Implementing the Interactive Television Applications Case Study using Epsilon

Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, Fiona A.C. Polack
Department of Computer Science
The University of York
YO10 5DD, York, United Kingdom
{dkolovos, paige, lmr109, fiona}@cs.york.ac.uk

ABSTRACT

In this paper we illustrate a technical solution to the Interactive Television Applications case study presented in [1] using languages and tools from the Epsilon GMT component based on the infrastructure provided by the Eclipse Modeling Framework (EMF).

1. INTRODUCTION

The case study presented in [1] presents the requirements of a system that supports designing interactive television applications using model engineering techniques and principles. We have implemented the main operations of such a system with languages and tools from the Epsilon GMT component [2], using the Eclipse Modeling Framework (EMF) [3] as the underlying modelling technology. The rest of the paper is organized as follows. In Section 2 we provide a brief discussion on Epsilon. In Section 3 we discuss model management operations implemented to define the system, and in Section 4 we draw conclusions on the usefulness of our approach and the needs for extending Epsilon as identified during the implementation process.

2. THE EPSILON MODEL MANAGEMENT PLATFORM

Epsilon [2], is a platform of integrated task-specific model management languages, and a component of the Eclipse Generative Modeling Technologies (GMT) research incubator project. Epsilon provides languages for direct manipulation of models (EOL) [4], model merging (EML) [5], model comparison (ECL) [6], model-to-model transformation (ETL), model validation (EVL) [7] and model-to-text transformation (EGL). All the languages of the platform build on a common OCL-based model navigation and manipulation language (EOL) and a common runtime environment, and are therefore highly interoperable. For example, an *operation* defined using EOL can be reused (imported) as-is by the model-to-model and the model-to-text transformation languages. With regard to supported modelling technologies, the architecture of Epsilon allows users to manage models of different technologies, such as MDR [8] and EMF [3] models and XML documents, and even facilitates support for additional model representation formats using a layered architecture (discussed in [4]). In the context of tool-support, Epsilon languages are supported by stable execution engines and Eclipse-based development tools, available at [2].

3. IMPLEMENTING THE CASE STUDY

In this section we identify the model management operations involved in the system and present a concrete implementation for each. Our intention here is not to provide a ready-to-use tool, but instead to demonstrate how the model-management-related components of such a tool can be implemented using Epsilon. Integrating the components in a usable application that can be operated by end-users would require a significant amount of trivial Java code, which is clearly out of scope of this work.

3.1 Designing the TVApp DSL

Our first step was to design a Domain Specific Language (DSL) that enables users to design Interactive TV Applications. We have designed our TVApp DSL atop EMF by defining its abstract syntax in terms of ECore (using Emfatic [9] as a convenience textual representation). A graphical overview of the DSL is presented in Figure 1.

3.2 Editing Support for the TVApp DSL

The next step was to provide editing support for the DSL so that users can start composing models that conform to the language. The available options were: to use a tree-based editor; to use a textual syntax (e.g. with oAW XText [10]); and/or to use a diagrammatical syntax (e.g. using GMF [11]). Since the TVApp DSL is predominantly hierarchical, we decided to adopt the tree-based editor approach. EMF provides a reflective tree-based editor that can be used to edit EMF models of arbitrary ECore metamodels. On top of this reflective editor we have implemented Exeed [12], an extension that enables users to customize the appearance (e.g labels, icons, property view) of the editor by attaching specific EAnnotations to the EClasses and EStructuralFeatures of the metamodel. For instance, to make both the *name* and the *information* of a *Text* appear on the editor (see Figure 3), the annotations displayed in Listing 1 were added to the definition of the *Text* EClass.

Listing 1: Definition of the Text EClass using Emfatic

```
@exeed(classIcon="text",
 label="return self.name + ': ' + self.information;")
class Text extends Content {

  @exeed(multiLine="true")
  attr String information;
  val TextHistory[*] history;
}
```

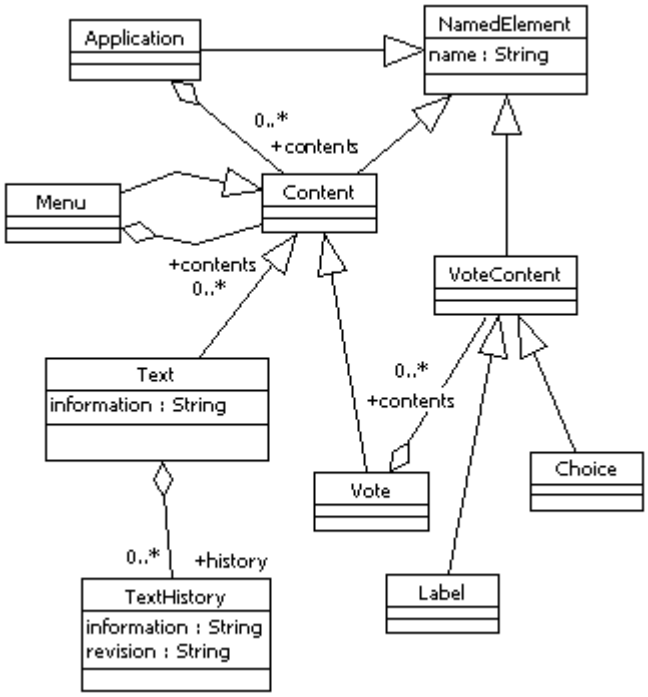

Figure 1: The TVApps Domain Specific Language

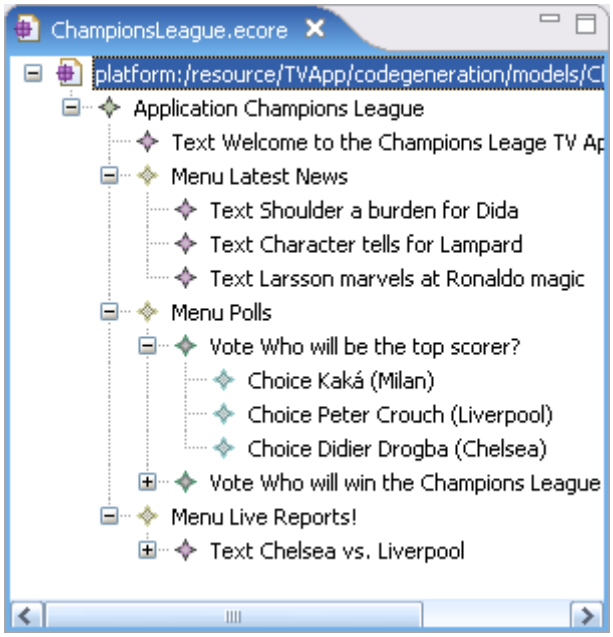

Figure 2: Designing TVApp models using the Reflective Editor

Exeed provides a number of additional annotations discussed in detail in [12]. In fact, the same customization can be achieved by generating a dedicated editor for the DSL and then customizing it using Java. However, as discussed in [12], we regard using Exeed as a more agile approach, particularly for developing prototype editors. The basic and enhanced views of a TVApp model in the reflective editor and Exeed are shown in Figures 2 and 3 respectively.

3.3 Generating a TVApp model for a Sports Competition

To address Case 1 of [1], we have designed an additional Competition metamodel for capturing information about competitions, groups and participating teams. A graphical overview of the metamodel is displayed in Figure 4. The design of the metamodel also satisfies the requirement of Case 2 for defining the team-related information first and then adding the teams into groups by modifying the *competitors* list of each group.

To transform a Competition model into a TVApp model, we have used the Epsilon Transformation Language (ETL) [2]. The transformation displayed in Listing 2 defines that an Application and a Vote will be generated from each Competition, and that the Vote will contain a flattened sequence of Labels (one for each Group) and Choices (one for each Competitor in the group).

Listing 2: ETL transformation that transforms a Competition model into a TVApp model

```
rule Competition2Application
  transform c : Competition!Competition
  to a : TVApp!Application, v : TVApp!Vote {

  a.name := c.name + ' Application';
  v.name := 'Who will win the ' + c.name + '?';
  a.contents.add(v);
```

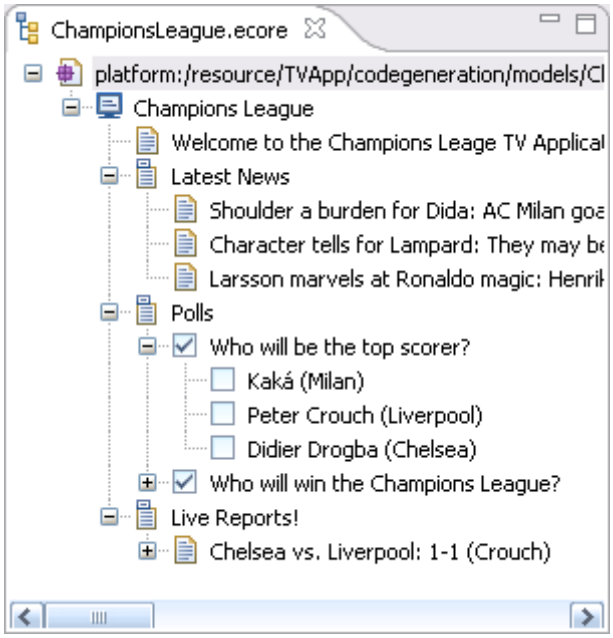

Figure 3: Designing TVApp models using Exeed

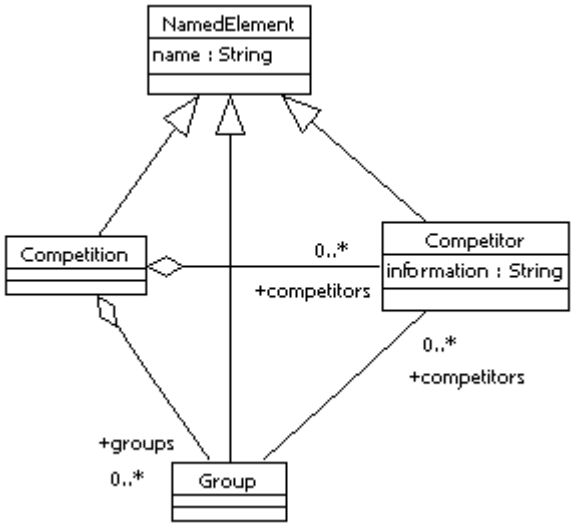

Figure 4: The Competition Domain Specific Language

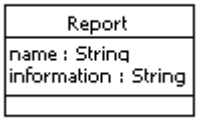

Figure 5: The Report Domain Specific Language

```
  for (g in c.groups) {
    v.contents.add(g.equivalent());
    for (memb in g.members) {
      v.contents.add(memb.equivalent());
    }
  }
}

rule Competitor2Choice
  transform co : Competition!Competitor
  to ch : TVApp!Choice {

  ch.name := co.name;
}

rule Group2Label
  transform g : Competition!Group
  to l : TVApp!Label {

  l.name := 'Group ' + g.name;
}
```

3.4 Integrating Live Reports

Case 3 of [1] requires that an external user must be able to update a particular text but not the structure of the application. To achieve this we have defined the minimal Report DSL. The rationale is that a user can be granted the permission to only compose Report models which will then be merged with the original application model. To satisfy the requirement set in Case 4 of [1], when merging the TVApp with a Report, the original information contained in the Texts which the Report updates are not lost but instead are stored in the form of new *TextHistory* model elements. We implement the merging using the Epsilon Merging Language (EML) [5] module displayed in Listing 3.

As discussed in [5], EML operates in two steps. The first step is to establish correspondences between elements that need to be merged (using match-rules) and the second is to merge those elements (using merge-rules) and optionally transform any non-matched elements into the target model (using transform-rules reused from ETL). In Listing 3, matching Reports from the Report model with Texts from an OldTVApp model is achieved through the MatchReportWithText match-rule that compares the names of the compared elements. Then, matching Reports and Texts are merged using the MergeReportWithText merge-rule which specifies that the Text should be updated and that the old version of the text should be stored in the form of a new TextHistory model element with a proper information and revision number.

Listing 3: EML module that merges a TVApp with a Report model

```
rule MatchReportWithText
  match t : OldTVApp!Text
  with r : Report!Report {

  compare : r.name = t.name
}

auto rule MergeReportWithText
  merge ot : OldTVApp!Text
  with r : Report!Report
  into nt : NewTVApp!Text {

  var h : new NewTVApp!TextHistory;
  h.information := ot.information;
  h.revision := ot.history.collect(h|h.revision).max(0) + 1;
  nt.history.add(h);
  nt.information := r.information;
}
```

3.5 Generating XML from a TVApp model

To generate XML from a TVApp model as required in Section 6 of [1], we use the pivot XML metamodel displayed in Figure 6. Thus, we transform the TVApp model into an XML model (Listing 4) using the Epsilon Transformation Language, and then we generate the textual representation of the XML model (Listing 5) using the Epsilon Generation Language, a template-based language for text generation. Using this approach promotes modularity by refraining from generating XML text directly from the TVApp model, but also conform with the requirement not to use a special (hard-coded) method for transforming XML models to text.

Listing 4: ETL transformation that transforms TVApp models to XML models

```
pre {
  var doc : new Xml!Document;
  doc.rootElement := TVApp!Application.
    allInstances.first().equivalent();
}

abstract rule NamedElement2Element
  transform ne : TVApp!NamedElement
  to n : Xml!Element {

  n.addAttribute('name', ne.name);
}

rule Application2Element
  transform a : TVApp!Application
  to n : Xml!Element extends NamedElement2Element{

  n.name := 'Application';
  n.contents := a.contents.equivalent();
}

rule Vote2Element
```

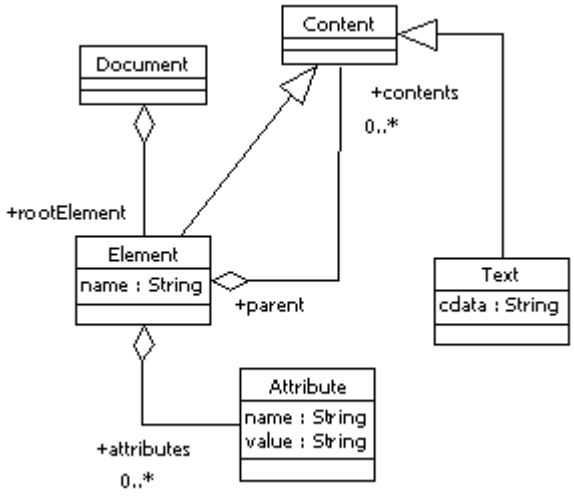

Figure 6: The XML Domain Specific Language

```
  transform v : TVApp!Vote
  to n : Xml!Element extends NamedElement2Element {

  n.name := 'Vote';
  n.contents := v.contents.equivalent();
}

rule Choice2Element
  transform c : TVApp!Choice
  to n : Xml!Element extends NamedElement2Element {

  n.name := 'Choice';
}

rule Label2Element
  transform c : TVApp!Label
  to n : Xml!Element extends NamedElement2Element {

  n.name := 'Label';
}

rule Text2Element
  transform t : TVApp!Text
  to e : Xml!Element extends NamedElement2Element {

  e.name := 'Text';
  var text : new Xml!Text;
  text.cdata := t.information;
  e.contents.add(text);
}

rule Menu2Element
  transform m : TVApp!Menu
  to e : Xml!Element extends NamedElement2Element {

  e.name := 'Menu';
  e.contents := m.contents.equivalent();
}

operation Xml!Element addAttribute
  (name : String, value : String) {
  var attr : new Xml!Attribute;
  attr.name := name;
  attr.value := value;
  self.attributes.add(attr);
}
```

The ETL transformation of Listing 4, demonstrates two important characteristics of ETL: rule inheritance and state-changing operations. By inheriting from the abstract rule *NamedElement2Element*, the rest of the rules are maintained simple and without duplication. Moreover, defining the *addAttribute()* state-changing operation simplifies the *NamedElement2Element* rule.

Listing 5: EGL template that generates XML text from XML models

```
<?xml version="1.0"?>
<!--Generated using EGL-->
[%=Document.allInstances().first().rootElement.toString(0)%]

[%
operation Element toString(indent : Integer) : String {
  var str : String;
  str := indent.getIndent() + '<' + self.name.normalize();
  for (a in self.attributes) {
    str := str + ' ' + a.name.normalize() +
      '="' + a.value.normalize() + '"';
    if (hasMore){
      str := str + ' ';
    }
  }
  str := str + '>\r\n';
  for (c in self.contents) {
    str := str + c.toString(indent + 1);
  }
  str := str + indent.getIndent() + '</' +
    self.name.normalize() + '>\r\n';
  return str;
}

operation Text toString(indent : Integer) : String {
  return (indent + 1).getIndent() +
    self.cdata.normalize() + '\r\n';
}

operation Integer getIndent() : String {
  var indent : String;
  for (i in Sequence{1 .. self}){
    indent := indent + ' ';
  }
  return indent;
}

operation String normalize() {
  var normalized : String := self;
  if (not normalized.isDefined()) {
    normalized := '';
  }
  else {
    normalized := normalized.replace('<', '&lt;');
    normalized := normalized.replace('>', '&gt;');
    -- etc
  }
  return normalized;
}
%]
```

Generating XML text from an XML model in Listing 5 involves much dynamic and little static text. Therefore, the vast majority of the serialization process is performed via string concatenation. By contrast, the header (processing instruction and comment) is only static text that is emitted as-is from the template.

3.5.1 Integrating the Model-to-Model and the Model-to-Text steps

To integrate the two steps presented above into a coherent process that transforms a TVApp model directly to textual XML, we use the ANT-based Epsilon Workflow [13]. Thus, in Listing 6, we define two tasks for loading the involved models (epsilon.loadModel), one that invokes the ETL transformation (epsilon.etl) and one that invokes the EGL model-to-text transformation on the intermediate XML model (epsilon.egl).

Listing 6: The wokflow that integrates the ETL and EGL tasks

```
<?xml version="1.0"?>
<project default="main">

  <epsilon.loadModel name="TVApp" type="EMF">
    <parameter name="modelFile"
      value="models/ChampionsLeague.ecore"/>
    <parameter name="metamodelFile"
      value="../metamodels/TVAppDsl.ecore"/>
    <parameter name="isMetamodel" value="false"/>
    <parameter name="isMetamodelFileBased" value="true"/>
    <parameter name="readOnLoad" value="true"/>
  </epsilon.loadModel>

  <epsilon.loadModel name="Xml" type="EMF">
    <parameter name="modelFile"
      value="models/TVAppXml.ecore"/>
    <parameter name="metamodelFile"
      value="../metamodels/Xml.ecore"/>
    <parameter name="isMetamodel" value="false"/>
    <parameter name="isMetamodelFileBased" value="true"/>
    <parameter name="readOnLoad" value="false"/>
  </epsilon.loadModel>

  <target name="main">

    <epsilon.etl src="TVApp2Xml.etl" sourcemodel="TVApp"
      targetmodel="Xml">

      <model ref="TVApp"/>
      <model ref="Xml"/>
    </epsilon.etl>

    <epsilon.egl src="Xml2Text.egl"
      target="output/TVApp.xml">
      <model ref="Xml"/>
    </epsilon.egl>

  </target>

</project>
```

3.6 Generating a Mock-up of the TV Application

A feature that is not required explicitly in [1], but which we found really useful is to be able to *preview* a TVApp model using a mock-up that closely resembles the appearance of the final deployed application. Therefore, we have used EGL to compose a model-to-text transformation that generates a set of linked HTML screens that emulate the look-and-feel and functionality of the deployed application [1]. Listing 7 presents the main template of the EGL solution that iterates the model and invokes the respective sub-templates (Text.egl, Menu.egl, Vote.egl) to generate the mockup HTML screens. The complete source code for this and all the other model management tasks presented in this paper is available at http://www.cs.york.ac.uk/~dkolovos/mdd-tif/TVApps.zip.

Listing 7: EGL template that generates mockup HTML screens for a TVApp model

```
[%
  import 'include\\Common.eol';

  TemplateFactory.setTemplateRoot('workspace\\MDD-TIF');
  TemplateFactory.setOutputRoot('workspace\\MDD-TIF\\html');

  var header : Template :=
    TemplateFactory.load('include\\Header.egl');
  var text : Template :=
    TemplateFactory.load('include\\Text.egl');
  var menu : Template :=
    TemplateFactory.load('include\\Menu.egl');
  var vote : Template :=
    TemplateFactory.load('include\\Vote.egl');
  var footer : Template :=
    TemplateFactory.load('include\\Footer.egl');

  var contents : String := 'default';

  if (Application.isType(content) or Menu.isType(content)) {
    menu.populate('menu', content);
    contents := menu.process();

    -- Recursively generate the pages
    -- for the contents of this
    -- application / menu
    for (child in content.contents) {
      var page : Template := TemplateFactory.load('Page.egl');

      page.populate('content', child);

      page.generate(child.filename());
    }

  } else {
    if (Text.isType(content)) {
      text.populate('text', content);
      contents := text.process();

    } else {
      if (Vote.isType(content)) {
        vote.populate('vote', content);
        contents := vote.process();
      }
    }
  }

%]
[%=header.process()%]
[%=contents%]
[%=footer.process()%]
```

[1] An exemplar mockup generated from the TVApp model displayed in Figure 3 is available at http://www.cs.york.ac.uk/∼dkolovos/epsilon/mdd-tif/Champions%20League.html

4. CONCLUSIONS

In this paper we have demonstrated using languages and tools from the Epsilon GMT component to implement the case study presented in [1].

Summarizing our experiences from the implementation process, we have found implementing the model management tasks using languages of the Epsilon platform to be quite straightforward. However, by implementing the case study we also realized that while it is easy to implement and execute model management tasks from a developer perspective, to make it accessible to end-users, a significant amount of Java code must also be written - particularly with respect to presenting the available tasks to the users and enabling them to invoke the tasks in a user-friendly manner. In our view, this is a very important aspect that must be addressed to enable widespread use of model engineering.

5. ACKNOWLEDGEMENTS

The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2006-2009).

6. REFERENCES

[1] Model-Driven Development Tool Implementers Forum Organizers. Interactive Television Applications Case Study, 2007. http://www.dsmforum.org/events/MDD-TIF07/InteractiveTVApps.pdf.

[2] Extensible Platform for Specification of Integrated Langauges for mOdel maNagement (Epsilon). http://www.eclipse.org/gmt/epsilon.

[3] Eclipse.org. Eclipse Modelling Framework. http://www.eclipse.org/emf.

[4] 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.

[5] Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. Merging Models with the Epsilon Merging Language (EML). In *Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, Genova, Italy, October 2006. LNCS.

[6] Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proc. 1st International Workshop on Global Integrated Model Management (GaMMa), ACM/IEEE ICSE 2006*, pages 13 – 20, Shanghai, China, 2006. ACM Press.

[7] Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. On The Evolution of OCL for Capturing Structural Constraints in Modelling Languages. Under review, 2007.

[8] Sun Microsystems. Meta Data Repository. http://mdr.netbeans.org.

[9] IBM alphaWorks. Emfatic Language for EMF Development, February 2005. http://www.alphaworks.ibm.com/tech/emfatic.

[10] openArchitectureWare, Official Web-Site. http://www.openarchitectureware.org/.

[11] Eclipse GMF - Graphical Modeling Framework, Official Web-Site. http://www.eclipse.org/gmf.

[12] Dimitrios S. Kolovos. Exeed: EXtended Emf EDitor - User Manual, 2007. www.eclipse.org/gmt/epsilon/doc/Exeed.pdf.

[13] Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. A Framework for Composing Modular and Interoperable Model Management Tasks. Under review, 2007.