Flexible Serialisation of Model Instances: A Cascading Mapping Mechanism

Alexander Paulsen

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Major parts of the code for a software system can be generated from models. These models come with functionality for editing and storing model instances in the XMI format. Not all developers might want to store instances in exactly this format, and the current alternatives for modifications are insufficient.

The goal of the thesis is to create a flexible serialiser for model instances. The serialiser relies on mappings to define the relations from the model to a XML representation.

The mapping mechanism developed is designed and implemented as an extension to the Eclipse Modeling Framework (EMF). The serialiser provides some of the functionality defined in the ISO standard for Petri nets. The serialiser is also able to serialise other formats than Petri nets.

Among other functionality, mappings can be defined cascading and the best fitting mapping is chosen. New mappings can be added to the serialiser using known methods in Eclipse and requires no programming. If no mappings are defined the serialiser relies on the standard, basic serialiser provided with EMF.

Resumé (Danish)

Store dele af koden til et softwaresystem kan genereres fra modeller. Disse modeller leveres med funktionalitet til redigering og lagring af modelinstanser i XMI-formatet. Nødvendigvis ønsker ikke alle udviklere at gemme instanser i nøjagtig dette format, og de nuværende alternativer til ændringer er utilstrækkelige.

Målet med dette speciale er at skabe en fleksibel serialiseringsmekanisme til modelinstanser. Mekanismen gør brug af mappings til at definere relationerne fra model til en XML-repræsentation.

Mekanismen er designet og implementeret som en udvidelse til Eclipse Modeling Framework (EMF). Mekanismen giver mulighed for at repræsentere nogle af de strukturerer defineret i ISO-standarden for Petri net. Det er også muligt at serialisere andre formater og modeller end Petri net.

Blandt anden funktionalitet kan mappings defineres kaskaderende og den mest præcise mapping vælges. Nye mappings kan tilføjes via normale arbejdsgange i Eclipse og kræver ingen programmering. Hvis ingen mappings passer, gøres brug af standard serialiseringsmekanismen i Eclipse.

Preface

This thesis was prepared from the 1st of September to the 1st of February at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring an MSc in Computer Science and Engineering. The thesis was prepared with Ekkart Kindler as supervisor

I would like to thank Ekkart Kindler for being the supervisor for this thesis, providing answers to many questions and knowledge on Petri nets and EMF. I would also like to thank my family and friends for support through the development of the thesis. A special thank to Thea Paulsen and Benedikte Larsen for corrections and feedback.

Lyngby, 01-February-2015

Alexander Paulsen

Alexander Paulsen

Contents

Summary **i**

Resumé (Danish) **iii**

Preface **v**

1 Introduction **1**
- 1.1 Wished Functionality 2
- 1.2 Evaluation 3
- 1.3 Report Structure 3

2 Problem **5**
- 2.1 Petri Nets 5
- 2.2 Models and Model Instances 6
- 2.3 Serialisation 8
- 2.4 Serialisation with XMI 8
- 2.5 Serialisation with Mappings 9

3 Analysis **11**
- 3.1 Features of the Serialiser 12
  - 3.1.1 Context Dependent Mappings 12
  - 3.1.2 Multiple Model Versions 14
  - 3.1.3 Objects into Attributes 16
  - 3.1.4 Objects on Multiple Levels 16
  - 3.1.5 References to Other Objects 17
  - 3.1.6 Minimising Effort of Defining Mappings 18
  - 3.1.7 Dynamic Mappings 20
- 3.2 State of the Art of Serialising 20
  - 3.2.1 Serialising With Simple 21

3.2.2 Serialising With XStream 22
3.2.3 Serialising With Java Architecture for XML Binding 23
3.2.4 Serialising With EMF XMI serialiser 24
3.2.5 Comparison of Serialisers 26

4 Technological Background 27
4.1 Serialisation in Eclipse Modeling Framework 27
4.1.1 Saving an Instance of a Model 30
4.1.2 Loading an Instance of a Model 32
4.1.3 Low Level Serialisation 35
4.2 Extending Functionality in Eclipse 36
4.3 Domain Specific Languages 37

5 Software Design 39
5.1 Overall Structure 39
5.2 Features of the Mapping Mechanism 40
5.2.1 Context Dependent Mappings 41
5.2.2 Multiple Versions 41
5.2.3 Standard Mappings 42
5.2.4 Adding Tables 42
5.2.5 Cascading Mappings 42
5.3 Concepts 43
5.3.1 Defining and Using Mappings 43
5.3.2 References to Other Objects 44
5.3.3 Defining Packages 44
5.4 Domain Specific Language for Mappings 45
5.4.1 Features of the New Serialiser (Options) 47

6 Handbook for Developers 49
6.1 Installing Eclipse with EMF and Ecore 49
6.2 Installing the Serialiser and Mapping Mechanism 50
6.3 Adding Mappings for an Ecore Model 51

7 Implementation 53
7.1 Overriding the Standard Serialiser 54
7.2 Mappings for the Serialiser 55
7.2.1 Plugging in Mappings 56
7.2.2 The Editor for Mappings and Packages 58
7.3 Realising the Serialiser 59
7.3.1 Saving Instance – `MySave` 61
7.3.2 Load Functionality – `MyLoad` and `MySAXHandler` 62
7.3.3 Table for mappings – `MappingTable` 65

8 Evaluation **69**
8.1 Test of Mapping Mechanism 69
8.1.1 Relying on the Default Serialiser 70
8.1.2 Adding Mappings – Object from Attribute 72
8.1.3 Saving Object as a Number 73
8.1.4 Most Specific Mapping 73
8.1.5 Add Tables 75
8.1.6 Precedence of Added Tables 75
8.1.7 Using References 76
8.1.8 Adding Extra Levels 77
8.1.9 Multiple mappings 79
8.2 Correctness of Serialisation 82
8.3 Relation to Standard for Petri Nets 82
8.4 Conclusion on Test 83
8.5 The Future of the Serialiser 83
8.5.1 Fully Supporting the ISO Standard for Petri nets 83
8.5.2 References in Instances 84
8.5.3 Error Handling in Mapping Editor 84
8.5.4 The mapping Editor 84
8.5.5 Developed Serialiser as Default 85

9 Conclusion **87**

Bibliography **90**

A Call Structure for Methods in EMF **91**

B Developed Software **99**

CHAPTER 1

Introduction

By using models, major parts of the code for software systems can be generated. The generated model code will be created with functionality for creating and modifying instances of models as well as all kinds of mechanisms for notifications when changed. The model code will also come with some mechanism for serialising the model instances e.g. the model instance is serialised into the XMI-format. All this gives the developer the ability to focus on developing the model, making the development more hassle-free.

Sometimes the developer do not want the standard serialisation of model instances. This could be a developer wanting to represent data in a format not specified by a certain technology i.e. XMI. It could also be a developer wanting to represent data in a very specific format. Serialisation technology available today often have options or other ways to configure the serialisation – but these do not allow for major changes in the structure of the serialisation.[1]

Throughout this report, the PNML format, used for storing Petri nets will be used as an example. Even though the format is formally defined as a standard as of ISO/IEC 15909-2, and is defined by simple rules and tables; XMI cannot be configured into serialising model instances as defined in the format. To do that, something more configurable than XMI is needed.

[1]A discussion of the current state of serialisers and why the options are insufficient can be found in Sect. 3.2

It would be a great help for developers, having a technology which could do that; A technology where the developer could define the relationship between a model and the representation in XML, so called mappings. And then provide the technology to load and save according to these mappings.

The aim of this thesis is to develop such a technology. The technology will be able to define how to represent instances of UML models in XML. The technology will be developed as an extension to the Eclipse Modeling Framework (EMF). Developers will then be able to define their own mappings, which by using the developed technology, will map according to these. But defining all mappings for even small projects might become too much work, so the technology should make it possible to only define the mappings which differs from XMI, and rely on XMI for the rest. Various features might be wanted by a developer, such features will be developed.

Serialising data is not a new concept, multiple solutions already exists. These will be discussed in the analysis, in Chap. 3. In the next section, Sect. 1.1, the wished functionality for the developed serialiser will be discussed.

1.1 Wished Functionality

The development of the technology will focus especially on the following points:

- New mappings should be possible to add by using plug-in extensions in Eclipse, without having to do any programming.
- To ease the developers work, it should only be necessary to define the relevant mappings. The technology should then rely on already defined standard mappings (XMI) when no mappings are available.
- Sometimes models are extended dynamically, and in regarding to that, the mappings should also be extended dynamically.
- Formats can have different versions, these different versions should all be possible to support. This could be supported by letting the user choose the version to be saved in, when saving. And having the table dynamically extend depending on which version is defined in the document, when loading.
- The same object might be used multiple times throughout a document, but is in need of being serialised differently. The technology should support these context dependent mappings.

- Sometimes the contents of an object is a subset of the content of another object. The technology should support linking in the model.
- The technology developed throughout this project should be configurable enough to be able to serialise parts of the PNML format defined in the ISO standard.

1.2 Evaluation

The technology is interesting for developers using model based software engineering and should thus be possible to use for all sorts of different formats. When a developer creates a new model, he might want a certain representation in XML.

The serialisation of Petri nets using Petri Net Markup Language (PNML) defined in ISO/IEC 15909-2:2009 and 15909-2:2011, will be used as an example, and some of the features of the standard should be supported.

The technology should be evaluated by serialising the different parts of some model instance and comparing the results with the expected results defined in the standard. Additionally it should be proven that the mappings affect the loading and saving in the same ways. It should also be proven that a loaded and saved instance of a model remains the same.

1.3 Report Structure

The structure of this report is as follows:

Chapter 1 — **Introduction** is a general introduction to this thesis on serialisation of model instances.

Chapter 2 — **Problem** defines the problem and covers concepts needed to understand the problem.

Chapter 3 — **Analysis** analyses the problem and already existing solutions.

Chapter 4 — **Technological Background** covers the technologies used in this project, and how to use them.

Chapter 5 — **Software Design** describes the design of the software for solving the problem.

Chapter 6 — **Handbook for Developers** is a detailed guide on installing and using the developed software.

Chapter 7 — **Implementation** describes how the software; the mapping mechanism and the serialiser is implemented.

Chapter 8 — **Evaluation** assesses by test and discussion the developed software.

Chapter 9 — **Conclusion** concludes the work done throughout this project.

Chapter 2

Problem

The following chapter defines the problem which will be solved in this report, but to define the problem, some concepts must be defined. Serialising model instances is a very general problem, so as an example Petri nets will be used throughout this report. The first section of this chapter, Sect. 2.1, will cover what Petri nets are. In Sect. 2.2 a model for Petri nets and an instance of a Petri net will then be covered. These instances of this model will need to be serialised so it can be loaded and saved later, which will be covered in Sect. 2.3, 2.4 and 2.5.

2.1 Petri Nets

A Petri net is a modeling language for behaviour. It is a directed bipartite graph containing places, transitions, arcs and tokens. A place is denoted by a circle and contains a condition for in which the place can fire. Transitions are denoted by a square and contains events which will occur. Places and transitions are linked together with arcs, which are directed arrows. This denotes where a token would flow when fired for a place. A token is denoted by a filled circle, a place can hold one or more tokens which can then be fired. In Fig. 2.1 the elements contained in a Petri net are shown. In the next section a model for defining the relations between the elements in a Petri net is described, along with a model of a such instance.

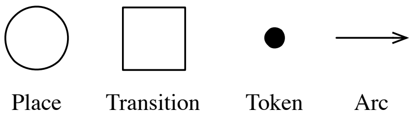


Figure 2.1: A Petri net contains the four elements; places, transitions, tokens and arcs.

An instance of a Petri net could be the simple Petri net shown in Fig. 2.2. In this example the top left place holds a token. If the condition for the place is met, the transition is ready to fire. That means it will execute the event contained and consume the token. The transition will then create tokens in all output places. In this case that means the lower right place will now have the token. This gives the transition to the left of it, the ability to fire.

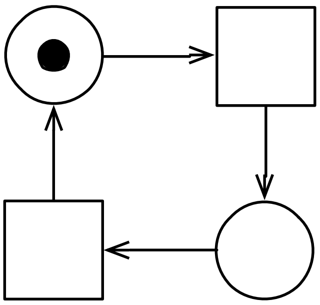


Figure 2.2: A simple instance of a Petri net

2.2 Models and Model Instances

In Fig. 2.3 the model for a Petri net is shown. The model represents the different concepts needed for a Petri net and the relationship between these.

A `PetriNet` consists of multiple `Nodes` and `Arcs`. A `Node` can either be a `Place` or a `Transition`. An `Arc` contains both a reference to a `source` and a `target`, which both is `Nodes`. A `Place` can contain one or more tokens.

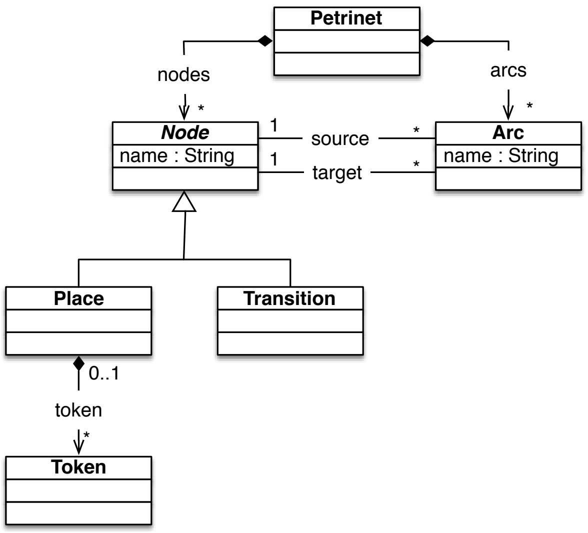


Figure 2.3: A model for Petri nets, showing the relationships between the different concepts needed to define a Petri net.

An instance of a such model could be the simple Petri net in Fig. 2.4. An instance of a model is the realisation of a model, and contains actual data, such as places, transitions and tokens. The model instance is a very simple Petri net containing two places and two transitions, with arcs in between.

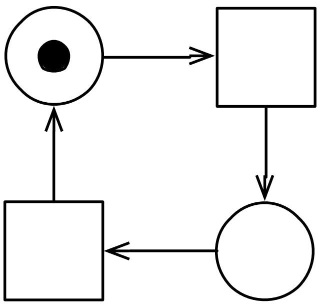
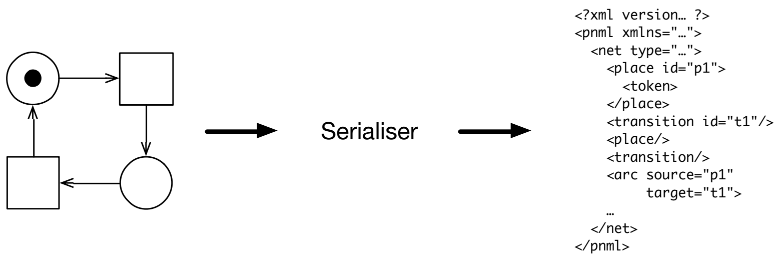


Figure 2.4: A simple instance of the Petri net model.

2.3 Serialisation

Serialisation, also known as marshalling, is the concept of representing objects in memory into a format that can later be stored. These translated objects should then later be able to be restored into their previous state. In Fig. 2.5 an example of the expected result when serialising the Petri net is shown.



```
<?xml version… ?>
<pnml xmlns="…">
  <net type="…">
    <place id="p1">
      <token>
    </place>
    <transition id="t1"/>
    <place/>
    <transition/>
    <arc source="p1"
         target="t1">
    …
  </net>
</pnml>
```

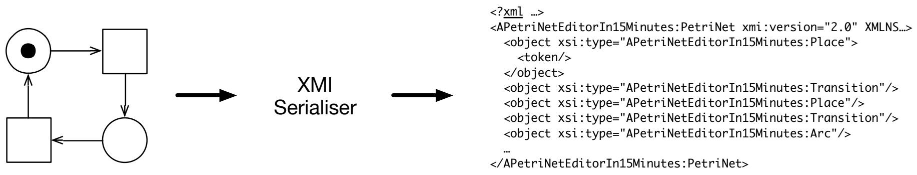
Figure 2.5: An instance Petri net model is serialised into XML format using a serialiser.

When deserialising, also known as unmarshalling, the process shown in Figure 2.5 is reversed. The stored file, in XML format, is passed to a serialiser, using (preferably the same) mappings. The file can now be translated into the same data which were serialised.

2.4 Serialisation with XMI

Using the standard built-in Eclipse XMI serialiser, the flow is as seen in Fig 2.6. The serialisation is almost what a developer wants, if ignoring all the namespaces. But; since all objects are related to the net in a variable called objects, the objects will be serialised into a tag called "object". Depending on which object it is, the object will then get a type corresponding to which object it is.

To change this, the structure of the representation must be changed. This is a major change and cannot be done in XMI.



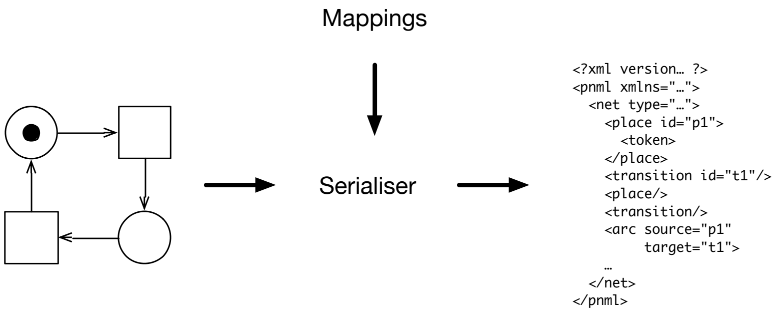
```
<?xml …>
<APetriNetEditorIn15Minutes:PetriNet xmi:version="2.0" XMLNS…>
  <object xsi:type="APetriNetEditorIn15Minutes:Place">
    <token/>
  </object>
  <object xsi:type="APetriNetEditorIn15Minutes:Transition"/>
  <object xsi:type="APetriNetEditorIn15Minutes:Place"/>
  <object xsi:type="APetriNetEditorIn15Minutes:Transition"/>
  <object xsi:type="APetriNetEditorIn15Minutes:Arc"/>
  …
</APetriNetEditorIn15Minutes:PetriNet>
```

Figure 2.6: An instance of a model, a Petri net, is serialised into XMI format using a serialiser.

2.5 Serialisation with Mappings

Using XMI to serialise is not sufficient, it must be possible to modify the representation even more than the built in serialiser can do. But first a way to define how a model should be represented must be defined.

Using mappings it is possible to define the relationship between a model and a corresponding representation. This could for example be a model defining a Petri net, and a representation of such in an XML format which a developer want to use. The developer can then map the model for Petri nets into the XML representation. Then using some software supporting the mappings, the developer can serialise instances of models defined by a model, into his storage format, which in this example is XML.



```
<?xml version… ?>
<pnml xmlns="…">
  <net type="…">
    <place id="p1">
      <token>
    </place>
    <transition id="t1"/>
    <place/>
    <transition/>
    <arc source="p1"
         target="t1">
    …
  </net>
</pnml>
```

Figure 2.7: An instance of a model, a Petri net, is serialised into XML format using a serialiser. The serialiser depends on mappings defining how the model instance is mapped into the storage format.

From the model defined in Fig. 2.3, and a model for the XML format in the example in Fig. 2.5, the mappings shown in Tabl 2.1 could be defined.This would support the translation of the Petri net model into the model for the XML format shown in Fig. 2.7.

Java object	**XML tag**	**Attributes**
Place	place	—
Transition	transition	—
Arc	arc	source, ...
⋮	⋮	⋮

Table 2.1: An example of possible mappings for the model defined in Fig 2.3 to map into an XML format.

The mappings are very simple. The Java objects are translated into elements in XML with the same name. If a Java object contains attributes, these are serialised along with the object. A such mapping table could be created by reading the model. In this project more interesting mappings will be considered, where a developer might want to change this default behaviour.

Chapter 3

Analysis

The following chapter contains an analysis of what is needed to create a serialiser suiting the needs for this project. The chapter covers which necessary features and concepts the solution must support and what features are supported by already existing applications.

To evaluate that the serialiser is usable, the serialiser must be able to serialise parts of PNML. That means the needed features to serialise PNML is also the needed features for this project. The PNML is chosen as it contains simple elements to serialise, but also contains multiple more advanced features needed to be serialised in a special way.

The following sections covers the features needed for the serialiser, these are:

Context Dependent Mappings where mappings of objects are dependent on in which context the object is in.

Multiple Versions where multiple models can be defined and changed in between.

Objects into Attributes where objects can be serialised into attributes on the containing element instead of new objects.

Objects on Multiple Levels where objects when serialised gets an extra level of elements to be contained in.

References in Data where objects can contain references to other objects

Defining Mappings where various concepts for defining mappings are analysed. Containing standard mappings, choosing the correct, most specific mapping and how mappings is to be defined

Dynamic Mappings how mappings dynamically is to be plugged in.

The chapter also covers the state of the art of four widely used serialisers. To give a broad image of the ways of serialise.

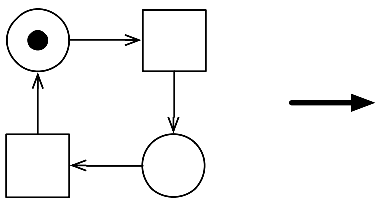
- Simple
- XStream
- JAXB
- the EMF serialiser

3.1 Features of the Serialiser

The next sections will cover the features needed to realise the serialiser.

3.1.1 Context Dependent Mappings

Sometimes the same kind of object is used different places in a model. But the way the object is to be serialised depends on the context the object is used in. As an example, in Fig. 3.1 a serialisation of a Petri net can be seen.



```
<net>
  <place>
    <token/>
  </place>
  <transition>
    <token/>
  </transition>
  <place/>
  <transition/>
  <arc source... >
  ...
</net>
```

Figure 3.1: The representation of a regular Petri net, with one token on the place.

A developer might for various reasons want to represent the token object differently depending on which object it is on. As an example, the developer want the token object to be a star on transitions. This is shown in Fig. 3.2 where the model is updated, in Fig. 3.3 an instance of the model is shown along with an XML representation. In this example, the token object must be represented differently, due to the context it is placed in.

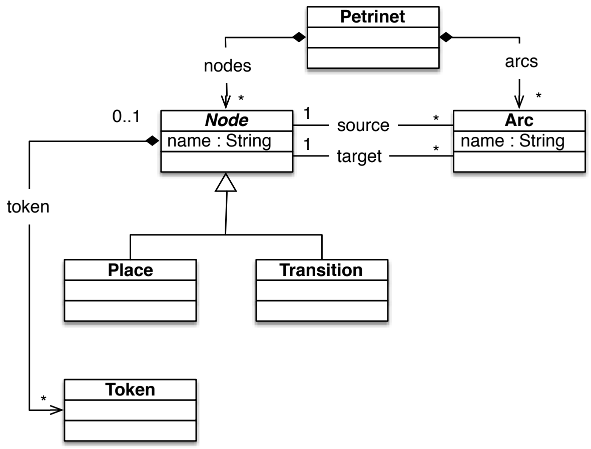
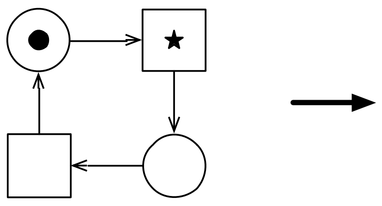


Figure 3.2: A model for Petri nets, extended from the original version in Fig. 2.3 on Page 7, to also include tokens on transitions.



```
<net>
  <place>
    <token/>
  </place>
  <transition>
    <star/>
  </transition>
  <place/>
  <transition/>
  <arc source... >
  ...
</net>
```

Figure 3.3: The representation of a Petri net, with the token on the transition displayed as a star.

Now the transition also contains a token. The model does not differentiate between a token on a place or transition. But in the XML representation we want to do exactly that. If the token is contained in a place, it is to be serialised into a token-element. But if the token is contained in a transition it needs to be serialised into a star-element. To support this, it must be taken into account in what context a token is in.

3.1.2 Multiple Model Versions

Models might be evolving for long periods of time, or models might be in need of updates for various reasons. This means that multiple versions of a model might exist. Imagine a developer having developed the model and the representation shown in Fig 3.1. But later the developer revises his model and realises that he needs to be able to define a colour on the token. The developer, for legacy reasons, still wants to be able to use the old version. The developer could then revise the model and create a second version. See the model in Fig. 3.4. In Fig. 3.5 an instance of a such model is shown along with an XML representation.

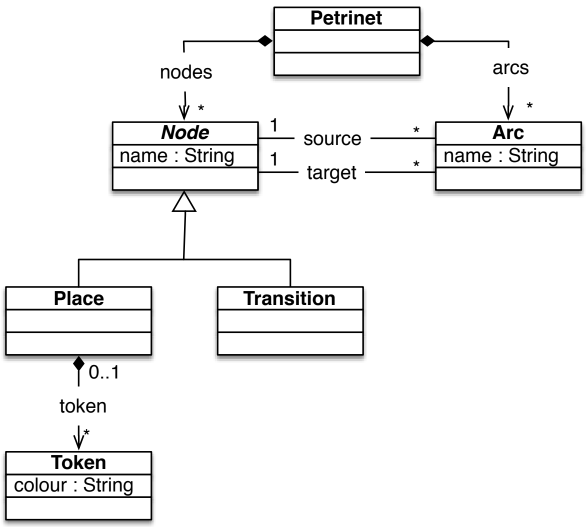
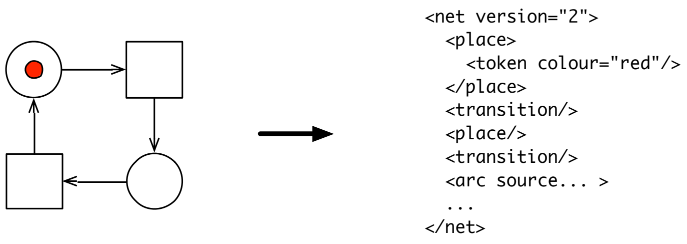


Figure 3.4: A second revised model for Petri nets, extended from the original version to also include colour on the token object.



```
<net version="2">
  <place>
    <token colour="red"/>
  </place>
  <transition/>
  <place/>
  <transition/>
  <arc source... >
  ...
</net>
```

Figure 3.5: An instance of the model shown in Fig. 3.4.

To be able to distinguish between the different versions a version attribute could be introduced in the net. Now the developer is able to serialise nets using the two different versions by defining which model should be used depending on the version defined in the representation.

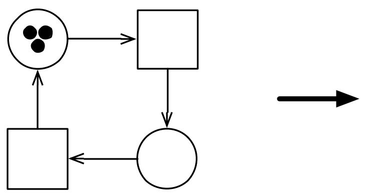
That means the developer is able to have, *one* model; but *two* representations. But the other way is also possible!

3.1.2.1 Multiple Representations

The developer might want to keep, and use both versions; but only have on representation. The developer should then be able to create mappings, so both versions would result in the same representation.

3.1.3 Objects into Attributes

For some representations a developer might want to change how some objects are serialised. In this case the developer wants the token objects to be serialised into an attribute describing how many tokens there are. In Fig. 3.6 a such instance of a model is shown.



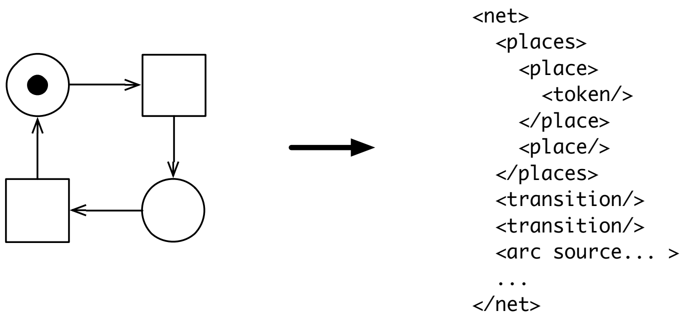
```
<net>
  <place tokens="3"/>
  <transition/>
  <place/>
  <transition/>
  <arc source... >
  ...
</net>
```

Figure 3.6: An instance of the original model shown in Fig. 2.3. Notice how the tokens are represented as an attribute instead of as objects.

Without changing the model to contain the number of tokens on the place, the mechanism must support this representation.

3.1.4 Objects on Multiple Levels

A developer might wants to group multiple objects in an element. In this example the developer want to group all places into an element called places. In that element he then want all places to be serialised. An example of a such representation is shown in Fig. 3.7. A such representation should be supported.

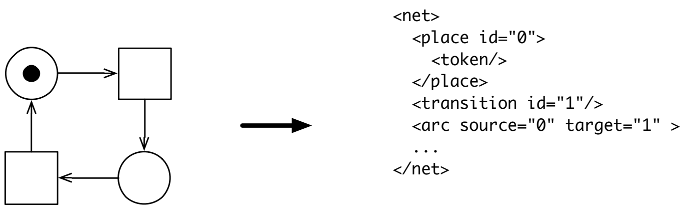


```
<net>
  <places>
    <place>
      <token/>
    </place>
    <place/>
  </places>
  <transition/>
  <transition/>
  <arc source... >
  ...
</net>
```

Figure 3.7: An instance of the original model shown in Fig. 2.3. Notice how places now is in an extra element level.

3.1.5 References to Other Objects

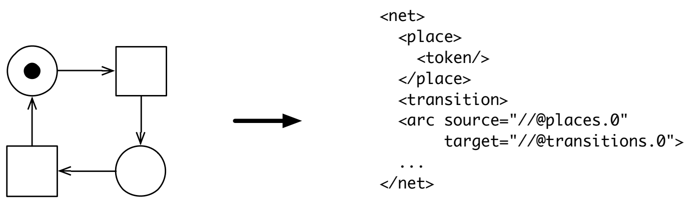
In some cases the content of one object, is a reference to another object. For instance when defining arcs in Petri nets. See Fig. 3.8. Giving the nodes identifiers, it is now possible to refer to a specific place when creating an arc.



```
<net>
  <place id="0">
    <token/>
  </place>
  <transition id="1"/>
  <arc source="0" target="1" >
  ...
</net>
```

Figure 3.8: An instance of the original model shown in Fig. 2.3. Notice that arcs now refer to the identifier now contained in places and transitions.

Another solution which does not require changes to the model, is to use XPath to define which nodes the arc is connecting. In Fig 3.9 a solution using XPath is shown.



```
<net>
  <place>
    <token/>
  </place>
  <transition>
  <arc source="//@places.0"
       target="//@transitions.0">
  ...
</net>
```

Figure 3.9: An instance of the original model shown in Fig. 2.3. Now using references using XPath.

Both references using XPath (no identifiers), and references using identifiers must be supported in the implemented solution.

3.1.6 Minimising Effort of Defining Mappings

It would be easy for a serialiser to serialise if every object has a corresponding mapping. That would mean that for each object the serialiser just have to find the mapping and map the object. From a developers point of view that might not be the best solution. If a developer was to design a model, he would not want to define a mapping for each object he already have defined.

A solution to that, could be to use general mappings. The more general the mappings could be, the less mappings would be needed. But this raises other issues. Some objects are special and need to be treated specially, to allow for general mappings, mappings that overlap must also be allowed. Overlapping mappings in the sense of some objects having multiple mappings. Allowing multiple mappings for objects raises the question of which mapping should be used[1], the most logical solution is to use the most specific mapping. Which means; it must be defined what is the most specific mapping. This will be covered with an example in Sec. 3.1.6.2.

3.1.6.1 Standard Mappings

Another important point is that when letting a developer define more and more general mappings, some mappings might be so general that they could be used

[1]Using both mappings when overlapping would lead to duplicates of data in the new representation.

for any project. Such mappings could already exist in the application, so the developer only needs to map the special cases.

Standard mappings are the least specific mappings, and should only be relied on if no other mappings can be used.

3.1.6.2 Most Specific Mapping

Which mapping is the most specific depends on various things:

- When defining context dependent mappings, it should be possible to define mappings with and without context. Context dependent mappings should take precedence over non-context dependent mappings.
- When defining multiple versions for a model, mappings for the used version must take precedence over mappings not related to any version. If no versions are specified in the mappings, or no versions fit the current version, mappings without version could be used.

#	Feature	Container	Element-name
1	token	—	token
2	token	place	token2

Table 3.1: An example of a mapping table for the standard model in Fig. 2.3.

In Tabl. 3.1, a mapping table can be seen. It contains two mappings for the same feature. The first mapping contains no information about the containing object, and the second mapping contains information on which context it applies. This means, in the case of another context i.e. parent object, is not a `place`, the first rule should apply. But if it is in the context of a `place`, the second mapping is more specific, and should apply.

3.1.6.3 No Programming

The serialiser could make a interface available for developers, developers could then implement their own additions to the serialiser. But this means that developers would be forced to understand all of the inner workings of the serialiser.

Instead a no-programming interface for the developer to define mappings in would be beneficial.

A good user interface should have a printable and useful representation of mappings as long as functionality to aid the developer in making the right mappings.

3.1.7 Dynamic Mappings

When developing a model, a set of mappings to define how the model is to be serialised must also be defined. These mappings could be defined and stored in different ways.

One way is to let these mappings be part of the serialiser, and thus needed at compile time.

But that is not very usable for a developer. Instead the mappings should be able to be plugged-in at any time. So the user do not have to acquire the source code in order to use the serialiser.

3.2 State of the Art of Serialising

This section covers the state of the art of a selection of the serialisers currently available. The four serialisers are chosen to give a broad image of how serialisers can be implemented. But also to show some of the most used serialisers. The chosen serialisers span from Simple and XStream, which are the most simple serialisers, to more configurable serialisers such as JAXB and the EMF Serialiser.

Simple is a very simple serialiser, supporting XML. The serialiser uses annotations directly in the Java classes to define which classes should be serialised into which elements.

XStream is a Java library for serialising into XML. XStream does not require any modifications to objects and is thus easy to setup.

Java Architecture for XML Binding (JAXB) is part of the Java SE platform and allows for serialising Java objects into XML representations. JAXB is widely used and configurable.

EMF XMI Serialiser XML Metadata Interchange (XMI) is the standard developed by Object Management Group and used in Eclipse Modeling Framework to exchange information in XML. The serialiser provided with EMF for XMI serialising is very configurable.

The next four sections will discuss the usage of each of the four different serialisers. Section 3.2.5 will contain a comparison of the serialisers.

3.2.1 Serialising With Simple

Simple is as the name suggest a simple XML serialiser. Part of the simplicity is that Simple requires no additional files with mappings or configuration to work. Simple relies on annotations in the Java code to define how objects are to be serialised.[1]

In List. 3.1 a Java class annotated and ready for simple to serialise is shown.

```
@Root
public class Example {

   @Element
   private String text;

   @Attribute
   private int index;

   public Example() {
      super();
   }

   public Example(String text, int index) {
      this.text = text;
      this.index = index;
   }

   public String getText() {
      return text;
   }

   public int getIndex() {
      return index;
   }
}
```

Listing 3.1: A Java class annotated so Simple is able to serialise[1].

To serialise an instance of the above object, a Persister is required. A persister is given a model instance and can then output XML. An example of how this is done is shown in List. 3.2

```
Serializer serializer = new Persister();
Example example = new Example("Example message", 123);
File file = new File("example.xml");

serializer.write(example, file);
```

Listing 3.2: A persister is used to serialise an instance of the Java class.[1]

The model is populated with data, the index is set to `123` and the text is set to `Example message`. The model is then serialised. The resulting XML representation is shown in List. 3.3.

```
<root index="123">
   <text>Example message</text>
</root>
```

Listing 3.3: The resulting XML representation after serialising a model instance with some data.[1]

3.2.2 Serialising With XStream

Using XStream the elements created when serialising is by default the same as the object names. Though it is possible to add annotations, this is done in a very similar way to Simple. See List. 3.4. All snippets are from the tutorial. [2]

```
XStreamAlias("message")
class RendezvousMessage {
  @XStreamAlias("type")
  private int messageType;

  public RendezvousMessage(int messageType) {
    this.messageType = messageType;
  }

}
```

Listing 3.4: Annotating a class for serialisation with XStream. [2]

The serialiser must then be informed to use these annotations, an example of a main method serialising the class can be seen in List. 3.5.

```
public static void main(String[] args) {
  XStream stream = new XStream();
  xstream.processAnnotations(RendezvousMessage.class);
  RendezvousMessage msg = new RendezvousMessage(15);
  System.out.println(stream.toXML(msg));
}
```

Listing 3.5: Main class for serialisation with XStream. [2]

And the resulting XML is as in List 3.6.

```
<message>
  <type>15</type>
</message>
```

Listing 3.6: Result of serialisation of the class in List 3.4 with XStream. [2]

The methods used are very similar to what is seen in Simple. Using annotations directly in the code, and informing the code of these, lets a developer customise the serialisation.

3.2.3 Serialising With Java Architecture for XML Binding

Java Architecture for XML Binding (JAXB) is a much more feature rich serialiser. The setup is a bit more dificult, but it also gives room for many more modifications. In List 3.7 an example of how JAXB can be used to serialise an object can be seen. Example from the documentation. [3]

```
//The class declares importing of three standard Java classes, five
     JAXB binding framework classes and the primer.po package:
import java.io.FileInputStream;
import java.io.IOException;
import java.math.BigDecimal;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;

import primer.po.*;

//A JAXBContext instance is created for handling classes generated
    in the primer.po.
JAXBContext jc = JAXBContext.newInstance( "primer.po" );

//An Unmarshaller instance is created, and the po.xml file is
    unmarshalled.
Unmarshaller u = jc.createUnmarshaller();
PurchaseOrder po = (PurchaseOrder)
    u.unmarshal(new FileInputStream("po.xml"));

//set methods are used to modify information in the address branch
    of the content tree.
```

```
USAddress address = po.getBillTo();
address.setName("John Bob");
address.setStreet("242 Main Street");
address.setCity("Beverly Hills");
address.setState("CA");
address.setZip(new BigDecimal
address.setName("John Bob");
address.setStreet("242 Main Street");
address.setCity("Beverly Hills");
address.setState("CA");
address.setZip(new BigDecimal("90210"));

// A Marshaller instance is created, and the updated XML content is
     marshalled to system.out. The setProperty API is used to
    specify output encoding; in this case it is formatted (human
    readable) XML.
Marshaller m = jc.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(po, System.out);
```

Listing 3.7: Serialising using JAXB.[3]

The serialiser comes with multiple features to customise. Among other features it is possible to provide better names, specify meaningful package names derived from an URI and change namespaces. It is possible to provide both inline annotations in a schema or provide declarations in an external file. [3]

3.2.4 Serialising With EMF XMI serialiser

The default serialiser for new models created with EMF, is the EMF XMI serialiser. This serialiser, is as, JAXB highly customisable. An example of how it is used is seen in List 3.8. [4]

```
import java.io.IOException;
import java.util.Collections;
import java.util.Map;

import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.resource.ResourceSet;
import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;

import datamodel.website.MyWeb;
import datamodel.website.Webpage;
import datamodel.website.WebsiteFactory;
import datamodel.website.WebsitePackage;
import datamodel.website.impl.WebsitePackageImpl;

public class CreateSaveTester {
```

```
/**
   * @param args
   */

  public static void main(String[] args) {
    // Initialize the model
    WebsitePackage.eINSTANCE.eClass();
    // Retrieve the default factory singleton
    WebsiteFactory factory = WebsiteFactory.eINSTANCE;

    // create the content of the model via this program
    MyWeb myWeb = factory.createMyWeb();
    Webpage page = factory.createWebpage();
    page.setName("index");
    page.setDescription("Main webpage");
    page.setKeywords("Eclipse, EMF");
    page.setTitle("Eclipse EMF");
    myWeb.getPages().add(page);

    // As of here we preparing to save the model content

    // Register the XMI resource factory for the .website extension

    Resource.Factory.Registry reg = Resource.Factory.Registry.
        INSTANCE;
    Map<String, Object> m = reg.getExtensionToFactoryMap();
    m.put("website", new XMIResourceFactoryImpl());

    // Obtain a new resource set
    ResourceSet resSet = new ResourceSetImpl();

    // create a resource
    Resource resource = resSet.createResource(URI
        .createURI("website/My2.website"));
    // Get the first model element and cast it to the right type,
        in my
    // example everything is hierarchical included in this first
        node
    resource.getContents().add(myWeb);

    // now save the content.
    try {
      resource.save(Collections.EMPTY_MAP);
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
}
```

Listing 3.8: Serialising using the default EMF XMI serialiser. Example from Vogella. [4]

As the serialiser is part of the EMF project it is an obvious choice to use. Mappings can be defined both as annotations in the model or directly in the code, see List. 3.9.

```
EAttribute someAttribute = ...
XMLMapImpl map = new XMLMapImpl();
XMLInfoImpl x = new XMLInfoImpl();
x.setXMLRepresentation(XMLInfoImpl.ELEMENT); //can be altered to
    change presentation
map.add(someAttribute, x);
saveOptions.put(XMLResource.OPTION_XML_MAP, map);
```

Listing 3.9: Changing the representation using programming.[5]

3.2.5 Comparison of Serialisers

Using annotations in generated models mean that these must be set manually, unless a method for generating annotations in these models were developed, which could be possible. This adds an extra layer of development to a solution such as Simple and XStream. Since it must be ensured each time the model code is generated, the mappings are not removed or changed.

Both Simple and XStream do not require much work to get working, but both suffer from not being very flexible. It is not possible to make changes to the structure in XStream, and the possible changes in structure in Simple is not enough for this project.

JAXB and the EMF Serialiser is much more configurable. The serialisers only allow for minor changes in the structure, and cannot support all features used to serialise Petri nets as the standard defines. Neither of the serialisers support customisation of the serialisation without programming, unless annotations are used.

The EMF serialiser is highly connected with EMF and contains some good features, this serialiser has a good base to extend on.

The serialiser JAXB also contains some great features, e.g. the functionality to create shorthand names for namespaces. Ideas from this serialiser could be used in a solution building on the EMF Serialiser.

Chapter 4

Technological Background

Throughout the development of this project various technologies has been used. This chapter covers the technologies used, what they do, and how they are used.

The chapter covers the Eclipse Modeling Framework, and more specifically how serialisation in the framework is to be used. As well as how to extend plug ins in Eclipse and the key aspects of how to create domain specific languages.

The solution for this thesis is build upon Eclipse, which means the Eclipse Modeling Framework has influenced both the design and implementation of the project. The next section covers the, for this thesis, important aspects of EMF.

4.1 Serialisation in Eclipse Modeling Framework

Usually when developing software, the code is written line by line. This is error-prone and tedious process. The concept of model based software engineering (MBSE) is trying to improve the development of software. With MBSE the developer can create a model of the needed software and then generate parts of the software.

The Eclipse Modeling Framework is an official framework developed by the Eclipse Foundation. The framework is used to generate code from models. The code generated by EMF is only the structural parts, i.e. the code defining objects, relations between objects and modifications of these objects. EMF then also comes with various methods for modifying these objects.

EMF bases the generated code on Ecore models. Ecore is a meta model similar to MOF, MOF is its own meta model and can be used to describe itself, but also to describe UML. Ecore consists of classes, attributes and operations.[6] EMF comes along with support for change notification; an reflective API for manipulating the objects generally. By default EMF comes with an XMI serialiser, which can serialise any model defined.

Given an Ecore model, EMF can generate code for the model, adapters and the editors. The following is a description of each and what it contains.

Model creates the Java interfaces and implemented classes for each class in the model; a factory for creating objects of each class, as well as a package containing meta data about the implementation.

Adapters generate the implementation classes which are used to adapt the model i.e. how the classes are edited and displayed. In EMF this can be done using socalled ItemProviders.

Editor provides a customisable simple editor that can be used to modify the data stored according to the model. The editor is a simple structural tree editor, as opposed to a graphical editor.

That means it is possible to: create, edit, store and delete model instances, but a method to change how these instances are saved is needed.

Content in EMF is stored in `Resources` and can then be plugged in, and used in the editor.[1] A `Resource` can be a project, folder or file. An appropriate factory creates the resource as a persistent document. This document contains the content as well as diagnostics, possible errors and/or other problems.[7] More interesting for this project; it also holds an URI for where to load and save the contents.

In this thesis the format for serialisation is XML, so the interface `XMLResource` is used, which builds upon `Resource`. In Fig 4.1 an overview of how XML resources is used can be seen. An `XMLResource` has an object for loading and for saving.

[1]How this is done is covered in Sect. 4.2

Both uses a helper for serialising into XML, the helper used in this project is the default one; `XMLHelper`.

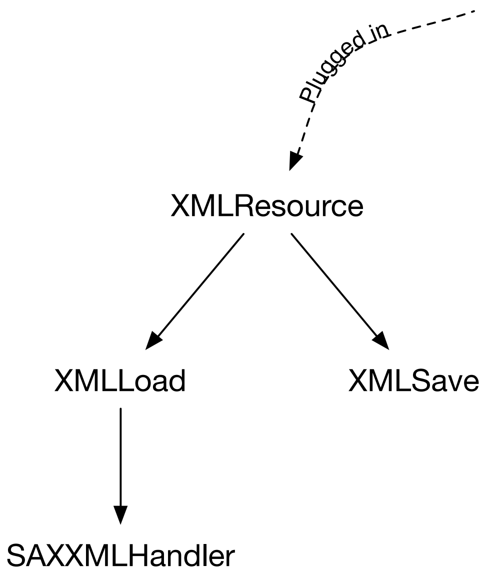


Figure 4.1: Resources are plugged in to Eclipse. A `XMLResource` needs an object for saving and loading. And for loading it needs a handler for the XML.

A `Resource` must implement public methods for saving and loading, this can be used when tailoring serialisation of own objects. To change the way objects are saved and loaded these methods can just be overridden to provide a custom serialisation of objects. When loading, a special handler for reading the XML is needed. It can either be a SAX or a DOM handler. A SAX handler is chosen as handler and covered in Chapter 7 — the implementation.

The next sections cover the serialisation in more detail. The first section, Sect 4.1.1, covers the most simple part of serialising; the process of saving. The next part, Sect 4.1.2, covers the opposite; the loading. The last section, Sect 4.1.3, covers the low level part, the `XMLHelper`; which is used both in the process of saving and loading.

4.1.1 Saving an Instance of a Model

When serialising, saving the model instance is the simplest. Saving a `XMLResource` in EMF is covered by the class `XMLSave`. The class contains a document object which is used to build up the serialised data.

In Fig 4.2 a very simple model is shown. In Fig 4.3 an instance of a such model is shown. The model contains a container object (the root object) and the elements A, B, C and D. An A contains a name and can contain a B or a D object. A B object can contain one or more C objects. And lastly a D object has also an attribute number.

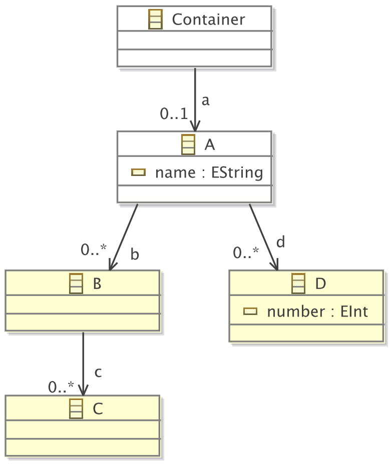
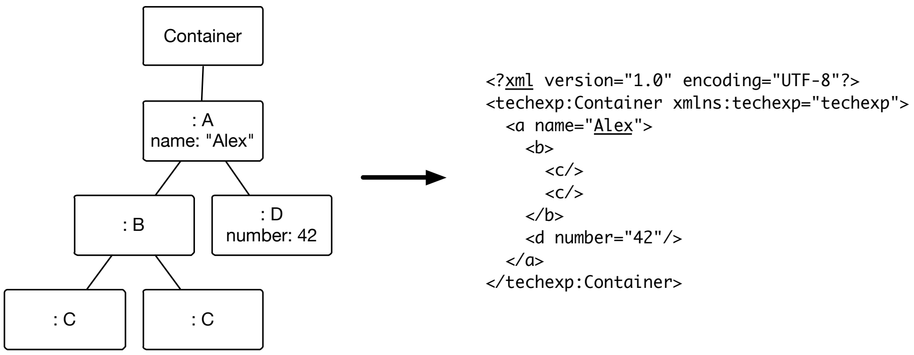


Figure 4.2: A simple model a developer wants to save instances of.



```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container xmlns:techexp="techexp">
  <a name="Alex">
    <b>
      <c/>
      <c/>
    </b>
    <d number="42"/>
  </a>
</techexp:Container>
```

Figure 4.3: Saving of a simple instance, of the model shown in Fig. 4.2

When serialising the model instance, the root object (Container) is handled by its own since it might contain special features. This is done in the method `save` after setting up the namespace and doctype. Each root object is traversed using the method `traverse`, this ensures that all root objects are saved. Each element is then saved using the method `saveFeatures`. Depending on the multiplicity of the element, either the method `saveContainedSingle` or `saveContainedMany` is called for single elements or higher multiplicity respectively. For each element the method `saveElement` is then called.

When elements and attributes have been saved on all root elements, the method `endSave` is called which handles the namespaces and cleans up the cache and various used lists.

That means, in the case of Fig. 4.3, when a save command is notified, the method will notice that there is only one root element. This root element Container, will be passed to the method `traverse` which will then send the element to `writeTopObject`. This will write the meta data and the container element to the document object.

The method `save` will then continue with the A object. The object will be passed to the method `saveFeatures` to save the features on the object. In this case it is the name attribute and the B and D objects. The name attribute will be passed to the method `saveContainedSingle` whereas both objects B and D will be send to the method `saveContainedMany` which both eventually will call `saveObject`. Which will obtain the values using the XMLHelper. When saving the object B, the method will realise it contains multiple objects which then will be passed on to the method `saveContainedMany`. When saving the D object, the method will realise it contains the number attributes which will be send to the method `saveContainedSingle`.

4.1.1.1 Key Methods of `XMLSave`

Tabel 4.1 is a description of some of the most important methods and the functionality of these.

save is the key method of this class, it receives the resource and the handler and calls the method `traverse` with the root elements.

traverse receives one or more root elements, the root elements are each saved using `writeTopObject`.

writeTopObject receives the root elements and handles the meta data if needed.

saveFeatures receives the object to save, loops over all the features on the object, and saves each feature depending on which kind the attached feature is.

saveContainedSingle works as above, but is only called if the attached feature is a contained single. The single element will then be saved using `saveElement`.

saveContainedMany is called in case of the feature is a "contained many" feature. This means that the feature is contained and has a multiplicity of many. Each value will be iterated over and saved using `saveElement`.

saveElement receives the objects to be saved.

Table 4.1: The most important methods of the `XMLSave` class.

Do note that due to the method save calls `traverse` with all the root elements, each root element is then looped over and each child object is received with `saveElement`. That means each object will be iterated over for each root object, resulting in every object being saved.

4.1.2 Loading an Instance of a Model

When loading, the data saved must be deserialised. When deserialising data the serialised format must be read and translated into the original data structure. The serialised format is XML so an XML file must be parsed and translated into the original EMF-objects which were saved. The process is the reverse as when saving, and can be seen in Fig. 4.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container xmlns:techexp="techexp">
  <a name="Alex">
    <b>
      <c/>
      <c/>
    </b>
    <d number="42"/>
  </a>
</techexp:Container>
```

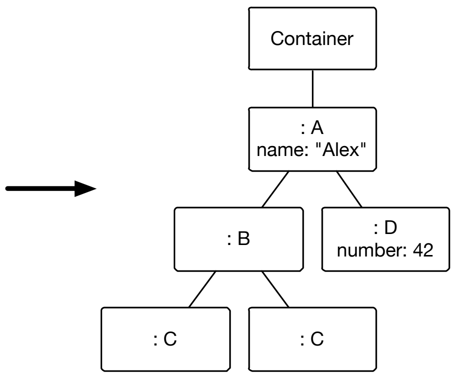


Figure 4.4: Loading of a simple model instance.

The serialisation is usually handled in `XMLLoad` with the help from a handler. The standard is a standard SAX parser; the `SAXXMLHANDLER`. `SAXXMLHANDLER` is, as the name suggest, a SAX parser; which in contrast to a DOM parser, only reports each parsing event as it is happening and then discards almost all of the information on the event when it is reported. A DOM parser would load the whole file into memory, having no events, and then it would be possible to modify the parsed data.[8]

Serialisation of the whole document is done in the implementation of `XMLLoad`. For the root element and recursively for the child elements to the root elements; the method `traverse` is called. This ensures that all child elements to the root element will be serialised; meaning all elements would be visited once. When the elements are traversed the information obtained is send to the handler.

Before any elements are handled the encoding is extracted and various meta data are set. The first element, the root element, is special in comparison with its child elements, and must be handled such. `createTopObject` and `createDocumentRoot` are methods created to handle the root of the document.

When an element is read, that being the document root or just any object, the method `startElement` is called. The attributes for the element is then handled in the method `setAttributes`. Then the element is processed in the method `processElement`, which tries to create the object either based on the type of feature or from a provided factory. At last in the handling of the start element the method `handleObjectAttribs` is called to handle the attributes on the object. For each attribute to the element, the method `setAttribValue` is called.

If the element has any child elements, these are now traversed by `XMLLoad`, any child element is traversed as described above, that results in any child element of the child element is traversed, resulting in traversing all the objects.

When all child elements have been traversed, the end element is reached and the method `endElement` is called. A list of open elements is part of the handler, and is handled by the handler. If no child elements exist, the end tag is reached, this also applies if the end tag is inlined in the tag, `endElement` is called.

That means when serialising the example in Fig. 4.4, the method `load` will be notified that changes should be saved. The load method will then traverse the root object Container. The root object will be passed to `createDocumentRoot` which creates the element and passes it onto `processTopObject`.

Then the `traverse` method will traverse the element in Container. The element A will be send to `createObject` to be created, and processed by `processObject`. The element is now created and processed, but attributes still needs to be set. The object is sent to `handleObjectAttribs` which will for each attribute; set the attribute using the method `setAttribValue`.

The rest of the elements will be created using `createObject` and then processed by `processObject`. If the element has any attributes, these will be handled by `setAttribValue`.

4.1.2.1 Key Methods of `XMLLoad` and `XMLHandler`

Tabel 4.2 is a description of some of the most important features and the functionality of these. This list covers methods in both `XMLLoad` and `XMLHandler`. Load and traverse are located in `XMLLoad`, the remaining methods are located in `XMLHandler`.

load is the most important method and is called when a save event is invoked.

traverse traverses any element. The element is traversed by creating and processing the object as well as setting the attributes. If the element contains other elements these are traversed as well.

createDocumentRoot Creates the root element, and hands it to `processTopObject` for processing.

processTopObject maintains the data structure for which objects are serialised.

createObject creates other objects based on the type of these. Hands the created objects to `processObject`.

processObject as `processTopObject`, maintains the data structure.

handleObjectAttribs handles the attributes on the object, each attribute is send to the method `setAttribValue`

setAttribValue sets the attribute value depending on which feature the attribute is.

Table 4.2: The most important methods of the classes `XMLLoad` and `XMLHandler`.

4.1.3 Low Level Serialisation

To be able to keep the serialisation at a high level of abstraction, and to be able to reuse functionality, the low level functionality of loading and saving is refactored into a helper. This helper is then used both in the stage of loading and saving.

XMLHelper is a helper provided by EMF. The helper class helps with most of the low level parts. It can retrieve the name of elements, attributes on elements, name space and like when loading. When saving, the helper can be used to create the XML elements.

One of the most frequent uses of the helper is to go from an object and a feature[2] to having an actual value to be saved for the feature. This is done in the implementation of `XMLSave`; the class `XMLSaveImpl` in the method `saveContainedSingle` on line `2398`, which can be seen in List. 4.1.

```
protected void saveContainedSingle(EObject o, EStructuralFeature f)
    {
  InternalEObject value = (InternalEObject)helper.getValue(o, f);
  if (value != null) {
    saveElement(value, f);
  }
}
```

Listing 4.1: Addition to `plugin.xml` to plug in a custom serialiser.

4.2 Extending Functionality in Eclipse

In Eclipse it is possible to let other plug-ins be responsible for functionality. This is done by letting plug-ins define extension points where other plug-ins then can plug in own functionality.

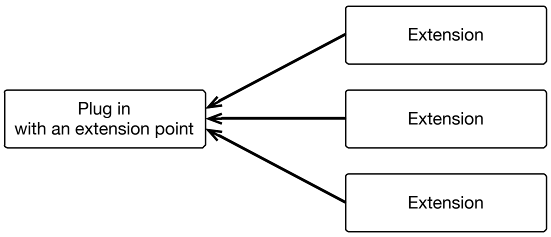


Figure 4.5: By defining extension points in Eclipse it is possible to let other plug-ins be responsible for functionality.

Various extension points are already available in EMF. For example the serialiser for a plug-in be specified as an extension point, which will override the standard serialiser. This will be used for plugging in the serialiser developed in this thesis.

Additionally, new extension points can be created. New extension points can be defined in the file `plugin.xml` in Eclipse. An extension point can contain strings, booleans or even Java classes. It can also contain a `Resource`. In

[2]Which is what is parsed by the handler.

Eclipse a `Resource` can be any file or folder, both local or non-local (e.g. from a repository).[9][10]

4.3 Domain Specific Languages

In contrast to programming languages which are general purpose languages, a Domain Specific Language (DSL) is specific to a certain domain. This means the language is not generally usable, but usable in some special context.

DSLs comes with some great advantages, but it also have its drawbacks. The largest advantage for this system is the level of abstraction. When abstracting away most of the implementation details, the developer does not need the same amount of knowledge about the serialiser to be able to use it.

Some of the drawbacks is that a developer now needs to understand an extra language, which might not be as widely used as a general purpose language. Additionally, editors and other programming tools might not be available.[11][12]

Chapter 5

Software Design

This chapter covers the software design choices made in this thesis. Based on the knowledge provided in the analysis, Chap. 3 and the technological background, Chap. 4, various design choices have been made. In Sect. 5.1 the overall structure of the solution is described. This section also covers the design of the features and concepts listed in the analysis:

- Context dependent mappings
- Multiple versions
- Standard Mappings
- Cascading Mappings
- Dynamic Mappings
- References in Data
- Loading Packages

And at last, in Sect. 5.4, a domain specific language to define mappings is defined.

5.1 Overall Structure

In Fig. 5.1 the structure of the solution can be seen. Some of the structure is already given by the chosen framework, EMF. Due to the way plug-ins are plugged in, in Eclipse; the product is split up into three stages.

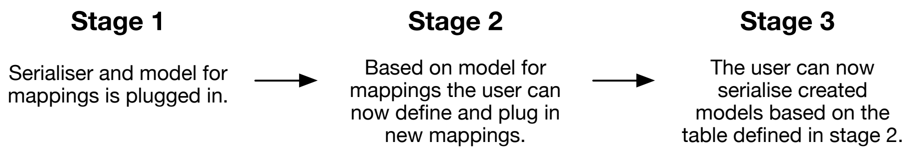

Stage 1
Serialiser and model for mappings is plugged in.
Stage 2
Based on model for mappings the user can now define and plug in new mappings.
Stage 3
The user can now serialise created models based on the table defined in stage 2.

Figure 5.1: The solution is contained in three parts. Each arrow illustrates a new started runtime workbench or install of the previous stage.

When a developer wants to use the solution, the developer must follow the three stages. In the first and second stage, various parts are plugged in. To be able to use the plugged in parts the parts must be installed or run in a runtime workbench. [13]

The First Stage is the development workbench. This stage will not be visible for regular users of the serialiser, but is visible when developing the serialiser and a model for mappings is visible in this thesis and power users wanting to get to know how the serialiser and model is developed.

The Second Stage is the runtime workbench when running the first stage, this will be called the mappings workbench. In this workbench the model for mappings is plugged in along with an editor to create these.[1] In this workbench a user is able to create his mappings, and plug these in, into Eclipse.

The Third Stage is the runtime workbench of the second stage. In this stage both the serialiser and the mappings defined in the second stage is plugged in. This means in this final stage the user is able to serialise his model instances according to the mappings defined in the second stage.

5.2 Features of the Mapping Mechanism

The following sections cover the chosen design for the features in the serialiser, the features are described in the analysis.

[1]Actually, the way the project is laid out, the serialiser is also plugged in at this stage, though the mappings must be defined before we can serialise. To avoid running three versions of Eclipse, the mappings can be created in the second stage and copied and plugged in, in the first stage.

5.2.1 Context Dependent Mappings

Some objects might need to be serialised differently depending on which context they are in.

This is done by letting the user specify which context the mapping should apply in. A mapping contains a reference to a container object which defines which parent object is needed for a mapping to apply. The container definition can be left blank if the developer wants to specify this mapping for all contexts.

Working with the example from Sect. 3.1.1 on page 12 where a developer wants to serialise the token as an element `star` only when contained in a transition. Given the same model, the following mapping table, Tabl. 5.1, would solve the developers problem.

#	**Feature**	**Container**	**Element-name**
1	token	transition	star
2	token	place	token

Table 5.1: An example of a mapping table solving the problem in Sect. 3.1.1.

5.2.2 Multiple Versions

Multiple versions of a model might exist in which the serialiser must be able to account for. To be able to know what model is in use, the representation should somehow reflect which model is used.

This could be done using an attribute e.g. in the root element. In PNML it is seen as the attribute `version` which then refers to a value of which version it could be, e.g. `2`.

This will be supported by being able to extend mappings, with other mappings when some attribute and value match.

#	Attribute	Value	Table to use
1	version	2008	2008.mappings
2	version	2014	2014.mappings

Table 5.2: An example of a mapping table solving the problem of multiple versions of mapping tables.

As an example, we say that we have a representation created in 2008 and a new representation created in 2014. The mappings for these representations are saved in the files `2008.mappings` and `2014.mappings` respectively. If the model then contain an attribute called version on the root element, we can difrentiate between the two models. If the attribute `version` is `2008`, we use the file `2008.mappings`. In case of the attribute is `2014`, the file `2014.mappings` is used instead.

5.2.3 Standard Mappings

To ease the work of the developers using this serialiser, it would be beneficial if only the special kind of mappings needs to be defined.

The design is build upon the serialiser already existing in EMF. This serialiser serialises into XMI and contains a good set of rules. If no other mapping is usable, the object is send to the already existing serialiser to be serialised.

5.2.4 Adding Tables

Now that having multiple mappings for each element is allowed, additional ways to add mappings could be useful. New tables could be added given their path, and when the mapping applies, the defined mapping table is added.

5.2.5 Cascading Mappings

Allowing developers to rely on a set of already defined mappings and be able to extend with their own mappings raises some issues. Now multiple mappings for one object and element relation can exist. To keep a sensible representation the serialiser should only create one element per object which means the serialiser will only rely on one mapping per object. Which mapping used should be chosen carefully from the following set of rules.

Standard Mappings take the least precedence, and should only be relied on if no other mappings apply.

Mappings Extending a Table should take precedence. Mapping tables can be extended with new mappings. The deepest mappings should take precedence.

More Precise Mappings should take precedence, that means e.g. mappings with a matching container should take precedence over mappings without a container defined.

5.3 Concepts

In the analysis multiple concepts were also described. In this section the concepts of the serialiser are described.

5.3.1 Defining and Using Mappings

Non-dynamic mappings would mean that a developer just wanting to serialise a model instance; would need to download the source code of the serialiser, compile and run it. Define the mappings, copy these mappings into the workbench with the source code, run it again. And then now would be able to serialise model instances.

Instead, the mappings defined in the second stage (refer to Fig. 5.1 on page 40) should be plugged in anywhere in Eclipse. As described in Sect. 5.1, to be able to use plugged in resources, the developer must be running an instance of Eclipse inside. This means that with dynamic mappings, the developer does not need to do anything in the first stage.[2] And the developer can now plug in mappings from any project, allowing the developer to have multiple projects sharing mappings, and projects adding mappings upon other project's mappings.

[2] If the developer want extra knowledge of the process of serialising, he should and will still be able to download and study the source code of the serialiser.

5.3.2 References to Other Objects

Model instances to be serialised might contain references to other objects. This could be an element having a reference to another element, which must be handled somewhere.

Before the serialisation, the object will have some sort of reference in memory to where the object is located. When serialising, this reference needs to be translated into a reference which is understandable in the given representation.

The design choice in Sect. 5.2.3 will help us in this case. Using mappings from XMI when there are no defined mappings, will solve this. XMI already handles references made between objects, if it is ensured that this feature is not removed (e.g. by overriding it) references will work.

5.3.3 Defining Packages

When defining mappings precisely, describing which object to be serialised is important. Mappings can be defined on objects to any package, not just the package the mapping is defined it. Due to the design of the Java language it can be assumed that object names in a package is unique.[14] Though it cannot, and it would not make sense, to assume that object names spanning multiple packages is unique.

Instead it should be possible to specify in which package a certain object is located in. If packages are already specified, these could be loaded lazily to save time. When describing in which package an object is located in, it would be easier to only specify a shorthand (unique) name of the package. If all packages are to be defined anyway, this could be combined into specifying a shorthand name and a url for all packages used. These could then be used when referring to objects. Shorthand names for packages should be available for the serialiser at least at compile time, otherwise the serialiser cannot locate the packages.

#	**Prefix**	**Package URL**
1	se	`http://student.dtu.dk/s093259/serialExample`
2	core	`http://org.pnml.tools/epnk/pnmlcoremodel`

Table 5.3: An example of giving two packages a prefix.

Table 5.3 shown two examples, where packages are given a prefix. The prefix can be any string, but choosing a long prefix defeats the purpose. Now when creating a mapping, e.g. the container object can be defined by the object name and package (given by the prefix).

5.4 Domain Specific Language for Mappings

To be able to contain all the features described in Sect. 5.2 a DSL is created. A DSL is chosen based on the knowledge of pros and cons from Sect. 4.3. Advantages of a DSL are that a developer without much knowledge about the implementation of the serialiser still can use the tool. And if the language is intuitive, the cost of learning the DSL should not be too high. The drawback of the lack of development tools and editors for a new DSL is solved by generating these using EMF.

In the DSL the developer must be able to define both mappings and references to packages. The DSL will then be plugged in into Eclipse. To avoid multiple files, thus keeping it simple, only one DSL is specified. This DSL contains both definitions of mappings and packages. References from packages do not need to be specified in the same file as the mapping in which they are used in.

Having only one DSL still allows developers to define packages and mappings separately in multiple files and plug all of them in.

An example of a definition of a prefix for a Package can be seen in Tabl. 5.3 on Page 44. The example shows how two long urls are related to a prefix. Now this prefix can be referred instead of writing the long URL.

A developer can also define mappings. Mappings can map various functionality; to support that, an enumeration defining what functionality should be mapped is part of the mapping.

An example of a mapping defined can be seen in Tabl. 5.4. The mapping uses the feature “Object from attribute” on the object `Object` contained in the object `Page`. The feature requires values for the attribute name, value (to be matched to create the element) and target (which element should be created).

Option	Feature	Container	Attribute	Value	Target
`OBJECT_FROM_ATTRIBUTE`	core:object	core:Page	type	place	Place

Table 5.4: An example of a mapping.

In Fig. 5.2 the meta model for mappings and packages can be seen.

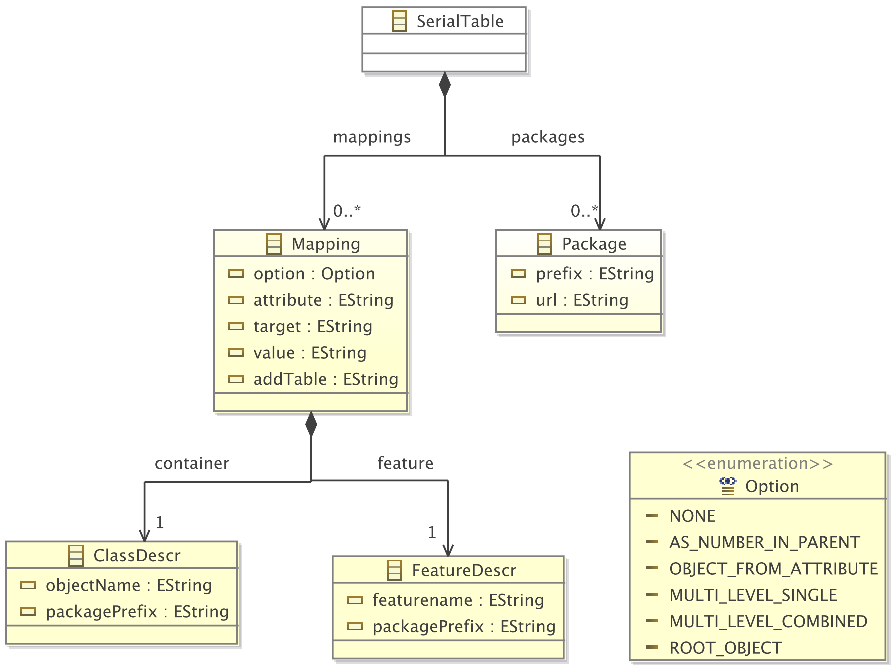


Figure 5.2: The Ecore diagram of the table defining mappings.

A mapping contains a field for which feature is to be mapped, features are an essential part of EMF and are used to specify a specific functionality e.g. a variable on an object. This is used to specify which object or part of object is to be serialised into what.

The container object is the `SerialTable` and contains a list of zero or more mappings and zero or more packages.

Packages are defined by a string defining the prefix, and a string giving the URL for the package, the URL for a package is unique. Internally in the serialiser, there exists a hashmap associating the prefix with the loaded models. The models are lazy loaded when needed, this is done to only store the necessary models. Storing any model defined in any mapping table plugged in could lead to a large memory usage.

A mapping consists of the definition of seven elements:

Option defines which feature is wanted for the mapping, five different features can be chosen, these are covered in the next section.

Feature defines which feature in the model the mapping is defining. The feature is defined by the feature name but also in which package it is located in. The package is described using a prefix defined in the mapping table.

Container defines the context the features are presented in. This can be left empty or contain a parent feature. A Container is defined by the name of the container but also in which package it is located in. This is defined by the prefix of the package.

Attribute defines a name for an attribute which with the corresponding value must match.

Value is the value needed to be matched.

Target is the target object for the mapping. E.g. which object to create in the "object from attribute"-feature.

Add Table can be used to append an extra mapping table when the mapping matches.

5.4.1 Features of the New Serialiser (Options)

The enumeration in the Ecore diagram shows the five different features which can be chosen, a value of `none` can be chosen in case of no extra feature is wanted. The choices can be seen in the following list.

None is chosen when no extra feature is needed. This could be when an object is to be ignored e.g. when only an add table feature is wanted.

As number in parent is chosen when an object needs to be serialised as attributes.

Object from attribute is used when an object is to be defined differently depending on attributes on the object.

Multi level single is used when the element need to be serialised on multiple levels. In this case the elements are serialised into an extra level only containing a single element.

Multi level combined is the same as the above feature. But instead of serialising into separate elements, all objects are serialised into the same element.

Root object is used when defining a special mapping for only the root object. This is to be used if extra mapping tables are to be loaded dependent of the root object. If the root object matches the mapping the corresponding table is added.

Chapter 6

Handbook for Developers

To be able to use the software developed in this thesis for a new Ecore model the following steps should be followed:

1. Installing the Eclipse with EMF and Ecore.
2. Installing the serialiser and mapping mechanism.
3. Adding mappings for the Ecore model.

The steps are described in the following three sections.

6.1 Installing Eclipse with EMF and Ecore

The following steps must be followed in order to install EMF and EMF (Ecore) models. [15]

- If not already installed, install JRE 6.0 or newer from `http://www.oracle.com`

- Download Eclipse classic from `http://www.eclipse.org`
- Extract Eclipse in a proper location and start the executable.
- Choose a proper path for the workspace when prompted.
- Go to the “Help”-menu and choose “Install New Software...”
  - Select the official software site for your installation of eclipse e.g. Luna.
  - Search for and install EMF Eclipse Modeling Framework SDK.
  - Search for and install Ecore Tools SDK.
- Again go to the “Help”-menu and choose “Install New Software...”
  - Enter the url for the ePNK update site `http://www2.imm.dtu.dk/~ekki/projects/ePNK/indigo/update/`
  - Search for and install all ePNK packages except experimental packages.

6.2 Installing the Serialiser and Mapping Mechanism

You should have received a package with the software developed in this project along with this report. A CD is part of the appendix of the printed version. The software can be requested by e-mail on `s093259@student.dtu.dk`

The software developed in this project is added to Eclipse by following the wizard from Import... in the File menu.

When creating new models, the standard XMI serialiser is still chosen as default. For projects where the serialiser developed in this project, along with the mapping mechanism is wanted it should be plugged in.

This is plugged in, by editing the `plugin.xml` file in the project directory. The serialiser is plugged in by adding the lines in List 6.1

```
<extension point="org.eclipse.emf.ecore.extension_parser">
  <parser type="serialExample"
       class="serializer.MyResourceFactory" />
</extension>
```

Listing 6.1: Addition to `plugin.xml` to plug in the developed serialiser.

6.3 Adding Mappings for an Ecore Model

When a model is created you kan now add mappings to the model. This is done by running the provided software to access the developed editor for mappings.

You can now create mappings for your Ecore model.

Now either another workbench can be started, or the mappings can be copied over to the development workbench (the workbench with the code for the serialiser).

To be able to use the mappings, these needs to be plugged in, this is again done in `plugin.txt` and can be done by adding the contents of List 6.2.

```
<extension
      id="1st"
      name="The first table"
      point="dk.dtu.se.SerialDefinition.TableExtension">
   <table
         auto_load_on_extension="serialexample"
         description="A table defining the mapping rules for ..."
         resource="mappings/1.serialdefinition">
   </table>
</extension>
```

Listing 6.2: Addition to `plugin.xml` to plug in a new mapping table.

Or alternatively by the graphical editor for `plugin.xml`, the Plug-In Manifest Editor, which is the standard editor for the file. Go to the tab extensions and press add. Now search for the extension point `dk.dtu.se.SerialDefinition.TableExtension` and fill out the required fields.

Chapter 7

Implementation

This section covers the implementation of the mapping mechanism and the serialiser.

As seen in Fig. 7.1, to be able to use the serialiser, the serialiser must be plugged in, and associated with the proper file extension. EMF Ecore provides an extension point for this. This is covered in Sect. 7.1. And as described in the design chapter, mappings needs to know what needs to be serialised and how. This is covered in Sect. 7.2.1, which also covers the editor for creating the mappings. The mappings can define various functionalies, these individual features are covered in the realisation of the serialiser section, Sect. 7.3.

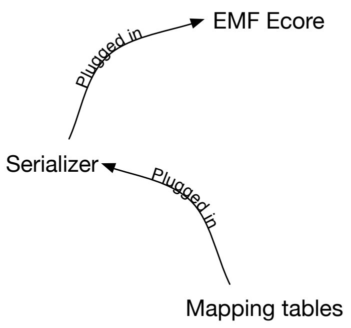


Figure 7.1: The implementation consists of a serialiser which is plugged in into EMF Ecore. The serialiser relies on mappings, a model for mappings is created, as well as a extension point so that the serialiser can be extended with mappings.

7.1 Overriding the Standard Serialiser

As described in Sect. 4.2, extension points are used to plug in own projects in to existing projects. EMF provides an extension point where it is possible to extend the resource with a custom parser, depending on the extension of the file. That means to be able to plug in a custom serialiser, it must be added to the extension point provided by EMF. This can be done by adding the contents of List 7.1 to the file `plugin.xml`.

```
<extension point="org.eclipse.emf.ecore.extension_parser">
  <parser type="serialExample"
      class="serializer.MyResourceFactory" />
</extension>
```

Listing 7.1: Addition to `plugin.xml` to plug in a custom serialiser.

This means that the developed serialiser, created by the factory `serializer.ABCResourceFactory`, will be used for files ending with `.serialexample`. This is done by the developer wanting to use the serialiser, as described in the handbook in Chap. 6.

The serialiser needs mappings to know what and how to serialise objects, the implementation of mappings is covered in the next section.

7.2 Mappings for the Serialiser

To know how to define mappings, it must first be defined what is needed to define a mapping; a model for mappings is needed. A table is an obvious choice to define mappings in, and is also the chosen format. Instead e.g. a linked list could be chosen. But since the mapping tables can contain a mapping table, creating a linked list. Mappings in a linked list is chosen for a more simple understandable design. Additionally tables (realised as object in an `ArrayList`) performs better than linked list (realised as `LinkedList`) if more lookups are needed than inserts. [16]

Creating a model with EMF, which already is in use in this project, is an obvious choice. The Ecore model defined can be seen in Fig. 7.2.

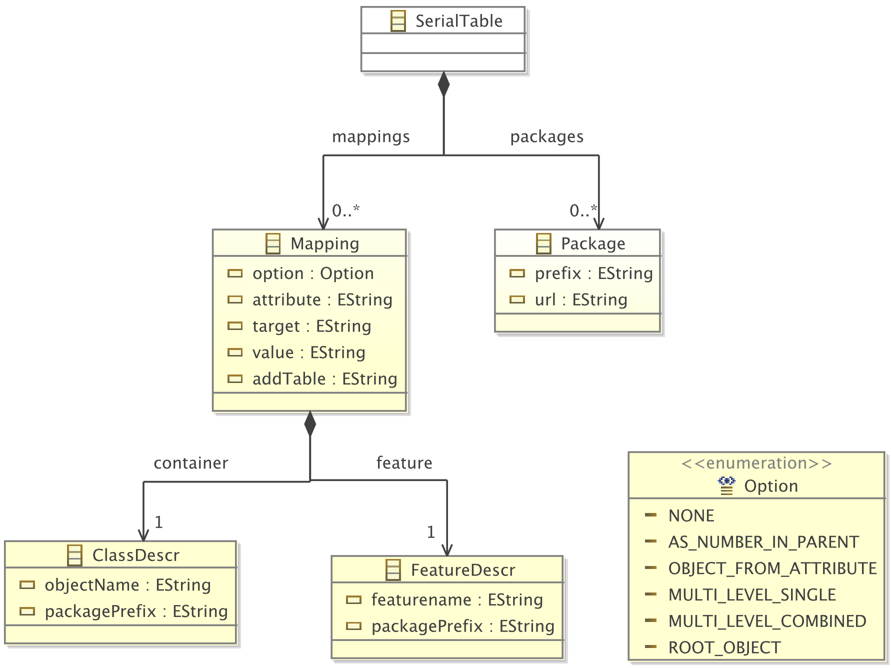


Figure 7.2: The Ecore diagram of the table defining mappings.

The design of the model is covered in Sec. 5.4. In Eclipse, Ecore models can be created using the Sample Ecore Model Editor, which is a simple tree editor. For this project an Ecore Diagram is created and edited with the accompanying Ecore Diagram Editor. The editor displays the model, as what is shown in Fig. 7.2.

To be able to generate the model code and editor, a EMF Generator Model is needed. When a such is created, various settings can be set, e.g. in which project should the code be generated. When the model, edit and editor code is generated, the editor can now be run in a run-time workbench.

Now a model and an editor for the mappings exist, but the mappings must be used in the serialiser. To be able to do so, the mappings should be plugged in, so these can be used by the serialiser.

7.2.1 Plugging in Mappings

In EMF, extension points are used to plug in own projects in to existing projects as covered in Sect. 4.2. This is good for extending projects where the user do not want to work with the source code of the project, or just easily want to plug in extra features or data for another project. This is exactly what a developer want to achieve. This makes using extension points to plugging in mapping tables the right choice.

In Eclipse new extension points can easily be created, these are to be defined in the file `plugin.xml`. An extension point is defined by just a name, identifier and an XML schema. The extension point for mapping tables is shown in List 7.2.

```
<extension-point id="TableExtension" name="Table Extension" schema
    ="schema/TableExtension.exsd"/>
```

Listing 7.2: Addition to `plugin.xml` to create new extension points, other developers can extend.

Now an extension point for mappings is created. But the XML schema must also be defined, defining what is needed when adding a table. The schema consists of an extension point, an identifier, a name and a sequence of mapping tables. In order for other people to define mapping tables, these must be exported by the project, this is default behaviour.

The extension point is used to define the kind of extension is expected, in this case it is a mapping table. An identifier and a name are used to identify the extension and refer to it. And the sequence of tables are the actual mapping tables needed in the serialiser along with a description for the tables. A way to know which tables should automatically be added to which models is also needed. Generally, when plugging in a serialiser, the file extension of the model instance is used to determine which serialiser is used. This can be reused to make it possible to define for which file extensions which mapping tables should be automatically added.

An example of a mapping table being plugged into the serialiser can be seen in List. 7.3.

```
<extension
        id="1st"
        name="The first table"
        point="dk.dtu.se.SerialDefinition.TableExtension">
  <table
        auto_load_on_extension="serialexample"
        description="A table defining the mapping rules for ..."
        resource="mappings/1.serialdefinition">
  </table>
</extension>
```

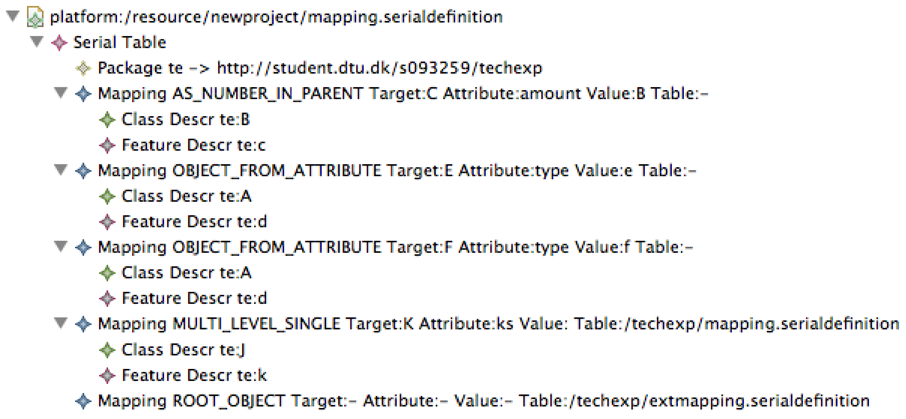
Listing 7.3: Addition to `plugin.xml` to plug in a new mapping table.

This is to be done for each new model, as described in the handbook in Chap. 6.

7.2.2 The Editor for Mappings and Packages

As described in the section about EMF, Sect. 4.1, when creating a model in EMF, the model comes with adapters and an editor.

The editor is a simple editor with ability to create, edit and delete new mappings. The mapping tables can then be saved in the standard XMI format, stored, and then loaded again.[1] A screenshot of the editor can be seen in Fig. 7.3.



```
platform:/resource/newproject/mapping.serialdefinition
  Serial Table
    Package te -> http://student.dtu.dk/s093259/techexp
    Mapping AS_NUMBER_IN_PARENT Target:C Attribute:amount Value:B Table:-
      Class Descr te:B
      Feature Descr te:c
    Mapping OBJECT_FROM_ATTRIBUTE Target:E Attribute:type Value:e Table:-
      Class Descr te:A
      Feature Descr te:d
    Mapping OBJECT_FROM_ATTRIBUTE Target:F Attribute:type Value:f Table:-
      Class Descr te:A
      Feature Descr te:d
    Mapping MULTI_LEVEL_SINGLE Target:K Attribute:ks Value: Table:/techexp/mapping.serialdefinition
      Class Descr te:J
      Feature Descr te:k
    Mapping ROOT_OBJECT Target:- Attribute:- Value:- Table:/techexp/extmapping.serialdefinition
```

Figure 7.3: The simple editor for editing mappings. The editor is part of what is generated by EMF.

As described in Sect. 4.1 the project also come with adapters which are used when modifying and displaying the model. By standard in EMF only the first attribute on an object is shown in the editor. This can be changed by changing the generated method which defines the textual representation of an object. For example the textual representation of a `Package` can be changed in the item provider class `PackageItemProvider` as shown in List 7.4. Note the JavaDoc annotation `@generated NOT` which ensures that EMF does not overwrite the method when generating code.

```
/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated NOT
 */
```

[1]It is even possible to create a mapping table for defining the serialisation of mapping tables.

```
  @Override
  public String getText(Object object) {
    String labelUrl = ((serialDefinition.Package)object).getUrl();
    String labelPrefix = ((serialDefinition.Package)object).
        getPrefix();
    return (labelUrl == null || labelUrl.length() == 0 ||
        labelPrefix == null || labelPrefix.length() == 0) ?
      getString("_UI_Package_type") :
      getString("_UI_Package_type") + " " + labelPrefix + " -> " +
          labelUrl;
  }
```

Listing 7.4: By changing the item provider for packages, the textual representation of a package can be changed. By standard only the URL was shown.

The editor is updated to show values for all attributes to give a better overview and support printing from the editor without losing information.

7.3 Realising the Serialiser

As described earlier, all files, folders and projects is realised in Eclipse as Resources. To plug in the developed serialiser, the save and load features can be overwritten of a standard resource-class from Eclipse. Since this project will only work with XML as format, the class `XMLResource` is an obvious choice. Since the `XMLResource` comes with some nice to have features for saving XML. A class diagram over the serialiser can be seen in Fig. 7.4 on Page 60.

In Eclipse, resources are to be created by a factory following the creational design pattern for factories. A such factory is created by extending the `ResourceFactory` already in EMF. The new factory is then extended with functionality to create our own resource, containing our own save and load functionality.

To be able to use the serialiser, the factory created, is then plugged in and associated with a proper file extension. As covered in Sect. 7.1. The resource contains the class `MappingTable` which is a class wrapping some functionality around the class `SerialTable`, which contains the actual mappings and packages.

In the following sections the save, load and table functionality will be described.

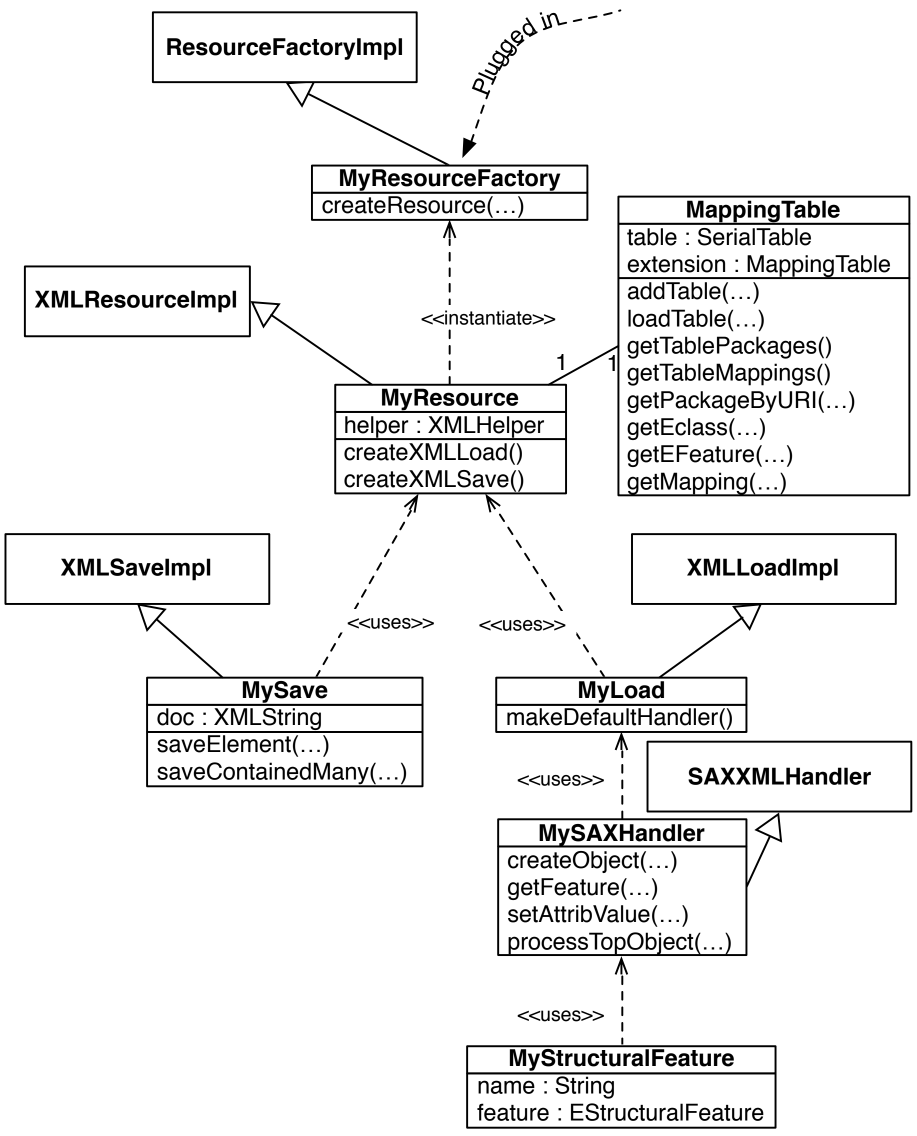


Figure 7.4: Class diagram for the serialiser. Extended classes are from the EMF Ecore package. This diagram contains all classes but only highlights the important attributes and methods.

7.3.1 Saving Instance – `MySave`

When saving (and loading), when no mappings are present, the serialiser should fall back to the standard XMI mappings. This means that when creating the save functionality, all functionality from the class `XMLSaveImpl` can be used, and only the special cases must be handled. The features needed for the serialiser defined in Sec. 5.4.1 can be realised by overriding just two methods. The method `saveElement` and `saveContainedMany`.

7.3.1.1 Saving an Element – `saveElement`

The method `saveElement` which takes an object (`InternalEObject`) and a feature (`EStructuralFeature`) is, as expected, responsible for saving the element. The object is the object being saved, and the feature is the feature from the model on where to save it, e.g. as an attribute on another object (and which).

The functionality added in `saveElement` is, if the object should not be saved (the none option) or the object is defined by an attribute (the object from attribute option) it is handled in `MySave`. Otherwise the object and feature is passed to its super-class, to determine what functionality is needed for the object and feature is handled by the mapping table, and covered in Sect. 7.3.3.

If the mapping for the object and feature fits with the none-feature, the element is just skipped and thus never saved. The serialising then continues with the next object.

If the mapping requests the object from attribute feature, the element name as defined in the mapping is created instead. This is done by adding the start tag to the document as well as the attribute as defined in the feature. The method `saveFeatures` is then called to save the rest of the features i.e. other attributes associated with the object. Do note, that the method `endElement` on the document should not be called as expected. That method is called in `endSaveFeatures` which is as the last thing in `saveFeatures`.

7.3.1.2 Saving Multiple Elements – `saveContainedMany`

The method is responsible for saving features on objects with more than one element associated with the feature. Such as the features multi level (single and combined) and the "as number in parent"-feature. The feature also takes an object and a feature as arguments, like the method `saveElement`.

If the mapping for the object and feature pair is the "as number in parent"-feature, the amount of elements is counted by receiving the list of elements associated with the feature from the helper and calculating the size. The objects must be casted correctly, the class of the object is unknown – but it is known that the objects at least extend the class `InternalEObject`. This is done by the following code in List. 7.5.

```
List<? extends InternalEObject> values = ((InternalEList<? extends
    InternalEObject>) helper.getValue(o, f)).basicList();
int size = values.size();
```

Listing 7.5: Reciving elements assosiated with a feature.

If the feature for multiple levels is needed, this is handled in almost the same way. The mapping can either be for multiple levels in each own element, or all elements in an extra level combined. Both is covered. The same functionality as when the number of elements is saved in the parent.

That means that, the elements associated with the feature is retrieved using `XMLHelper` as in List. 7.5. But instead of saving the number of elements in an attribute, the extra level of elements is created using the document object. Depending on whether the combined or single variant of extra level is needed, elements is created and ended using `startElement`, and `endElement`. The element is then saved using the method `saveElement`.

7.3.2 Load Functionality – `MyLoad` and `MySAXHandler`

When loading the method `load` is called from the class `MyLoad`. The class extends the class `XMLLoadImpl` which contains the methods for reading the XML and traversing the objects. For this project only the default handler must be overwritten. This is done by overwriting the class `makeDefaultHandler`, and instead providing my own handler, the class `MySAXHandler`.

`MySAXHandler` extends the regular SAX handler from EMF, the class `SAXXML-Handler`. The handler is responsible for getting content and handling objects and their attributes.

As with the save functionality, if there is no applicable mapping, it should use the standard functionality. That means only four methods is overwritten to support the new features.

The following list contains the four methods, and are covered in the next sections.

createObject responsible for creating objects. Used for the "object from attribute", "multi level" and none-feature.

getFeature responsible for receiving the feature for the methods `handleFeature` and `setAttribValue`. Is used for the "multi level"-feature.

setAttribValue responsible for creating the values on objects. Used for the "as number in parent" and "object from attribute" feature.

processTopObject is responsible for processing the root object(s). Is used to add tables if a mapping is of the feature "root object".

7.3.2.1 Creating Objects – createObject

This method is responsible for creating objects. It receives, a peekobject (the parent object) and the feature to create. The method usually creates a new object and passes it along to the method `processObject` which finally adds the object.

For this project in some cases it is needed to handle it differently. A none-feature is added, that means if that mapping apply, nothing should be done. This is implemented as a simple check to see if the mapping applies, and the method then returns without having created any object.

The method also supports the "object from attribute"-feature. If a such mapping applies, another object than what EMF would create should be created. From the mapping it is known what object in which package should be created. The proper factory instance is found. When the new object is created it is added to the parent object. Now the newly created object might have additional attributes which must be handled. The normal chain of called methods are called. The method `handleObjectAttribs` to handle the attributes on the object, and the method `processObject` which maintains the created lists of objects. Listing 7.6 shows the method calls.

```
EPackage ePackage = table.getPackageByName(m.getContainer().
    getPackagePrefix());
EClassifier eClassifier = ePackage.getEClassifier(m.getTarget());
EClass eClass = (EClass) eClassifier;

EObject newObject = ePackage.getEFactoryInstance().create(eClass);

List<EObject> objs = (List<EObject>) peekObject.eGet(feature);

objs.add(newObject);

...
```

```
handleObjectAttribs(newObject);
processObject(newObject);
```

Listing 7.6: Recieving the proper factory, creating a new object and adding it to the parent object. And lastly handling the newly created object as usual.

To be able to create the multi level feature, a temporary connection must be created when loading the first element, since the extra level must be ignored. To do this the class `AssocClass` from ePNK is used. The new object is given the feature, and the source for the first object. We do not know the inner object yet, which is why the class must be used. Then, when serialising later and provided with an object of the type `AssocClass`, the correct object can be created. The `AssocClass` object contains the source and feature, and now the connection can be created and the object can be processed.

7.3.2.2 Retrieving Features – getFeature

To be able to handle the newly added class `AssocClass`, a wrapper class around the feature `EStructuralFeature` to also contain the name of which object it must be associated with, must be added. The wrapper class `MyStructuralFeature` is created to contain the feature and the name, when a such class is met in the method `getFeature` the new class is used instead.

7.3.2.3 Setting Values on Objects – setAttribValue

If earlier in the method for creating objects the feature "object from attribute" was used, the object to be created was represented in XML as an attribute on the object. That means that in this method, an attribute we have already handled in an earlier method (by creating the proper object) will be received. This attribute must be ignored, because the regular EMF serialiser does not understand this feature since it is not part of the model.

The "as number in parent" feature is also implemented in this method. When the method reaches an attribute matching a mapping with that feature the objects are created. As in the "object from attribute" feature, the proper factory is found which creates the appropriate number of objects. The amount of objects created is found by the method `getValue` on the attributes object, containing all the attributes.

7.3.2.4 Proccessing Root Objects – processTopObject

In some cases the developer might want to add new tables depending on values on the top element. This is done by adding a mapping with the feature chosen as `root element`. If the attribute and value matches the values in the mapping a table is added.

This feature only adds a table and does not handle the root object, that means the root object is just passed along to its super and handled as usual independently of a new table is added or not.

7.3.3 Table for mappings – `MappingTable`

The class `MappingTable` is responsible for the table of mappings and packages as well as features for retrieving information from these.

The following list is the key methods of the class, the methods are covered in more details in the following sections.

addTable responsible for adding tables to the internal structure. Uses `load-Table` if given a path to a table.

loadTable returns a loaded table when given a path.

getMapping responsible for returning a list of mappings. Multiple methods exists to only return relevant features given different type of variables.

getPackageByName responsible for returning the proper model-package by searching the list of defined prefixes in the mapping table.

7.3.3.1 Loading Mapping Tables – `loadTable`

The method for loading tables is only used by the method `addTable` if a path is given. The path is given as a string, so a URI used internally in Eclipse to locate the table is created. A new `XMIResource` is created to contain the table, the resource is set up and the path is set. The model is then loaded and casted to the proper object for the tables used within the project. Note that the resource contains a list of models, only the first element exist and returned, which is the one loaded. For Java code, refer to List. 7.7.

```java
URI uri = URI.createPlatformPluginURI(path, false);

Resource resource = new XMIResourceImpl();
serialDefinition.SerialTable model = null;

resource.unload();
resource.setURI(uri);

try {
  resource.load(null);

  // XMIModel : Create a class for XMI Model
  model = (serialDefinition.SerialTable) resource.getContents()
      .get(0);

} catch (Exception e) {
  //...
}

return model;
```

Listing 7.7: Loading and returning a table given a path.

7.3.3.2 Adding Mapping Tables – `addTable`

As described in the Design chapter, Chap. 5, the mapping tables can be extended with new mapping tables in a linked list. When a mapping table is created it contains no mappings. When the method retrieves mappings to be added it checks if the list of mappings is created. If the list is `null` the mappings are added, otherwise the table is extended with a new table for the mappings. The new table is created and the same method, `addTable` is called. See List 7.8.

```java
if (table == null) {
  table = newTable;
} else {
  extension = new MappingTable();
  extension.addTable(newTable);
}
```

Listing 7.8: Adding tables.

7.3.3.3 Searching for Mappings – `getMapping`

Both when saving and loading, the relevant mappings will be searched to find a match. The method is overloaded with multiple different parameters for only returning the relevant set of mappings. E.g. matching a feature or matching the name of an attribute.

The methods uses the method `getTableMappings` to receive all the mappings on the table and recursively on all tables extending. This can be seen in List 7.9. Before returning the list of mappings, the mappings are sorted, see Line 7. This ensure that the mappings are sorted for each level of tables. The sorting is simple, it moves mappings with a context to the front of the list to obey the rule that these take precedence. This is done by creating two lists, one list with mappings with context, the other for mappings without. The mappings without context are then appended the list of mappings with context. This ensures mappings with context to take precedence, but does not interfere with any other order of the mappings.

```
1  List<Mapping> mappings = new ArrayList<Mapping>();

   if (extension != null) {
     mappings.addAll(extension.getTableMappings());
5  }
   if (table != null) {
     mappings.addAll(sortMappings(table.getMappings()));
   }
   return mappings;
```

Listing 7.9: Recursively retrieving all mappings.

These mappings are then searched to match e.g. the proper attribute.

7.3.3.4 Get Package from Name – `getPackageByName`

Uses the method `getTablePackages` to receive all packages on the mapping table, and recursively the packages on all extended mapping tables. The method then searches through the mapping tables until a matching prefix is found, and if found it is returned. Otherwise the serialisation is stopped since this introduces the serialiser to a problem it cannot continue from.

CHAPTER 8

Evaluation

The following sections contain the evaluation of the project. The first section is test; This covers a test of each of the features implemented, and their relation to Petri nets. When testing a feature the serialiser must be able to save in the correct representation but also be able to read the serialised file — while keeping the same content.

This chapter also covers the correctness of the serialisation, describing why the implemented features are enough, and correct.

The last section of this chapter gives a look on the future for this project.

8.1 Test of Mapping Mechanism

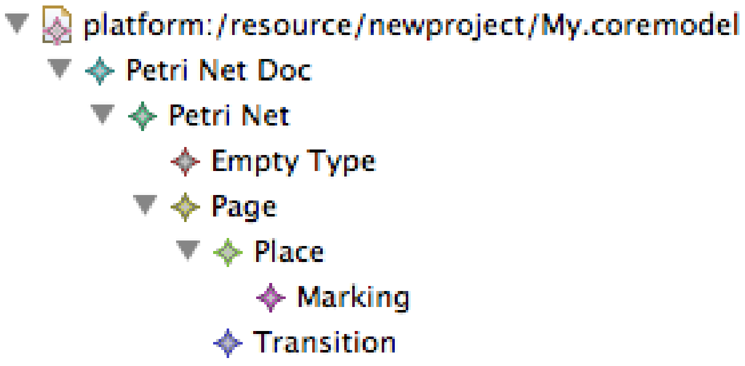
This section will cover the test of the serialiser. Throughout this project numerous features are developed, the test of the functionality of these features will be documented in this section. Additionally, this section will cover if these features cover the features needed to be able to serialise Petri nets according to the iso standard for Petri nets; PNML.

The testing is performed on an Eclipse installation installed as described in the developer handbook in Chap. 6. This section will test the core model of PNML, which is defined in the Ecore model located in `org.pnml.tools.epnk.model.PNMLCoreModel.ecore`. The model can be seen in Fig. 8.1 on Page 71.

For testing purposes this model is created in its own project, and the serialiser developed is associated with the file extension.[1] This means that the usual editor for Petri nets cannot be used and the simple tree editor will be used instead. This is also how the model instances will be displayed in this section.

8.1.1 Relying on the Default Serialiser

A simple model instance of a such model can be seen in Fig. 8.2. List. 8.1 shows the XML file when saving.



```
▼ platform:/resource/newproject/My.coremodel
  ▼ Petri Net Doc
    ▼ Petri Net
        Empty Type
      ▼ Page
        ▼ Place
            Marking
          Transition
```

Figure 8.2: A simple instance of the model in Fig. 8.1. A Petri net with no type, a place with a marking and a transition.

The purpose of this test is to see if a model is serialised without any mappings, the resulting representation is the same as using the default EMF serialiser.

```
<?xml version="1.0" encoding="UTF-8"?>
<coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
  <net>
    <type xsi:type="coremodel:EmptyType"/>
    <page>
      <object xsi:type="coremodel:Place">
        <labels/>
```

[1]The file extension for this test is .coremodel

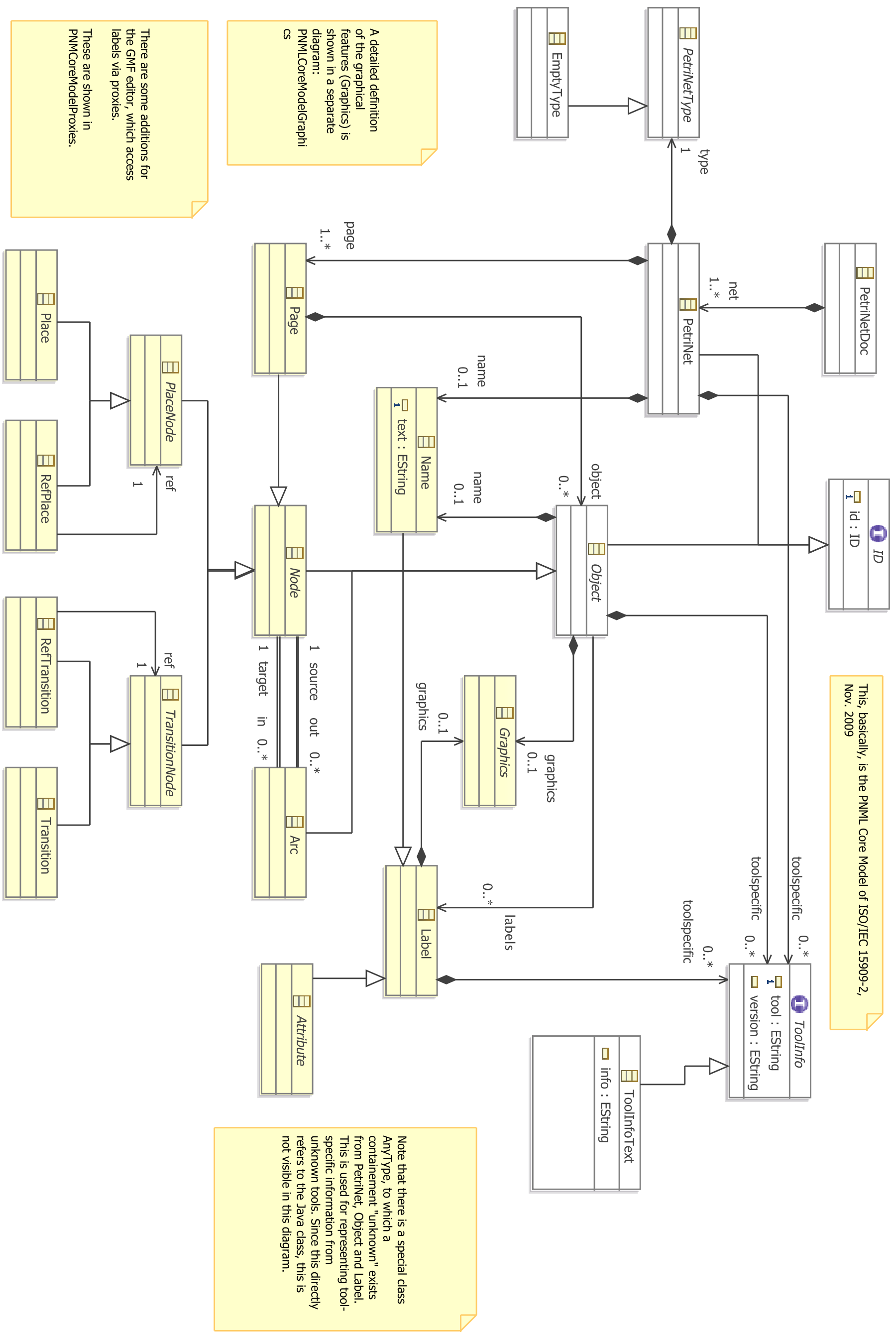

PetriNetType
EmptyType
type
1
PetriNetDoc
net
1..*
PetriNet
page
1..*
Page
ID
id : ID
Name
text : EString
name
0..1
name
0..1
object
0..*
Object
Node
PlaceNode
ref
1
Place
RefPlace
TransitionNode
ref
1
RefTransition
Transition
Arc
1 source out 0..*
1 target in 0..*
Graphics
graphics
0..1
graphics
0..1
Label
labels
0..*
Attribute
ToolInfo
tool : EString
version : EString
ToolInfoText
info : EString
toolspecific
0..*
toolspecific
0..*
toolspecific
0..*
A detailed definition of the graphical features (Graphics) is shown in a separate diagram: PNMLCoreModelGraphics
There are some additions for the GMF editor, which access labels via proxies. These are shown in PNMLCoreModelProxies.
This, basically, is the PNML Core Model of ISO/IEC 15909-2, Nov. 2009
Note that there is a special class AnyType, to which a containement "unknown" exists from PetriNet, Object and Label. This is used for representing tool-specific information from unknown tools. Since this directly refers to the Java class, this is not visible in this diagram.

Figure 8.1: The Ecore diagram for the core model of PNML as modeled in `org.pnml.tools.epnk.model.PNMLCoreModel.ecore`.

```
      <labels/>
    </object>
    <object xsi:type="coremodel:Transition"/>
  </page>
</net>
</coremodel:PetriNetDoc>
```

Listing 8.1: Output when saving the model instance shown in Fig. 8.2

The instance is saved, with no loss of data and can be loaded again to reflect the instance shown in Fig. 8.2. The instance is saved as the standard XMI serialiser shipping with EMF. This means, with no mappings – the serialiser relies on the standard XMI serialiser as intended. When saving the above model instance with the standard EMF serialiser the result i the same as in List. 8.1.

Now mappings can be added.

8.1.2 Adding Mappings – Object from Attribute

A mapping table is created and added. The mapping table is then plugged in and associated with the file extension `.coremodel`. The mapping table is as shown in Tabl. 8.1, the Petri net remains the same and the output can be seen in List. 8.2.

Option	Feature	Container	Attribute	Value	Target
`OBJECT_FROM_ATTRIBUTE`	core:object	core:Page	type	place	Place

Table 8.1: The mapping table to map into List. 8.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
  <net>
    <type xsi:type="coremodel:EmptyType"/>
    <page>
      <object type="place">
        <labels/>
        <labels/>
      </object>
      <object xsi:type="coremodel:Transition"/>
    </page>
  </net>
</coremodel:PetriNetDoc>
```

Listing 8.2: Output when saving the model instance shown in Fig. 8.2, with the mapping table shown in Tabl. 8.1

8.1.3 Saving Object as a Number

Using Tabl. 8.2, instead of having labels as individual object, the amount of labels can be saved as an attribute.

Option	Feature	Container	Attribute	Value	Target
`AS_NUMBER_IN_PARENT`	core:Object	core:labels	amount	Object	Label

Table 8.2: The mapping table to map into List. 8.3.

The resulting XML file is shown in Lst. 8.3. Note the attribute `amount` on line 6.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
    <coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
        instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
      <net>
        <type xsi:type="coremodel:EmptyType"/>
 5      <page>
          <object xsi:type="coremodel:Place" amount="2"/>
          <object xsi:type="coremodel:Transition"/>
        </page>
      </net>
10  </coremodel:PetriNetDoc>
```

Listing 8.3: Output when saving the model instance shown in Fig. 8.2, with the mapping table shown in Tabl. 8.2

The instance can be saved and loaded with the new feature without dataloss.

8.1.4 Most Specific Mapping

General for mappings the serialiser should select the most precise mapping. This is defined in Sect. 5.2.5. With the test in Sec. 8.1.2 it was shown that plugged in mappings takes precedence over standard mappings. In this test it will be shown that more precise mappings (i.e. mappings with a context) takes precedence over mappings without.

The order of mappings is the last thing to decide which mapping is used. Extending Tabl. 8.2, to the table Tabl. 8.3, shows this in List. 8.4.

Option	Feature	Container	Attribute	Value	Target
AS_NUMBER_IN_PARENT	core:Object	core:labels	amountFirst	Object	Label
AS_NUMBER_IN_PARENT	core:Object	core:labels	amount	Object	Label

Table 8.3: Tabl. 8.2 extended with a new mapping with a new value for the element name.

```
<?xml version="1.0" encoding="UTF-8"?>
<coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
  <net>
    <type xsi:type="coremodel:EmptyType"/>
    <page>
      <object xsi:type="coremodel:Place" amountFirst="2"/>
      <object xsi:type="coremodel:Transition"/>
    </page>
  </net>
</coremodel:PetriNetDoc>
```

Listing 8.4: Output when saving the model instance shown in Fig. 8.2, with the mapping table shown in Tabl. 8.2

When the container is removed from the table, as shown in Tab. 8.4 the attribute name is chosen from the second mapping, which shows the mapping with context takes precedence. Which is seen in List. 8.5

Option	Feature	Container	Attribute	Value	Target
AS_NUMBER_IN_PARENT	—	core:labels	amountFirst	Object	Label
AS_NUMBER_IN_PARENT	core:Object	core:labels	amount	Object	Label

Table 8.4: Tabl. 8.3 without container defined on the first mapping.

```
<?xml version="1.0" encoding="UTF-8"?>
<coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
  <net>
    <type xsi:type="coremodel:EmptyType"/>
    <page>
      <object xsi:type="coremodel:Place" amount="2"/>
      <object xsi:type="coremodel:Transition"/>
    </page>
  </net>
</coremodel:PetriNetDoc>
```

Listing 8.5: Output when saving the model instance shown in Fig. 8.2, with the mapping table shown in Tabl. 8.4

8.1.5 Add Tables

Using the option for adding tables on root objects, new tables can be added. For this test a mapping table is added based on the root object. This is shown in Tabl 8.5.

Do note that the column add table is available on all mappings but kept out of the tables due to space issues in this paper. Refer to the DSL defining mappings in Sect. 5.4

Option	**...**	**Add Table**
`ROOT_OBJECT`	...	dk.dtu.se.coremodel/mappings/2.serialdefinition

Table 8.5: To add a new table on root object. The shown columns of this table is reduced to fit in this paper. All not shown fields are empty.

The table is added based on the URI in Eclipse for the table. The table is added when loading and the mappings are used as expected. The referred URI is the URI for the mappings in Tabl. 8.4. The resulting representation is the same as the listing in List. 8.5 and shows the mappings work.

8.1.6 Precedence of Added Tables

As defined in Sect. 5.2.5, the latest added mappings must take precedence over others. Using the tables in the previous sections this can be shown working. Table 8.5 is combined with the table mapping the attribute for "as number in parent" in Tabl. 8.6.

Option	**Feature**	**Container**	**Attribute**	**Value**	**Target**	**Add Table**
`AS_NUMBER_IN_PARENT`	core:Object	core:labels	amountFirst	Object	Label	—
`ROOT_OBJECT`	—	—	—	—	—	dk.dtu.se...*

Table 8.6: Table 8.5 is combined with the table mapping the attribute for "as number in parent".

The URI in Tabl. 8.6 refers to the table in Tabl. 8.7.

Option	Feature	Container	Attribute	Value	Target	Add Table
`AS_NUMBER_IN_PARENT`	core:Object	core:labels	amount	Object	Label	—

Table 8.7: Extension of Tabl. 8.6. Notice the differences in the column Attribute.

The resulting representation is the same as in List. 8.5 that means with the correct selected mapping with the attribute `amount`. Which shows the precedence of mappings is correct.

8.1.7 Using References

And now to something completely different. Before using the new serialiser, it was possible to reference objects in the instance. This should still be possible and work as expected with the new serialiser. An `Arc` is introduced to the instance from the place to the transition. The resulting XML file can be seen in List. 8.6.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
    <coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
        instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
      <net>
        <type xsi:type="coremodel:EmptyType"/>
 5      <page>
          <object xsi:type="coremodel:Place" amount="2"/>
          <object xsi:type="coremodel:Transition"/>
          <object xsi:type="coremodel:Arc" source="//@net.0/@page.0/
              @object.0" target="//@net.0/@page.0/@object.1"/>
        </page>
10    </net>
    </coremodel:PetriNetDoc>
```

Listing 8.6: Output when saving the model instance shown in Fig. 8.2, with the new table in

The arc references to the correct place and transition as expected, refer to Line 8 in List. 8.6. This is tested by creating multiple places and transitions. And then only deleting the referenced one – the editor will then remove this reference since the object does not exist any more.

The editor relies on XPath when there is no identifiers on the objects. If identifiers are added to the object, the identifier is used instead. This can be seen in List. 8.7.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
    <coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
        instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
```

```
  <net>
    <type xsi:type="coremodel:EmptyType"/>
    <page>
      <object xsi:type="coremodel:Place" id="p1" amount="2"/>
      <object xsi:type="coremodel:Transition" id="t1"/>
      <object xsi:type="coremodel:Arc" source="p1" target="t1"/>
    </page>
  </net>
</coremodel:PetriNetDoc>
```

Listing 8.7: Output when saving the model instance shown in Fig. 8.2, with identifiers on place and transition.

The transition is given the identifier `t1` and the place the identifier `p1`. The arc is now using these new identifiers and it is now easier to verify that the editor relates to the correct objects by creating more with new identifiers. When deleting the object with the referenced identifier, the reference is removed and it can be verified to be the correct object referenced.

8.1.8 Adding Extra Levels

The features "multi level single" and "multi level combined" can also be used by a developer. The features are used for adding an extra level of elements in the XML representation. As an example the developer want all labels to be enclosed in the element `extraelement`. This can be done with the defined mappings in tabl. 8.8.

Option	Feature	Container	Attribute	Value	Target
`MULTI_LEVEL_COMBINED`	core:labels	core:Object	extralevel	—	Label

Table 8.8: Table adding an extra level to the representation of labels.

Serialising an instance with three labels on the object results in the representation in List. 8.8

```
<?xml version="1.0" encoding="UTF-8"?>
<coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
  <net>
    <type xsi:type="coremodel:EmptyType"/>
    <page>
      <object xsi:type="coremodel:Place" id="p1">
        <extralevel>
          <labels/>
          <labels/>
          <labels/>
        </extralevel>
      </object>
      <object xsi:type="coremodel:Transition" id="t1"/>
      <object xsi:type="coremodel:Arc" source="p1" target="t1"/>
    </page>
  </net>
</coremodel:PetriNetDoc>
```

Listing 8.8: Output when saving the model instance shown in Fig. 8.2, with identifiers on place and transition.

The previous mapping used the feature “multi level combined” to combine the extra level into one. Instead each object could be in each own extra element. The feature used is changed as seen in Tabl. 8.9 and results in the representation seen in List. 8.9.

Option	Feature	Container	Attribute	Value	Target
`MULTI_LEVEL_SINGLE`	core:labels	core:Object	extralevel	—	Label

Table 8.9: Table adding an extra level to the representation of labels in each own element.

```
<?xml version="1.0" encoding="UTF-8"?>
<coremodel:PetriNetDoc xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:coremodel="http://dk.dtu.se.coremodel">
  <net>
    <type xsi:type="coremodel:EmptyType"/>
    <page>
      <object xsi:type="coremodel:Place" id="p1">
        <extralevel>
          <labels/>
        </extralevel>
        <extralevel>
          <labels/>
        </extralevel>
        <extralevel>
          <labels/>
        </extralevel>
      </object>
      <object xsi:type="coremodel:Transition" id="t1"/>
      <object xsi:type="coremodel:Arc" source="p1" target="t1"/>
    </page>
  </net>
</coremodel:PetriNetDoc>
```

Listing 8.9: Output when saving the model instance shown in Fig. 8.2, with three labels using the mapping table in Tabl. 8.9.

8.1.9 Multiple mappings

The previous tests have shown the features individually in a context of Petri nets. This test will show another model than Petri nets, and will make use of almost all the features at once.

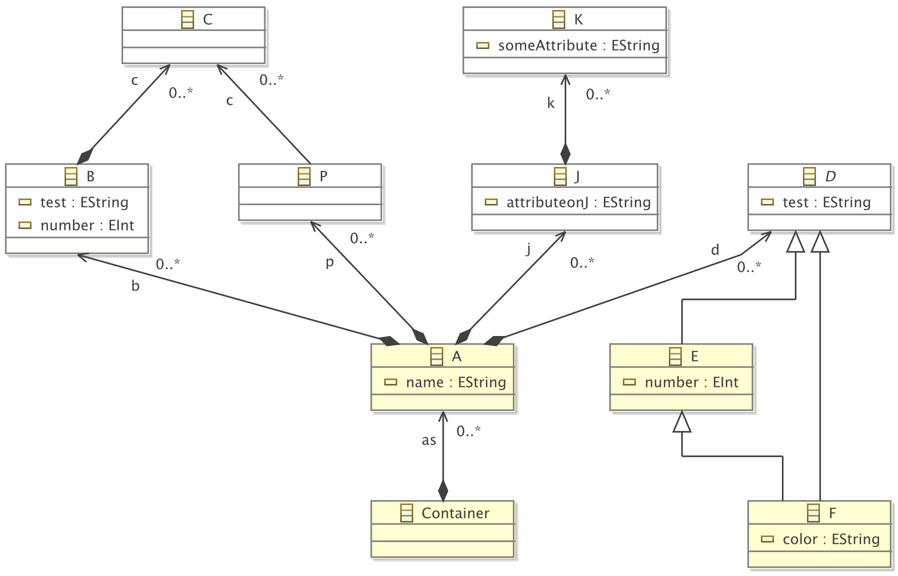


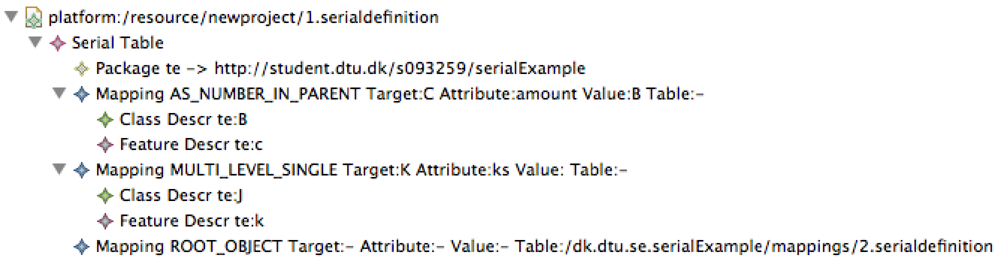
Figure 8.3: The model for the test of all features.

The model can be seen in Fig. 8.3 and contains an object A in the container object. The A object contains

- A list of B objects that contains a list of C objects.
- A list of P objects also containing a list of C objects.
- A list of J objects containing a list of K objects
- A list of D objects (which are abstract) and can either be a E or F object.

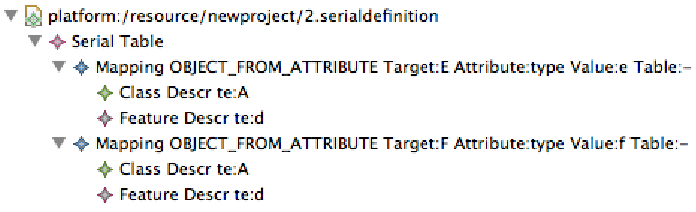
The mappings defined can be seen in Fig. 8.4. The mappings extend into the file `2.serialdefinition`, which can be seen in Fig. 8.5

The mappings are the following



```
platform:/resource/newproject/1.serialdefinition
  Serial Table
    Package te -> http://student.dtu.dk/s093259/serialExample
    Mapping AS_NUMBER_IN_PARENT Target:C Attribute:amount Value:B Table:-
      Class Descr te:B
      Feature Descr te:c
    Mapping MULTI_LEVEL_SINGLE Target:K Attribute:ks Value: Table:-
      Class Descr te:J
      Feature Descr te:k
    Mapping ROOT_OBJECT Target:- Attribute:- Value:- Table:/dk.dtu.se.serialExample/mappings/2.serialdefinition
```

Figure 8.4: The mappings for the model in Fig. 8.3, `1.serialdefinition`, as seen in the mapping editor.



```
platform:/resource/newproject/2.serialdefinition
  Serial Table
    Mapping OBJECT_FROM_ATTRIBUTE Target:E Attribute:type Value:e Table:-
      Class Descr te:A
      Feature Descr te:d
    Mapping OBJECT_FROM_ATTRIBUTE Target:F Attribute:type Value:f Table:-
      Class Descr te:A
      Feature Descr te:d
```

Figure 8.5: The mappings for the model in Fig. 8.3, `2.serialdefinition`, as seen in the mapping editor.

- The url for the package serial example is added. [2]
- C objects on B objects must be serialised as a number in parent with the attribute amount.
- K objects on the J object must be serialised with an extra level called `ks` in between.
- When loading the root object, the contents of `2.serialdefinition` must be added.
- When the attribute `type` on the object D is `e` the created element should be of the type E.
- When the attribute `type` on the object D is `f` the created element should be of the type F.

Now an instance of the model shown is serialised. The model instance is shown in Fig. 8.6, and the result of serialisation is as expected and is shown in List. 8.10.

[2]Note that this package definition is also usable in the added mapping table.

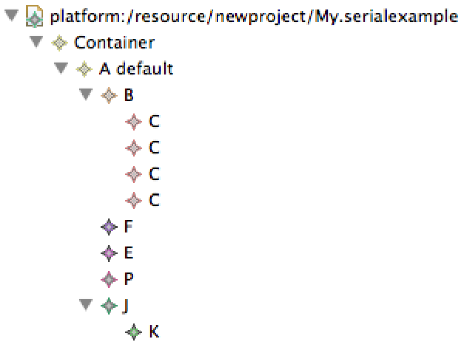


Figure 8.6: An instance of the model in Fig. 8.3. Not shown in the editor is the reference in P to the 2nd C object.

```
<?xml version="1.0" encoding="UTF-8"?>
<serialExample:Container xmlns:serialExample="http://student.dtu.dk
    /s093259/serialExample">
  <as>
    <b amount="4"/>
    <d type="f"/>
    <d type="e"/>
    <p c="//@as.0/@b.0/@c.1"/>
    <j>
      <ks>
        <k/>
      </ks>
    </j>
  </as>
</serialExample:Container>
```

Listing 8.10: The result of the serialisation.

The file can be loaded and saved without dataloss. Notice that:

- The amount of Cs is saved in b as an attribute.
- The objects E and F is saved as D with the attribute type defining which object to create.
- Since the mapping defining the above is used, that means the addition of new tables are working.
- The reference in P to the 2nd C object is working. [3]
- The K object is serialised into J with an extra element `ks` in between.

[3]Can only be tested in the editor, the reference works correctly, when the second C object on B is deleted, it is also deleted from the P object, proving it is a reference to the same object.

8.2 Correctness of Serialisation

This section is an evaluation of the correctness of the serialisation. For the serialisation mechanism to be useful it must be correct. To ensure this the loading and saving of features are kept as identical as possible. The mappings for saving and loading is the same. The code is kept as defensive as possible, meaning only when a mapping applies the functionality is overridden.

Throughout the test section, Sect. 8.1 numerous model instances has been loaded and saved. Without dataloss.

Though, allowing multiple mappings for the same elements raises some issues. When saving and loading an attribute on an element it cannot be insured the same mapping applies, when multiple exist. The order; and which mapping is the best fit, of course is the same — due to the same sorting of mappings for both saving and loading. But if multiple mappings exist for creating attributes for the "as number in parent" exist, as an example the table of mappings from Tabl. 8.10, when testing the precedence of mappings.

Option	Feature	Container	Attribute	Value	Target
`AS_NUMBER_IN_PARENT`	—	core:labels	amountFirst	Object	Label
`AS_NUMBER_IN_PARENT`	core:Object	core:labels	amount	Object	Label

Table 8.10: Reappearance of Tabl. 8.4.

When serialising, the second mapping will be chosen, and the attribute name will be `amount`. When saving, with this table, the attribute will never be `amountFirst`.

Though if a mapping with the attribute `amountFirst` is loaded, this will actually be supported! This could be exploited by a developer to support reading of older formats, but still saving in a new. But it also means the serialiser can use a mapping while loading which is not used while saving. It will not lead to dataloss but it can lead to adding additional data.

8.3 Relation to Standard for Petri Nets

Multiple features from the PNML standard is implemented and tested.

- To support multiple versions of the Petri net standard, support for multiple tables is implemented and tested.
- To support high level Petri nets with additional added levels. (e.g. `Terms` which gets an extra element `structure` when serialised. See Page 90 of the iso standard for an XML representation and Page 14 for a meta model.
- To better support attributes defining which object to be created the feature "Object from attribute" is added. This is used when defining the identifier on the net type. See page 89 in the iso standard for an example.
- The feature "as number in parent" is not part of the standard, but could be used in new standards.

8.4 Conclusion on Test

Multiple features of the Petri net format are supported. The serialiser can save the new features and is able to load them without errors or loss of data. The serialiser is also able to serialiser other formats than PNML.

The serialiser relies on the plugged in mappings and chooses the correct mapping based on the rules defined.

8.5 The Future of the Serialiser

The serialiser has some functionality and features, but some features and ideas was deemed out of scope of the project due to the limited time frame of the project. The serialiser and mapping functionality could be extended with the functionality from the following sections.

8.5.1 Fully Supporting the ISO Standard for Petri nets

The serialiser has some base functionality and supports multiple features of the ISO standard for Petri nets. The base functionality developed for this serialiser could be extended to fully support the PNML standard and also the much larger HLPNG standard and can be described in abstract syntax.

8.5.2 References in Instances

By standard the EMF XMI Serialiser uses XPath for references, since this serialiser relies on this implementation the same support for XPath is used. The implementation is rather unconventional by using XPath with indexing from zero, instead of the usual index starting from one.

This could be solved by adding a custom implementation for XPath references.

Additionally some developers might want another way to define references, which also could be added. Functionality for developers to extend the serialiser with their own functionality for defining references could also be added.

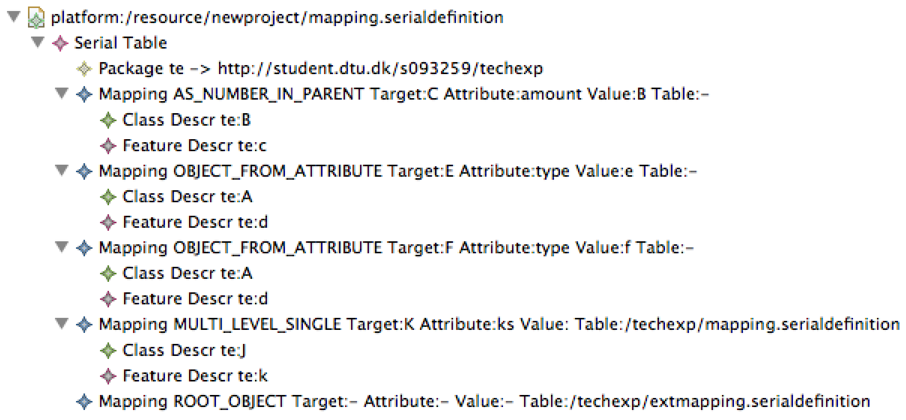
8.5.3 Error Handling in Mapping Editor

When defining mappings various things can go wrong. E.g. if a developer refers to a feature which creates an object which might not exist. This yields an error. These mistakes could be avoided by introducing features in the mapping editor as suggested in the next section.

When an error occurs, it could be interesting to both prompt the user, but also show a marking in the mapping table on which mapping rule yielded an error.

8.5.4 The mapping Editor

The simple tree editor for editing mappings is not a very useful tool. The editor can be used to define mappings, but additional features to ease the definition of mappings could be very useful.



```
platform:/resource/newproject/mapping.serialdefinition
  Serial Table
    Package te -> http://student.dtu.dk/s093259/techexp
    Mapping AS_NUMBER_IN_PARENT Target:C Attribute:amount Value:B Table:-
      Class Descr te:B
      Feature Descr te:c
    Mapping OBJECT_FROM_ATTRIBUTE Target:E Attribute:type Value:e Table:-
      Class Descr te:A
      Feature Descr te:d
    Mapping OBJECT_FROM_ATTRIBUTE Target:F Attribute:type Value:f Table:-
      Class Descr te:A
      Feature Descr te:d
    Mapping MULTI_LEVEL_SINGLE Target:K Attribute:ks Value: Table:/techexp/mapping.serialdefinition
      Class Descr te:J
      Feature Descr te:k
    Mapping ROOT_OBJECT Target:- Attribute:- Value:- Table:/techexp/extmapping.serialdefinition
```

Figure 8.7: The simple editor for editing mappings.

- When defining packages, the url must be manually typed, instead the url could be chosen from a list of relevant urls already plugged in.
- When defining a mapping and choosing the feature, the feature must be manually typed. It would be easier to define mappings if this feature could be selected in the specified model.
- When defining a container element, it would be easier if this was not manually typed but instead was chosen from a list of possible container elements. This could somehow be determined from the model.
- As suggested above, adding labels in the mapping table if an error is met.

8.5.5 Developed Serialiser as Default

The developed serialiser must be plugged in for each developed project. The user experience could be greatly improved if the serialiser was plugged in as default. Which is possible in EMF by adding the serialiser for all extensions. The serialiser is developed to rely on the default serialiser if no mappings are present so it should not interfere with other projects if no mappings are defined.

Chapter 9

Conclusion

The goal of this project was to develop a flexible serialiser for model instances. A such has been developed, documented and tested. The technology is developed as an extension to EMF.

The serialiser uses mappings to define the relationship between model and representation, these mappings are defined without programming. Mappings can be cascading and tables of mappings can be extended with other tables. The serialiser chooses the most precise mapping defined by a set of rules.

Using the serialiser and the mapping mechanism multiple features of the standard for Petri nets are supported. Though, the serialiser supports customisation of the serialisation of any UML model instance. Both instances of Petri nets and instances of other models are tested to work with the serialiser.

Ideas to further support serialisation of model instances and to ease the definition of mappings is proposed.

Bibliography

[1] Simple. XML Serialization, http://simple.sourceforge.net/download/stream/doc/tutorial/tutorial.php.

[2] Codehaus. XStream Tutorials, http://xstream.codehaus.org/tutorial.html.

[3] Oracle Corporation. Java Architecture for XML Binding (JAXB), http://docs.oracle.com/javase/tutorial/jaxb/.

[4] Lars Vogel. Eclipse Modeling Framework – Persisting models via XMI, http://www.vogella.com/tutorials/EclipseEMFPersistence/article.html.

[5] IBM Redbooks. Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework, http://www.redbooks.ibm.com/abstracts/sg246302.html.

[6] Eclipse Foundation. Eclipse Modeling Framework (EMF), http://eclipse.org/modeling/emf/.

[7] Eclipse Foundation. EMF Javadoc — Interface Resource, http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.3/org/eclipse/emf/ecore/resource/Resource.html.

[8] Oracle Corporation. Java Documentation – Reading XML Data into a DOM, http://docs.oracle.com/javase/tutorial/jaxp/dom/readingXML.html.

[9] Lars Vogel. Eclipse Extension Points and Extensions, http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html.

[10] Eclipse Foundation. Eclipse Documentation – Package org.eclipse.core.resources, `http://help.eclipse.org/`.

[11] Eric Steven Raymond. The Art of Unix Programming – Minilanguages, `http://www.faqs.org/docs/artu/minilanguageschapter.html`.

[12] Martin Fowler. DSL Q&A, `http://martinfowler.com/bliki/DslQandA.html`.

[13] Eclipse Foundation. Eclipse Documentation – Running the plug-in, `http://help.eclipse.org/`.

[14] Oracle Corporation. Naming a Package, `http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html`.

[15] Ekkart Kindler. ECNO: Installing Eclipse, `http://www2.imm.dtu.dk/~ekki/projects/ECNO/version-0.2.0/eclipse-installation.html`.

[16] Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

Appendix A

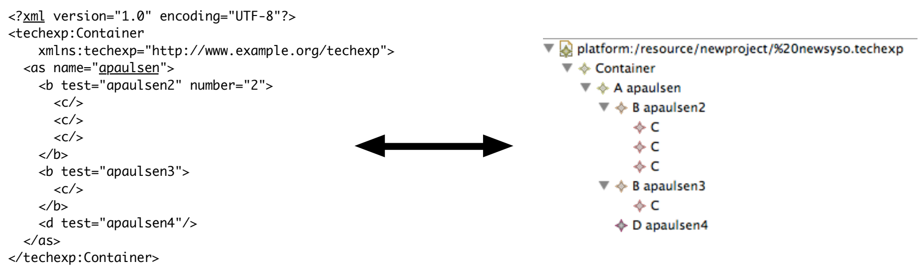
Call Structure for Methods in EMF

This appendix contains the results of debugging the method calling structure of the standard EMF serialiser.

For this example a model is created. An instance of a such model can be seen in Fig. A.1 and the model can be seen in Fig. A.2.

The calling structure for saving can be found in Fig. A.3 and Fig. A.4[1]. Loading can be found in Fig. A.5, Fig. A.6 and Fig. A.7.

[1]The images are split vertically and horizontally respectively and can be hard to read. Refer to App. B which contains a digital copy of both images



```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container
    xmlns:techexp="http://www.example.org/techexp">
  <as name="apaulsen">
    <b test="apaulsen2" number="2">
      <c/>
      <c/>
      <c/>
    </b>
    <b test="apaulsen3">
      <c/>
    </b>
    <d test="apaulsen4"/>
  </as>
</techexp:Container>
```

```
▼ platform:/resource/newproject/%20newsyso.techexp
  ▼ Container
    ▼ A apaulsen
      ▼ B apaulsen2
          C
          C
          C
      ▼ B apaulsen3
          C
        D apaulsen4
```

Figure A.1: An instance of the test model.

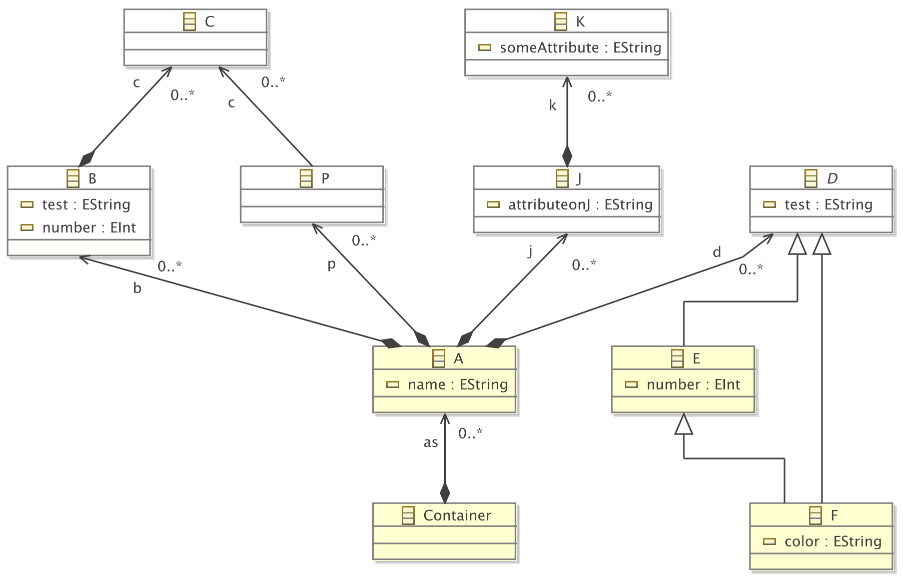


Figure A.2: The test model.

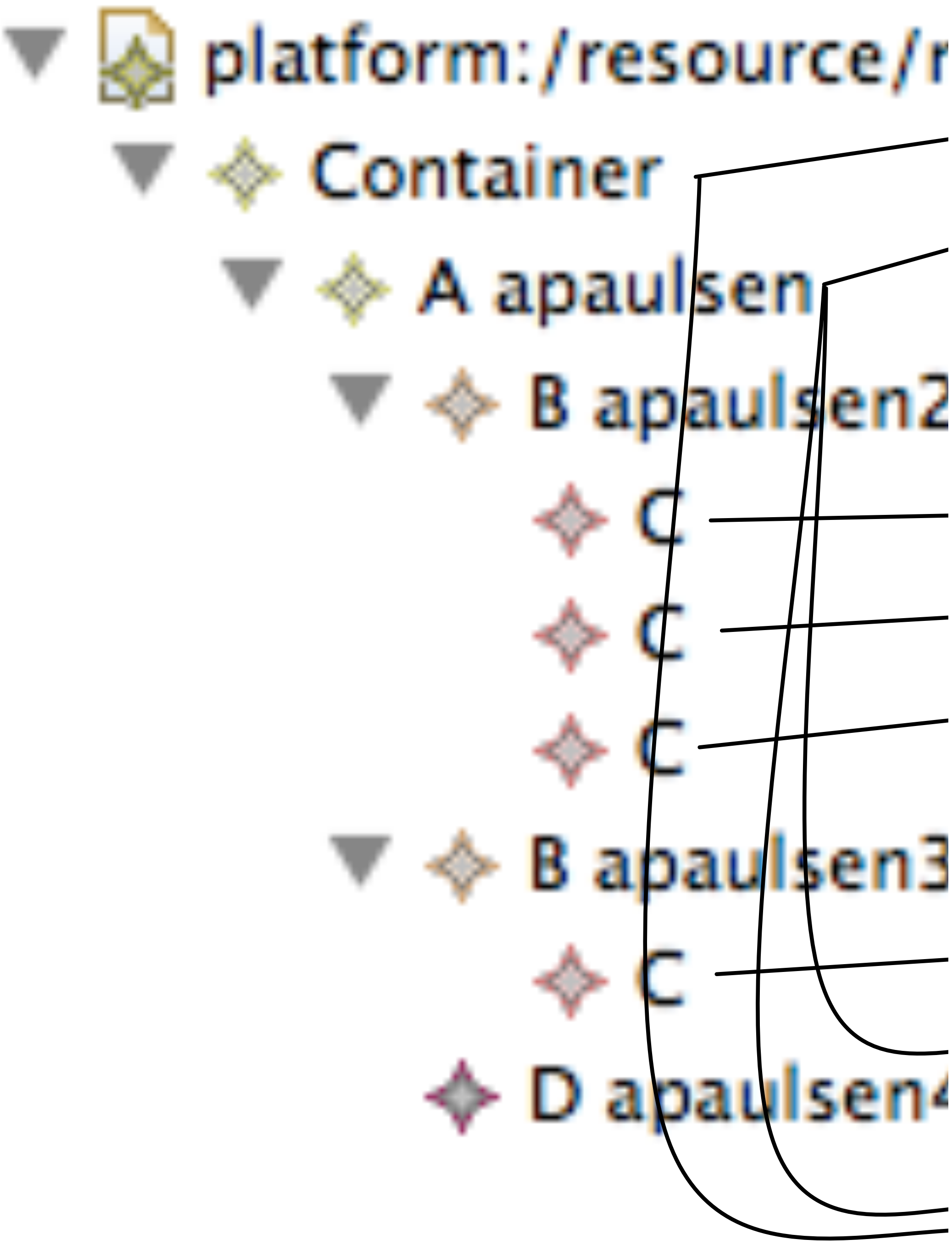
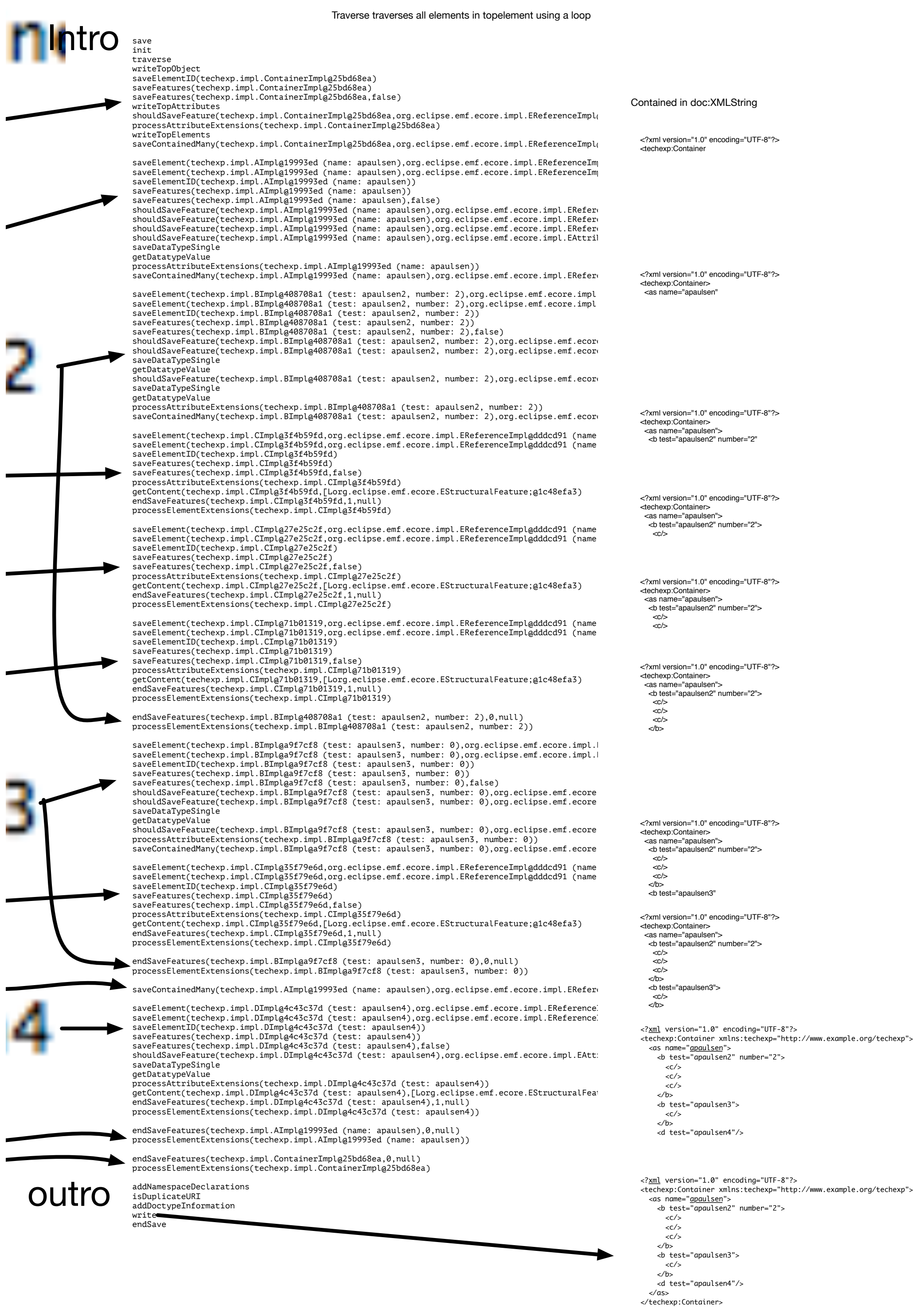


Figure A.3: The calling structure for saving. Left side.


Traverse traverses all elements in topelement using a loop
Intro
Contained in doc:XMLString
2
3
4
outro

```
save
init
traverse
writeTopObject
saveElementID(techexp.impl.ContainerImpl@25bd68ea)
saveFeatures(techexp.impl.ContainerImpl@25bd68ea)
saveFeatures(techexp.impl.ContainerImpl@25bd68ea,false)
writeTopAttributes
shouldSaveFeature(techexp.impl.ContainerImpl@25bd68ea,org.eclipse.emf.ecore.impl.EReferenceImpl(
processAttributeExtensions(techexp.impl.ContainerImpl@25bd68ea)
writeTopElements
saveContainedMany(techexp.impl.ContainerImpl@25bd68ea,org.eclipse.emf.ecore.impl.EReferenceImpl(

saveElement(techexp.impl.AImpl@19993ed (name: apaulsen),org.eclipse.emf.ecore.impl.EReferenceImp
saveElement(techexp.impl.AImpl@19993ed (name: apaulsen),org.eclipse.emf.ecore.impl.EReferenceImp
saveElementID(techexp.impl.AImpl@19993ed (name: apaulsen))
saveFeatures(techexp.impl.AImpl@19993ed (name: apaulsen))
saveFeatures(techexp.impl.AImpl@19993ed (name: apaulsen),false)
shouldSaveFeature(techexp.impl.AImpl@19993ed (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
shouldSaveFeature(techexp.impl.AImpl@19993ed (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
shouldSaveFeature(techexp.impl.AImpl@19993ed (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
shouldSaveFeature(techexp.impl.AImpl@19993ed (name: apaulsen),org.eclipse.emf.ecore.impl.EAttri
saveDataTypeSingle
getDatatypeValue
processAttributeExtensions(techexp.impl.AImpl@19993ed (name: apaulsen))
saveContainedMany(techexp.impl.AImpl@19993ed (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer

saveElement(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2),org.eclipse.emf.ecore.impl
saveElement(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2),org.eclipse.emf.ecore.impl
saveElementID(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2))
saveFeatures(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2))
saveFeatures(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2),false)
shouldSaveFeature(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2),org.eclipse.emf.ecor
shouldSaveFeature(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2),org.eclipse.emf.ecor
saveDataTypeSingle
getDatatypeValue
shouldSaveFeature(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2),org.eclipse.emf.ecor
saveDataTypeSingle
getDatatypeValue
processAttributeExtensions(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2))
saveContainedMany(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2),org.eclipse.emf.ecor

saveElement(techexp.impl.CImpl@3f4b59fd,org.eclipse.emf.ecore.impl.EReferenceImpl@dddcd91 (name
saveElement(techexp.impl.CImpl@3f4b59fd,org.eclipse.emf.ecore.impl.EReferenceImpl@dddcd91 (name
saveElementID(techexp.impl.CImpl@3f4b59fd)
saveFeatures(techexp.impl.CImpl@3f4b59fd)
saveFeatures(techexp.impl.CImpl@3f4b59fd,false)
processAttributeExtensions(techexp.impl.CImpl@3f4b59fd)
getContent(techexp.impl.CImpl@3f4b59fd,[Lorg.eclipse.emf.ecore.EStructuralFeature;@1c48efa3)
endSaveFeatures(techexp.impl.CImpl@3f4b59fd,1,null)
processElementExtensions(techexp.impl.CImpl@3f4b59fd)

saveElement(techexp.impl.CImpl@27e25c2f,org.eclipse.emf.ecore.impl.EReferenceImpl@dddcd91 (name
saveElement(techexp.impl.CImpl@27e25c2f,org.eclipse.emf.ecore.impl.EReferenceImpl@dddcd91 (name
saveElementID(techexp.impl.CImpl@27e25c2f)
saveFeatures(techexp.impl.CImpl@27e25c2f)
saveFeatures(techexp.impl.CImpl@27e25c2f,false)
processAttributeExtensions(techexp.impl.CImpl@27e25c2f)
getContent(techexp.impl.CImpl@27e25c2f,[Lorg.eclipse.emf.ecore.EStructuralFeature;@1c48efa3)
endSaveFeatures(techexp.impl.CImpl@27e25c2f,1,null)
processElementExtensions(techexp.impl.CImpl@27e25c2f)

saveElement(techexp.impl.CImpl@71b01319,org.eclipse.emf.ecore.impl.EReferenceImpl@dddcd91 (name
saveElement(techexp.impl.CImpl@71b01319,org.eclipse.emf.ecore.impl.EReferenceImpl@dddcd91 (name
saveElementID(techexp.impl.CImpl@71b01319)
saveFeatures(techexp.impl.CImpl@71b01319)
saveFeatures(techexp.impl.CImpl@71b01319,false)
processAttributeExtensions(techexp.impl.CImpl@71b01319)
getContent(techexp.impl.CImpl@71b01319,[Lorg.eclipse.emf.ecore.EStructuralFeature;@1c48efa3)
endSaveFeatures(techexp.impl.CImpl@71b01319,1,null)
processElementExtensions(techexp.impl.CImpl@71b01319)

endSaveFeatures(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2),0,null)
processElementExtensions(techexp.impl.BImpl@408708a1 (test: apaulsen2, number: 2))

saveElement(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0),org.eclipse.emf.ecore.impl.
saveElement(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0),org.eclipse.emf.ecore.impl.
saveElementID(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0))
saveFeatures(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0))
saveFeatures(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0),false)
shouldSaveFeature(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0),org.eclipse.emf.ecore
shouldSaveFeature(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0),org.eclipse.emf.ecore
saveDataTypeSingle
getDatatypeValue
shouldSaveFeature(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0),org.eclipse.emf.ecore
processAttributeExtensions(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0))
saveContainedMany(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0),org.eclipse.emf.ecore

saveElement(techexp.impl.CImpl@35f79e6d,org.eclipse.emf.ecore.impl.EReferenceImpl@dddcd91 (name
saveElement(techexp.impl.CImpl@35f79e6d,org.eclipse.emf.ecore.impl.EReferenceImpl@dddcd91 (name
saveElementID(techexp.impl.CImpl@35f79e6d)
saveFeatures(techexp.impl.CImpl@35f79e6d)
saveFeatures(techexp.impl.CImpl@35f79e6d,false)
processAttributeExtensions(techexp.impl.CImpl@35f79e6d)
getContent(techexp.impl.CImpl@35f79e6d,[Lorg.eclipse.emf.ecore.EStructuralFeature;@1c48efa3)
endSaveFeatures(techexp.impl.CImpl@35f79e6d,1,null)
processElementExtensions(techexp.impl.CImpl@35f79e6d)

endSaveFeatures(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0),0,null)
processElementExtensions(techexp.impl.BImpl@a9f7cf8 (test: apaulsen3, number: 0))

saveContainedMany(techexp.impl.AImpl@19993ed (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer

saveElement(techexp.impl.DImpl@4c43c37d (test: apaulsen4),org.eclipse.emf.ecore.impl.EReference
saveElement(techexp.impl.DImpl@4c43c37d (test: apaulsen4),org.eclipse.emf.ecore.impl.EReference
saveElementID(techexp.impl.DImpl@4c43c37d (test: apaulsen4))
saveFeatures(techexp.impl.DImpl@4c43c37d (test: apaulsen4))
saveFeatures(techexp.impl.DImpl@4c43c37d (test: apaulsen4),false)
shouldSaveFeature(techexp.impl.DImpl@4c43c37d (test: apaulsen4),org.eclipse.emf.ecore.impl.EAtt
saveDataTypeSingle
getDatatypeValue
processAttributeExtensions(techexp.impl.DImpl@4c43c37d (test: apaulsen4))
getContent(techexp.impl.DImpl@4c43c37d (test: apaulsen4),[Lorg.eclipse.emf.ecore.EStructuralFea
endSaveFeatures(techexp.impl.DImpl@4c43c37d (test: apaulsen4),1,null)
processElementExtensions(techexp.impl.DImpl@4c43c37d (test: apaulsen4))

endSaveFeatures(techexp.impl.AImpl@19993ed (name: apaulsen),0,null)
processElementExtensions(techexp.impl.AImpl@19993ed (name: apaulsen))

endSaveFeatures(techexp.impl.ContainerImpl@25bd68ea,0,null)
processElementExtensions(techexp.impl.ContainerImpl@25bd68ea)

addNamespaceDeclarations
isDuplicateURI
addDoctypeInformation
write
endSave
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container>
 <as name="apaulsen"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container>
 <as name="apaulsen">
  <b test="apaulsen2" number="2"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container>
 <as name="apaulsen">
  <b test="apaulsen2" number="2">
   <c/>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container>
 <as name="apaulsen">
  <b test="apaulsen2" number="2">
   <c/>
   <c/>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container>
 <as name="apaulsen">
  <b test="apaulsen2" number="2">
   <c/>
   <c/>
   <c/>
  </b>
```

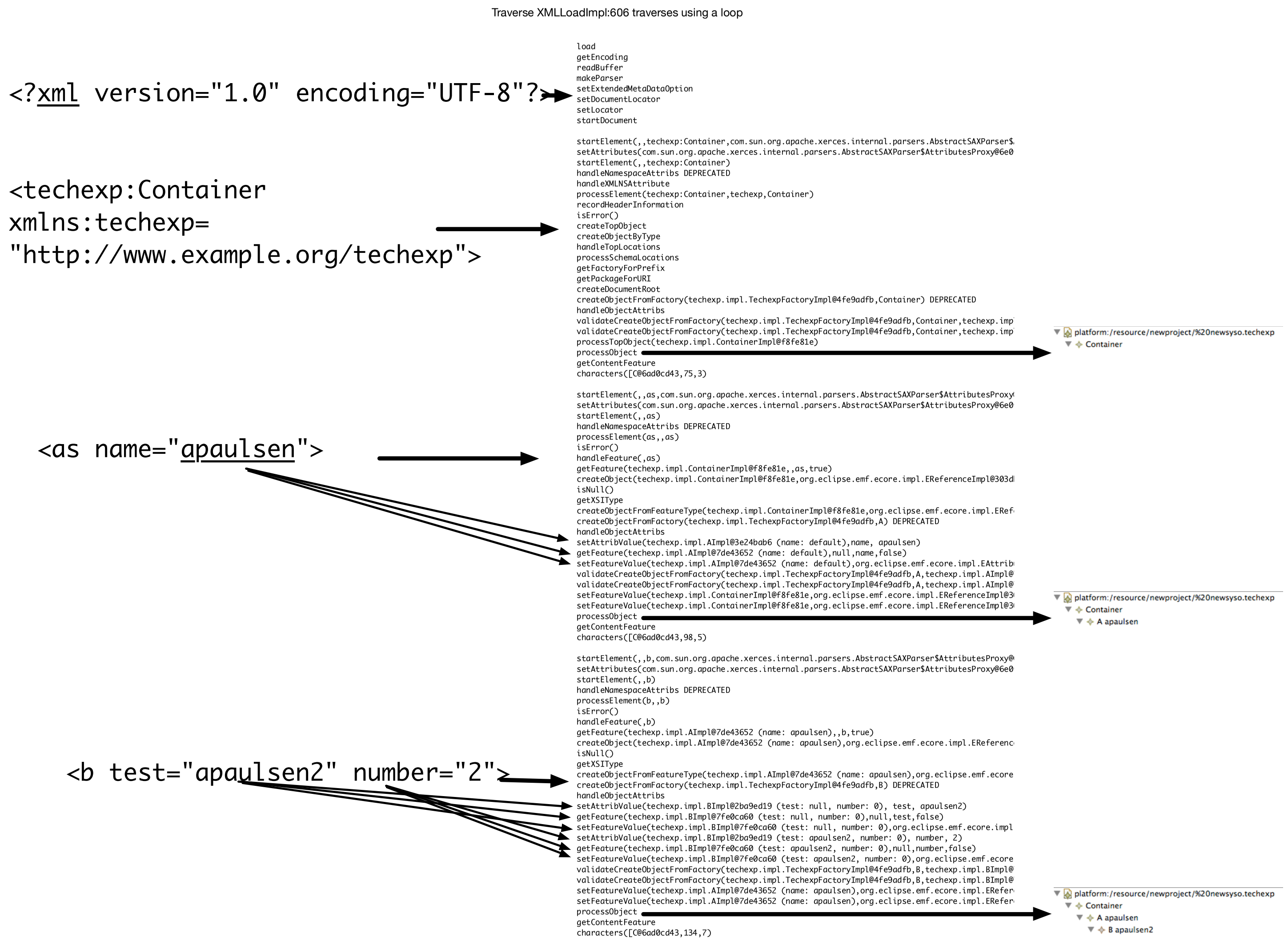
```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container>
 <as name="apaulsen">
  <b test="apaulsen2" number="2">
   <c/>
   <c/>
   <c/>
  </b>
  <b test="apaulsen3"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container>
 <as name="apaulsen">
  <b test="apaulsen2" number="2">
   <c/>
   <c/>
   <c/>
  </b>
  <b test="apaulsen3">
   <c/>
  </b>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container xmlns:techexp="http://www.example.org/techexp">
  <as name="apaulsen">
    <b test="apaulsen2" number="2">
      <c/>
      <c/>
      <c/>
    </b>
    <b test="apaulsen3">
      <c/>
    </b>
    <d test="apaulsen4"/>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<techexp:Container xmlns:techexp="http://www.example.org/techexp">
  <as name="apaulsen">
    <b test="apaulsen2" number="2">
      <c/>
      <c/>
      <c/>
    </b>
    <b test="apaulsen3">
      <c/>
    </b>
    <d test="apaulsen4"/>
  </as>
</techexp:Container>
```

Figure A.4: The calling structure for saving. Right side,

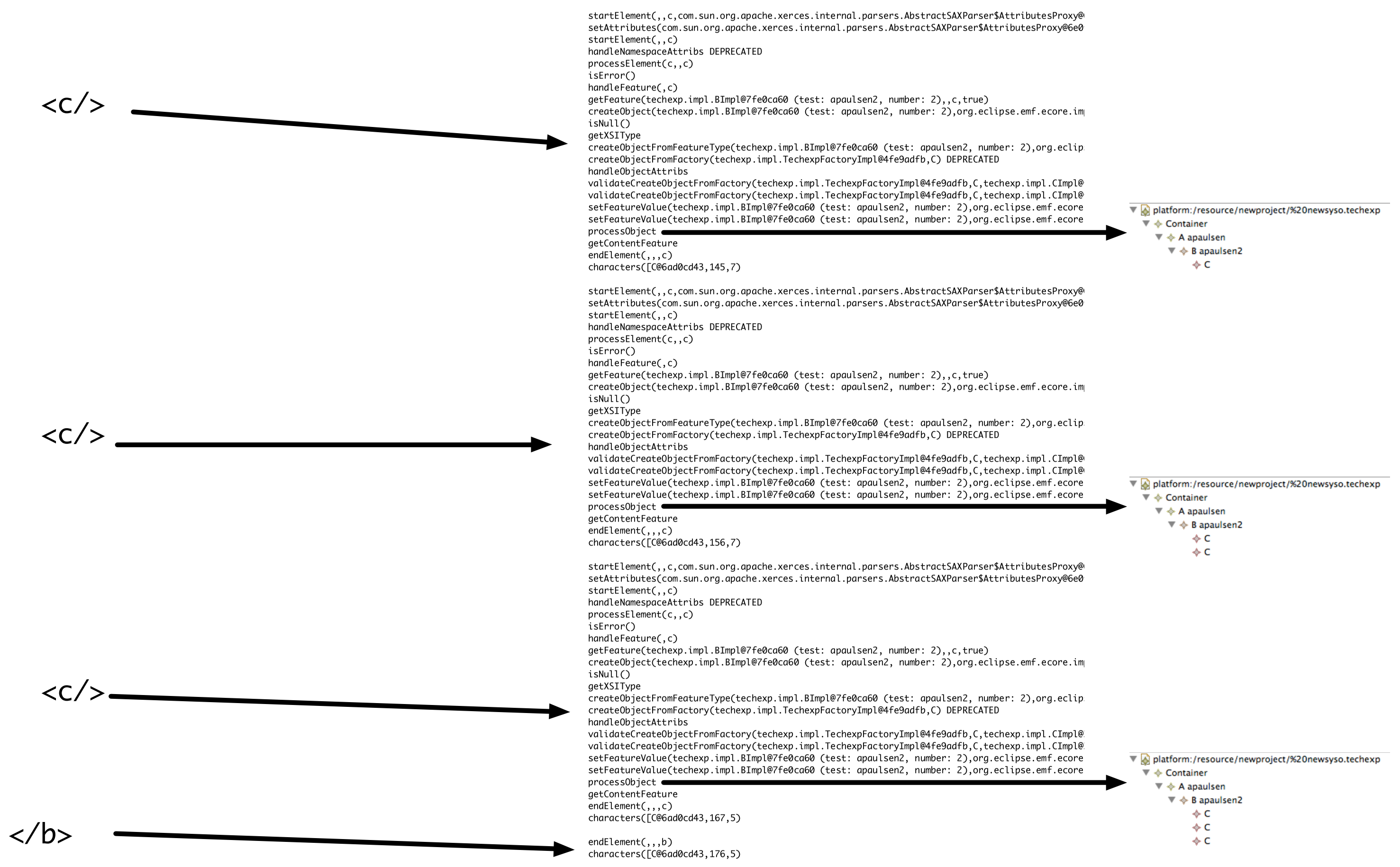

Traverse XMLLoadImpl:606 traverses using a loop

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

<techexp:Container
xmlns:techexp=
"http://www.example.org/techexp">

  <as name="apaulsen">

    <b test="apaulsen2" number="2">
```

```
load
getEncoding
readBuffer
makeParser
setExtendedMetaDataOption
setDocumentLocator
setLocator
startDocument

startElement(,,techexp:Container,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,techexp:Container)
handleNamespaceAttribs DEPRECATED
handleXMLNSAttribute
processElement(techexp:Container,techexp,Container)
recordHeaderInformation
isError()
createTopObject
createObjectByType
handleTopLocations
processSchemaLocations
getFactoryForPrefix
getPackageForURI
createDocumentRoot
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,Container) DEPRECATED
handleObjectAttribs
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,Container,techexp.imp
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,Container,techexp.imp
processTopObject(techexp.impl.ContainerImpl@f8fe81e)
processObject
getContentFeature
characters([C@6ad0cd43,75,3)

startElement(,,as,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,as)
handleNamespaceAttribs DEPRECATED
processElement(as,,as)
isError()
handleFeature(,as)
getFeature(techexp.impl.ContainerImpl@f8fe81e,,as,true)
createObject(techexp.impl.ContainerImpl@f8fe81e,org.eclipse.emf.ecore.impl.EReferenceImpl@303d
isNull()
getXSIType
createObjectFromFeatureType(techexp.impl.ContainerImpl@f8fe81e,org.eclipse.emf.ecore.impl.ERef
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,A) DEPRECATED
handleObjectAttribs
setAttribValue(techexp.impl.AImpl@3e24bab6 (name: default),name, apaulsen)
getFeature(techexp.impl.AImpl@7de43652 (name: default),null,name,false)
setFeatureValue(techexp.impl.AImpl@7de43652 (name: default),org.eclipse.emf.ecore.impl.EAttrib
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,A,techexp.impl.AImpl@
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,A,techexp.impl.AImpl@
setFeatureValue(techexp.impl.ContainerImpl@f8fe81e,org.eclipse.emf.ecore.impl.EReferenceImpl@3
setFeatureValue(techexp.impl.ContainerImpl@f8fe81e,org.eclipse.emf.ecore.impl.EReferenceImpl@3
processObject
getContentFeature
characters([C@6ad0cd43,98,5)

startElement(,,b,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,b)
handleNamespaceAttribs DEPRECATED
processElement(b,,b)
isError()
handleFeature(,b)
getFeature(techexp.impl.AImpl@7de43652 (name: apaulsen),,b,true)
createObject(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.EReferenc
isNull()
getXSIType
createObjectFromFeatureType(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,B) DEPRECATED
handleObjectAttribs
setAttribValue(techexp.impl.BImpl@2ba9ed19 (test: null, number: 0), test, apaulsen2)
getFeature(techexp.impl.BImpl@7fe0ca60 (test: null, number: 0),null,test,false)
setFeatureValue(techexp.impl.BImpl@7fe0ca60 (test: null, number: 0),org.eclipse.emf.ecore.impl
setAttribValue(techexp.impl.BImpl@2ba9ed19 (test: apaulsen2, number: 0), number, 2)
getFeature(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 0),null,number,false)
setFeatureValue(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 0),org.eclipse.emf.ecore
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,B,techexp.impl.BImpl@
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,B,techexp.impl.BImpl@
setFeatureValue(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
setFeatureValue(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
processObject
getContentFeature
characters([C@6ad0cd43,134,7)
```

platform:/resource/newproject/%20newsyso.techexp
Container
platform:/resource/newproject/%20newsyso.techexp
Container
A apaulsen
platform:/resource/newproject/%20newsyso.techexp
Container
A apaulsen
B apaulsen2

Figure A.5: The calling structure for loading. First page.



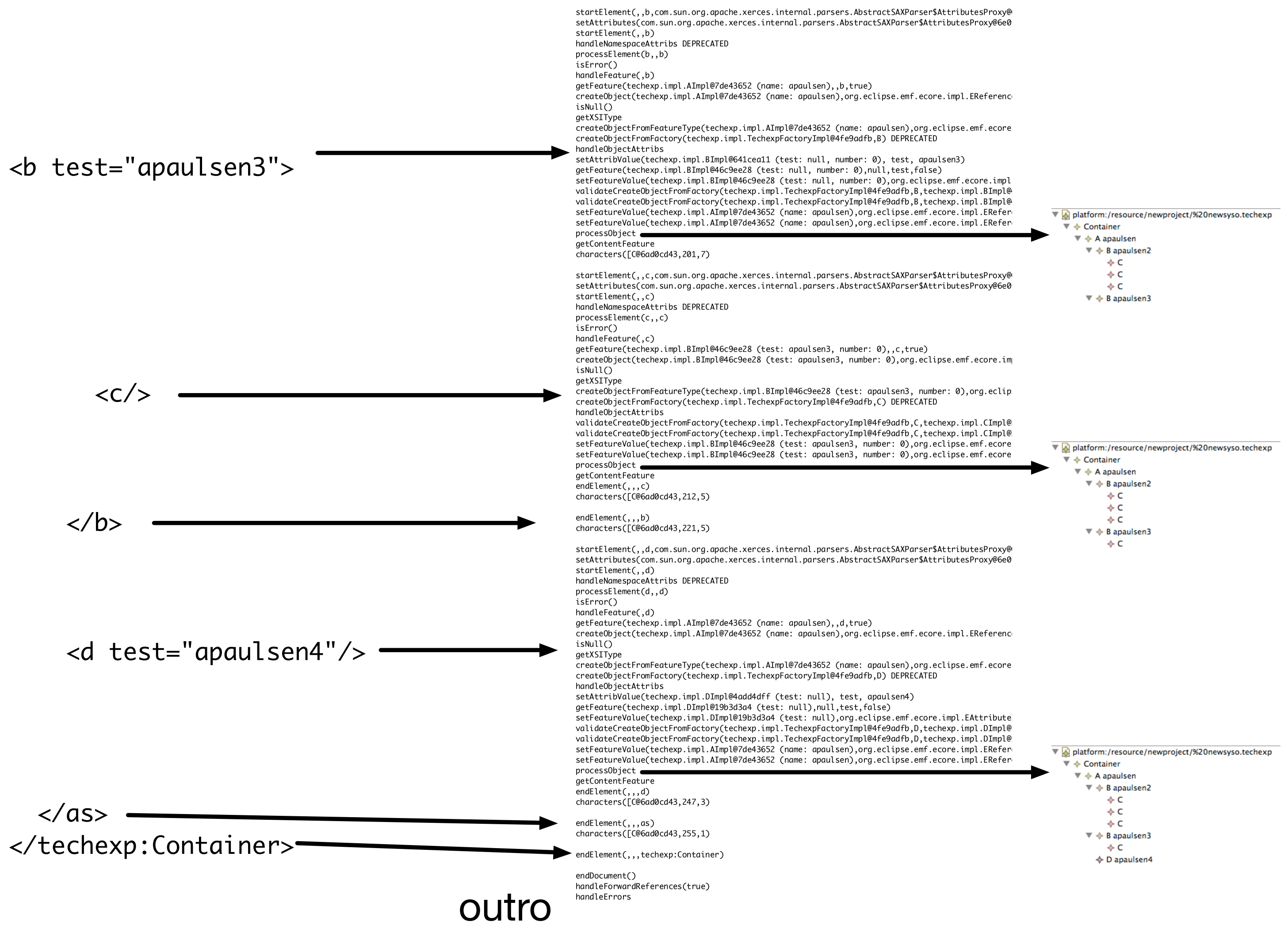
```
startElement(,,c,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,c)
handleNamespaceAttribs DEPRECATED
processElement(c,,c)
isError()
handleFeature(,c)
getFeature(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),,c,true)
createObject(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore.im
isNull()
getXSIType
createObjectFromFeatureType(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclip
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C) DEPRECATED
handleObjectAttribs
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C,techexp.impl.CImpl@
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C,techexp.impl.CImpl@
setFeatureValue(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore
setFeatureValue(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore
processObject
getContentFeature
endElement(,,,c)
characters([C@6ad0cd43,145,7)

startElement(,,c,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,c)
handleNamespaceAttribs DEPRECATED
processElement(c,,c)
isError()
handleFeature(,c)
getFeature(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),,c,true)
createObject(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore.im
isNull()
getXSIType
createObjectFromFeatureType(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclip
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C) DEPRECATED
handleObjectAttribs
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C,techexp.impl.CImpl@
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C,techexp.impl.CImpl@
setFeatureValue(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore
setFeatureValue(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore
processObject
getContentFeature
endElement(,,,c)
characters([C@6ad0cd43,156,7)

startElement(,,c,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,c)
handleNamespaceAttribs DEPRECATED
processElement(c,,c)
isError()
handleFeature(,c)
getFeature(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),,c,true)
createObject(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore.im
isNull()
getXSIType
createObjectFromFeatureType(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclip
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C) DEPRECATED
handleObjectAttribs
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C,techexp.impl.CImpl@
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C,techexp.impl.CImpl@
setFeatureValue(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore
setFeatureValue(techexp.impl.BImpl@7fe0ca60 (test: apaulsen2, number: 2),org.eclipse.emf.ecore
processObject
getContentFeature
endElement(,,,c)
characters([C@6ad0cd43,167,5)

endElement(,,,b)
characters([C@6ad0cd43,176,5)
```

```
▼ platform:/resource/newproject/%20newsyso.techexp
  ▼ Container
    ▼ A apaulsen
      ▼ B apaulsen2
          C
```

```
▼ platform:/resource/newproject/%20newsyso.techexp
  ▼ Container
    ▼ A apaulsen
      ▼ B apaulsen2
          C
          C
```

```
▼ platform:/resource/newproject/%20newsyso.techexp
  ▼ Container
    ▼ A apaulsen
      ▼ B apaulsen2
          C
          C
          C
```

Figure A.6: The calling structure for loading. Second page.



```
startElement(,,b,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,b)
handleNamespaceAttribs DEPRECATED
processElement(b,,b)
isError()
handleFeature(,b)
getFeature(techexp.impl.AImpl@7de43652 (name: apaulsen),,b,true)
createObject(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.EReferenc
isNull()
getXSIType
createObjectFromFeatureType(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,B) DEPRECATED
handleObjectAttribs
setAttribValue(techexp.impl.BImpl@641cea11 (test: null, number: 0), test, apaulsen3)
getFeature(techexp.impl.BImpl@46c9ee28 (test: null, number: 0),null,test,false)
setFeatureValue(techexp.impl.BImpl@46c9ee28 (test: null, number: 0),org.eclipse.emf.ecore.impl
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,B,techexp.impl.BImpl@
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,B,techexp.impl.BImpl@
setFeatureValue(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
setFeatureValue(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
processObject
getContentFeature
characters([C@6ad0cd43,201,7)

startElement(,,c,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,c)
handleNamespaceAttribs DEPRECATED
processElement(c,,c)
isError()
handleFeature(,c)
getFeature(techexp.impl.BImpl@46c9ee28 (test: apaulsen3, number: 0),,c,true)
createObject(techexp.impl.BImpl@46c9ee28 (test: apaulsen3, number: 0),org.eclipse.emf.ecore.im
isNull()
getXSIType
createObjectFromFeatureType(techexp.impl.BImpl@46c9ee28 (test: apaulsen3, number: 0),org.eclip
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C) DEPRECATED
handleObjectAttribs
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C,techexp.impl.CImpl@
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,C,techexp.impl.CImpl@
setFeatureValue(techexp.impl.BImpl@46c9ee28 (test: apaulsen3, number: 0),org.eclipse.emf.ecore
setFeatureValue(techexp.impl.BImpl@46c9ee28 (test: apaulsen3, number: 0),org.eclipse.emf.ecore
processObject
getContentFeature
endElement(,,,c)
characters([C@6ad0cd43,212,5)

endElement(,,,b)
characters([C@6ad0cd43,221,5)

startElement(,,d,com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@
setAttributes(com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$AttributesProxy@6e0
startElement(,,d)
handleNamespaceAttribs DEPRECATED
processElement(d,,d)
isError()
handleFeature(,d)
getFeature(techexp.impl.AImpl@7de43652 (name: apaulsen),,d,true)
createObject(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.EReferenc
isNull()
getXSIType
createObjectFromFeatureType(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore
createObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,D) DEPRECATED
handleObjectAttribs
setAttribValue(techexp.impl.DImpl@4add4dff (test: null), test, apaulsen4)
getFeature(techexp.impl.DImpl@19b3d3a4 (test: null),null,test,false)
setFeatureValue(techexp.impl.DImpl@19b3d3a4 (test: null),org.eclipse.emf.ecore.impl.EAttribute
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,D,techexp.impl.DImpl@
validateCreateObjectFromFactory(techexp.impl.TechexpFactoryImpl@4fe9adfb,D,techexp.impl.DImpl@
setFeatureValue(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
setFeatureValue(techexp.impl.AImpl@7de43652 (name: apaulsen),org.eclipse.emf.ecore.impl.ERefer
processObject
getContentFeature
endElement(,,,d)
characters([C@6ad0cd43,247,3)

endElement(,,,as)
characters([C@6ad0cd43,255,1)

endElement(,,,techexp:Container)

endDocument()
handleForwardReferences(true)
handleErrors
```

Figure A.7: The calling structure for loading. Third page.

Appendix B

Developed Software

This appendix contains the developed software for this project, the tests and vector images of App A.

For the printed version a CD is provided on the following page. If this paper was provided without software, the software can be requested by e-mail to `s093259@student.dtu.dk`

The contents of the CD or zip file is the following list:

appendix a.zip contains vector images of App A.

plugin.zip contains the developed software, the file can be loaded into Eclipse, refer to the handbook in Chap. 6. The file contain four projects: [1]

- **dk.dtu.se.serializer** is the serialiser developed in the project.
- **dk.dtu.se.SerialDefinition** is the mapping mechanism.
- **dk.dtu.se.coremodel** is the coremodel tested in Sect. 8.1.
- **dk.dtu.se.serialExample** is the additional model tested in Sect. 8.1.9.

[1] Three of the projects relies on generated code, this generated code is also provided in the projects `*.edit`, `*.editor` and `*.tests`

runtime.zip contains files for testing the functionality in the runtime workbench. Files can be imported or copied in, when in the runtime workbench.