A Lightweight Approach for Managing XML Documents with MDE Languages

Dimitrios S. Kolovos, Louis M. Rose,
James Williams, Nicholas Matragkas, and Richard F. Paige

Department of Computer Science, University of York,
Derramore Lane, Heslington, York, YO10 5GH, UK.
{dkolovos,louis,jw,nikos,paige}@cs.york.ac.uk

Abstract. The majority of contemporary model management languages that support MDE tasks (such as model transformation, validation and code generation) require models to be captured using metamodelling architectures such as Ecore and MOF. In practice, a limited subset of modelling tools – with the exception of some UML tools – build atop such architectures. For many modelling languages and tools outside of the UML/Ecore/MOF family, plain XML is a widely-used model storage and exchange format. In this paper, we argue for the importance of integrating XML-based models in the MDE process. We identify the challenges involved in integrating XML-based models into MDE processes, and we present a technical solution that addresses these challenges, which enables developers to perform a wide range of model management tasks on models captured in XML.

1 Introduction

Model Driven Engineering (MDE) focuses on elevating machine-processable *models* to first-class artefacts of the software development process. MDE is technology-agnostic in the sense that it does not prescribe a specific architecture or framework atop which models should be captured, or a particular format in which they should be stored. Therefore, in principle, any structured machine-processable document can play the role of a model in an MDE process.

The majority of recent research on MDE has focused on 3-level metamodelling architectures where models conform to metamodels which are defined in terms of architecture / framework-specific metamodelling languages such as MOF [1] or Ecore [2]. As a result, most contemporary model management languages that support tasks such as model transformation, code generation, model validation etc., require models to be captured atop such architectures. In practice however, very few modelling tools actually use MOF/Ecore to manage and store their models; XML appears to be the most commonly used model persistence format [3].

Although XML is clearly inferior for MDE purposes to elaborate object-oriented metamodelling architectures from a technical perspective, due to its popularity and simplicity, it has the potential of lowering the entry barrier and

playing the role of a stepping stone for the wider adoption of automated model management and MDE. In an effort to make model management languages and MDE techniques more accessible to XML-literate developers, in this paper we propose a lightweight approach for providing first-class support for managing XML documents within Epsilon [4], a mature and well-established family of model management languages. By first-class in this context, we mean support for XML documents in their native standard W3C DOM[1] representation, and not through an implicit or a behind-the-scene injection to a proprietary representation (e.g. as instances of an Ecore-based XML metamodel) that Epsilon already provides support for.

The remainder of the paper is organised as follows. In Section 2 we discuss the importance of XML for MDE and highlight the need for providing first-class support for XML documents in model management languages. In Section 3 we discuss how we implemented such support in the context of Epsilon and in Section 4 we present a case study that illustrates using languages of the Epsilon platform to perform model management tasks on XML documents. In Section 5 we discuss related work and in Section 6 we conclude and provide directions for further work on this subject.

2 Background and Motivation

XML is ubiquitous in the world of software: a vast number of off-the-shelf tools either use XML as a native format for storing structured data they manage, or provide import/export capabilities from/to XML. Also, literally hundreds of modelling languages have been defined atop XML [3] such as the Systems Biology Markup Language (SBML)[2], the Financial products Markup Language[3] and exchange formats such as the Graph Exchange Language[4]. This is consistent with the experience obtained through our interaction with industrial collaborators, which also indicates that XML is particularly popular as a native representation format for bespoke modelling tools developed in-house.

Compared to contemporary metamodelling architectures such as EMF and MOF, plain XML is technically inferior as it only supports capturing tree-structured metadata and does not provide support for types. XML Schema remedies these limitations by adding support – among other – for formalising cross-references between XML elements, and for defining complex and primitive types but is still geared more towards the concrete representation rather than towards the abstract syntax of the metadata it models.

Despite its technical limitations, we argue that plain XML has the potential to lower the entrance barrier for developers that have not been previously exposed to MDE; it can be used to enable developers to capture primitive *models* that contain domain-specific information of interest and start managing them

[1] http://www.w3.org/DOM/
[2] http://sbml.org/
[3] http://www.fpml.org/
[4] http://www.gupro.de/GXL/

in an automated manner with MDE languages, without requiring them to first become familiar with metamodelling architectures such as EMF and MOF. In the sequel, and if automated model management (e.g. code generation, model transformation, validation) appears to be delivering results, a transition to a contemporary metamodelling architecture that addresses the limitations of XML is the next logical step.

In the following sections we demonstrate an approach for contributing first-class support for managing plain XML documents to the Epsilon family of MDE languages. Our aim with this work is to render MDE languages useful and attractive to developers that are experienced with XML but not with metamodelling architectures, thus providing means that lower the entrance barrier to MDE.

2.1 Epsilon

Epsilon [4] is a mature and well-established family of interoperable languages for model management. Languages in Epsilon can be used to manage models of diverse metamodels and technologies. At the core of Epsilon is the Epsilon Object Language (EOL) [5], an OCL-based imperative language that provides features such as model modification, multiple model access, conventional programming constructs (variables, loops, branches etc.), user interaction, profiling, and support for transactions. Although EOL can be used as a general-purpose model management language, its primary aim is to be reused in task-specific languages. Thus, a number of task-specific languages have been implemented atop EOL, including those for model transformation (ETL), model comparison (ECL), model merging (EML), model validation (EVL), model refactoring (EWL) and model-to-text transformation (EGL).

With regard to the types of models supported, Epsilon provides the Epsilon Model Connectivity (EMC) layer that offers a uniform interface for interacting with models of different modelling technologies. Currently, EMC drivers have been implemented to support EMF [2] (XMI 2.x), MDR [6] (XMI 1.x) and Z [7] specifications in LaTeX using CZT [8] Also, to enable users to compose complex workflows that involve a number of individual model management tasks, Epsilon provides ANT [9] tasks and an inter-task communication framework discussed in detail in [10].

3 Managing XML Documents in Epsilon

In this section we illustrate how we have implemented first-class support for managing XML documents in all the languages provided by Epsilon to perform tasks such as model transformation, validation, comparison, refactoring, merging and code generation.

3.1 The Epsilon Model Connectivity Layer

The Epsilon Model Connectivity (EMC), shown in Figure 1, is an abstraction layer for managing models in Epsilon. Via EMC, the model management lan-

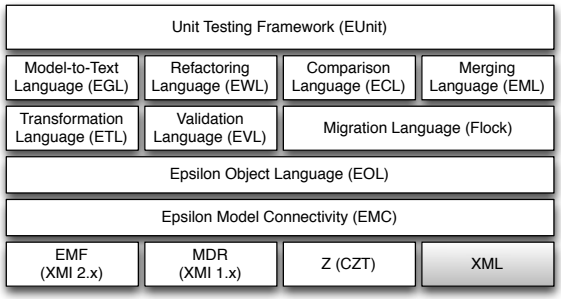

Fig. 1. Overview of the architecture of Epsilon

guages of Epsilon can query and modify models of varying modelling technologies without needing to be aware of the low-level details of each technology.

EMC enables developers to implement *drivers* – essentially classes that implement the `IModel` interface of Figure 2 – to support diverse modelling technologies. This work illustrates the design and implementation of an additional driver (on top of the existing drivers for managing EMF, MDR and Z models) for interacting with schema-less XML documents.

In addition to abstracting over the technical details of specific modelling technologies, EMC facilitates the concurrent management of models expressed with different technologies. For instance, Epsilon can be used to transform an EMF-based model into an MDR-based model, to perform inter-model validation between a Z model and an EMF model, or to develop a code generator that consumes information from an EMF-based and an XML model at the same time.

3.2 The Plain XML EMC Driver

To support management of XML documents with languages of the Epsilon family, a new driver has been implemented atop EMC. The XML driver uses the standard W3C DOM Java implementation as the underlying representation for XML documents and this, combined with the ability of Epsilon languages to invoke Java operations enables developers to access the complete standard DOM API[5] in their model management programs.

By contrast to drivers for 3-tier architectures such as EMF/MOF, in this driver, in the absence of a metamodel or a schema, the developer needs to assist Epsilon in navigating the XML model and performing type coercion / casting. Therefore, the plain XML driver (shaded box in Figure 1) uses predefined naming conventions to allow developers to programmatically access and modify XML documents in an elegant and concise way. It is worth noting that providing support for XML documents in Epsilon did not require any other changes beyond

[5] http://www.w3.org/DOM/

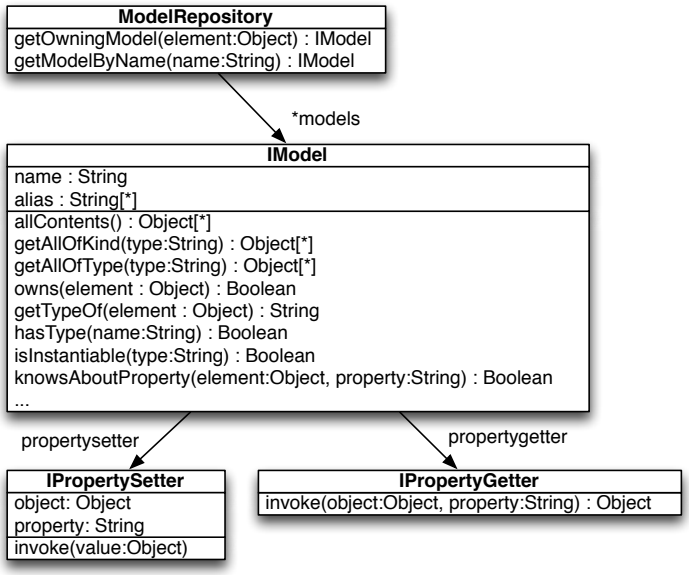

Fig. 2. The Model Connectivity Layer of Epsilon

the addition of the XML driver. This section outlines the supported conventions using the document of Listing 1.1 as a running example.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<library>
  <book title="Eclipse␣Modeling␣Framework" pages="744">
    <author>Dave Steinberg</author>
    <author>Frank Budinsky</author>
    <author>Marcelo Paternostro</author>
    <author>Ed Merks</author>
    <published>2009</published>
  </book>
  <book title="Eclipse␣Modeling␣Project:␣A␣Domain-Specific␣
      Language␣(DSL)␣Toolkit" pages="736">
    <author>Richard Gronback</author>
    <published>2009</published>
  </book>
  <book title="Official␣Eclipse␣3.0␣FAQs" pages="432">
    <author>John Arthorne</author>
    <author>Chris Laffra</author>
    <published>2004</published>
  </book>
</library>
```

Listing 1.1. Example XML document

Accessing elements by tag name The `t_` prefix before the name of the tag is used to represent a type, instances of which are all the elements with that tag. For instance, `t_book.all` can be used to retrieve all elements tagged as `<book>` in the document, `t_author.all` to retrieve all `<author>` elements etc. Also, if `b` is an element with a `<book>` tag, then `b.isTypeOf(t_book)` shall return true.

```
// Get all <book> elements
var books = t_book.all;

// Get a random book
var b = books.random();

// Check if b is a book
// Prints 'true'
b.isTypeOf(t_book).println();

// Check if b is a library
// Prints 'false'
b.isTypeOf(t_library).println();
```

Listing 1.2. Accessing elements by tag name

Getting and Setting Attribute Values of Elements An attribute name, prefixed by `a_`, can be used as a property of the element object. For example, if `b` is the first book of the XML document of Listing 1.1, `b.a_title` will return EMF Eclipse Modeling Framework. Attribute properties are readable and writable.

In this example, `b.a_pages` will return `744` as a string. For `744` to be returned as an integer, the `i_` prefix should be used instead (i.e. `b.i_pages`. The driver also supports the following prefixes: `b_` for boolean, `s_` for string (alias of `a_`) and `r_` for real values.

```
// Print all the titles of the books in the library
for (b in t_book.all) {
  b.a_title.println();
}

// Print the total number of pages of all books
var total = 0;
for (b in t_book.all) {
  total = total + b.i_pages;
}
total.print();

// ... the same using collect() and sum()
// instead of a for loop
t_book.all.collect(b|b.i_pages).sum();
```

Listing 1.3. Getting and setting attribute values

Getting/setting the text of an element The `.text` property can be used to read/write the value of the textual content of an element.

```
1 for (author in t_author.all) {
2   author.text.println();
3 }
```

Listing 1.4. Getting and setting the text of an element

Accessing the parent of an element The `.parentNode` read-only property can be used to retrieve the parent node of an element.

```
1 // Get a random book
2 var b = t_book.all.random();
3
4 // Print the tag of its parent node
5 // Prints 'library'
6 b.parentNode.tagName.println();
```

Listing 1.5. Getting the parent of an element

Retrieving the children of an element The `.children` read-only property can be used to retrieve all the child-nodes of an element.

```
1 // Get the <library> element
2 var lib = t_library.all.first();
3
4 // Iterate through its children
5 for (b in lib.children) {
6   // Print the title of each child
7   b.a_title.println();
8 }
```

Listing 1.6. Getting the children of an element

Getting child elements with a specific tag name Using what has been discussed so far, this can be achieved using a combination of the `.children` property and the `select`/`selectOne()` EOL operations. However, the driver also supports `e_` and `c_`-prefixed shorthand properties for accessing one or a collection of elements with the specified name respectively. `e_` and `c_` properties are read-only.

```
1 // Get a random book
2 var b = t_book.all.random();
3
4 // Get its <author> children using the
5 // .children property
6 var authors = b.children.select(a|a.tagName = "author");
```

```
7
8  // Do the same using the shorthand
9  authors = b.c_author;
10
11 // Get its <published> child and print
12 // its text using the
13 // .children property
14 b.children.selectOne(p|p.tagName = "published").text.
       println();
15
16 // Do the same using the shorthand
17 // (e_ instead of c_ this time as
18 // we only want one element,
19 // not a collection of them)
20 b.e_published.text.println();
```

Listing 1.7. Getting children with a specific tag name

Creating new elements The standard `new` operator can be used to create new elements in the XML document.

```
1  // Check how many <books> are in the library
2  // Prints '3'
3  t_book.all.size().println();
4
5  // Creates a new book element
6  var b = new t_book;
7
8  // Check again
9  // Prints '4'
10 t_book.all.size().println();
```

Listing 1.8. Creating new elements

Add a child to an existing element The `.appendChild(child)` operation can be used to add a child-node to an element. If the node to be added is already a child of another node, it is first detached from its previous parent.

```
1  // Create a new book
2  var b = new t_book;
3
4  // Get the library element
5  var lib = t_library.all.first();
6
7  // Add the book to the library
8  lib.appendChild(b);
```

Listing 1.9. Adding a child to an existing element

Setting the root element of an XML document The `.root` property of the model can be used to set the root element of an XML document.

```
1 XMLDoc.root = new t_library;
```

Listing 1.10. Setting the root element of an XML document

The XML driver also supports (optional) caching so that expensive operations such as collecting all elements with a particular tag do not need to be performed repetitively.

3.3 Alternative Design Choices

As discussed above, the plain XML driver presented in this section makes use of particular naming conventions – such as the `t_`, and `c_` prefixes – to specify XML model element types, to distinguish between child elements and attribute values, to specify the expected result type when retrieving children of an element by name (single element vs. collection of elements), and to perform type-casting of the values of attributes. Given that Epsilon is dynamically typed, the prefixes could have been eliminated in an alternative design, but this would have introduced several inconveniences, which are now discussed.

Ordinarily, Epsilon throws a runtime error when trying to use undefined variables or types of model element that do not exist. These runtime errors provide valuable information to users, alerting them to problems with their programs. Schema-less models, such as plain XML models, do not provide type information. Without the `t_` prefix for XML model element types, the EOL interpreter would become unable to distinguish between undefined variables and XML model element types. Consequently, undefined variables would have to be treated as XML model element types, and the user would not be alerted to the potential error in their program. In addition, had we not used `c_` and `e_` to distinguish between single and multiple children, all child element navigations would need to return a collection of elements. Also, explicitly specifying attribute value type-casting using the `i_`, `r_`, `b_` prefixes avoids unintended type casts.

In our view, employing these prefixes makes up to an extent for the lack of a formal metamodel and makes code easier – albeit slightly more verbose – to write and maintain.

4 Case Study

In this section we present a case study that demonstrates how the XML driver that was presented in the previous section can be used to validate and transform the XML-based OO model of Listing 1.11 to a respective EMF-based UML model. This case study has been intentionally kept simple for brevity reasons.

```
1 <?xml version="1.0"?>
2 <model>
3   <class name="Customer">
```

```
4     <property name="name" type="String"/>
5     <property name="address" type="Address"/>
6   </class>
7   <class name="Invoice">
8     <property name="serialNumber" type="String"/>
9     <property name="customer" type="Customer"/>
10    <property name="items" type="InvoiceItem" many="true"/>
11  </class>
12  <class name="InvoiceItem">
13    <property name="quantity" type="Integer"/>
14    <property name="product" type="Product"/>
15  </class>
16  <class name="Product">
17    <property name="name" type="String"/>
18    <property name="unitPrice" type="Float"/>
19  </class>
20  <class name="Address">
21    <property name="number" type="String"/>
22    <property name="postCode" type="String"/>
23  </class>
24  <datatype name="String"/>
25  <datatype name="Integer"/>
26  <datatype name="Float"/>
27 </model>
```

Listing 1.11. OO model captured using XML

Listing 1.12 illustrates a constraint expressed using the Epsilon Validation Language (EVL) which checks that the type of each property in the XML model of Listing 1.11 corresponds to a defined type (class or datatype). Line 2 defines that the constraint applies to all elements tagged as `property` and line 5 checks that there is an element tagged as `datatype` or `class` whose name matches the value of the `type` attribute of the `property`. If such an element is not found, in lines 7-9 a diagnostic message is produced.

```
1  import "util.eol";
2  context t_property {
3    constraint TypeMustBeDefined {
4
5      check : typeForName(self.a_type).isDefined()
6
7      message : "Property " + self.a_name + " of class " +
8        self.parentNode.a_name + " is of unknown type: " +
9        self.a_type
10   }
11 }
```

Listing 1.12. XML validation constraint expressed in EVL

Listing 1.13 illustrates a model-to-model transformation expressed using the Epsilon Transformation Language (ETL) that transforms the XML model of

Listing 1.11 to an EMF-based UML model. The transformation consists of 4 rules which transform elements tagged as `model`, `class`, `property` and `datatype` to respective Models, Classes, Properties and DataTypes in the target UML model. This transformation illustrates how EMC enables programs in all Epsilon languages to manage models that conform to different technologies concurrently.

```
1  import "util.eol";
2
3  rule t_model2Model
4   transform s : XML!t_model
5   to t : UML!Model {
6
7   t.packagedElement.addAll(s.children.equivalent());
8  }
9
10 rule t_class2Class
11  transform s : XML!t_class
12  to t : UML!Class {
13
14  t.name = s.a_name;
15  t.ownedAttribute.addAll(s.children.equivalent().
16    select(e|e.isTypeOf(UML!Property)));
17 }
18
19 rule t_property2Property
20  transform s : XML!t_property
21  to t : UML!Property {
22
23  t.name = s.a_name;
24  var type = typeForName(s.a_type);
25  t.type = type.equivalent();
26
27  if (s.b_many) { t.upper = -1; }
28
29  if (not type.isTypeOf(XML!t_datatype)) {
30   var association = new UML!Association;
31   association.ownedEnds.add(t);
32   var opposite = new UML!Property;
33   opposite.type = s.parentNode.equivalent();
34   association.ownedEnds.add(opposite);
35   UML!Model.all.first().packagedElement.
36     add(association);
37  }
38
39 }
40
41 rule t_datatype2DataType
42  transform s : XML!t_datatype
43  to t : UML!DataType {
44
```

```
45   t.name = s.a_name;
46
47 }
```

Listing 1.13. XML to UML transformation expressed in ETL

```
1  operation typeForName(type : String) {
2   return allTypes().selectOne(t|t.a_name = type);
3  }
4
5  operation allTypes() : Sequence {
6   return XML!t_class.all.includingAll(XML!t_datatype.all);
7  }
```

Listing 1.14. Utility methods (util.eol) used in Listings 1.13 and 1.12

5 Related Work

The importance of XML has been recognised by the developers of the Eclipse Modelling Framework (EMF) and as a result EMF provides support for managing schema-based XML documents. To support schema-based XML documents, EMF provides a built-in transformation that can produce an Ecore metamodel from an XML schema, a parser that can parse XML files that conform to an XSD into in-memory models that conform to the respective Ecore metamodel, and a serialiser that can then persist in-memory models back to XML. While Ecore and XSD share many common features such as being able to define complex structures (e.g. through EClasses in Ecore and Complex Types in XSD), inheritance, references with cardinality etc. they also differ in some respects. For instance, XSD can define anonymous complex types while Ecore cannot define anonymous EClasses, EMF models can contain multiple root objects while XML documents can only have one root node, Ecore does not have equivalent constructs for the XSD `<choice>` element or the `mixed` feature, etc. In an effort to compensate for these differences, the XSD to Ecore transformation employs conventions that, while necessary, can lead to non-straightforward Ecore metamodels.

For example, the XML Schema of listing 1.15 is transformed into the Ecore metamodel illustrated in Figure 3. In the Ecore metamodel the reader can observe the ItemType and ItemType1 EClasses which have been generated by the anonymous complex types in lines 8 and 21 of the XSD. Also, in order for a developer to access the text content of an item element, they need to query the `mixed` feature of ItemType (or ItemType1) – which is not straightforward for a developer with no EMF expertise.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4  <xs:element name="invoice">
```

```
5    <xs:complexType>
6      <xs:sequence>
7        <xs:element name="item">
8          <xs:complexType mixed="true">
9            <xs:sequence>
10             <xs:element name="unitPrice" type="xs:float"/>
11           </xs:sequence>
12         </xs:complexType>
13       </xs:element>
14     </xs:sequence>
15   </xs:complexType>
16 </xs:element>
17 <xs:element name="order">
18   <xs:complexType>
19     <xs:sequence>
20       <xs:element name="item">
21         <xs:complexType mixed="true">
22           <xs:sequence>
23             <xs:element name="quantity" type="xs:int"/>
24             </xs:sequence>
25           </xs:complexType>
26         </xs:element>
27       </xs:sequence>
28     </xs:complexType>
29   </xs:element>
30 </xs:schema>
```

Listing 1.15. Example XML Schema

The Atlas Transformation Language (ATL) provides support for schema-less XML documents through an injection transformation that converts an XML document to a respective EMF model that conforms to a simple Ecore-based XML metamodel, and an extraction transformation that does the reverse. As such, the syntax for managing XML documents in ATL is particularly verbose as illustrated by Listings 1.16 and 1.17.

```
1 XML!t_book.all.first().a_title.println();
```

Listing 1.16. EOL statement that prints the title of the first book

```
1 XML!Element.allInstances()->select(e|e.name = 'book')->
      first()
2   .children->select(c|c.oclIsTypeOf(XML!Attribute)
3     and c.name = 'title')->first().value.println();
```

Listing 1.17. Equivalent ATL statement that prints the title of the first book

Xlinkit [11] is a tool for checking consistency issues in distributed documents. Using Xlinkit, developers can specify cross-document constraints that can be automatically evaluated to reveal inconsistencies. For the specification of constraints, Xlinkit defines an XML-based language that uses XPath [12] for

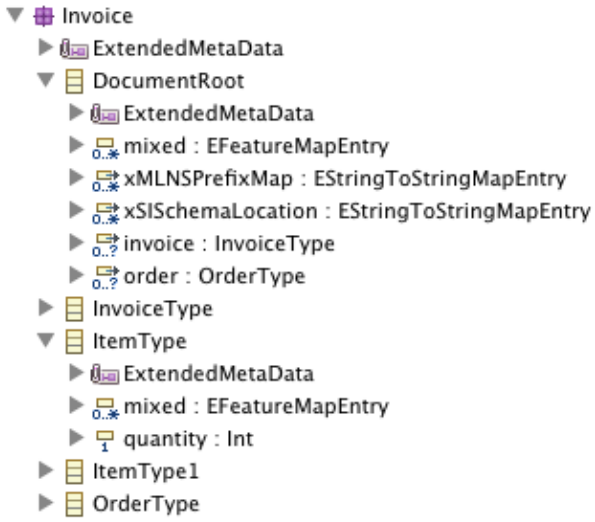

Fig. 3. Ecore metamodel generated from the XML Schema of Listing 1.15

document navigation. Listing 1.18 demonstrates an exemplar Xlinkit constraint that applies on a UML and a Java model and states that for each class in the UML model, a class with the same name must exist in the Java model. In our view, the main shortcoming of this approach is that the concrete syntax of the expression language is based on XML and that, as illustrated in Listing 1.18, results in lengthy and challenging to read and maintain specifications.

```
<globalset id="classes"
  xpath="//Foundation.Core.Class[@xmi.id]"/>
<globalset id="javaclasses" xpath="/java/class"/>
<consistencyrule id="r1">
 <forall var="c" in="classes">
  <exists var="j" in="javaclasses">
   <equal
     op1="c/Foundation.Core.ModelElement.name/text()"
     op2="j/@name"/>
  </exists>
 </forall>
</consistencyrule>
```

Listing 1.18. Example Xlinkit constraint

6 Conclusions and Further Work

In this paper we have highlighted the importance of XML in the context of MDE; in particular we have discussed the role of XML both as a legacy format in which

a significant amount of data is already encoded, and as a means of lowering the entrance barrier for newcomers in MDE. Following that we illustrated a technical solution for adding first-class support for XML to the Epsilon MDE platform so that plain XML documents can be used in a wide range of MDE tasks such as model validation, transformation, comparison, merging and code generation as they are and without needing to first transform them to models that conform to metamodelling architectures such as MOF or EMF.

Although in this paper we have illustrated a solution for adding support for managing XML documents to a particular family of model management languages, it is worth noting that this approach is also directly applicable to other model management languages (such as ATL[13] or MOFScript[14]) that provide a layer of indirection between the language run-time and the concrete modelling technologies they support.

Acknowledgements. The work in this paper was supported by the European Commission via the MADES and INESS projects, co-funded under the 7^{th} Framework programme (grants #218575 (INESS), #248864 (MADES)).

References

1. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
2. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks. *EMF: Eclipse Modelling Framework*. Eclipse Series. Addison-Wesley Professional, 2 edition, December 2008.
3. CoverPages. XML Applications and Initiatives, June 2005. http://xml.coverpages.org/xmlApplications.html.
4. Eclipse Foundation. Epsilon Modeling GMT component. http://www.eclipse.org/gmt/epsilon.
5. Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.
6. Sun Microsystems. Meta Data Repository. http://mdr.netbeans.org.
7. Jim Woodcock and Jim Davies. *Using Z : Specification, Refinement, and Proof*. Prentice Hall, March 1996.
8. Community Z Tools. http://czt.sourceforge.net.
9. The Apache Ant Project. http://ant.apache.org.
10. Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. A Framework for Composing Modular and Interoperable Model Management Tasks. In *Proc. Workshop on Model Driven Tool and Process Integration (MDTPI), ECMDA*, Berlin, Germany, June 2008.
11. Christian Nentwich, Licia Capra, Wolfgang Emmerich and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002.
12. W3C. XML Path Language (XPath), Official Web-Site. http://www.w3.org/TR/xpath.
13. Frédéric Jouault and Ivan Kurtev. Transforming Models with the ATL. In Jean-Michel Bruel, editor, *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, volume 3844 of *LNCS*, pages 128–138, Montego Bay, Jamaica, October 2005.

14. Jon Oldevik. MOFScript User Guide. http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf.