Chapter 3

The RML Tutorial

Author: Joost Jacob

When reading this tutorial you could go directly to Sect. 3.4 Installing and running now if you are in a hurry and just want to learn RML quickly. There are examples in Sect. 3.5 Examples that will introduce everything incrementally and step by step. You can treat the examples as exercises and try to solve them before looking at the solutions. When trying to solve such exercises you can use Table 2.1 for an overview of all the RML constructs that are defined in the current version (September 23, 2008) of RML. Sections 3.1–3.3 are meant for readers who prefer a little more introduction and explanations.

RML (Rule Markup Language) was designed for ease of use. You do not have to be an experienced programmer to use RML. My experiences with the existing transformation methods for XML were such that I felt it was a good idea to come up with something much more simple and elegant.

It is assumed that the reader does have at least a superficial knowledge of XML, like what are XML element names and attributes and for example the fact that well-formed XML only has one root element. With this tutorial the reader can learn how to transform XML with RML, according to transformation rules that are defined using the input XML itself.
Other approaches for XML transformations do not make much use of the problem domain XML for defining transformation rules. They define the transformations in (complex!) special purpose XML [Cla] or they are more low level, defined in various programming languages.

It is not my goal to replace existing technology for XML transformations, but to add a new, easy to use and interoperable technology. If you have a transformation that is easily done with XSLT for instance, then by all means use XSLT. But sometimes you will need transformations that are hard to program with XSLT, for example removing duplicate child elements, and then RML provides an easy solution for your transformation problem.

Your ideas for improving this tutorial are welcome. I have been trying to make this tutorial easy to understand and readable, but English is not my native language so there are probably lots of grammar and style errors. Please do send your suggestions by email to Joost.Jacob@gmail.com.

3.1 The XML vocabulary for the examples

The example XML vocabulary in this tutorial is an XML vocabulary for modeling business processes. Such a vocabulary can be defined with a DTD or with an XML Schema but to save space here Fig. 3.1 gives just an informal definition.

<table>
<thead>
<tr>
<th>element</th>
<th>attributes</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td></td>
<td>the root element</td>
</tr>
<tr>
<td>process</td>
<td>id, name</td>
<td></td>
</tr>
<tr>
<td>role</td>
<td>id, name</td>
<td></td>
</tr>
<tr>
<td>collaboration</td>
<td>id, name</td>
<td></td>
</tr>
<tr>
<td>event</td>
<td>id, name</td>
<td></td>
</tr>
<tr>
<td>object</td>
<td>id, name</td>
<td></td>
</tr>
<tr>
<td>triggering</td>
<td>id, name</td>
<td>must contain a from and to element</td>
</tr>
<tr>
<td>realisation</td>
<td>id, name</td>
<td>must contain a from and to element</td>
</tr>
<tr>
<td>use</td>
<td>id, name</td>
<td>must contain a from and to element</td>
</tr>
<tr>
<td>from</td>
<td>href</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>href</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: The XML vocabulary for the examples
3.2 How RML works

Details will be explained in Sect. 3.5 but here is already a short description of how RML works.

The simplest RML tool is called applyrule. It is a Python[vR95] library that can also be used as a command-line program, and it takes as input some problem domain XML and an RML-rule. Both the XML and the RML-rule will normally be provided in files. The applyrule program then transforms the problem XML according to the rule and outputs the result.

3.2.1 Rules

An empty rule looks like shown in Figure 3.2.

```xml
<div class="rule">
  <div class="antecedent">
    <!-- Insert matching pattern here -->
  </div>

  <div class="consequence">
    <!-- Insert output pattern here -->
  </div>
</div>
```

Figure 3.2: An empty RML-rule

As can be seen in Fig. 3.2 an empty rule consists of div elements like in XHTML. A rule consists of an antecedent (input) and a consequence (output), marked with class attributes of div elements. The XML comments in Fig. 3.2 show what must be done to change this empty rule template to rules that actually do something. How this is done is explained later. The
empty rule defines an empty transformation: the output is the same as the input.

3.2.2 Literal matching

Any XML in the antecedent of a rule that is pure problem domain XML is matched literally. An exception is how the attributes are matched: if the (attributes, value) pairs of an element in the antecedent are a subset of the pairs of an element in the input, and if the elements have the same name, then it is considered a match. This means that input elements can have more attributes that are not involved in the matching process.

3.2.3 Wildcard matching

A core idea of RML is to define XML notation for an XML version of constructs like the * and ? and + and others in expressions for text matching with wildcards, as illustrated in Fig. 3.3.

```
d:\tutorial>dir

Directory of d:\tutorial

11/21/2003 05:14 PM <DIR> .
11/21/2003 05:14 PM <DIR> ..
11/21/2003 05:11 PM 19 g.bat
11/21/2003 05:07 PM 74 make.bat
11/21/2003 05:07 PM 7,764 tutorial.dvi
11/21/2003 05:07 PM 6,962 tutorial.tex

d:\tutorial>dir tutorial.*

Directory of d:\tutorial

11/21/2003 05:07 PM 7,764 tutorial.dvi
11/21/2003 05:07 PM 6,962 tutorial.tex
```

Figure 3.3: The * wildcard in action in the Windows XP shell
3.2. HOW RML WORKS

These wildcard constructs can then be used for matching (parts of) XML. The input that matches a wildcard can then be remembered in RML-variables. The constructs are called XML wildcard expressions.

RML uses XML wildcard expressions to bind parts of the input XML into RML-variables. There are XML wildcard elements and XML wildcard attributes. The XML wildcards elements are special RML elements that can be mixed with the problem domain XML in the antecedent of a rule. The XML wildcard attributes are used for binding attribute values into RML-variables. Exactly how it is done will be made clear later with examples. All the RML constructs are shown in Table 2.1 in the Appendix. As can be seen it all fits in one table, and the RML constructs are designed to be easy to remember. Future versions of RML may add extra constructs, but the ones shown in Table 2.1 are all that were needed so far for the XML transformations I encountered in practice.

3.2.4 Search and replace

The applyrule program tries to find the pattern in the antecedent of a rule in the input XML. If it finds a piece of input XML that matches the pattern, then it replaces that piece of input by the consequence of the rule.

3.2.5 The dorules tool

The dorules program takes as input some problem XML and a list of RML rules. The RML rules are passed as filenames separated by + characters. For instance:

dorules -i myinput.xml -r rule1.xml+rule2.xml

It then applies the first rule of the list to the input, just like the applyrule program, and if there is a match (the output is different from the input) then the program restarts, taking the generated output and the list of rules as input. If a rule does not match the input then the next rule in the list is tried. If no rule in the list matches the input then the program stops. This program turned out to be useful in practice, alleviating the writing of commandline scripts. Such commandline scripts can perform complex transformations by executing applyrule and dorules repeatedly with varying rules.
3.2.6 The dorecipe tool

To avoid writing commandline scripts at all, the dorecipe program is available. With the dorecipe program the user can define a sequence of applyrule and dorules executions in XML. The XML used is Recipe RML (RRML). An example RRML recipe is shown in Fig. 3.4.

```xml
<rrml-recipe>
  <apply>
    <rule>
      <variable name="ID" value="commandline-ID"/>
      <directory name="rules"/>
      <filename name="effect1.xml"/>
    </rule>
  </apply>
  <iterate>
    <rule>
      <directory name="rules"/>
      <filename name="effect3.xml"/>
    </rule>
    <rule>
      <directory name="rules"/>
      <filename name="effect2.xml"/>
    </rule>
  </iterate>
</rrml-recipe>
```

Figure 3.4: An RRML recipe

An RRML recipe has a root element called rml-recipe. This root element can contain apply elements and iterate elements. The apply element corresponds to the execution of the applyrule program and the iterate element corresponds to the dorules program. An apply element must contain exactly 1 rule element, an iterate element must contain 1 or more rule elements. Finally, a rule element must contain a filename element and it can contain a directory element and 0 or more variable elements. Later in this tutorial there will be examples showing the usage of recipes.
3.3. FUTURE VERSIONS OF RML

3.2.7 XML parsing details

RML was designed to transform XML elements and their text content. In the current version of RML only XML elements and their text contents are preserved. This means that for instance XML comments or processing instructions are removed. The reason for this is to make the RML tools as portable as possible, independent of the capabilities of the available XML parsers on a platform. There are many XML tools available outside RML to extract and handle things like processing instructions so I suggest you use those if you need to use XML constructs that are not XML elements. However, contact the author if you have suggestions.

In XML elements produced by the RML tools, the order of attributes in the set of attributes may be changed with respect to the input, this depends on the XML parser on your system that is used by the RML tools. This should not be a problem: relying on attribute order in XML is generally considered bad XML usage.

The RML tools look for the pyRXP module and use that if available. The pyRXP module uses the very fast RXP parser, that is an example of a parser that does not preserve attribute order. If your system is missing the pyRXP parser then the old xmllib Python parser is used, this one is available since Python version 1.5.2 from 1999. Support for more parsers may be added in the future.

3.3 Future versions of RML

Some ideas for future versions of RML that would be simple to implement but were not necessary in the XML languages in the projects it was used for are listed here.

- String concatenation.
  Example: `<newprefix+rml-X />` in the consequence. If X is bound (in the antecedent) to "MyName" then this will produce a `<newprefixMyName />` element.

- Arithmetic
  Example: `<... ...= 10+rml-X />` in the consequence. If X is bound to a string representing a number, then the sum of 10 and this number becomes the attribute value.
• XPath support.
  Example: `<rml-if xpath="model/*/process" />` will let a match only succeed if the previous element is a `process` element with a `model` ancestor.
3.4 Installing and running

Unzip the supplied zipfile, this will create a directory called rml, containing several other subdirectories. The RML tools, `applyrule` and `dorules` and `dorecipe`, are in the rml directory. It is assumed the reader does know how to run commandline programs and how to go to directories on his or her operating system. If you have a default Python [vR95] installation on a Windows computer then you can call Python files as executable programs, because the .py filename extension will be marked as executable. To get Python, go to http://www.python.org and download the executable installer. In the example commands you can then use

```
C:\mystuff\rml\applyrule.py -i myinput.xml -r myrule.xml
```

from any other directory, assuming you did install RML in C:\mystuff. With linux or unix systems you usually have to prepend `python` to commands, for instance

```
$ python applyrule.py -i myinput.xml -r myrule.xml
```

If one of the tool programs is run without arguments then a short usage help is output.

The tutorial examples are in subdirectory examples below tutorial below rml. If you go to the examples directory you can run the `applyrule` program on file ”input.xml” there using the empty rule in file ”rule.empty.xml” by issuing the

```
..\..\applyrule.py -i input.xml -r rule.empty.xml
```

command. This will output the contents of the file ”input.xml”, since nothing was matched. The output is pretty printed, with more indentation for elements deeper down the tree hierarchy, and with attributes indented from the element name and below each other. Figure 3.5 shows an input and the output from applying the empty rule.
CHAPTER 3. THE RML TUTORIAL

Input:

```xml
<model xmlns="Concepts.ArchiMate.Generic"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="Concepts Generic.xsd">

  <event id="008" name="request for insurance"/>
  <process id="009" name="investigate"/>

  <triggering id="015" name="triggers">
    <from href="008"/>
    <to href="009"/>
  </triggering>

</model>
```

Output:

```xml
<model
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="Concepts.ArchiMate.Generic"
       xsi:schemaLocation="Concepts Generic.xsd">

  <event
        name="request for insurance"
        id="008"/>

  <process
        name="investigate"
        id="009"/>

  <triggering
        name="triggers"
        id="015">
    <from
         href="008"/>
    <to
         href="009"/>
  </triggering>

</model>
```

Figure 3.5: Input and output when using the empty rule
3.5 Examples

3.5.1 Deleting an element

Deleting an element using a literal match

Example 001 Suppose you want to transform

```xml
<model>
  <event name="request for insurance" id="008" />
  <process name="investigate" id="009" />
  <process name="formalize request" id="010" />  
</model>
```

available in file input.001.xml
to

```xml
<model>
  <event name="request for insurance" id="008" />
  <process name="formalize request" id="010" />  
</model>
```

, removing the process element with name investigate. This can be done with the RML rule

```xml
<div class="rule">  
  <div class="antecedent">  
    <process name="investigate" id="009" />  
  </div>
  <div class="consequence" />
</div>
```

rule.001.xml

If you have the input in file input.001.xml and the rule in file rule.001.xml then you can do this transformation on the commandline with the command

```
applyrule --input input.001.xml --rule rule.001.xml
```

The files for the examples, in this case input.001.xml and rule.001.xml, are in your examples directory.
How does RML apply this rule to the input? What happens is that the **applyrule** program looks in the **antecedent** of the rule and finds the **process** element there. This is a literal element, there are no special RML features in the element. The program then loads the input in its memory and searches the input for such an element. In our case it finds such an element and then replaces it by the contents of the **consequence** of the rule (in this case nothing). The program then outputs the modified input. If the program would not have found a matching element then the output would be the same as the input, but “pretty-printed” (see Fig. 3.5).

Output is printed to screen, there are no special features to produce files. The normal I/O redirection of the operating system can be used to produce files, for instance

```bash
applyrule --input input.xml --rule rule.xml > myoutput.xml
```

**Deleting an element with a specific attribute**

**Example 002**  Now suppose you want to remove a **process** element from **input.001.xml** and you do know it has a **name="investigate"** and an **id** attribute but you don’t know the value of the **id** attribute. This RML rule does what you want:

```xml
<div class="rule">
  <div class="antecedent">
    <process name="investigate" id="rml-X" />
  </div>
  <div class="consequence" />
</div>
```

`rule.002.xml`

This is an RML rule with a special RML feature in it: the "**rml-X**" attribute value of the **id** attribute. In RML, if an attribute value starts with **rml-** then it is considered an RML variable. The name of the variable is what comes after the **rml-**. If you use this rule, what happens is that the **applyrule** program does the same thing as in example 001, but instead of looking for an **id="009"** it now only looks for an **id** attribute and it puts the value of the attribute it finds into RML variable **X**. This variable **X** is not used anywhere else in the rule so for the rest of **applyrule**’s program execution it is just ignored.
3.5. EXAMPLES

But what if there are 2 process elements with name="investigate" and an id attribute? Then the first one in the input XML will be removed, the first being the first one encountered when reading the XML input file from top to bottom.

Example 002 can reuse the file input.001.xml so there is no need for a input.002.xml file, but there is a rule.002.xml in your examples directory.

Deleting an element with a specific attribute value

Example 003 The following rule removes a process element if it has an id attribute with value 009, ignoring all other attributes and values.

```xml
<div class="rule">
  <div class="antecedent">
    <process id="009" />
  </div>
  <div class="consequence" />
</div>
```

rule.003.xml

This rule works because RML does subset matching: if the element name and all the attributes of a pattern element match, then it is considered a match even if the matching input element has more attributes.

If `<process id="009" />` is replaced by `<process id="rml-X" />` then even the attribute value does not matter: the first process element with an id attribute is removed.

3.5.2 Changing an element

So far we did see only rules that delete elements. The emphasis was on how to match input elements, how to select elements that are to be deleted. In this Section we will see how element names and attributes can be changed into something else. This will also explain more about RML-variables.

Changing an element name

Example 004 Suppose you want to change the name of the `<process name="investigate" id="009" />` element to MyProcess. The rule in file rule.004.xml does just that.
<div class="rule">
  <div class="antecedent">
    <process name="investigate" id="009" />
  </div>
  <div class="consequence">
    <MyProcess name="investigate" id="009" />
  </div>
</div>

rule.004.xml

But this rule only works for a process with attributes name="investigate" and id="009".

Changing an element name and copying all attributes

**Example 005** What if you just want to change the element name like in Example 004 and copy all the other attributes. This is useful for instance if you don’t know all the attributes. Then you need the special RML attribute rml-others. It puts all attributes that do not appear elsewhere in the element into an RML variable. An RML variable is denoted by a leading rml-. Rule rule.005.xml shows the solution.

<div class="rule">
  <div class="antecedent">
    <process rml-others="rml-X" />
  </div>
  <div class="consequence">
    <MyProcess rml-others="rml-X" />
  </div>
</div>

rule.005.xml

Changing all process element names: Iterating a rule

**Example 006** But rule.005.xml only changes 1 element. If you want to change all process elements to MyProcess elements then you could just repeatedly use applyrule until there are no more process elements left. But you can also use the dorules program here. The dorules program has a
--rules parameter instead of a --rule parameter, it takes a set of rules as the value of that parameter, where rules are separated by a + character. In this example there is only one rule. Run dorules -i input.002.xml -r rule.005.xml for the desired effect.

Changing attribute values

Example 007  Example 006 did bind a set of attributes to an RML variable. We can also bind an attribute value to an RML variable. We did that already in Example 002, but here we will also use the value of the attribute in the output.

Rule rule.007.xml shows how to search for a process with name="investigate" and to change it to a process with name="SomethingElse". The id attribute with its value is copied to the output. This works even when you don’t know the value of the id attribute, the RML variable with name A is used for that.

```xml
<div class="rule">
  <div class="antecedent">
    <process name="investigate" id="rml-A" />
  </div>
  <div class="consequence">
    <process name="SomethingElse" id="rml-A" />
  </div>
</div>
```

3.5.3 RML variables for elements

Section 3.5.2 introduced RML variables. This Section will say more about RML variables. The antecedent of an RML rule contains XML from the problem domain XML vocabulary, mixed with other RML constructs. Together they form a matching pattern. When the rule is applied, this matching pattern is matched against the XML input. When a match occurs, the RML variables in the antecedent are filled with values. The type of these values can be:

- a string (element name, or attribute value),
- a set of attributes and their values,
- one XML element,
- a list of XML elements from the problem domain.

The last two are introduced next.

**RML variables for lists of elements and their children**

**Example 008** Duplicate all children of a `model` element.

```xml
<div class="rule">
  <div class="antecedent">
    <model>
      <rml-list name="A" />
    </model>
  </div>
  <div class="consequence">
    <model>
      <rml-use name="A" />
      <rml-use name="A" />
    </model>
  </div>
</div>
rule.008.xml
```

The `rml-list` RML element stores a list of elements at that position in the pattern into an RML variable. The RML variable can be output in the consequence of a rule with the `rml-use` RML element. All children elements of elements in the list will also be copied.

**RML variables for complete elements and their children**

**Example 009** Duplicate the first child of a `model` element.
The `rml-tree` element matches only 1 element (and all its possible children), the `rml-list` element after it in the antecedent of the rule matches the rest of the elements in that list.

### Defining RML variables for elements or lists of element with `rml-bind`

New in RML 1.6.

You can bind RML variables for elements (or for lists of elements) in the matching process, but you can also define them “manually” with `rml-bind`:

```xml
<model>
  <rml-tree name="A" />
  <rml-list name="B" />
</model>

<model>
  <rml-use name="A" />
  <rml-use name="A" />
  <rml-use name="B" />
</model>
```
This rule replaces all children of model with the MyNewElement element. Here <MyNewElement/> is bound to RML variable A. Instead of only <MyNewElement/> you can also put a list of elements there, and you can use RML variables to define the elements (for example RML variables for element names and attribute values). The rule.009a.xml is equivalent with a rule that has no rml-bind element and just has <MyNewElement/> in the consequent of the rule in place of the rml-use element there. So why bother with rml-bind? In Section 3.5.5 in example 011a we will see how manually binding with rml-bind can be useful.

3.5.4 RML variables for text content

The <rml-text name="SomeName" /> construct is used to bind XML text-content. It is used in the same way as the rml-tree element, but it will only match if there is XML text-content to be found in the matching position of the input. So you can not match an element and put it in the SomeName variable, if you want that, then you have to use rml-tree. You can use the SomeName RML variable just like any other RML variable; use it like rml-SomeName for an element name in the output or for an attribute name or attribute value, or use it like <rml-use name="SomeName" /> for text-content of an XML element.
3.5.5 Adding constraints with rml-if

Example 010  Transform

```
<model>
  <collaboration id="004" name="Negotiation">
    <role id="001" name="Intermediary"/>
    <role id="002" name="Customer"/>
  </collaboration>
  <process id="009" name="investigate"/>
  <role id="002" name="Customer"/>
  <process id="010" name="formalize request"/>
  <role id="001" name="Intermediary"/>
</model>

input.010.xml
```

into

```
<model>
  <collaboration id="004" name="Negotiation">
    <role id="001" name="Intermediary"/>
    <role id="002" name="Customer"/>
  </collaboration>
  <process id="009" name="investigate"/>
  <role id="002" name="Customer"/>
  <process id="010" name="formalize request" />
  <role id="001" name="Intermediary"/>
</model>
```

by executing this rule:
In the example the Intermediary is added as a child of model. The rule looks for a model element with a collaboration child that in turn has exactly 2 children elements, where the second child elements is also a child of the model element. If that is the case, the rule adds the first child to the children of model.

The rml-if elements are elements that constrain whether a match succeeds or not. The rml-if child="SomeVar" element only succeeds if the element bound to SomeVar appears somewhere in the current list of elements. SomeVar has to be bound earlier in the rule with an rml-tree name="SomeVar" element. There is also a rml-if nochild=X /> element that succeeds only if X does not appear in the current list. These 2 constraint adding elements are more complex than I would like, but I have found good
usage for them in practice.

**Example 011** If you repeat `rule.010.xml`, using the output as input, then you add Intermediary elements every time. To prevent that, rewrite the rule as:

```xml
<rule>
  <antecedent>
    <model>
      <collaboration rml-others="CollAttrs">
        <rml-tree name="A" />
        <rml-tree name="B" />
      </collaboration>
      <rml-list name="L1" />
      <rml-if child="B" />
      <rml-if nochild="A" />
      <rml-list name="L2" />
    </model>
  </antecedent>
  <consequence>
    <model>
      <collaboration rml-others="CollAttrs">
        <rml-use name="A" />
        <rml-use name="B" />
      </collaboration>
      <rml-use name="L1" />
      <rml-use name="L2" />
      <rml-use name="A" />
    </model>
  </consequence>
</rule>
```

`rule.011.xml`

The only difference is the line with `<rml-if nochild="A" />` in the antecedent of the rule. If I execute `..\..\dorules.py -i input.010.xml -r rule.010.xml` on my computer, then the program hangs until the system runs out of memory: It tries to add an infinite number of Intermediary elements and I have to use the Break key to stop it. But if I use `dorules`
with rule.011.xml then it stops with the desired effect. It stops because the second time it tries to apply the rule, the match fails because nochild fails because there is an Intermediary. The dorules program stops when it can not change the input anymore with any of its rules.

**Example 011a** In example 011 we were able to stop iteration by simple adding a `<rml-if nochild="A">` element. The value for the A variable was found earlier in the rule. But sometimes we can not do this, especially when the value that we want to bind to the variable is not present in the input. An example is when we want to create a completely new element and bind it to an RML variable such that we we can use the RML variable in `rml-if` constructs.

Suppose we want to transform input.011a.xml:

```
<model>
  <collaboration id="004" name="Negotiation">
    <role id="001" name="Intermediary"/>
    <role id="002" name="Customer"/>
  </collaboration>
</model>
```

input.011a.xml

into output.011a.xml:

```
<model>
  <collaboration id="004" name="Negotiation">
    <role id="001" name="Intermediary"/>
    <role id="002" name="Customer"/>
  </collaboration>
  <Intermediary id="001"/>
</model>
```

output.011a.xml

creating a new element with name Intermediary. Then we can use rule.011a.xml:
<div class="rule">
  <div class="antecedent">
    <model>
      <collaboration rml-others="CollAttrs">
        <rml-list name="PreRoles"/>
        <role id="rml-idA" name="rml-A"/>
        <rml-list name="PostRoles"/>
      </collaboration>
      <rml-list name="Tail"/>
    </model>
  </div>
  <div class="consequence">
    <model>
      <collaboration rml-others="CollAttrs">
        <rml-use name="PreRoles"/>
        <role id="rml-idA" name="rml-A"/>
        <rml-use name="PostRoles"/>
      </collaboration>
      <rml-use name="Tail"/>
      <rml-A id="rml-idA"/>
    </model>
  </div>
</div>

rule.011a.xml

But when we apply rule.011a.xml to the result (in output.011a.xml) again, it produces another Intermediary element. We want to stop that, and instead add a Customer element, like in output.011b.xml:

<model>
  <collaboration id="004" name="Negotiation">
    <role id="001" name="Intermediary"/>
    <role id="002" name="Customer"/>
  </collaboration>
  <Intermediary id="001"/>
  <Customer id="002"/>
</model>

output.011b.xml
To achieve this, use rule.011b.xml:

```xml
<div class="rule">
  <div class="antecedent">
    <model>
      <collaboration rml-others="CollAttrs">
        <rml-list name="PreRoles"/>
        <role id="rml-idA" name="rml-A" />
        <rml-list name="PostRoles"/>
      </collaboration>
      <rml-bind name="New">
        <rml-A id="rml-idA" />
      </rml-bind>
      <rml-if nochild="New"/>
      <rml-list name="Tail"/>
    </model>
  </div>
  <div class="consequence">
    <model>
      <collaboration rml-others="CollAttrs">
        <rml-list name="PreRoles"/>
        <role id="rml-idA" name="rml-A" />
        <rml-use name="PostRoles"/>
      </collaboration>
      <rml-use name="Tail"/>
      <rml-A id="rml-idA" />
    </model>
  </div>
</div>
```

When we apply rule.011b.xml iteratively until the output is stable (with the dorules tool: "dorules.py -i input.011a.xml -r rule.011b.xml") we get the desired output.011b.xml. The rule works by binding the desired new element to an RML variable with the name `New`, and using this `New` variable in the `<rml-if nochild="New"/>` test; preventing a match if the new element is already there.
Example 012  Suppose you want to output the second role in the first collaboration in input.010.xml, producing

\[
<\text{role id="002" name="Customer"} />
\]

This rule:

\[
<\text{div class="rule">}
<\text{div class="antecedent">}
<\text{model}>
<\text{collaboration}>
<\text{role}>}
<\text{rml-tree name="A"} />
</collaboration>
</model>
</div>

<\text{div class="consequence">}
<\text{rml-use name="A"} />
</div>
</div>
\]

rule.012a.xml

produces the desired output. And when applying that rule to

\[
<\text{model}>
<\text{collaboration id="004" name="Negotiation">}
<\text{role id="001" name="Intermediary"} />
<\text{role id="002" name="Customer"} />
<\text{role id="003" name="Office"} />
</collaboration>
<\text{process id="009" name="investigate"} />
<\text{role id="002" name="Customer"} />
<\text{process id="010" name="formalize request"} />
</model>
\]

input.012.xml

it also outputs the Customer. But what if you want the rule only to work if there are exactly 2 child elements in the collaboration? Then use rule:
The `<rml-if last="true"/>` element makes the matching of the rule only succeed if the previous element is the last in a list. If you apply `rule.012.xml` to `input.012.xml` then you get the contents of the input back: no match. But `rule.012.xml` does work on `input.010.xml`.

### 3.5.6 Match choice with `rml-type="or"`

**Example 013** If you want to match

```xml
<model>
  <role id="001" name="Intermediary"/>
</model>
```

`input.013a.xml`

or

```xml
<model>
  <role id="002" name="Customer"/>
</model>
```

`input.013b.xml`
but not

```xml
<model>
  <role id="003" name="Insurance Company"/>
</model>
```

input.013c.xml

then you can use rule rule.13.xml:

```xml
<div class="rule">
  <div class="antecedent">
    <model>
      <role id="001" name="Intermediary" rml-type="or" />
      <role id="002" name="Customer" rml-type="or" />
    </model>
  </div>
  <div class="consequence">
    <matched how="allright" />
  </div>
</div>
```

rule.13.xml

With the special attribute `rml-type="or"`, the RML tools try the next element if a match fails on an element, but only if that next element also has this special attribute.

### 3.5.7 How to remove duplicate siblings

**Example 014**  Some of the inspiration leading to RML came when I had to transform something like
<model>
    <role id="003" name="Insurance Company"/>
    <whatever />
    <role id="003" name="Insurance Company"/>
    <evenmore />
</model>

input.014.xml
to

<model>
    <role id="003" name="Insurance Company"/>
    <whatever />
    <evenmore />
</model>

, just removing duplicate siblings. This looks simple but I found out it was very hard to do with for example XSLT. The RML rule for this is not difficult:

<div class="rule">
    <div class="antecedent">
        <model>
            <rml-list name="list1" />
            <role rml-others="A" />
            <rml-list name="list2" />
            <role rml-others="A" />
            <rml-list name="list3" />
        </model>
    </div>
    <div class="consequence">
        <model>
            <rml-list name="list1" />
            <role rml-others="A" />
            <rml-list name="list2" />
            <role rml-others="A" />
            <rml-list name="list3" />
        </model>
    </div>
</div>

rule.014.xml
3.5. EXAMPLES

This rule also shows a typical usage of rml-list elements: all the elements, and their children, around the elements you are interested in, are remembered in variables (here list1 and list2 and list3. With this pattern you can “preserve the context” of the elements you want to match. Example 016 also shows this pattern.

Example 015  And for Example 014 even this rule works:

```xml
<div class="rule">
  <div class="antecedent">
    <model>
      <rml-list name="list1" />
      <rml-tree name="A" />
      <rml-list name="list2" />
      <rml-use name="A" />
      <rml-list name="list3" />
    </model>
  </div>

  <div class="consequence">
    <model>
      <rml-use name="list1" />
      <rml-use name="A" />
      <rml-use name="list2" />
      <rml-use name="list3" />
    </model>
  </div>
</div>
```

rule.015.xml

, not only for duplicate role elements, but for any duplicate elements in the model. This rule makes use of the fact that variable A has been bound in the line with `<rml-tree name="A" />`, and then a `<rml-use name="A" />` is allowed not only in the consequence, but also in the antecedent of a rule.

3.5.8 Iterating sets of rules

The dorules tool accepts a list of rulefilenames in parameter --rules (or -r), instead of just one rule like the applyrule tool.
It then applies the first rule of the list to the input, just like the `applyrule` program, and if the rule can be successfully applied (the output is different from the input) then the program restarts, taking the generated output and the list of rules as input. If a rule does not match the input then the next rule in the list is tried. If no rule in the list matches the input then the program stops.

Iterating sets of rules is often useful. A typical usage pattern is when you want to create new elements with data from 2 original elements, but you don’t know the order of the 2. Then you write 2 rules and let `dorules` handle it.

**Example 016  Transform**

```xml
<model>
  <triggering id="016" name="triggers">
    <from href="009"/>
    <to href="010"/>
  </triggering>
  <triggering id="015" name="triggers">
    <from href="008"/>
    <to href="009"/>
  </triggering>
</model>
```

`input.016.xml`

into

```xml
<model>
  <triggering id="016" name="triggers">
    <from href="008"/>
    <to href="010"/>
  </triggering>
</model>
```

`input.016.xml`

by executing `dorules` with this rule:
and this rule:
issuing the command:

dorules -i input.016.xml -r rule.016a.xml+rule.016b.xml

Note that the order of the rules in the list is significant. The order is not important in this example but in another it could be. This example is also another example of “context preserving” with rml-list elements, storing the context in Prelude, Between and Epilog.
3.5.9 Turning a list into a hierarchy

Example 017 Transform

```xml
<input.017.xml

<model>
  <role id="001" name="Intermediary"/>
  <collaboration id="004" name="Negotiation"/>
  <process id="009" name="investigate"/>
  <process id="010" name="formalize request"/>
</model>

input.017.xml

into

<model>
  <role id="001" name="Intermediary">
    <collaboration>
      <process id="009" name="investigate">
        <process id="010" name="formalize request"/>
      </process>
    </collaboration>
  </role>
</model>

by executing dorules with this rule:
rule.017.xml

This rule "deepens" all lists in the input when applied iteratively with dorules.

3.5.10 Pre-binding string variables on the command-line

Example 018  If you have this input:

```xml
<model>
  <event name="request for insurance" id="008" />
  <process name="investigate" id="009" />
  <process name="investigate" id="010" />
</model>
```

available in file input.018.xml

and you would like to remove the process with id="010", then you need something very similar to the rule.002.xml you wrote earlier. But that rule
removes the id="009" process, because that is the first that matches the antecedent of the rule (<process name="investigate" id="rml-X" />).

A solution is to bind commandline variables to RML variables. If you issue the command:

```
applyrule -i input.018.xm -r rule.002.xml X=010
```

then the id="010" will be removed.

What happens is that the string 010 is bound to RML variable X, meaning that the RML tools now treat "rml-X" in a rule as "010".

### 3.5.11 Using recipes

RML recipes are stated in Recipe RML (RRML), an XML vocabulary for RML recipes. With RML recipes the user can define sequences of `applyrule` and `dorules` executions. See Sect. 3.2.6 and Fig. 3.4, there is an example recipe. There are plans for a future version of RRML with `rule` elements that can execute XSLT transformations too. Or that can execute arbitrary programs...let me know what you would like.

I hope you enjoyed this tutorial. Good luck with your XML transformations!