.1.
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's 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". 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 ElementDTOs is returned to the engine, because the result may be more than one ElementDTO. This will be explained in section 3.3.

3.3. Actions
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.

4. Results

OPT is the test bed for the performance comparison of the performance of the stream based solution with XSLT based solutions. 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, 1.45GB 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 1, 2 and 3. 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 with 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 that with the duplication of SPT file size, the time also duplicates.

Table 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 2: 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 3: 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

5. Conclusions

The final conclusion is that, in terms of performance, is to use stream, since it is capable of handling the most common EMF migrations, and provides extensibility for possible future requisites or special transformation cases. In fact, 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 the OPT, since the transformation process is no longer a limitation for the growth of SPT files.

References

[1] Extensible markup language (xml). `http://www.w3.org/XML/`. Accessed: 2015-10-02.

[2] Xml schema. `http://www.w3.org/XML/Schema`. Accessed: 2015-10-02.

[3] Pierre Genevès, Nabil Layaïda, and Vincent Quint. Impact of xml schema evolution. *ACM Trans. Internet Technol.*, 11(1):4:1–4:27, July 2011.

[4] Xsl transformations (xslt). `http://www.w3.org/TR/xslt`. Accessed: 2015-10-02.

[5] Eclipse modeling framework (emf). `https://eclipse.org/modeling/emf/`. Accessed: 2015-10-02.

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

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

[8] Edapt - migrating emf models. `https://www.eclipse.org/edapt/`. Accessed: 2015-10-15.

[9] Markus Herrmannsdoerfer. The edapt solution for the gmf model migration case.

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

[11] Enrico Biermann, Karsten Ehrig, Christian Khler, Gnter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 425–439. Springer Berlin Heidelberg, 2006.

[12] Moritz Eysholdt, Sören Frey, and Wilhelm Hasselbring. Emf ecore based meta model evolution and model co-evolution. *Softwaretechnik-Trends*, 29(2):20–21, 2009.

[13] Maximilian Koegel. Model migration with edapt. Presented at EclipseCon NA 2015, 2015.

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

[15] Xml path language (xpath). `http://www.w3.org/TR/xpath/`. Accessed: 2015-10-02.