Bachelor's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Transforming XML documents based on user-defined rules

Jakub Pavlát

May 2017

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Jakub Pavlát

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: Transformace XML dokumentů na základě uživatelsky definovaných pravidel

Pokyny pro vypracování:

Prostudujte možnosti jazyka XSLT[1] a v součinnosti se zadavatelem BP vyberte vhodný způsob zápisu pravidel pro transformaci XML dokumentů ze zdrojového formátu EMF XMI[2] do cílového formátu PNML[3]. Zdrojový dokument popisuje model specifikovaného autonomního systému, cílový formát popisuje realizaci tohoto systému pomocí Petriho sítě. Pravidla pro popis transformace by měla být zapsána formálně přesným a na jazyku XSLT nezávislým způsobem. Z tohoto popisu bude vygenerován kód v XSLT, který transformaci uskuteční. Vygenerovaný XSLT kód otestujte v součinnosti se zadavatelem na vhodných reálných datech.

Seznam odborné literatury:

[1] XSLT - http://www.w3schools.com/xml/xsl_intro.asp
[2] EMF v1.0 Users' Guide
http://www.eclipse.org/modeling/emf/docs/1.x/UG/EMF_v1.0_Users_Guide.html
[3] Petri Net Markup Language (PNML) - http://xml.coverpages.org/pnml.html

Vedoucí: doc. Ing. Karel Richta, CSc.

Platnost zadání do konce letního semestru 2017/2018

L.S.

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
4

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 2.11.2016

Acknowledgement / Declaration

First and foremost allow me to thank my supervisor, Doc. Ing. Karel Richta, CSc., whose expertise and exceptional open minded attitude made this thesis possible. I would also like to thank Ing. Tomáš Richta, as well as his colleagues, for providing me with insight into his research and having the patience to explain it to me over and over again. Let me also express my gratitude to my family for supporting me and everyone else who has made this work to be completed possible.

Abstrakt / Abstract

Tato bakalářská práce se zaměřuje na možné využití XSLT transformací v konverzi z formátu Eclipse Modeling Frameworku (EMF), do Petri Net Markup Language (PNML). Motivace za vývojem takové transformace je její možné využití v oblasti návrhu dynamických architektur pro distribuované embedded řídící systémy, neboť tyto systémy používají k popisu své struktury výše uvedené PNML, zatímco EMF může být, jako již existující formát s nástroji, použit k návrhu a popisu takovýchto systémů a přitom, na rozdíl od PNML, si udržet vysokou úroveň čitelnosti pro člověka.

This bachelor's thesis focuses on the possible use of XSLT transformations for a conversion from the Eclipse Modeling Framework (EMF) format to the Petri Net Markup Language (PNML). The motivation for developing this particular tranformation is its possible further use in the field of dynamic software architecture for distributed embedded control systems design, as these systems use the aforementioned PNML to define their structure, whereas EMF can be used as an existing format and tools to design and describe these systems, while, unlike PNML, retaining a human-readable form.

/ Contents

Chapter 1
Introduction

1.1 Overview

In this thesis we will be examining the possibility of using XSLT[1] to implement a format conversion. The formats in question are Domotic ecore, a domain specific language built as an extension to the Eclipse Modeling Framework (EMF)[7], and Petri Net Markup Language (PNML)[8] and the transformation in question is from EMF to PNML. Both formats are still xml-style documents with a tree structure of the elements. The choice of these two particular standards is motivated by their connection in the field of dynamic software architecture for distributed embedded control system design.

This thesis came to be as a small part of an ongoing research conducted by Tomáš Richta and Vladimír Janoušek from Brno University of Technology, Faculty of Information Technology and Fernando Macías along with Adrian Rutle from Department of Computing, Mathematics and Physics of Western Norway University of Applied Sciences. Their research deals with the use of domain specific languages (along with other components) for distributed control system description. It should be noted, that while this thesis aims to construct a prototype transformation that might aid in their research, its general goal is to examine capabilities of XSLT.

In this context PNML is used to describe the system - its components, data exchange arcs between these component and the rules for such exchanges. A document written in PNML can be translated into executable code and run on a Petri Net Operating System (PNOS)[6]. However, as will become transparent later on, when graphed out (or even more so as a raw document) PNML gives very little idea to the reader of how the system looks, what function each component has and what type of data exchange goes on, which makes the system design inefficient and possibly quite complex. That is why we consider EMF for design. EMF is a modeling framework primarily used for code generation based on a class model. Code generation aside, the modeling features of EMF allow for easy design of basically any structure similar to a class diagram - in our case a distributed system.

Chapter 2 of this thesis is about existing resources, such as EMF and PNML - the two formats between which we will be implementing the transformation. Chapter 3 is where we go over the actual transformation that will need to be implemented as well as its implementation in XSLT. Chapter 4 is reserved for evaluation and assessment of how well and to what extent we have accomplished all the goals of this thesis and if we have not, then why.

Chapter 2
Existing formats and resources

2.1 Petri Nets

A Petri net is a graphical modeling tool used for description and definition of distributed systems. Although initially used for the description of systems consisting of several chemical reactions, Petri nets and the term *distributed system* are not limited to the field of chemistry, but can be applied in electronics and software, as is our case.

A Petri net is made up of 2 major components - states (also called places or, more abstractly, conditions) and transitions. Since the target described systems were originally chemical reactions, states represented substances and transitions reactions. States are represented by circles, transitions by squares or rectangles. They are interconnected by arrows, indicating the flow direction. In our use of Petri nets, the flow always goes both ways so we do not use the term arrow, but rather we call them arcs (from general graph theory). When showing steps of a particular process, tokens, represented as black dots, can be placed inside states and be moved to other states via transitions. This can be useful to visualise a chronological order of operations, however their use is not mandatory.

Based on the structure of the net and certain set restrictions, we can differentiate between classes (or types) of Petri nets. As an example of such a class, we may present *production nets*, simply showing different states of a process, as seen in fig. 2.1, or *cyclic nets*, which describe processes, that may be repeated infinitely (in a *cycle*). [4]

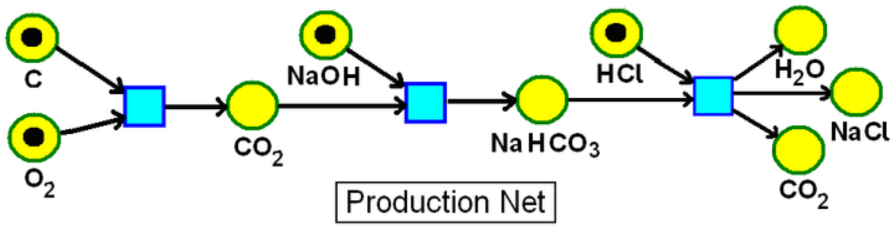

Figure 2.1. an example of a Petri net describing a chemical process [4]

The dynamic software architecture research [5] is based on using *(extended) workflow nets* and that is the class of Petri nets that we will be dealing with. Ultimately however, making the distinction between classes is not necessary in the scope of this thesis, as Petri nets are not the focus of this thesis.

2.2 Petri Net Markup Language

Petri Net Markup Language, or PNML, is a XML-style based format used for high-level Petri net definition interchange. The format specifications are defined in the ISO/IEC 15909-2:2011 (a second part of ISO/IEC 15909) standard. This particular standard was created as a resolution of a purely technical problem of sharing, importing and exporting of Petri nets between tools. A new standard had to be created because there are many types of Petri nets where usually each Petri net type uses a different format for definition. As well as being readable and universal, extraction of information from the document should be possible even without knowing which exact type of Petri net is being described. The format focuses on 3 key features, namely

- Readability - human readable (non-binary) using a conventional text editor
- Universality - every Petri net type can be represented
- Mutuality - easy information extraction, even if the Petri net type is not known

Readability comes from PNML being a subtype of XML. Even though there may be formats that provide more readability, or may be slightly better fitted for the task, XML was chosen because of its overwhelming popularity. Universality is guaranteed by attaching additional information of a Petri net type to *objects* in the net. This information is stored by labeling the net and the objects in it (for example in the form of element attributes). Mutuality can be guaranteed by conventions, which describe a set of standardized labels and their semantics and typical use. [8]

2.2.1 Concepts

The most general format of PNML is a labeled graph, with 2 types of nodes - places transitions. Along with arcs, we call these *objects* of a Petri net. Please note that PNML describes 3 other object types, namely *page*, *reference place* and *reference transition* - we will explain what these are later on.

One file in PNML can contain several Petri nets. Each of these nets consists of objects (previously defined as being of type *place*, *transition* or *arc*). Every object has a unique identifier. The identifier must be unique across the entire document, not just a page, or a Petri net.

Every object can also have a label. Labels are typically place markings, transition guards or arc inscriptions. Labels come in two variants: *annotations* and *attributes*. Typically labels are plain text, that should be displayed near the object it corresponds to, while attributes have a small domain of values which alter the visuals of the object itself (color, shape, etc.). The impact of an attribute on the object is however not defined in PNML and is a problem left for the tool that uses the given net.

Objects and annotations can be enriched by graphical information so that not only the net structure is preserved, but the graphical layout is kept as well. Our main goal is to prepare a Petri net definition so that it can be used to compile code runnable on PNOS, where the graphical information is superfluous. It could be argued that for debugging for example, the graphical information may be important - luckily however, tools for viewing and editing Petri nets described with PNML are usually capable of adding this information on their own automatically, or they let users manipulate the objects and store the resulting information into the PNML file.

Tools that deal with PNML may also need to store additional data, specific to and only usable by this one particular tool. PNML can store this information. This information is marked as tool specific and even contains the name of the tool it is intended for, making it very easy for other tools to simply ignore this information.

The PNML standard also tackles the problem of describing nets, that are too large to fit onto a single page. Obviously the page could be scaled up, but the visual representation of the net would then be poor. Therefore PNML introduces 2 distinct principles for dealing with vast nets.

We have mentioned *pages* and *reference nodes* earlier when we talked about the base objects of every Petri net. The essential idea is, that a part of a net can be encapsulated in a <page> element to create a simulation of drawing parts of the net on different pages, while staying confined within the same document. The problem with drawing a net on different pages is that each page is secluded and cannot be connected to the other pages. This is where *references* come in. If we need to reference a node defined on another page (e.g. to create a transition between 2 places) we can use a *referencePlace* element with attributes *id* (which is of the same type as the *id* attribute in *place*) and *ref*. While *id* has the same meaning as with all other elements, the *ref* attribute is a reference to an *id* attribute of the node we are referencing, defined on another page.

The concept of *modules* is slightly more complex in comparison with pages and references. It builds on the way many systems and software in general is designed, which is defining *modules* and then reusing these *modules* several times in the design. It is a principle easily compared to creating a *class* in a higher programming language and then instantiating it several times. Modules are more complex, however they often exploit the unique features of a particular Petri net type and therefore are not as widespread nor supported. [8] Since we will not be using modules at all, as they are not widely supported, we will skip examining any code fragments.

2.2.2 Syntax

We will now go back to the previously explained concepts and describe how they can actually be expressed in a PNML document.

The following code fragment describes a *place* node (with additional graphical information).[8] *Place* nodes may later be simply referred to as `places`.

```
<place id="p1">
  <graphics>
    <position x="20" y="40"/>
  </graphics>
  <name>
    <value>ready to produce</value>
    <graphics>
      <offset x="-10" y="10"/>
    </graphics>
  </name>
  <initialMarking>
    <value>P</value>
  </initialMarking>
</place>
```

We can see a use of two labels. Both *name* and *initialMarking* are annotations.

It should be noted that the *graphic* element for *place* and *name* has different structure. Graphical information for a *place* are just its Cartesian position, whereas an annotation such as *name* is described with an offset respective to its parent element.

Please note, that the PNML standard requires the *id* attribute of an object to satisfy the restrictions of xs:ID, which include the first character of the ID to be either a letter or an underscore. [2] The compilation routine of PNML for PNOS however is unable to cope with any IDs that contain an underscore or letters, so we are forced to deviate from the PNML standard here and use numerical IDs only.

Another object type was a *transition* (this time with additional tool specific information).[8]

```
<transition id="t1">
  ...
```

```
    <toolspecific tool="PN" version="1.0">
      <hidden/>
    </toolspecific>
  </transition>
```

As mentioned before, the *toolspecific* element does not interest us very much, but if one would want to store some tool specific data, the grammar requires declaring an element *toolspecific* with the attributes *tool* and *version* referencing an existing tool and its verison respectively.

The last of the basic objects was an *arc*

```
  <arc id="a1" source="p1" target="t1">
    <inscription>
      <value>x</value>
    </inscription>
    <type value="inhibitor"/>
  </arc>
```

Since some definitions of Petri net prohibit multiple arcs between a place and a transition, it could be argued that an arc does not need an *id* as it can be uniquely identified by the two objects it connects (referenced by the attributes *source* and *target*), however as stated in The Petri Net Markup Language [8], PNML aims to suffice for the description of even extended Petri nets and therefore should not place a limit on the number of arcs between 2 objects.[4] [8]

Also, because the definition of Petri nets 2.1 states that an arc cannot connect 2 places or 2 transitions (must be from a place to a transition or vice versa), then if the value of *source* is an *id* of a place, then the value of *target* must be an *id* of a transition (and vice versa).

Lastly we talked about how PNML deals with large net structures that are hard to visualize on a single page. We will examine the page and reference constructs without going into too much detail, because our main objective is to produce a PNML document and feed it to a compiler and except for debugging, this feature has no practical use for us at this moment.

```
  <net id="n1" type="HLnet">
    <name>
      <value>Example high-level net</value>
    </name>
    <place id="p1">
      ...
    </place>
    <page id="pg1">
      <referencePlace id="rp1" ref="p1">
        <name>...</name>
        <graphics>
          <position x="20" y="20"/>
        </graphics>
      </referencePlace>
      ...
    </page>
    ...
  </net>
```

The referencing mechanism is demonstrated using the *place* with *id* $p1$. We define it on the top level of the net and it is valid in that context. If we want to connect it to some objects on *page pg1*, we have to define it within the context of the page. Simply declaring it again is prohibited however (because of the *id* principle violation), which is why we use the $referencePlace$ object. If we were to create an arc on the top level from this place, we would use the reference $p1$, however if we wanted to create an arc inside *page pg1*, we would use the reference $rp1$. [8]

The use of $pages$ and $references$ does not give the net any new functionality, however it could be used to organize the net into pages, where each page can be viewed on its own. This could be useful if the system effectively decomposed a problem and each page would tackle one part.

2.3 Petri Net Operating System

Although the workings of the Petri Net Operating System are beyond the scope of this thesis, it plays a key role in Richta's, Janoušek's, Macías' and Rutle's research.

A primary application area of their research are distributed control systems based on wireless sensor networks. The nodes of this net contain PNOS, which is comprised of the kernel and the platform net. The kernel contains PNVM (Petri Net Virtual Machine) which interprets Petri nets that are installed in the system in form of a bytecode (PNBC). PNOS also supports access of the application program to the hardware inputs and outputs, as well as serial communication port. [6]

PNML with the appropriate inscriptions can be used to generate the byte code for PNOS. This is why in this thesis we explore the possibility of a transformation from EMF to PNML.

2.4 Eclipse Modeling Framework

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model.[7]

As the name suggests, the framework is built atop the Eclipse IDE. At its core, without any extensions, the most basic use of the framework would be using a graphical user interface (GUI) to create a data model (also called domain model) for some data you are working with. This meta model is called Ecore. Based on this model, EMF can generate Java classes and interfaces and their factories (in the object oriented programming sense). [7]

The process in this case would be first creating a domain model in the EMF modeling tool. We can create entities, classes, packages and relations between them. If we were modeling a enterprise application of a library for example a part of the model might look something like fig. 2.2.

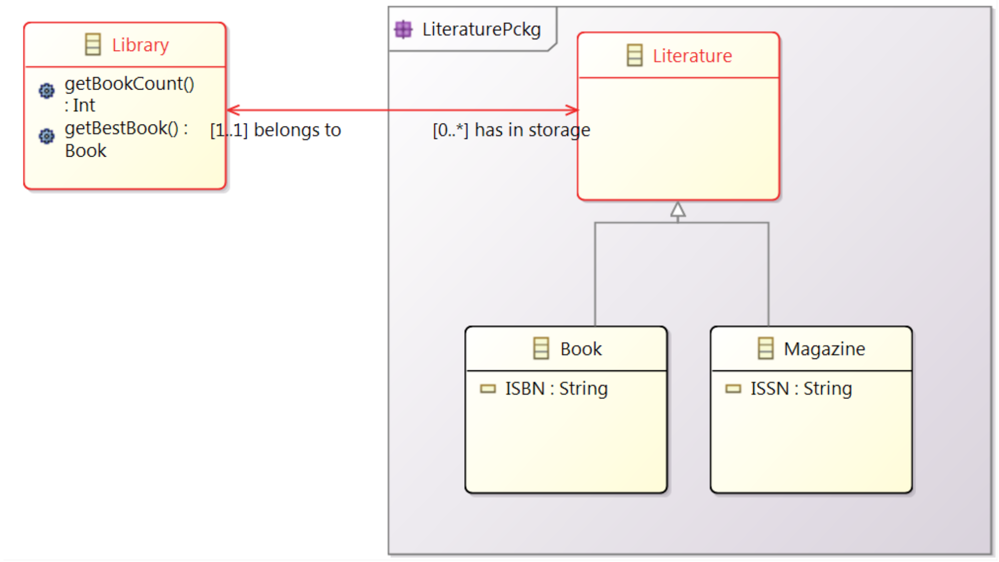

Figure 2.2. an example of a domain model in EMF

When we have our domain model created, we can use EMF to generate interfaces and classes for the model entities. The interface for the *Book* entity, would look like this

```
public interface Book extends Literature {
    String getISBN();
    void setISBN(String value);
}
```

As per the model, the interface inherits from *Literature* and has a getter and a setter for the *ISBN* attribute.

An important thing to note is that EMF stores our UML model in an XML document. More precisely it is an XMI document, but we are not going to be taking any advantage from it being XMI so we may consider it being simple XML. A XMI code fragment that describes the Book entity would look like this

```
<eClassifiers xsi:type="ecore:EClass" name="Book"
     eSuperTypes="#//Literature">
  <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="ISBN" ordered="false" eType="ecore:EDataType
        http://www.eclipse.org/emf/2003/XMLType#//String"/>
</eClassifiers>
```

We can see that this document stores the name of the class, its type and attributes as well as the class from which it inherits.

Although EMF seems like a great tool, we may struggle to find use for it in the field of Petri nets and distributed systems. The thing that makes EMF a great tool to design embedded distributed systems is the fact that the default set of symbols, annotations and relations that are used in the model can be expanded by custom extensions. This means that we do not have to be limited to the predefined entities and relations in the model. That is where Domotic ecore comes in. It is a set of new components and relations that can be used in the model.

At this time, documentation for Domotic ecore is scarce because it is currently in the stages of research and development by Richta, Janoušek, Macías and Rutle.

As the name implies, Domotic ecore contains mainly components that can be used to describe real life components found in an ordinary house - components like knobs, valves, thermostats, boilers, simple computational units and so on. We can use EMF to draw up a model with these components and link them together. For example we might create a *Thermostat* and a *Knob* and link them both to a *Computational* unit and place them into a package.

Hopefully the similarity between creating and packaging classes and interfaces and relations between them and creating and packaging components like thermostats and boilers and relations between them is apparent. It should be clear that the modeling principle remains the same with different components and symbols.

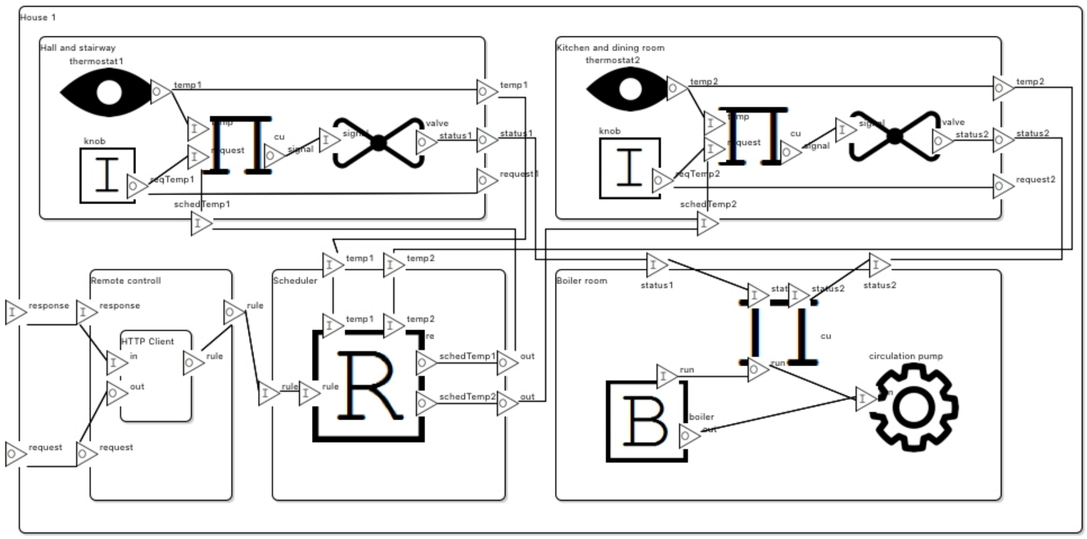

Figure 2.3. an example of a house model in Domotic ecore [5]

Domotic ecore even inherits the packaging principle (placing components into other components). In this particular example we can see that we divided the top level package *House* into four sub-packages based on the room they would be physically in.

The code that corresponds with this model is fairly long so we will not be going through it all, but let us take a look at some fragments of it. This first example is to show how the packaging principle works in practice.

```
<contains xsi:type="domotic:Component" name="House 1">
  <contains xsi:type="domotic:Component" name="Hall and stairway">
    <contains xsi:type="domotic:Sensor" name="thermostat1">
      ...
    </contains>
    <contains xsi:type="domotic:ComputationalUnit" name="cu">
      ...
    </contains>
    ...
  </contains>
  ...
</contains>
```

Instead of using classes, we can use elements with xsi:type *domotic* : *Component* which can contain other *contains* elements. The difference is that there is no limit to how many layers the structure can have, unlike in Ecore where there cannot be a package inside a package. This obviously comes from the fact that there cannot be a package inside a package in Java.

There is also a difference in describing relations in Domotic ecore and basic Ecore.
Ecore has a couple of ways to set up relations between components - SuperType, Reference, Bi-directional Reference and Composition. These relations however are all stored in the class as seen in this code fragment taken from the definition of Library in fig. 2.2.

```
<eClassifiers xsi:type="ecore:EClass" name="Library">
  <eStructuralFeatures xsi:type="ecore:EReference"
        name="has in storage" upperBound="-1"
        eType="#//LiteraturePckg/Literature"
        eOpposite="#//LiteraturePckg/Literature/belongs%20to"/>
  ...
```

As we can see, the relation between Library and Literature (from package LiteraturePckg) is in fact stored in a child element to the Library class element. This is not the case with Domotic ecore. In Domotic ecore to create a relation (two components having data exchange), we use the element *dataFlows* which has attributes *name* and *ports* like so

```
...
<contains xsi:type="domotic:Sensor" name="thermostat1">
  <outputPorts name="temp1"/>
</contains>
<contains xsi:type="domotic:ComputationalUnit" name="cu">
  <inputPorts name="temp"/>
</contains>
<dataFlows name="df" ports="//@contains.0/@contains.0/@outputPorts.0
      //@contains.0/@contains.0/@contains.1/@inputPorts.1"/>
...
```

This fragment declares 2 components - a sensor and a computational unit and their relation. As we can see, the relation is defined and declared outside the declaration of the 2 components.

We should also explain how the relation set by dataFlows works. There are two expressions in the ports attribute separated by a space. Each of these refers to an element in the document. These expression are in principle similar to xPath, yet have different rules of application. No use of traditional xPath axis is allowed except for the *child* axis. Every element must be preceded by a @ symbol, followed by its position (in the sequence of its siblings) designated by an integer after a . sign. It should be noted that we count from 0 rather than 1. Take for example the first expression *//@contains.0/@contains.0/@outputPorts.0*. As stated before, these expressions do not use any other axis than *child* and therefore the first two slashes // are not interpreted as *descendant-or-self* but rather simply as the beginning of the expression. The integer 0 means that we want the 1st element (in case there were more than one). The same rules apply for the rest of the expression.

We can look at the following minimalistic Domotic ecore document to see how the expressions are built up. The elements and their corresponding expression are on the same line.

```
<domotic:Component>
  <contains ...> //@contains.0
    <contains ...> //@contains.0/contains.0
      <outputPorts .../> //@contains.0/contains.0/@outputPorts.0
    </contains>
    <contains ...> //@contains.0/contains.1
      <inputPorts ...> //@contains.0/contains.1/@inputPorts.0
    </contains>
  </contains>
</domotic:Component>
```

Please note that these pseudo-xPath expressions do ignore the root element (the first token of the expression would always be the same).

This overview should be enough to provide basic working knowledge of EMF and Domotic ecore in the extent we will need later on.

Chapter 3 Transformation

3.1 Formal parameters

In order to use EMF for PNML generation, we will need to find a mapping from one format to the other. It should be noted, that while the transformation could be done in both directions - EMF to PNML and PNML to EMF, we are interested only in the former, as transforming PNML to EMF has no use for us right now and we will not be exploring this option further. The formal requirements for the transformation are as follows:

- Transformation input
  1. *EMF* file, as described in 2.4, describing the system structure
- Transformation output
  1. a *.pnml* file describing the top level of the petri net
  2. *n* *.pnml* files each describing the internal structure of a top level component
- Lightweight-*ness* - we want the transformation to as minimalistic as possible, requiring very little other software and resources
- Platform independence - continuing from the previous point - we want this transformation to employ a WORA (write once, run anywhere) principle, to make the area of usability as wide as possible. Using XSLT ensures that any platform with a XSLT processor can use these transformations without much (or ideally none) further implementation.
- Commercial independence - we want this transformation process to not be reliant on any commercial products, or products under licensing so that its use can be universal.

3.2 Petri net top level

The more complex of the 2 outputs of the transformation is creating the top level of the petri net. We will look at the algorithm required to describe the net top level and at its implementation.

3.2.1 Transformation algorithm

As described in 2.4 and 2.2, both PNML and EMF are xml style documents with different nested elements as well as being subjects to different XML Schema Definitions. However, not all EMF elements are relevant for PNML generation, meaning the mapping is not a *one-to-one (bijection)*. To describe the transformation implementation we will first decompose the EMF document into elements that are of interest to us.

- The term *depth* will be used to signify the number of elements in the *ancestor xPath* axis

Format decomposition 3.1.

1. Let C be a set of all $< contains >$ elements in depth 1, such that they match the $xPath$ `Component/contains/contains`
2. Let D be a set of all $< dataFlow >$ elements in depth 1, such that they match the $xPath$ `Component/contains/dataFlows`

Please note that we will refer to this definition later several times.

The desired mapping from EMF to PNML can be achieved by implementing the following algorithm:

Algorithm 3.2.

1. For $\forall C_i \in C$ create a place P_i such that the attribute *id* of P_i is unique across the entire document.
2. For $\forall D_i \in D$ create a transition T_i such that the attribute *id* of T_i is unique across the entire document.
3. For $\forall D_i \in D$ create a pair of arcs A_1 and A_2 such that the attribute *id* of A_1 and A_2 is unique across the entire document and
   a) attribute $from$ of element A_1 has the value of attribute *id* of a place P which was created from an element C referenced by the first expression in attribute *ports*
   b) attribute *to* of element A_1 has the value of attribute *id* of transition T which was created from $dataFlow$ D_i
   c) attribute $from$ of element A_2 has the value of attribute *id* of a place P which was created from an element C referenced by the second expression in attribute *ports* of D_i
   d) attribute *to* of element A_2 has the value of attribute *id* of transition T which was created from $dataFlow$ D_i

To prove the validity and completeness of this algorithm, we would need to formalise the behaviours of both the input and output models as well as the language that represents them. Firstly, this would be a very complex problem on its own, probably exceeding the scope of this thesis. Secondly, as previously mentioned, the Domotic ecore model is still in the early stages of its development and so far no major papers have been published. For these reasons we will only provide a *sketch* of a proof that this solution works.

Sketch of proof would be as follows:

Let G be the Petri net graph in the output. Due to step 1, all components of EMF model C will have appropriate place in the Petri net graph G. Due to step 2, all dataflows in D will have an appropriate transition in the Petri net graph G. In step 3, all transitions created in the step 3 will be linked by input and output arcs according to the source dataflow. [3]

3.2.2 XSLT implementation

In this section, we will try to implement the algorithm described in 3.2.1.

The first approach we take will be fairly straight forward - going through the steps 3.2 we will create a functional section of XSLT for each one, starting with the first line of the algorithm. *For* $\forall C_i \in C$ *create a place* P_i *such that the attribute id of* P_i *is unique across the entire document.* In XSLT this should be a simple operation. Using the constructs *template* and *apply-templates* we can easily achieve this bijective *(one to one)* mapping.

```
<xsl:template match="contains">
  <place>
    <xsl:attribute name="id">
      <xsl:value-of select="generate-id()"/>
    </xsl:attribute>
    <name>
      <text>
        <xsl:value-of select="@name"/>
      </text>
    </name>
  </place>
</xsl:template>
```

The structure of elements *place* -> *name* -> *text* is in accordance with PNML specifications as described in 2.2. The result PNML fragment might look like this:

```
<place id="13">
  <name>
    <text>Kitchen</text>
  </name>
</place>
```

Please note, that the function *generate-id()* has an alphanumerical output (containing both letters and numerals) [2]. However, as was stated in 2.2, the attribute *id* cannot contain any alphabetical characters and has to be strictly numerical. For this purpose we refrain from using *generate-id()* and we will use the following instead:

```
<xsl:attribute name="id">
  <xsl:number format="00000" level="any"/>
</xsl:attribute>
```

This gives us a numeral-only value, as it is a *xsl:number* type node. This node will have a value of all the preceding elements to the one the XSLT processor is currently in. The length of the string in the *format* attribute is obviously dependent on how many elements requiring a unique identification there are overall.

Using this template, we have successfully satisfied step 1 of the algorithm 3.2 - every element *contains* is mapped to an element *place* with a unique *id* attribute across the document. The uniqueness of the *id* attribute is guaranteed by the *one-to-one* mapping between *contains* and *place* and the nature of the attribute value selected. This is because only one element *place* is created every time the XSLT processor reaches an element of type *contains* and in every element *contains* the expression above gives a unique number, because no two elements in one document can have the same number of preceding nodes.

Let us continue with step 2 of 3.2, *For* $\forall D_i \in D$ *create a transition* T_i *such that the attribute id of* T_i *is unique across the entire document.*

As with step 1 this will be very simple. Using the same constructs, we can use almost the same code, changing only the *match* expression from *contains* to *dataFlow*, as well as some of the inner structure of the result *transition* element.

```
<xsl:template match="dataFlow">
  <transition>
    <xsl:attribute name="id">
```

```
        <xsl:value-of select="generate-id()"/>
      </xsl:attribute>
      <create>
        <text>

        </text>
      </create>
    </transition>
  </xsl:template>
```

We have intentionally left the contents of the element *text* blank, because the contents need to have the form of $id1 : output(portname1, id2 : input(portname2)$, where *id* are identifiers of places and *port name* are values of *name* attributes of these places. Using the algorithm we are currently exploring, it would be a complex task to fetch these values, but a more severe problem arises with step 3 of this algorithm, so we will ignore this element for now, and assume that it can be created according to specification nevertheless.

The section above has so far described steps 1 and 2 of the algorithm 3.2. Continuing with step 3, that is

For $\forall D_i \in D$ *create a pair of arcs* A_1 *and* A_2 *such that the attribute id of* A_1 *and* A_2 *is unique across the entire document and*

a) *attribute* $from$ *of element* A_1 *has the value of attribute id of a place* P *which was created from an element* C *referenced by the first expression in attribute ports*
b) *attribute to of element* A_1 *has the value of attribute id of transition* T *which was created from* $dataFlow$ D_i
c) *attribute* $from$ *of element* A_2 *has the value of attribute id of a place* P *which was created from an element* C *referenced by the second expression in attribute ports of* D_i
d) *attribute to of element* A_2 *has the value of attribute id of transition* T *which was created from* $dataFlow$ D_i

Obviously without the succeeding conditions this is a trivial task. We would create a template just like with steps 1 and 2, with different content, namely there being 2 *arc* element instances. Taking into account even just the first condition makes using this approach impossible.

At this point, the XSLT processor has read a $dataFlow$ element. The only directly accessible information is what dataFlow contains which is basically just the *ports* attribute that tells us the source of the routed data and its destination. Although the expression is not actual xpath, let us assume that it could be transformed into a valid xpath expression. As per the 1st condition, we need to get the *id* of a place, that we created from an element (that matches the xpath in *ports*) of the original document. Now even if we could dynamically resolve the xpath in *ports*, we still would not be able to get to the *id* of the place created from it, because the *id* was created for the new document, and has no tie to the original.

To go step-by-step through this problem, let us consider the following input:

```
  <contains>
    <contains>
      <outputPort/>
    </contains>
    <contains>
      <inputPort>
  </contains>
  <dataFlows ports="//@contains.0/@contains.0/@outputPort.0
        //@contains.0/@contains.1/@inputPort.0">
</contains>
```

The transformation of the two *contains* elements are obvious and will produce something like the following

```
<place id="1">...</place>
<place id="2">...</place>
```

Now the processor moves on to the dataFlows element. The processor knows which elements from the original files it wants to link together, but there is no way of knowing what *ids* the places they represent have. XSLT is a declarative programming language which means that we do not describe the control flow or specific instruction that should be executed, but rather describe the logic of the computation.

XSLT therefore lacks any sort of data structures resembling a Key, $Value$ map. If there was a possibility to create such a structure, we could simply keep a map where the expression is the key and the id is the value. When creating places we would store these pairs, and retrieved the values when iterating over the dataFlows elements. As stated before however, there is no such construct in XSLT.

It may seem that merging steps 1 and 3 of the algorithm 3.2 might help - creating the places along with the transitions between them. It is true, that while processing the first dataFlows element, we can create the two place elements, storing their *ids* in two variables and then using these *ids* create a transition. This obviously works, but fails in the next iteration, where the places already have assigned *ids* that you cannot retrieve.

This is why we propose a different approach. In the previous paragraph we talked about merging steps of the algorithm 3.2 together, which only delays the problem, while what we can do is separate step 1 from the rest. What is meant by that, is that we run 2 transformations in a sequence, feeding the result of the first one as input to the other. The problem came from the fact that there was no usable link between the original contains elements and their *ids* in the resulting document - if we knew, what place (and subsequently *id*) each contains element mapped to, the problem would be gone. This can be done in several ways, so we picked what seemed the most straight forward one.

Specifically what we want to do is for the first transformation to transform the relevant contains elements into places, while leaving the rest of the contents intact, while the second transformation implements steps 2 and 3.

We will demonstrate how this solves the problem we could not tackle in just one transformation. The input for the first step is the same as before - Domotic ecore document. Let us reuse some definitions, specifically *Let C be a set of all $<$ contains $>$ elements in depth 1, such that they match the xPath* `Component/contains/contains`.

The objective of the first transformation is simple - accomplish step one of the algorithm 3.2 and copy the entire document. We have already shown a way to achieve the former. Copying the document is simple enough, although it should be noted, that a XSLT expression such as this

```
<xsl:template match="/">
  <xsl:copy-of select="."/>
</xsl:template>
```

is incorrect in this case, as copy-of creates a shallow copy only. What we need is a template that can be applied recursively, to create a deep copy. Fortunately there is a recommended template for that, which is as follows

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

This template matches any node or attribute and therefore effectively copies the entire document. This may seem like we cannot create any new elements, since every node matches this template, however XSLT is very clear about template conflicts and gives more specific match expressions precedence. This means that we can use the same template we used before to transform certain contains elements into places. To both create a new place and keep the original element, we must add a template application on the node itself to create a copy.

The input for the second transformation is then a document that contains all the original information, with the addition of every contains element being preceded (or succeeded, depending on the implementation) by a place element. This gives us a stable relation in the input document between a contains element and its place element's *id*. With this document we can run the second transformation, that will create transitions and arcs.

Because both arcs and transitions are created while iterating over the same set ($\forall D$), we will do steps 2 and 3 simultaneously, in one template. This is for 2 reasons -

1. It makes more sense to iterate over the set once, with every iteration creating 2 arcs and 1 transition, rather than iterating over the set twice (creating transition in one iteration, arcs in the other)
2. If we indeed split them up, when creating the arcs, we would not know what transition *id* to use in the from/to attribute of the arc. This issue is similar to the one we are currently trying to solve - the reason this can be solved by merging the steps together comes from the fact that the $|D| = |T|$ (the number of all dataFlow elements is equal to the number of transition elements), whereas with D and P we can make no such claim.

The idea now is, that if we know to which contains element an expression in the ports attribute of dataFlows is pointing to, we can get the *id* of this place as well. Let us say that in the input file we now have the following

```
<contains ...>
  <place id="1">...</place>
  <contains name="Kitchen" .../>
  <place id="2">...</place>
  <contains name="Bathroom" .../>
  <dataFlows ports="//@contains.0/@contains.0
        //@contains.0/@contains.1"/>
</contains>
```

and we want to create the transition and arcs from place 1 to place 2 (in practice there would be input and output ports, but the principle stands). Let us say that we create a transition with $id = 3$. Now we need to create 2 arcs (from place 1 to transition 3, from transition 3 to place 2). Transforming the expressions from the *ports* attribute into valid xPath could be done, but we run into yet another issue. While XSLT can obviously resolve such xPath expression, it can only do so if it is in the form of a literal, not a variable whose value is computed at runtime.

XSLT 2.0 or lower (without extensions), which are currently the only non-commercial versions, are incapable of dynamic *xPath* resolution. If you consider the following document fragment

```
...
<element id="1"/>
<xpath>//element[@id = "1"]</xpath>
...
```

there is no way to resolve the *xPath* in element <xpath> and thus get a reference to element <element id=1>. The only way to use xPath in these versions of XSLT is to include it in the XSLT file itself, making it useless since we do not know what the expression in the *ports* attribute will be. This is a severe drawback of using the non-commercial versions which requires yet another workaround.

Using commercial implementations of XSLT 3.0 is not possible as per the stated requirements in 3.1 just as using extensions for XSLT 2.0 (or lower). The workaround we propose is fairly simple. Rather than looking at the *ports* expression and calculating which element its xPath equivalent points to, we calculate and add these expressions to every element and then simply compare if the expression matches. We will have to revisit the 1st transformation, as all these values need to be added before the document is usable for the 2nd transformation.

It should be noted, that unlike our simplified examples, a dataFlow must create a port-to-port relation, rather than place-to-place relation. We will therefore have to ensure, that the document after the first processing has both the contents of the original EMF document and in some form stored information about *place*s and their *id*s and their ports along with the appropriate pseudo-xPath expressions for each port.

Again, there are many ways to accomplish this and we suggest the following structure

```
...
<place id="1" px="//@contains.0/@contains.0">
  <ports name="request" px="//@contains.0/@contains.0/@inputPorts.0"
        io="input"/>
  <ports name="temperature" px="//@contains.0/@contains.0/@outputPorts.0"
        io="output"/>
</place>
...
```

where *px* stands for pseudo-xPath and io for input/output to indicate, whether the original element was *inputPort* or *outputPort*.

We have already described a template for the appropriate *contains* elements that creates *place*s, so we can just extend the code of this template. The *id*, *name* and *io* attributes are trivial. What is interesting is how we can derive the expression in *px*.

When creating the attribute itself, we will apply a very general template with a *path* mode, that will trace back to the root element - it will apply itself on every node in the given axis. Using a template with a mode is convenient so that it does not clash with any of our current templates.

```
<xsl:apply-templates select="ancestor-or-self::*" mode="path"/>
```

The template itself looks as follows

```
1  <xsl:template match="*" mode="path">
2    <xsl:text>/@</xsl:text>
3    <xsl:value-of select="concat(name(), '.')"/>
4    <xsl:value-of select=
5         "count(preceding-sibling::*[name()=name(current())])"/>
6  </xsl:template>
```

These lines of code can be interpreted as

- 2 - adds a prefix to every element
- 3 - appends the name of the element and a dot sign
- 4 and 5 - counts how many sibling elements of the same name precede the current element

We also stated, that these expressions always start with `//`. Also this proposed template would match with the root element, which should not be in the expression as stated in 2.4. We can rid ourselves of both problems by adding an if-else structure to the template as such

```
<xsl:choose>
  <xsl:when test="count(ancestor::*) = 0">
    <xsl:text>/</xsl:text>
  </xsl:when>
  <xsl:otherwise>
      ...
  </xsl:otherwise>
</xsl:choose>
```

where the otherwise branch contains the code we used in the template earlier. In the choose construct we simply check if the current element is root and if so, do nothing else than produce a `/`.

Now we have the document in a state with which we can easily implement the rest of the algorithm. It may seem that we are now further from solving the problem than we were at the beginning, since

1. *place* elements now contain superfluous data
2. we have not implemented steps 2 and 3 of the algorithm 3.2.

Filtering the extra data in the *place* elements is trivial. We simply copy everything, that is neither the *px* attribute, or a *ports* element.

Creating the transitions and arcs at this point however becomes almost trivial as well.

When processing a *dataFlows* element (inside a template), we create a transition and store its generated *id* in a variable. We then declare several other variables as follows

```
<xsl:variable name="place1px"
      select="substring-before(@ports, ' ')"/>
<xsl:variable name="place1"
      select="//place[./ports[contains($place1px, @px)]]"/>
<xsl:variable name="place1port"
      select="$place1/ports[contains($place1px, @px)]"/>
```

- `place1px` is just the expression identifying the first element of the relation

- `place1` is an element, that has in its *px* attribute the same expression we are looking for
- `place1port` is the corresponding port element

We duplicate these variables for the second element in that relation as well (substituting `substring-before` for `substring-after`).

Now constructing the arcs is in fact trivial because we have references to both places from the *ports* attributes. When we need to retrieve the *ids* for `place1` and `place2`, we can use

```
<xsl:value-of select="$place1/@id"/>
```

and

```
<xsl:value-of select="$place2/@id"/>
```

respectively.

As per the PNML format definition2.2, there are required contents of the elements, but the mapping is very straight forward and we will therefore not go through those parts of the code.

With this we have accomplished all 3 steps of the algorithm 3.2.

3.2.3 Implementation overview

We will provide a brief and more shallow overview of the final implementation.

- Transformation 1: Copy contents of EMF and add $\forall C_i \in C$ a place that contains all its ports (along with their pseudo-xPath expressions)
- Transformation 2:
  - Copy all places, while filtering all the content not specified in PNML
  - For $\forall D_i \in D$ create a transition and a pair of arcs as is specified in PNML, where the correct *ids* can be found by searching the space of places, with the attribute *px*, that matches the expression in the *ports* attribute of D_i

3.3 Underlying nets

As per point 2 of the formal parameters 3.1, we need to produce separate PNML files for elements one below the top level. The contents of these files are still a subject of research, but so far it seems that the contents will be static and the transformation should act as a simple map, mapping a specific Domotic ecore type to predefined PNML code. Based on the future definition of the contents of these files, we would have to consider during which processing (we run 2 XSLT transformations) this should be dealt with. For now, let us say that we tackle this problem in the second transformation.

The generalized problem can be described as *create n files, where n is the number of specific elements in the input document, with content based on the element's attribute value.*

Firstly, we need to let the XSLT processor know we want it to process these elements. Let us examine the following code fragment

```
1  <xsl:template match="/">
2   <pnml>
3       ...
4   </pnml>
5   <xsl:apply-templates select=
          "domotic:Component/contains/contains/contains"/>
6  </xsl:template>
```

Lines 2 through 4 represent the already explained transformations. On line 5 we are declaring the processing of the elements at what we referred to as *one below the top level* (since we have defined the top level as elements matching $domotic : Component/contains/contains$).

If we want the files to have type-specific content, all we have to do is create a template for every type and restrict it accordingly. For example

```
<xsl:template match="contains[@xsi:type = 'domotic:Sensor']">
```

would be a template that only fits a Sensor type component.

The final problem was creating multiple output files. Fortunately XSLT has a $result-document$ node that does just that [2]. We can use it as follows

```
<xsl:template match="contains[@xsi:type = 'domotic:Sensor']">
  <xsl:result-document method="xml" href="sensor_{generate-id()}.pnml">
    <pnml ...>
      ...
    </pnml>
  </xsl:result-document>
</xsl:template>
```

By wrapping any arbitrary XSLT code in a $result - document$ node, we redirect the output to another document - in this case a document with a unique (based on the generate-id function) name. Now when this template is called, no output is written in the place of the template call - everything inside the $result - document$ is resolved and written in a separate document.

Chapter 4 Conclusion

We have successfully shown that it is in fact possible to convert EMF, or rather Domotic ecore, into PNML using XSLT.

We have also shown however, that what could commonly be understood as using only XSLT - which would be running a transformation with one XSLT stylesheet, is not in fact enough to define rules for such a transformation. Using two succeeding XSLT transformations eliminates the critical problems that arise in a one transformation solution. Managing two transformations, where the product of the first one is fed into the second one requires some, although minimal, management of something other than XSLT. An argument could be however made, that launching the XSLT processor always requires minimal action from a third party. Therefore we would like to think that it could be said that this problem is in fact solvable using only XSLT.

Another interesting finding was the approach XSLT 2.0 uses in dealing with and resolving xPath expressions. While some parts of the xPath expression can be dynamic (e.g. conditions can contain variables), simply feeding a variable (the content of which has been resolved also in runtime) to a field that resolves xPath expressions will not work. However we found that if need be, standard XSLT processors can be upgraded with extensions that support such functionality, as well as the now commercial version 3.0 of XSLT.

After seeing that the transformation process had to be divided into 2 parts, to work around a fairly simple mapping issue, XSLT should be chosen with caution when dealing with logic-heavy transformations.

4.1 Complete transformation example

We have attached a minimalistic example of the prototype transformation. Attached you can find a minimal EMF document with two top level components, *Hall and stairway* and *Scheduler*. They each contain another component - a *Sensor* and a *Rule Engine* respectively. There is also a defined data flow between these two components.

Following is a list of files in the `example` directory and its description.

- `emf.xml` - Domotic ecore file, describing a house, that serves as input for the first transformation
- `emf_pnml_1.xslt` - the transformation stylesheet describing the first part of the transformation
- `emf_pnml_2.xslt` - the transformation stylesheet describing the second part of the transformation
- `pnml.tmp` - a working hybrid file containing both EMF and PNML elements generated by the first part of the transformation, that also serves as input for the second transformation
- `pnml.pnml` - PNML final output file
- `sensor_d1e14.pnml` - PNML output file, describing the `Sensor` element (`d1e14` is a string returned by the generate-id() function)

Demonstrated are all the issues discussed in 3.2.2. Extending existing EMF with place elements, correctly resolving *id*s for data flow elements and creating files for 2nd level components can be examined closer in the example.

References

[1] Kay, M. *XSL Transformations (XSLT) Version 2.0* Retrieved from https://www.w3.org/TR/xslt20 on 10-02-2017

[2] Marsh, J., Veillard, D., Walsh, N. *W3C Recommendation* Retrieved from https://www.w3.org/TR/xml-id on 22-03-2017

[3] Pavlát, J., Richta, K., Richta, T., Janoušek, V. *Model Transformations via XSLT* In: Proc. of DATESO 2017, pp. 43–54, ISBN 978-80-01-06138-1

[4] Petri, C. A., Reisig, W. *Petri nets* Retrieved from http://www.scholarpedia.org/article/Petri_net on 20-03-2017

[5] Richta, T., Janoušek, V., Kočí, R. *Dynamic Software Architecture for Distributed Embedded Control Systems* In: ADECS '15. Bruxelles: CEUR-WS.org, 2015, pp. 1-15, ISBN 1-234-56789-X, ISSN 1613-0073

[6] Richta, T., Janoušek, V., Kočí, R. *Petri Nets-Based Development of Dynamically Reconfigurable Embedded Systems* In: CEUR Workshop Proceedings, 2013, vol. 2013, no. 989, pp. 203-217, ISSN 1613-0073

[7] The Eclipse Foundation, *Eclipse Modeling Framework (EMF)* Retrieved from http://www.eclipse.org/modeling/emf on 15-03-2017

[8] Weber, M., Kindler, E. *The Petri Net Markup Language*, 2003