A GROOVE Solution for the BPMN to BPEL Model Transformation

Maarten de Mol and Maria Zimakova

August 20, 2009
Content

Chapter 1. Introduction......................................................................................................... 3
Chapter 2. The Case Study ................................................................................................... 4
Chapter 3. Overview of the Solution.................................................................................... 5
  3.1. Graph Transformations in GROOVE ................................................................. 5
  3.2. Expressing BPMN in GROOVE ........................................................................... 6
  3.3. Analyze and Connect Gateways .......................................................................... 6
  3.4. Analyze Sequences and Mark Blocks ................................................................. 7
  3.5. Contract Patterns into BPEL Syntax Tree ........................................................... 8
  3.6. Expressing GROOVE Results in BPEL .............................................................. 9
  3.7. Simulation and Results ....................................................................................... 10
Chapter 4. Conclusion ........................................................................................................ 11
Bibliography ........................................................................................................................ 12
Appendix A. XSLT Transformations ................................................................................ 13
  A1. XSLT Transformation From BPMN To GROOVE Format................................. 13
  A2. XSLT Transformation From GROOVE Format To BPEL .................................. 17
Appendix B. Test Case Samples ....................................................................................... 21
Chapter 1.

Introduction

In this paper we present a solution of a model transformation between two standard languages for business process modeling BPMN and BPEL, using the GROOVE tool set. GROOVE is a tool for graph transformations that uses directed, edge labelled simple graphs and the SPO approach [Ren04]. Given a graph grammar \((G, P)\), composed of a start graph \(G\) and a set of production rules \(P\), the tool allows to compute a labelled transition system (LTS) corresponding to all possible derivations in this grammar. The tool is freely available for download. The latest version and documentation can be found on the website http://sourceforge.net/projects/groove. The graph grammar presented here as well as detailed description of the sample realization to the case study is available in the attachment.
Chapter 2.

The Case Study

The goal of the BPMN-BPEL case study [1] is to define a transformation from the subset of BPMN [OMG08] into BPEL [OAS07], using a graph transformation approach. It was considered three variants of this task corresponding to different levels of completeness of the transformation:

1. Transformation restricted to structured process models as defined in [ODH08].
2. Transformation restricted to structured and quasi-structured process models as defined in [ODH08].
3. Transformation covering structured and quasi-structured process models, as well as synchronizing process models as defined in [OAD06].

In our solution we implemented the first and second (partially) levels of the introduced classification.
Overview of the Solution

The rule system that we implemented in GROOVE realizes the mapping described in Figure 2 of [ODH08], as well as part of the mapping described in Figure 3 of [ODH08]. To reduce the complexity of the rules, we have split the system into two parts. First, the structure of the BPMN input is analyzed, which results in the addition of information to the model. Then, the incremental contraction is carried out itself. The separation of analysis and contraction makes the individual rules easier to define, and results in a structured system that is understandable and maintainable. On the other hand, the separation is not strictly necessary and results in a larger system that traverses the model more than once.

The rule system as a whole can roughly be characterized as follows:

1. **Initialize**: express the BPMN model as a graph in GROOVE (Section 3.2)
2. **Analyze gateways**: create explicit connections between the opening and closing gateways of patterns (Section 3.3)
3. **Analyze sequences**: mark the beginning and the end of sequences (and of singleton blocks) (Section 3.4)
4. **Contract patterns**: contract patterns into BPEL syntax tree in GROOVE (Section 3.5)
5. **Final phase**: express the final syntax tree in GROOVE as a BPEL model (Section 3.6).

In the following subsections we will describe these five phases separately.

### 3.1. Graph Transformations in GROOVE

A graph production system (GPS) is a set of graph production rules, each of which can transform a source graph into a new graph called the target graph. The rule specifies both the conditions under which it applies and the changes it makes to the source graph. Technically, a graph production rule consists of two partially overlapping graphs, a left hand side $L$ and a right hand side $R$, and a set of negative application conditions $N$, which are also (connected) graphs partially overlapping with $L$. In order to apply the rule, the left hand side $L$ is matched to (a part of) the source graph $G$, after which the image of $L$ in $G$ is replaced by a copy of $R$; but a matching is only valid if it cannot be extended to any of the graphs in $N$ – in other words, the structure in the negative application conditions is forbidden in the source graph.

In our visual presentation of a rule used in this paper (which is taken from the Groove tools) we combine all these elements together into one graph, made up of four types of elements:

- **Readers**: elements present in both $L$ and $R$. They have to be present in the source graph for $L$ to match and are preserved in the target graph;
- **Erasers**: elements present in $L$ but not in $R$. They are matched in the source graph but are not preserved in the target graph, i.e. they are removed.
- **Creators**: elements absent in $L$ but present in $R$. They are introduced to the target graph.
- **Embargoes**: elements absent in $L$ but present in one of the negative application conditions in $N$.

To distinguish these four types visually, each element has a distinct color and form, as shown in Figure 1: readers are black, erasers are dashed blue (darker gray in black-and-white presentations) creators are bold green (light gray in black-and-white presentations) and embargoes are bold, dashed red (dark gray in black-and-white presentations).

![Figure 1. The graph production rule elements in GROOVE](image)

### 3.2. Expressing BPMN in GROOVE

For this case study we have transformed BPMN models to GROOVE graphs, for which purpose an XSLT transformation (see Appendix A for detailed information) is provided that allows to transform BPMN diagrams in XML-based format from the test case into the GROOVE format.

The graphs in GROOVE are structured as follows:

- The basic units of BPMN are transformed to nodes with as label either **Task**, **ReceiveTask**, **Event** or **Message**. Each such node has an edge named `id` to a string attribute, which represents the identifier (name) of the unit.
- The gateways of BPMN are transformed to nodes with as label either **Fork**, **Join**, **DataXOR**, **EventXOR** or **Merge**.
- The unnamed connections of BPMN are transformed to edges with label `next`. The positive condition connections are transformed to edges with label `if`, leading to inserted nodes with label `Cond`. These condition nodes have an outgoing edge `value` to a string attribute, holding the condition itself. All negative condition connections, finally, are also transformed to edges with label `if`, but lead to inserted nodes with label `Default`.

The only real differences between a BPMN model and its representation in GROOVE are the encoding of unit and condition names by means of string attributes, and the encoding of conditions by explicit `Cond` and `Default` nodes.

Otherwise, the GROOVE graph is simply the same as the BPMN model.

### 3.3. Analyze and Connect Gateways

In this phase the opening and closing gateways of patterns are identified without performing contraction at all. This is achieved with the following algorithm:
1. For each outgoing edge next of each gateway, create a Connect node that establishes a link between the gateway and the target node of next;

2. Propagate Connect over each basic unit, and recursively over each recognized pattern. This results in direct links between connected gateways;

3. Analyze the connections between gateways:
   - If all connections of a Fork gateway lead to a single Join gateway (and the Join does not have any other incoming connections), then a Flow pattern has been recognized (and likewise for Switch and Pick).
     (The flow recognition rule in GROOVE is shown in Figure 2).
   - If a Merge gateway is connected to a DataXOR gateway, and the DataXOR is also connected backwards to the Merge, then a While, Repeat or Repeat+While pattern has been recognized.

The analyzed gateway connections are also used to turn some quasi-structured patterns into well-structured ones. The rule in Figure 3, for example, recognizes a Fork and a Join where all outgoing edges of the Fork connect to the Join, but the Join has more incoming connections. It then moves these other connections to a newly created Join gateway, which ensures that the original pattern becomes well-formed. Similar rules can be created for duplicating the Fork, as well as for duplications in Switch and Pick patterns.

3.4. Analyze Sequences and Mark Blocks

In this phase the next edges in the graph are renamed to better reflect their role. Edges leading to a ‘block’ (which is either the largest possible sequence or a single unit that is not part of a sequence) are renamed to begin, and edges leading out of a block are renamed to end. This allows sequences to be contracted from left to right in the next phase. Also, edges leading out of a DataXOR gateway in a recognized Switch pattern are renamed to if, and edges leading out of a EventXOR in a recognized Pick pattern are renamed to event.
Furthermore, explicit Empty nodes are inserted:

1. in empty Switch and Pick paths (which ensures that all paths contain at least one unit); and

2. between the Merge and DataXOR in a While and vice-versa between the DataXOR and the Merge in a Repeat (which ensures that only the Repeat+While pattern remains).

Finally, the superfluous default selector at the exit of a Repeat+While pattern is removed. These changes are only cosmetic in nature, and make the rules in the contracting phase easier to read.

### 3.5. Contract Patterns into BPEL Syntax Tree

In this phase the structured patterns are recognized and contracted into one node (following patterns in [ODH08]) to construct a BPEL syntax tree. This is achieved with the following algorithm:

1. For each task or event node, create an appropriate node in the syntax tree and a link bpel between Contracted node and the syntax tree node. The mapping between task/event nodes and syntax tree nodes is following:
   - Task with the attribute id ⇐ Mapping with the attribute id
   - Event Message/Clock with the attribute id ⇐ Receive/Wait with the attribute name
   - Empty ⇐ Empty

   (The task contraction rule and message contraction rule are shown in Figure 4).

2. Recognize and contract the Sequence patterns as a sequence of Contracted nodes between begin and end edges which are connected by next edges. The appropriate construction in the syntax tree is Sequence; the order of sequence elements is shown by edges next.

3. Recognize the Switch, Pick and Flow patterns with the following algorithm:
3.6. Expressing GROOVE Results in BPEL

For this case study we have transformed the resulting syntax tree to BPEL XML-based format with another XSLT transformation (see Appendix A for a transformation description and Appendix B for the results). The transfer algorithm is following:

- For all nodes in the GROOVE syntax tree, create an xml-tag in the BPEL representation.
- If the node in the GROOVE syntax tree has an attribute then xml-tag in the BPEL representation has an attribute with the same name.
- Tags Wait, Receive, Empty do not contain other nested tags.
Some example cases of BPEL models are available in the Appendix B for the reader to try out.

3.7. Simulation and Results

The graph grammar presented here is able to tackle all provided test cases (including 'quasi-structured'), with the only exception of 'synchronized'. The rule system as well as detailed description of the sample realization to the case study is available in the SHARE system [MZ09].
Chapter 4.

Conclusion

Using a number of advanced features of GROOVE, namely attributed graphs, quantified rules, rule parameters and controlled simulation, we have been able to very quickly and easily model the BPMN-BPEL test case as it was described in [Dum09].

We have shown results for the sample realization to the case study in the Appendix B to this paper.

The rule system as well as detailed description of the sample realization to the case study is available in the SHARE system [MZ09] for the reader to try out.

Acknowledgements: The research in this paper was carried out in the GRASLAND project (NWO) and the CHARTER project (EU Artemis).
Bibliography


Appendix A.

XSLT Transformations

A1. XSLT Transformation From BPMN To GROOVE Format

```xml
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="text"/>
</xsl:stylesheet>
```
A2. XSLT Transformation From GROOVE Format To BPEL

<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="text"/>
  <xsl:template match="graph">
    <xsl:text>&lt;?xml version="1.0" encoding="UTF-8"?&gt;
</xsl:text>
    <xsl:text>&lt;process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract" name="
</xsl:text>
    <xsl:value-of select="@id"/>
    <xsl:text"> targetNamespace="http://samples.org/"
</xsl:text>
    <xsl:value-of select="@id"/>
    <xsl:text">&gt;
</xsl:text>
    <xsl:text>&lt;variables&gt;
</xsl:text>
    <xsl:for-each select="edge[attr/string='condition']">
      <xsl:text>   &lt;variable name="
</xsl:text>
      <xsl:variable name="varop" select="@to"/>
      <xsl:variable name="varname" select="../edge[@from=$varop and
      @to=$varop]/attr/string/text()"/>
      <xsl:value-of select="substring-before(substring-after($varname,
      '"'), "'")
</xsl:text>
      <xsl:value-of select="'/"/>
      <xsl:text>&gt;&lt;/variable&gt;
</xsl:text>
    </xsl:for-each>
    <xsl:text>&lt;/variables&gt;
</xsl:text>
    <xsl:call-template name="temp_tree_root"/>
    <xsl:text> &lt;/process&gt;
</xsl:text>
  </xsl:template>
</xsl:stylesheet>
</xsl:when>
<xsl:when test="$vartag='While'">
  <xsl:for-each select="..//edge[@from=$varop and @from!=@to and
./attr/string/text()='condition']">
    <xsl:variable name="varattr" select="@to"/>
    <xsl:for-each select="../edge[@from=$varattr
and @from=$to]">
      <xsl:text>condition="</xsl:text>
      <xsl:variable name="varval" select="substring-before(substring-after(./attr/string/text(), ' quot;'), ' quot;')"/>
      <xsl:value-of select="$varval"/>
      <xsl:text>"></xsl:text>
    </xsl:for-each>
  </xsl:for-each>
  </xsl:when>
  </xsl:for-each>
  <xsl:text>&gt;</xsl:text>
</xsl:for-each>
<xsl:for-each select="..//edge[@from=$varop and @from!=@to and
./attr/string/text()='body']">
  <xsl:variable name="varnext" select="@to"/>
  <xsl:for-each select="../edge[@from=$varnext and @from=@to]">
    <xsl:call-template name="temp_node"/>
  </xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:when>
</xsl:for-each>
<xsl:when test="$vartag='Case'">
  <xsl:text>&gt;</xsl:text>
</xsl:for-each>
<xsl:for-each select="..//edge[@from=$varop and @from!=@to and
./attr/string/text()='condition']">
  <xsl:variable name="varattr" select="@to"/>
  <xsl:for-each select="../edge[@from=$varattr
and @from=@to]">
    <xsl:text>&lt;condition&gt;</xsl:text>
    <xsl:variable name="varval" select="substring-before(substring-after(./attr/string/text(), ' quot;'), ' quot;')"/>
    <xsl:value-of select="$varval"/>
    <xsl:text>&lt;/condition&gt;</xsl:text>
  </xsl:for-each>
</xsl:for-each>
  </xsl:for-each>
  <xsl:for-each select="..//edge[@from=$varop and @from!=@to and
./attr/string/text()='body']">
  <xsl:variable name="varnext" select="@to"/>
  <xsl:for-each select="../edge[@from=$varnext and @from=@to]">
    <xsl:call-template name="temp_node"/>
  </xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:when>
</xsl:for-each>
<xsl:for-each select="..//edge[@from=$varop and @from!=@to and
attr/string/text()!='id' and attr/string/text()!='condition' and
attr/string/text()!='body']">
  <xsl:variable name="varnext" select="@to"/>
  <xsl:for-each select="../edge[@from=$varnext and @from=@to]">
    <xsl:call-template name="temp_node"/>
  </xsl:for-each>
</xsl:for-each>
</xsl:when>
</xsl:for-each>
<xsl:otherwise>
  <xsl:variable name="vartext" select=""/>
  <xsl:if test="$varop=3">
    <xsl:value-of select="$vartext"/>
  </xsl:if>
  <xsl:if test="$varop=4">
    <xsl:value-of select="$vartext"/>
  </xsl:if>
  <xsl:text>&lt;/</xsl:text>
</xsl:otherwise>
Appendix B.

Test Case Samples

Sample 1. Sequence

(a) Initial BPMN diagram

(b) Phase 1. Initial GROOVE representation

(c) Phase 2-3. Analyze gateways and sequences

(d) Phase 4. Contract patterns into BPEL syntax tree

(e) Phase 5. Express (d) in a BPEL format

(f) BPEL graphical representation

Fig. 1. Transformation phases for the sample Sequence
Sample 2. While

(a) Initial BPMN diagram

(b) Phase 1. Initial GROOVE representation

(c) Phase 2. Analyze gateways

(d) Phase 3. Analyze sequences

(e) Phase 4. Contract patterns into BPEL syntax tree
(f) Phase 5. Express (e) in a BPEL format

(g) BPEL graphical representation

Fig. 2. Transformation phases for the sample While
Sample 3. doWhile

(a) Initial BPMN diagram

(b) Phase 1. Initial GROOVE representation

(c) Phase 2. Analyze gateways

(d) Phase 3. Analyze sequences

(e) Phase 4. Contract patterns into BPEL syntax tree
(f) Phase 5. Express (e) in a BPEL format

(g) BPEL graphical representation

Fig. 3. Transformation phases for the sample doWhile
Sample 4. smallStructured

(a) Initial BPMN diagram

(b) Phase 1. Initial GROOVE representation

(c) Phase 2. Analyze gateways

(d) Phase 3. Analyze sequences

(e) Phase 4. Contract patterns into BPEL syntax tree
(f) Phase 5. Express (e) in a BPEL format

(g) BPEL graphical representation

Fig. 4. Transformation phases for the sample smallStructured
Sample 5. largeStructured

(a) Initial BPMN diagram

(b) Phase 1. Initial GROOVE representation

(c) Phase 2. Analyze gateways
(d) Phase 3. Analyze sequences

(e) Phase 4. Contract patterns into BPEL syntax tree
(f) Phase 5. Express (e) in a BPEL format

(g) BPEL graphical representation

Fig. 5. Transformation phases for the sample largeStructured
Sample 6. parallelFlows

(a) Initial BPMN diagram

(b) Phase 1. Initial GROOVE representation

(c) Phase 2. Analyze gateways

(d) Phase 3. Analyze sequences

(e) Phase 4. Contract patterns into BPEL syntax tree
<xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract" name="s50" targetNamespace="http://samples.org/s50">
<variables/>
<sequence>
<invoke name="ca18" partnerLink="local" portType="localPT" operation="ca18" inputVariable="ca18_data_in" outputVariable="ca18_data_out"/>
<invoke name="dvpl" partnerLink="local" portType="localPT" operation="dvpl" inputVariable="dvpl_data_in" outputVariable="dvpl_data_out"/>
<invoke name="ca2m" partnerLink="local" portType="localPT" operation="ca2m" inputVariable="ca2m_data_in" outputVariable="ca2m_data_out"/>
<flow>
<sequence>
<invoke name="fc7d" partnerLink="local" portType="localPT" operation="fc7d" inputVariable="fc7d_data_in" outputVariable="fc7d_data_out"/>
<invoke name="fou9" partnerLink="local" portType="localPT" operation="fou9" inputVariable="fou9_data_in" outputVariable="fou9_data_out"/>
</sequence>
<sequence>
<invoke name="ii1v" partnerLink="local" portType="localPT" operation="ii1v" inputVariable="ii1v_data_in" outputVariable="ii1v_data_out"/>
<flow>
<invoke name="g792" partnerLink="local" portType="localPT" operation="g792" inputVariable="g792_data_in" outputVariable="g792_data_out"/>
<invoke name="lt5w" partnerLink="local" portType="localPT" operation="lt5w" inputVariable="lt5w_data_in" outputVariable="lt5w_data_out"/>
</flow>
<invoke name="lt9q" partnerLink="local" portType="localPT" operation="lt9q" inputVariable="lt9q_data_in" outputVariable="lt9q_data_out"/>
</sequence>
</flow>
</sequence>
</process>

(f) Phase 5. Express (e) in a BPEL format

(g) BPEL graphical representation

Fig. 6. Transformation phases for the sample parallelFlows
Sample 7. quasiStructured

(a) Initial BPMN diagram
(b) Phase 1. Initial GROOVE representation
(c) Phase 2. Analyze gateways
(d) Phase 3. Analyze sequences
(e) Phase 4. Contract patterns into BPEL syntax tree
(f) Phase 5. Express (e) in a BPEL format

(g) BPEL graphical representation

Fig. 7. Transformation phases for the sample quasiStructured