Domain specific modeling and analysis

Jacob, J.F.

Chapter 2

RML and its application to UML

Author: Joost Jacob

2.1 Introduction

The work in this paper was initiated and motivated by work in the IST project OMEGA (IST-2001-33522, [OME]) sponsored by the European Commission. The main goal of OMEGA is the correct development of real-time embedded systems in the Unified Modeling Language [SWB03]. This goal involves the integration of formal methods based on model-checking techniques [BDJ+03] and deductive verification using PVS [ORR+96].

The eXtensible Markup Language XML (XML [XML]) is used to encode the static structure of UML models in OMEGA. The XML encoding is generated by Computer Aided Software Engineering (CASE) tools; it captures classes, interfaces, associations, state machines, and other software engineering concepts. The OMEGA tools for model-checking and deductive verification are based on a particular implementation of the semantics of the UML models in a tool-specific format ([BGM01], [DJPV03], [ORR+96]). This complicates interoperability of such tools. In order to ensure that these different implementations are consistent, a formal semantics of UML models is developed in OMEGA in the mathematical formalism of transition systems

[Plo81]. However, it still requires considerable effort to ensure that these different implementations are indeed compatible with the abstract mathematical semantics. Some of the motivation for RML came in helping with this effort. Since the models produced by the CASE tools are encoded in XML it was a natural choice to look for an XML transformation technique instead of encoding a model and semantics in a special-purpose format. Simulating and analyzing *in* XML adds the interoperability benefit of XML and the many available XML tools can be used on the results.

In this paper a general-purpose method for XML transformations is introduced and its application to the specification and execution of UML models. The underlying idea of this method is to specify XML transformations by means of rules which are formulated in a problem domain XML vocabulary of choice: the rules consist of a mix of XML from the problem domain and the Rule Markup Language (RML, Sect. 2.3). The input and output of a transformation are pure problem domain XML; RML is only used to help to define transformation rules. The RML approach re-uses the problem domain XML as much as possible, with a "programming by example" technique. With this rule–based approach it becomes possible to define transformations that are very hard to do when using for example XSLT [XSL], the official W3C [W3C] Recommendation for XML transformations, as discussed in Section 2.2.1.

The RML tools are available as platform-independent command–line tools so they can easily be used together with other tools that have XML as input and output.

RML is not trying to solve *harder* or *bigger* transformations than other approaches. Instead of concentrating on speed or power, RML is designed to be something that is very *usable* and interoperable. Experience in several projects has shown that programmers can learn to use RML in only a few hours with the tutorial in Chapter 3, and even non-programmers put RML to good use. With respect to the RML application to UML models, only knowledge of XML and RML suffices to be able to define and execute their semantics.

As such, RML provides a promising basis for the further development of XML-based debugging and analysis tools for UML models.

XML itself is not intended for human consumption, but we have developed the ASCII Markup Language (AML) representation that helps considerably in this respect. The example model in this paper is presented in AML because AML is more readable than XML, but otherwise equivalent for this purpose.

More details about AML and an AML to XML translation, and back, are available at [Jaca].

Plan of the paper The next section starts with describing XML. Section 2.3 presents RML as a new approach to solve XML transformation problems and describes how to use RML for defining transformation rules. Section 2.4 shows examples of applications of RML, the main example being an application that results in executable UML models. The conclusion and a discussion of related work is in Sect. 2.5.

2.2 XML and XML transformations

With XML, data can be annotated and structured hierarchically. There are several ways to do this and there is no single best way under all circumstances: designing good XML vocabularies is still an art. For instance, suppose you want to describe a family in XML: a grandmother named Beth, a father named John, a mother name Lucy and son named Bill. One way to do this is:

```
<family>
    <grandma name="Beth" />
    <father name="John" />
    <mother name="Lucy" />
    <son name="Bill" />
</family>
```

The example shows five different XML elements: `family`, `grandma`, `father`, `mother`, and `son`. The XML hierarchy is a tree, with nodes called XML elements, and there has to be one and only one XML element that is the root of the tree, in the example the `family` element. An XML element consists of its name, optional attributes and an ordered list of subelements, where a subelement can also be a string. Attributes of XML elements are mappings from keys to values, where the keys are text strings and the values are text strings too.

A string enclosed with angle brackets is called a tag. A minimum tag only contains the element name, like the `<family>` in the example. The element name is not the only thing that can appear between the angle brackets, there can also be attributes like `name="John"` in the example. Attributes consist of the attribute name, an `=` and the attribute value (a text string) enclosed in double quotes.

An XML element that does *not* contain other elements, a so called empty element, has its tag closed by an `/`, as in `<X />`, where `X` is the element name. An XML element that has children consists of two tags: one for the element name (and its attributes), and one for closing the element after its children. In the example the `family` element is the only element with children. There are several rules that define if XML is *well formed*, for instance every opening tag `<X>` has to be closed by a closing tag `</X>`, and these rules can be checked by tools.

But the XML in the example does not reflect the tree like structure of the family. Another way is:

```
<family>
    <female>
        <name>Beth</name>
        <male marriedTo="Lucy">
            <name>John</name>
            <male>
                <name>Bill</name>
            </male>
        </male>
    </female>
    <female marriedTo="John">
        <name>Lucy</name>
    </female>
</family>
```

Here `Beth` is not the value of an attribute but it is the text content of a `name` element. The structure of this example may better indicate that Beth is the mother of John, but the XML is more verbose than the first example.

An XML vocabulary can be formally defined in a DTD (see the XML Specification in [XML]) or an XML Schema [XMS], both W3C Recommendations. There is also an ISO standard for defining vocabularies called RelaxNG [Cla01]. The definition can express that for instance every `female` in the example must have a `name` child and can have optional `female` or `male` childs. With such a definition, called schema, there are XML tools available that can *validate* if XML is conforming to a schema. Note that validating is different from checking well-formedness. It is possible to refer to the definition of the vocabulary used from inside XML, and there are many more XML concepts that can not be discussed here due to lack of space, for which I refer to the XML Specification [XML].

A schema only defines syntax, the *meaning* of the XML constructs defined has to be defined somewhere else. The W3C is working on standard ways for doing this (viz. the Semantic Web project) but currently this is usually

done in plain text documents. The family from the example has a tree like structure so in this respect it is an easy example to describe in XML that also has a tree structure. But a tree is not the only kind of structure that can be modeled conceptually with a schema. Other structures can also be modeled in XML because the XML elements can refer to each other by means of cross-references with identifiers, as in the second example with the `marriedTo` attributes.

A lot of XML vocabularies have been designed in recent years, for all kinds of problem domains. Often these vocabularies are W3C Recommendations, like RDF, XSLT and MathML [Mat03]; another example is XMI [XMI] as developed for UML [SWB03] by the Object Management Group [OMG]. The OMEGA project works with XMI and other XML vocabularies for software.

In general, when having a structure stated in XML data, a dynamics of the structure can be captured by rules for transforming the XML data. The rules that define XML transformations can be stated in XML itself too, but the problem domain XML vocabulary will usually not be rich enough to be able to state a rule. For this it has to be combined with XML that is suitable for expressing transformation rules, containing for example constructs to point out what to replace with what, and where. Section 2.2.1 shows how an XML vocabulary that is different from the problem domain vocabulary can be used, which is the current way of doing transformations in industry, and Sect. 2.3 shows the new RML approach that is based on extension.

2.2.1 XSLT

Extensible Stylesheet Language Transformations (XSLT, [XSL]) is a W3C recommendation for XML transformations. It is designed primarily for the kinds of transformations that have to do with visual presentation of XML data, hence the *style* element in the name. A popular use of XSLT is to transform a dull XHTML page to a colorful and stylized one. Or to generate visualizations from XML data. However, nowadays XSLT is being used more and more for general purpose XML transformations, from XML to XML, but also from XML to text. In the OMEGA project we have used XSLT to do transformations of software models (UML models stated in XMI [XMI]) resulting in models encoded in other XML vocabularies and also resulting in textual syntax like PVS [ORR+96]. The static structure of the models was transformed. But when we wanted to capture the semantics of execution of these software models we found that XSLT is not very usable for the partic-

ular kind of transformations that describe dynamics. These transformations use a match pattern that is distributed over several parts of an XML tree, whereas the matching technique used in XSLT is designed to match in a linear way, from root to target node in a tree. This linear matching is not suitable for matching of a pattern with several branches.

For instance, matching duplicate children of an element is very hard with XSLT. The MathML expression

```
<math>
    <apply>
        <and />
        <ci>p</ci>
        <ci>p</ci>
        <ci>q</ci>
    </apply>
</math>
```

meaning $p \wedge p \wedge q$ in propositional logic, is logically equivalent to

```
<math>
    <apply>
        <and />
        <ci>p</ci>
        <ci>q</ci>
    </apply>
</math>
```

meaning $p \wedge q$. In the MathML the `<ci>` element is used for pointing out constant identifiers and the `apply` element is used for building up mathematical expressions. Suppose we would like to transform all $p \wedge p \wedge q$ into $p \wedge q$, where p and q can be anything but two p's in an expression are equal. To perform such a transformation a tool has to look for a pattern with two identical children and then remove one of the children. Since XSLT is a Turing–complete functional programming language, it *is* possible to do this transformation, but XSLT templates for these kinds of transformations are extremely long and complex. XSLT simply was not designed for these kinds of transformations; the designers did not feel much need for them in the webwide world of HTML and webpublishing. The MathML+RML rule for removing duplicate children is simple, it is one of the examples in the RML tutorial in Chapter 3.

2.3 RML

This section first introduces the idea of *XML wildcard elements*. After that the RML syntax is introduced in Sect. 2.3.2, before Section 2.3.3 describes the RML tools.

2.3.1 XML Wildcard Elements

The transformation problem as shown in Sect. 2.2.1 reminds one of the use of wildcards for solving similar problems in text string matching. The idea of using an XML version of wildcards is a core idea of our method and of this paper. The idea is to define XML notation for an XML version of constructs like the `*` and `?` and `+` and others in text–based wildcards as in Perl regular expressions, and then using these constructs for matching and variable binding. These constructs are called *XML wildcard elements*. They consist of complete XML elements and attributes as can be formally defined with a schema, but they also consist of *extensions* for denoting wildcard variables *inside* a problem domain XML. These variables are just like the variables as they are used in the various languages available for text–based wildcard matching, for instance Perl regular expressions. The variables have a name, and they are given a value when a match succeeds. This value can then be used in the output of a transformation rule.

2.3.2 The RML syntax

RML rules are stated in XML. The basis of a rule is that in the antecedent of the rule the input is matched, and then whatever matched is replaced by the consequent of the rule.

RML was designed to be *mixed* with any problem domain XML, to be able to define transformations while re-using the problem domain XML as much as possible. RML is a mixture of XML elements, conventions for XML-attribute names, and conventions for attribute values, to mix in with XML from the problem domain vocabulary at hand. RML introduces some new XML elements and uses an element from XHTML. From XHTML only the `div` element is used and it is used to distinguish a rule, the antecedent of a rule, and the consequence of a rule by means of the `class` attribute of the `div` tag. We use the `div` tag from XHTML for reasons that have to do with a presentation in browsers.

The Table in Fig. 2.1 lists all the current RML constructs with a short explanation of their usage in the last column. These have been found to be sufficient for all transformations encountered so far in practice in the projects where RML is being used. An `X` in the XML tags can be replaced by a string of choice. The *position* that is sometimes mentioned in the explanations is the position in the sequential list that results from a root-left-right tree traversal

of the XML tree for the rule. It corresponds with how people in the western world reading an XML document encounter elements: top-down and left to right. A position in the rule tree corresponds with zero or more positions in the input tree, just like the `*` in the wildcard expression `a*b` corresponds with `c` on input `acb` and with `cd` on input `acdb` and with nothing on input `ab`. With the constructs in the Table in Fig. 2.1 the user can define variables for element names, attribute names, attribute values, whole elements (with their children) and lists of elements. An XML+RML version of the `a*b` wildcard pattern is

```
<a /> <rml-list name="Star" /> <b />
```

and this can be used as part of the antecedent of an RML rule that uses the contents of the `Star` variable in the consequent of the rule. The `a` and `b` elements are from some XML vocabulary, the `rml-list` element is from RML and described in the Table in Fig. 2.1. Section 2.4.1 shows a small but complete RML rule. Examples of input and rules take up much space and although we would have preferred to present more rule examples here now, there is simply not enough space to do that and we strongly invite the reader to look at the examples in the RML tutorial in Chapter 3 where there are examples for element name renaming, element replacing, removing duplicates, copying, attribute copying, adding hierarchy and many more.

It is easy to think of more useful elements for RML than in the Table, but not everything imaginable is implemented because a design goal of RML is to keep it as simple and elegant as possible. Only constructs that have proven themselves useful in practice are added in the current version.

The execution of a rule consists of binding variables in the matching process, and then using these variables to produce the output. Variable binding in RML happens in the order of a root-left-right traversal of the input XML tree. If an input XML tree contains more than one match for a variable then only the first match is used for a transformation. The part of the input that matches the rule antecedent is *replaced* by the consequent of the rule. If a rule does not match then the unchanged input is returned as output. If a rule matches input in more than one place and you want to transform all matches then you will have to repeat applying the rule on the input until the output is stable. There is a special RML tool called `dorules` for this purpose.

2.3.3 The RML tools and libraries

Open-source tools and libraries can be downloaded and also the RML tutorial in Chapter 3 is available online. The tutorial contains information about installing and running the tool, and there is also more technical information on the website, for instance about a matching algorithm. The tools and libraries have been successfully used under Windows, Linux, Solaris, and Apple.

A typical usage pattern of the `applyrule` command–line tool is `$ python applyrule.py --rule myrule.xml --input myinput.xml` that will print the result to console, or it can be redirected to a file.

The rule and the input are parameterized, not only as command–line parameters but also in the internal `applyrule` function; this makes the tool program also usable as a library and thus suitable for example for programming a simulation engine. There is an interface to additional hook functions so tools can be extended, for instance for programming new kinds of constraints on the matching. However, the set of RML constructs in the current version has proven to be sufficient for various XML transformation work, so adding functions via the hooks will normally not be necessary. An example of when it *is* desirable to add functions is when a tool designer for example wants to add functionality that does calculations on floating point values in the XML. The RML tools are written in Python [vR95] and the Python runtime can use the fast `rxp` parser written in, and compiled from, C, so the XML parsing, often the performance bottleneck in such tools, is as fast and efficient as anything in the industry. If the rxp module is not installed with your Python version then RML automatically uses another XML parser on your system.

The RML tutorial in Chapter 3 also describes a very simple XML vocabulary for defining RML recipes, called Recipe RML (RRML), and a tool called `dorecipe` for executing recipe–based transformations. RRML is used to define sequences of transformations and has proven itself useful in alleviating the need for writing shell scripts, also called batch files, containing sequences of calls to the RML tools.

2.4 RML examples

The main example presented in this paper is the executable specification of the semantics of UML models in XML and this is the topic of Section 2.4.1. Section 2.4.2 briefly mentions other projects wherein RML is applied.

2.4.1 Executable UML models

The application of RML to the semantics of UML models and its resulting execution platform is based on the separation of concerns betweeen coordination/communication and computation. This exploits the distinction in UML between so-called triggered and primitive operations. The behavior of classes is specified in UML statemachines with states and transitions, and every transition can have a trigger, guard, and action. A transition does not need to have all three, it may for example have only an action or no trigger or no guard. Triggered operations are associated with events: if an object receives an event that is a trigger for a transition, and the object is in the right location for the transition, and the guard for that transition evaluates to True, then the action that is specified in the transition is executed. The triggered operations can be synchronous (the caller blocks until an answer is returned) or asynchronous. Events can be stored in event queues, and the queues can be implemented in several ways (FIFO, LIFO, random choice, ...). There are also primitive operations: they correspond to statements in a programming language, without event association or interaction with an event mechanism. The primitive operations are concerned with computations, i.e. data-transformations, the triggered operations instead are primarily used for coordination and communication. More details can be found in [DJPV03].

This distinction between triggered and primitive operations and the corresponding separation of concerns between coordination/communicaion and computation is reflected in the RML specification and execution of UML models which delegates (or defers) the specification of the semantics and the execution of primitive operations to the underlying programming language of choice. This delegation is not trivial, because the result of primitive operations has to be reflected in the values of the object attributes in the XML, but the details of the delegation mechanism can not be given here due to a lack of space.

In our example the problem domain is UML and we will use a new XML

vocabulary that is designed for readability and elegance. This language is called *km*, for kernel model; a RelaxNG [Cla01] schema is at http://-homepages.cwi.nl/~jacob/km/km.rnc.

The online example is a prime sieve, it was chosen because it shows all the different kinds of transitions and it has dynamic object creation. It generates objects of class `Sieve` with an attribute `p` that will contain a prime number. But the user can edit the example online or replace it with his or her own example, if the implementation language for actions and guards is the Python programming language. A similar application can be written for the Java language, and UML models from CASE tools can be translated automatically to the km language. The example can be executed online in an interactive webapplication on the internet at [Jacd]. In the km application the user fills in a form with an object identity and a transition identity, and pressing a button sends the form to the webapplication that performs the corresponding transition. Instead of a user filling in a form, a program can be written that calls the website and fills in the form, thus automating the tool. We did so, but for this paper we consider a discussion of the automated version out of scope.

The km language defines XML for class diagrams and object diagrams. The classes consist of attribute names and a statemachine definition. The statemachines have states and transitions, where the transitions have a guard, trigger and action like usual in UML. The objects in the object diagram have attributes with values and an event queue that will store events sent to the object. An example of an object is

```
objectdiagram
    obj class=Sieve id=2 location=start target=None
        attr name=p value=None
        attr name=z value=None
        attr name=itsSieve value=None
        queue
            op name=e
                param value=2
```

where the object is of type `Sieve`, finds itself in the `start` state of the statemachine of the `Sieve` class, and has an eventqueue with one event in it with name `e` and event parameter `2`.

A detailed description of the km language and its design would take too much space here, but the interested reader who knows UML will have no trouble recognizing the UML constructs in the models since the km language was designed for readability.

In the km language the event semantics is modelled, but the so-called *primitive* operations that change attribute values are deferred to a programming language. So the models will have event queues associated with objects and executing a model will for example show events being added to queues, but operations that are not involved with events but only perform calculations are stored in the model as strings from the programming language of choice. Such an operation can be seen in the example as

```
transition id=t3
    source state=state_3
    target state=state_1
    action
        implementation
            """x = x + 1"""
```

where we see a transition in the statemachine with an action, the statement executed by the programming language (Python in this case) is `x = x + 1`. Transitions can also have a guard with an expression in a programming language, also encoded as text content of an `implementation` element.

We can now show a simple example RML rule.

```
<div class="rule" name="set location">
  <div class="antecedent">
    <obj id="rml-IDOBJ" location="rml-L" target="rml-T" rml-others="rml-O" >
      <rml-list name="ObjChildren"/>
    </obj>
  </div>
  <div class="consequence">
    <obj id="rml-IDOBJ" location="rml-T" target="None" rml-others="rml-O">
      <rml-use name="ObjChildren"/>
    </obj>
  </div>
</div>
```

This is a rule that is used after a transition has been taken successfully by an object modeled with km. With this rule the `location` attribute of the object is assigned the value of the `target` attribute and the `target` attribute is set to `None`. An example of the effect of the rule would be that

```
<obj id="id538" location="state_3" target="state_5" ... >
  <queue>
    ...
  </queue>
</obj>
```

is changed into

```
<obj id="id538" location="state_5" target="None" ... >
  <queue>
    ...
  </queue>
</obj>
```

for an object with identifier `id538`.

When applying this rule, the RML transformation tool first searches for an `obj` element in the input, corresponding with the `obj` element in the `antecedent` of the rule. These `obj` elements match if the `obj` in the input has an `id` attribute with the value bound to the RML `IDOBJ` variable mentioned in the antecedent, in the example this value is `id538` and it is bound to the RML variable `IDOBJ` *before* the rule is applied. This pre-binding of some of the variables is how the tool can manage and schedule the execution of the RML transformation rules. The `IDOBJ` is a value the user of the online webapplication supplies in the form there. If the `obj` elements match, then the other RML variables (`L`, `T`, `O` and `ObjChildren`) are filled with variables from the input `obj`. The `L`, `T` and `O` variables are bound to strings, the `ObjChildren` variable is bound to the children of the `obj` element: a list of elements and all their children. The `consequence` of the rule creates a new `obj` element, using the values bound to the RML variables, and *replaces* the `obj` element in the input with this new `obj` element.

Due to lack of space we restrict the description of the formalization in RML to the rule for the removal of an event from the event-queue, the antecedent is shown in AML notation:

```
km
    classdiagram
        ...
        class name=rml-ClassName
            statemachine
                transition id=rml-IDTRANS
                    trigger
                        op name=rml-TriggerName
                            rml-list name=Params
                ...
    objectdiagram
        ...
        obj class=rml-ClassName id=rml-IDOBJ
                        rml-others=rml-OtherObjAttrs
            queue
                rml-list name=PreEvents
                op name=rml-TriggerName
                rml-list name=PostEvents
```

and this contains some lines with `...` in places where `rml-list` and `rml-use` constructs are used to preserve input context in the output. Here we see that in RML a pattern can be matched that is distributed over remote parts in the XML, the remoteness of the parts is why the rule has so many lines. In short, this rule looks for the name of the trigger that indicates the event that has to be removed from the event-queue, and then simply copies the event-queue

without that event. But to find that name of the trigger, a search through the whole km XML model has to take place, involving the following steps.

During application of this rule, the matching algorithm first tries to match the input with the antecedent of the rule, where IDOBJ and IDTRANS are pre-bound RML variables, input to the tool. With these pre-bound variables it can find the correct `obj`, then it finds the `ClassName` for that object. With the `ClassName` the `class` of the object can be found in the `classdiagram` in km XML. When the class of the object is found, the transition in that class with id `TRANSID` can be found and in that `transition` element in the input we can finally find the desired `TriggerName`. The algorithm then looks for an `op` (operation) event with name `TriggerName` in the event-queue of the `obj`, and binds all other events in the event-queue to RML variables `PreEvents` and `PostEvents`. In the consequence of the rule then, all these bound RML variables are available to produce a copy of the input, with the exception that the correct event is removed. As given, the rule removes the first event that matches. It is trivial to change the rule to one that removes only the first event in a queue (by removing the `PreEvents`), or only the last. This is an example that shows that the semantics defined in the RML rules can be easily adapted, even *during* a simulation, and this makes such rules particulary suitable for experimental analysis.

The km application gives comments, for example about the result of the evaluation of a guard of a transition. If the user for instance selects a transition identity that does not correspond with the current state of the object, in the online example if you select (ObjID,TransitionID)=(1,t1) twice, a message is displayed on top of the model, like

```
#   Exiting: Wrong location (object:state_1 transition:start)
```

meaning that transition `t1` can not be taken because the object is in state `state_1` and the transition is defined for a `source` state with name `start`. Such messages do not interfere with the model itself, they are encoded as comments, and the model is unchanged after this message.

The only software a user needs to use the interactive application is a standards compliant browser like Mozilla or Internet Explorer. A user can not only go forward executing a model, but also go *backward* with browser's Back button. This is an example of the benefit of interoperability that XML offers, together with a software architecture and design that is platform independent.

2.4.2 Other examples

If a formalism is expressed in mathematics, then MathML is a generally usable way to express structure, and RML rules extending MathML can capture the dynamics. As an example of this, RML is used for an online interactive theorem prover that can be used to derive proofs for tautologies in propositional logic[1].

Although defining models and semantics in MathML will appeal to the mathematically educated, sometimes it is better to define a new special-purpose XML vocabulary; to make it more concise, better readable, more efficient, and for several other reasons. This was the case in the Archimate [Arc] project where RML has been applied successfully to Enterprise Architecture and Business Models. Rule–based transformations are being used for analysis of models and for visualizations. The RML tutorial in Chapter 3 and the downloadable RML package contain examples in the Archimate language.

2.5 Related work and conclusion

Standards related to RML are XML [XML], MathML [Mat03] and XSLT [XSL]. MathML is a W3C specification for describing mathematics in XML, and it is the problem domain language for the proof example in this paper. XSLT is a W3C language for transforming XML documents into other XML documents, and is discussed in Section 2.2.1. There are also standards that are indirectly connected with XML transformations, like XQuery that can treat XML as a database that can be queried, but a discussion of the many XML standards here is out of scope.

The RuleML community [RUL] is working on a standard for rule-based XML transformations. Their approach differs from the RML approach: RML re-uses the problem domain XML, extended with only a few constructs (in the table in Fig. 2.1) to define rules; whereas RuleML superimposes a special XML vocabulary for rules. This makes the RuleML approach complex and thus difficult to use in certain cases. The idea of using wildcard elements for XML has not been incorporated as such in the RuleML approach, but perhaps it can be added to RuleML and working together with the RuleML community in the future can be interesting.

[1] http://homepages.cwi.nl/~jacob/MathMLcalc/MathMLcalc.html

There are a number of tools, many of them commercial, that can parse XML and store data in tables like those in a relational database. The user has to define rules for extracting the data, to define what is in the columns and the rows of the tables, to define an entity-relationship model, and other things. Once the data the user is interested in is in the database, a standard query language like SQL can be used to extract data. And then that data can be used to construct new XML. The XML application called XQuery can be used in a similar way, and it is the approach taken by ATL [BDJ+03]. It would be possible to do any transformation with these techniques, but it would be very complex.

The Relational Meta-Language [Pet94] is a language that is also called RML, but intended for compiler generation, which is much more roundabout and certainly not usable for rapid application development like with RML in this paper.

An example of another recent approach is fxt [BS02], which, like RML, defines an XML syntax for transformation rules. Important drawbacks of fxt are that it is rather limited in its default possibilities and relies on hooks to the SML programming language for more elaborate transformations. For using SML a user has to be proficient in using a functional programming language. An important disadvantage of a language like SML is that it is not a mainstream programming language like Python with hundreds of thousands or users worldwide, which makes it unattractive to invest in tools based on SML. The fxt tools are available online but installing them turned out to be problematic.

The experience with several tools as mentioned above leads to the concept of *usability* of a tool in general. Here, a tool is not considered usable enough if it is too difficult to install and configure it and get it to run, or if the most widely used operating system Windows is not supported, or if working with the tool requires a too steep or too high learning curve, for example because the user has to learn a whole new programming language that is not a mainstream programming language. Although the fxt article [BS02] interestingly mentions "XML transformation ... for non-programmers", fxt is unfortunately an example of an approach that is not usable enough according to this usability definition.

XML is still gaining momentum and becoming more important and as a result there are many more tools from academic research available, rather too much to mention here as an internet search for "XML tool" reveals hundreds of search results. Unfortunately none of them turned out to be useful in

practice for our work according to the above definition of usability, after spending considerable time trying them out.

Other popular academic research topics that could potentially be useful for rule–based XML transformations are term–rewriting systems and systems based on graph grammars for graph reduction. However, the tested available tools for these systems suffer from the same kind of problems as mentioned above: the tools are generally not portable and most will never be portable for technical reasons, and using these tools for XML transformations is an overly complex way of doing things. To use these kind of systems, there has to be first a translation from the problem XML to the special-purpose data structure of the system. And only then, in the tool–specific format, the semantics is defined. But the techniques used in these systems are interesting, especially for very complex or hard transformations, and it looks worthwhile to see how essential concepts of these techniques can be incorporated in RML in the future.

Compared with the related work mentioned above, a distinguishing feature of the RML approach is that RML *re-uses* the language of the problem itself for matching patterns and generating output. This leads in a natural way to a much more usable and clearly defined set of rule–based transformation definitions, and an accompanying set of tools that is being used successfully in practice.

Elements that designate rules				
`div`	`class="rule"`			
`div`	`class="antecedent" context="yes"`			
`div`	`class="consequence"`			
element	attribute	A	C	meaning
Elements that match elements or lists of elements				
`rml-tree`	`name="X"`	*		Bind 1 element at this position to RML variable X.
`rml-text`	`name="X"`	*		Bind XML text-content to variable X.
`rml-list`	`name="X"`	*		Bind a sequence of elements to X.
`rml-use`	`name="X"`		*	Output the contents of the RML variable X at this position.
Matching element names or attribute values				
`rml-X`	...	*		Bind element name to RML variable X.
`rml-X`	...		*	Use variable X as element name.
...	`...="rml-X"`	*		Bind attribute value to X.
...	`...="rml-X"`		*	Use X as attribute value.
...	`rml-others="X"`	*		Bind *all* attributes that are not already bound to X.
...	`rml-others="X"`		*	Use X to output attributes.
...	`rml-type="or"`	*		If this element does not match, try the next element in the antecedent if that also has rml-type="or".
Elements that add constraints				
`rml-if`	`child="X"`	*		Match if X is already bound to 1 element, and occurs *somewhere* in the current sequence of elements.
`rml-if`	`nochild="X"`	*		Match if X does not occur in the current sequence.
`rml-if`	`last="true"`	*		Match if the preceding sibling of this element is the last in the current sequence.
A `*` in the `A` column means the construct can appear in a rule antecedent. A `*` in the `C` column is for the consequence.				

Figure 2.1: All the RML constructs