Using Model-to-Text Transformation for Dynamic Web-based Model Navigation

Dimitrios S. Kolovos, Louis M. Rose, and James R. Williams

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

Abstract. One of the main objectives of modelling is to enable collaborative decision making and communication among the stakeholders of the system. It is essential that both technical and non-technical stakeholders can access and comment on the models of the system at any time. In this paper we propose a model-to-text transformation approach for producing dynamic, web-based views of models – captured atop different modelling technologies and conforming to arbitrary metamodels – so that stakeholders can be provided with web-based, on-demand and up-to-date access to the models of the system using only their web browser. We demonstrate the practicality of this approach through case studies and identify a number of open challenges in the field of web-based model management.

1 Introduction

One of the main objectives of modelling is to enable collaborative decision making and communication among the stakeholders of the system – particularly so in the early stages of the software development lifecycle. To this end, stakeholders must be able to access – possibly different parts of – the *models* constructed by the designers of the system. In principle, the simplest way to achieve this is to establish a centralised repository where designers share the models that they construct with other stakeholders, so that the latter can navigate the models using the same modelling tools that the designers used to create them. However, experience obtained from interacting with our industrial partners – some of which was summarised in an earlier paper [1] – suggests that this is not always feasible or desirable, for a number of reasons:

– **Cost:** Purchasing licenses of expensive modelling tools only to view and provide feedback on the models of the system may be impractical or too expensive.
– **Time:** In some industrial environments, installing new software requires a formal approval process which can take significant time to complete.
– **Complexity/Training:** Modelling tools are typically complex because they accommodate the needs of software designers. As such, training is typically required for non-expert users, even to support them in relatively straightforward tasks.

– **Access control:** It may be desirable that some stakeholders are only granted access to some parts of the models.

We have encountered the above issues in the context of ongoing work with one of our major industrial collaborators. To address such issues in practice, modellers typically construct word processor *design documents* which are then disseminated to the stakeholders, who therefore are not required to purchase, install and learn to use any new software. However, this approach has known shortcomings. Assembling and synchronising such documents can be labour-intensive and error-prone, particularly if they are not supported natively by the modelling tool. Moreover, for large models, designers need to create correspondingly large documents, which ultimately become difficult to navigate and read. Finally, and as a consequence of the other shortcomings, such design documents quickly become out-of-date and do not reflect the current version of the models.

To eliminate the overhead of creating and distributing snapshots of the current versions of models in the form of design documents, we propose a model-to-text transformation approach for producing dynamic web-based views of models – captured atop a range of different modelling technologies – so that stakeholders can be provided with web-based, on-demand and up-to-date access to the models of the system from their web browser and without needing to purchase or install any additional tooling.

The remainder of the paper is organised as follows. Section 2 introduces the proposed transformation-based approach and discusses the details of the technical solution we have developed to realise it. Section 3 provides two case studies that demonstrate using the proposed approach to implement web application for navigating and animating state machine models, and for browsing Ecore metamodels in a Javadoc-like fashion. Section 4 discusses related work and Section 5 concludes the paper and provides interesting directions for further work on the subject.

2 Dynamic Web-Based Model Navigation

To overcome the shortcomings identified above, we propose an approach that allows stakeholders to have direct, on-demand and up-to-date access to the (parts of the) models in which they are interested. Additionally, the approach proposed in this section – like the design document approach – does not require stakeholders to purchase, install, or use any additional software other than their web browser.

To enable web-based access to models that have been defined atop a range of modelling technologies, in this work we have integrated the Epsilon Generation Language [2], which is a template-based model-to-text transformation language, with a Java-based servlet container and web-server (Tomcat). Like most model-to-text transformation languages, EGL was originally designed to support batch code generation; in this work, by integrating EGL with Tomcat, we can use EGL templates as server-side scripts to generate HTML content from models on demand, as displayed in Figure 1.

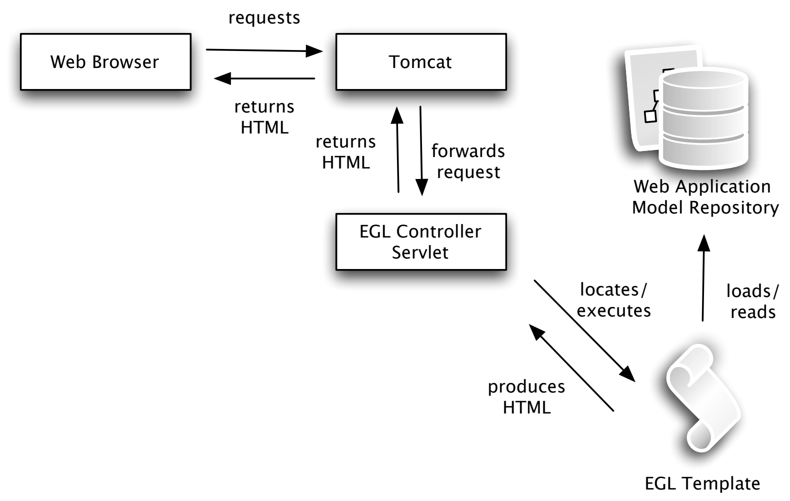


Fig. 1: EGL as an on-demand server-side scripting language in Tomcat

In the following sections we provide a brief overview of EGL and its underlying infrastructure and discuss the process and challenges involved in using it as a server-side scripting language. Before detailing the technical aspects of our work, we stress that, although in this work we use EGL and Tomcat as supporting technologies for our implementation, the proposed approach is not bound to a specific model-to-text transformation language or web-server. In principle, it can be implemented using any other model-to-text transformation language, such as Xpand[3], MOFScript[4] and Acceleo[1]. We selected EGL and Tomcat due to our technical expertise with them.

2.1 The Epsilon Generation Language

EGL is a template-based model-to-text transformation language implemented atop the Epsilon model management platform [5]. Epsilon provides a layered architecture that enables the construction of interoperable task-specific model management languages for tasks such as model transformation, validation, comparison, merging and refactoring. To enable model management languages built atop it to manage models captured using different modelling technologies, Epsilon provides an abstraction layer called EMC (Epsilon Model Connectivity) which specifies an API against which *drivers* for different modelling technologies are implemented. To date, EMC *drivers* for technologies such as EMF, MDR, Z (through CZT [6]), and plain XML have been implemented – and as such, all

[1] www.eclipse.org/acceleo

model management languages in Epsilon can manage models captured with all of these technologies. A more detailed discussion on EMC is available in [7].

As EGL is implemented atop Epsilon, it is interoperable with all of the modelling technologies listed above. Therefore, although in this work we demonstrate using EGL to build web applications around EMF-based models, any of the supported modelling technologies can also be used.

2.2 Integration with Apache Tomcat

Apache Tomcat is a widely used web server and Java servlet container. Tomcat comes with built-in support for the Java Server Pages (JSP) server-side scripting language for producing dynamic web pages, but also provides a flexible architecture which allows developers to extend it with support for additional server-side languages. To integrate EGL with Tomcat, we added a new *servlet mapping* that instructs Tomcat to redirect all requests for URLs that end with *.egl* to a dedicated controller servlet that is responsible for processing these requests.

As illustrated in Figure 1, when Tomcat receives a request for a URL that ends with *.egl*, it forwards the request to the EGL controller servlet which in turn locates and parses the respective EGL template, and if no errors occur during parsing, it executes the template and returns the produced text to Tomcat – which finally returns it to the browser. Similarly to the majority of server-side programming languages, EGL templates have access to a number of predefined variables for accessing *request* parameters and setting/getting *session* and *application* properties. Moreover, since the expression language on which EGL builds can reflectively access Java objects, EGL templates can interoperate seamlessly with existing Java libraries, and can be used in the context of frameworks such as Apache Struts[2], which facilitates the creation of J2EE applications.

Accessing Models To minimise the overhead of loading and storing models in individual EGL templates, each web application is provided with a dedicated model repository which EGL templates can access – to load and store models – via the built-in *modelManager* variable. The application model repository caches models so that they can be readily accessed by all of the EGL templates in an application. Templates have read/write access to the models in the repository; however, the ability for multiple users to modify models in the repository concurrently depends on whether the underlying modelling technology is thread-safe or not. In the current EGL–Tomcat integration, only support for EMF models has been implemented, and as EMF is not thread-safe, all of the applications that we have constructed so far have read-only access to the underlying models.

Template Factories EGL provides several types of built-in template [2]. For example, *EglTemplate* is used for generating plaintext and *EglFileGeneratingTemplate* for generating files on disk. Templates are accessed via the built-in *TemplateFactory* variable. Extenders of EGL can also specify their own template

[2] http://struts.apache.org/

types and their own template factories for capturing and re-using other code generation logic. For instance, the example described in Section 3.1 uses a custom template type and factory to produce an image file from the text generated by a template.

The dedicated EGL servlet queries the metadata of each web application to determine which type of template factory will be used to execute the templates of that application. Users can specify the fully-qualified Java class name of the template factory for their application as a parameter to the EGL servlet definition.

Caching To facilitate scalability of the applications developed atop the approach described in this section, the EGL servlet provides two types of caching, which can be used in web-based EGL applications. Caching is achieved from EGL templates via a built-in *cache* object.

Page Caching allows repeated requests for the same URL to be served without invoking any EGL templates. The EGL controller servlet maintains a cache that maps requests (URLs) to responses (the HTML generated by invoking an EGL template). A response is cached the first time that it is requested, and subsequent requests for to same URL are served from the cache. Applications can control which requests should be cached via the built-in *cache* object. Similarly, a URL can be marked as expired using the built-in *cache* object, and the next request for that URL will be served by invoking an EGL template rather than from the cache.

Fragment Caching allows unchanging page elements – such as headers and footers – to be shared between requests for different URLs, and requires that the EGL application be decomposed into separate subtemplates. The EGL controller servlet maintains a cache that maps subtemplates to partial responses (part of the HTML generated for a request to a particular URL). As with page caching, applications can control the fragment caching strategy via the built-in *cache* object, which provides methods for caching and expiring fragments. The case study in Section 3 uses fragment caching to cache headers, footers and sidebars.

Page caching results in less server-side processing than fragment caching, but is more brittle. For example, consider the effects of changes to a model on page and fragment caches. Pages and fragments that have been affected by changes to a model must be expired (removed from the cache). In the worst case then, the page and fragment caches must be completely emptied to ensure that stakeholders can view and navigate the updated model. The page cache will become fully populated only when every page of the web application is visited. By contrast, fragments for say, shared headers and footers, are cached by a request to any page, and future requests to any other page benefit from the cached fragment.

3 Case Studies

In this section, we present two case studies that demonstrate using the proposed approach for browsing behavioural and structural models. The first case study employs the proposed approach to visualise state machines and concentrates on automated diagram generation. The second case study illustrates a web application for navigating Ecore metamodels in a Javadoc-like style, through which we demonstrate the caching mechanisms discussed in Section 2.2 and evaluate the scalability of our implementation.

3.1 Visualisation and Animation of State Machine Models

Figure 2 shows the metamodel for a Mealy machine. A Mealy machine is finite-state machine that produces an output string in response to an input string. Each transition between states matches a character from the input string and generates an output character when fired. A Mealy machine must have one initial state and the execution ends when the input string has been completely parsed.

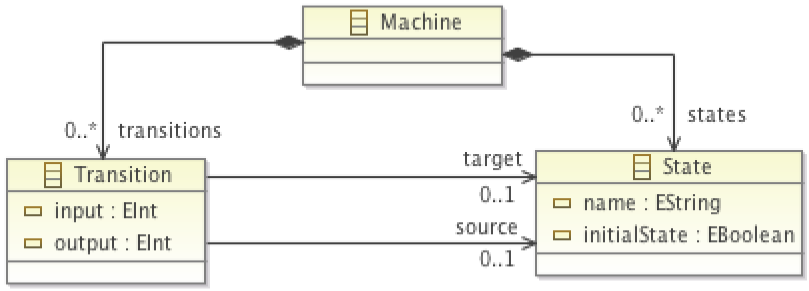


Fig. 2: The metamodel for a Mealy machine

In order to facilitate animation of the Mealy machine, asynchronous calls to two different server-side EGL scripts are required – one to generate an image of the model's current state, and another to request the output of the machine upon firing the selected transition. The user initially requests a standard HTML page, which, upon loading, makes a request to `MealyHandler.egl` to generate an image of the machine in its initial state. Clicking on a successor state will cause another request to `MealyHandler.egl` to return the image representing the machine in the new state, and will also make a request to `MealyOutput.egl` to display the results of executing that transition.

Before discussing the two templates used to animate the Mealy machine, we first briefly summarise the structure of a typical EGL template. EGL templates comprises dynamic sections, which contain executable code, and static sections, which contain text to be emitted. Consider Listing 1.2. Dynamic sections, such

```
1  [% modelManager.registerMetamodel('MealyMachine.ecore');
2    modelManager.loadModel('Example', 'My.mealymachine', 'MealyMachine');
3
4    var currentState = null;
5
6    if (request.getParameter('st').isDefined()) {
7      currentState = request.getParameter('st');
8    } else {
9      currentState = State.all.select(s|s.initialState==true).first().name;
10   }
11
12   var dotTemplate : Template := TemplateFactory.load('Mealy2Dot.egl');
13
14   dotTemplate.populate('currentState', currentState);
15   dotTemplate.generate("mealy_" + currentState + ".svg"); %]
```

Listing 1.1: Generating an SVG image of the Mealy machine

```
1  digraph G{
2    center=true;
3    nodesep=1.5;
4
5    node [ shape=circle, fontname=Tahoma, fontsize=10 ];
6      [%for (s in State.all) { %]"[%=s.name%]";[% }%]
7    "[%=currentState%]" [ color=deepskyblue, style=filled, fillcolor=lightgrey
         ];
8
9    [% for (t in Transition.all.select(t|t.source.name=currentState)) {%]
10     "[%=t.target.name%]" [color=deepskyblue, href="javascript:top.
           doTransition('[%=currentState%]', '[%=t.target.name%]')" ]
11   [%}%]
12
13   [% for (t in Transition.all) {%]
14     "[%=t.source.name%]" -> "[%=t.target.name%]" [ label="[%=t.input%]/[%=t.
           output%]" ]
15   [%}%]
16 }
```

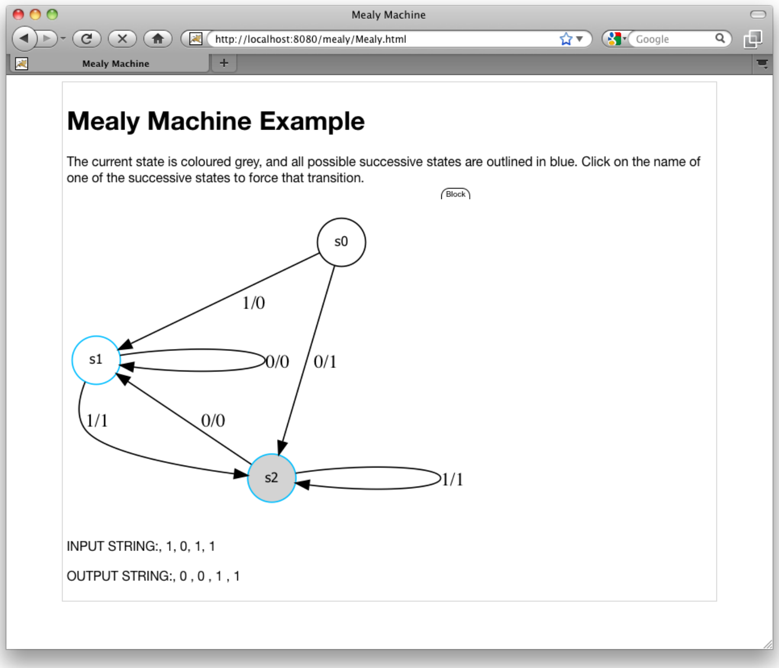
Listing 1.2: Generating the Graphviz representation of the Mealy machine

as the one on line 9, are enclosed in `[% %]` tags; static sections, such as the one on lines 1-5, are not enclosed in `[% %]` tags. Dynamic sections can take an alternate form, using a `[%= %]` tag (such as the one on line 7) to emit a dynamically computed value (the `currentState` variable in this case).

`MealyHandler.egl` (listing 1.1) generates the image of the machine using `Mealy2dot.egl`. The call to `generate` (line 15) delegates to an image generating template factory, which invokes DOT to convert the output of `Mealy2dot.egl` to an SVG (Scalable Vector Graphics) image.

`Mealy2dot.egl` (listing 1.2) outputs a Graphviz[3] DOT language description of the machine and the template factory executes the DOT description, creating an SVG file. Animation is achieved by passing the current state name as a parameter in the URL query string to `MealyHandler.egl` (e.g. `MealyHandler.egl?st=s0`) which populates the `currentState` variable in the

[3] www.graphviz.org



Mealy Machine Example

The current state is coloured grey, and all possible successive states are outlined in blue. Click on the name of one of the successive states to force that transition.

INPUT STRING:, 1, 0, 1, 1

OUTPUT STRING:, 0 , 0 , 1 , 1

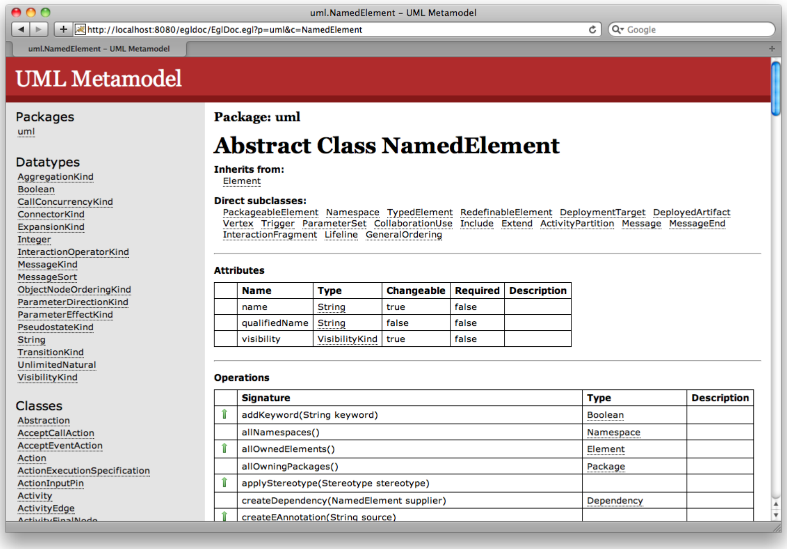
Fig. 3: Example Mealy machine animation

`Mealy2dot.egl` template (listing 1.1, line 14). `Mealy2dot.egl` highlights the current state by changing its background colour, and assigns URLs to any nodes reachable by outgoing transitions from the current state, making them clickable in the generated image. The URLs execute a JavaScript function, `doTransition(src,tgt)`, which handles the asynchronous request to generate the new image and replace the existing image with the newly generated one.

After the asynchronous request to generate the image has been made, `doTransition()` also makes an asynchronous request to `MealyOutput.egl` which returns a string containing the input and output strings from the transition. These are then printed underneath the image, showing the input and output history of the animation. Figure 3 shows the output of animating a Mealy machine over four transitions.

3.2 Navigation of Ecore metamodels

This section presents a web-application for on-demand navigation of Ecore metamodels in a Javadoc-like style and explores the way in which the scalability of the proposed approach is affected by the server-side processing load and the type of caching employed. The results presented in this section suggest that the



uml.NamedElement – UML Metamodel

http://localhost:8080/egldoc/EglDoc.egl?p=uml&c=NamedElement

UML Metamodel

Packages
uml

Datatypes
AggregationKind
Boolean
CallConcurrencyKind
ConnectorKind
ExpansionKind
Integer
InteractionOperatorKind
MessageKind
MessageSort
ObjectNodeOrderingKind
ParameterDirectionKind
ParameterEffectKind
PseudostateKind
String
TransitionKind
UnlimitedNatural
VisibilityKind

Classes
Abstraction
AcceptCallAction
AcceptEventAction
Action
ActionExecutionSpecification
ActionInputPin
Activity
ActivityEdge

Package: uml

Abstract Class NamedElement

Inherits from:
Element

Direct subclasses:
PackageableElement Namespace TypedElement RedefinableElement DeploymentTarget DeployedArtifact Vertex Trigger ParameterSet CollaborationUse Include Extend ActivityPartition Message MessageEnd InteractionFragment Lifeline GeneralOrdering

Attributes

	Name	Type	Changeable	Required	Description
	name	String	true	false	
	qualifiedName	String	false	false	
	visibility	VisibilityKind	true	false	

Operations

	Signature	Type	Description
↑	addKeyword(String keyword)	Boolean	
	allNamespaces()	Namespace	
↑	allOwnedElements()	Element	
	allOwningPackages()	Package	
↑	applyStereotype(Stereotype stereotype)		
	createDependency(NamedElement supplier)	Dependency	
↑	createEAnnotation(String source)		

Fig. 4: EglDoc for the NamedElement class of UML 2.2 [9].

approach proposed in Section 2 is well-suited to viewing and navigating models via the web for a significant number of concurrent users.

EglDoc: Metamodel Documentation This section demonstrates the approach proposed in Section 2 by describing the way in which an existing EGL application, EglDoc [8, Ch.5], has been ported to provide web-based model navigation and view extraction. EglDoc generates HTML documentation for Ecore metamodels (and is similar to Javadoc for Java in this respect). Figure 4 shows the output produced by EglDoc for the `NamedElement` class of the UML 2.2 [9] metamodel. EglDoc can be applied to produce documentation for any Ecore metamodel.

EglDoc comprises several templates, which are used to generate the header, footer, navigation bar and content of each page. An extract of the EGL template that generates documentation for attributes is shown in Listing 1.3. EGL is a template-based model-to-text language. Listing 1.3 generates an HTML table of attributes, listing the name and type of each attribute. Lines 10-12 generate a link to other pages, using the `toUrl()` operation on lines 19-21.

The existing version of EglDoc generates one HTML file for each element of the Ecore metamodel. Porting EglDoc to interoperate with the web-based model navigation and view extraction approach proposed in Section 2 involved parameterising the existing EGL templates to facilitate the identification of metamodel elements via the URL of a request to the web server. For example, a request for

```
[% if (class.eAllAttributes.size() > 0) { %]
  <h4>Attributes</h4>
  <table cellspacing="0">
    <tr> <th>Name</th> <th>Type</th> </tr>
    [% for (attribute in class.eAllAttributes.sortBy(a|a.name)) { %]
      <td>[%=attribute.name%]</td>

      [% if (attribute.eType.isDefined()) { %]
          <td>
            <a href="[%=attribute.eType.toUrl()%]">
              [%=attribute.eType.name%]
            </a>
          </td>
      [% } else { %]
          <td>&nbsp;</td>
      [% } %]
  [% } %]
[% } %]
[% operation EClassifier toUrl() : String {
    return self.ePackage.name + '-' + self.name + '.html';
} %]
```

Listing 1.3: View extraction for attributes in EGLDoc

```
1  [%
2      modelManager.registerMetamodel('Ecore.ecore');
3      modelManager.loadModel('Sample', 'UML.ecore', 'http://www.eclipse.org/emf
           /2002/Ecore');
4
5      if (request.getParameter('p').isDefined()) {
6        package = EPackage.all.selectOne(p|p.name = request.getParameter('p'));
7      }
8  %]
```

Listing 1.4: Model loading in EGLDoc

the URL `EglDoc.egl?p=uml&c=NamedElement` to the web-based EglDoc returns HTML for the `NamedClass` class of the `uml` package (in the UML 2.2 metamodel). Listing 1.4 shows the way in which the UML metamodel is loaded (lines 2-3) and the `p` parameter of the request is interpreted (lines 6-8). Note the use of the *modelManager* built-in variable for loading a model, which was discussed in 2.2. In Listing 1.4, the model to be loaded (`'UML.ecore'`) is hard-coded for clarity. In practice, the location of the model is specified as a URI, which can be configured by the user.

4 Related Work

Several mature Java-based template languages such as JSP, Velocity[4] and FreeMarker[5] are available as server-side scripting languages for Java-based web servers such as Tomcat. In principle we could have used any of these languages

[4] http://velocity.apache.org
[5] http://freemarker.sourceforge.net

in combination with the reflective API of EMF – or using code generated from the respective Ecore metamodels in order to achieve a more concise syntax. For dynamic languages such as Velocity and FreeMarker, developers could even implement EMF-specific extensions to enable a concise navigation style without needing to generate code from the Ecore metamodels. However, compared to these languages, we strongly believe that EGL is more suitable for the task as it provides first-order logic OCL-based collection navigation operations, built-in support for accessing mutliple models concurrently, and a number of existing *drivers* for interacting with a number of modelling technologies. Even more importantly, since the model connectivity framework discussed in Section 2.1 provides a uniform interface for different modelling technologies, the underlying modelling technology can be substituted later on if necessary (e.g. switch to a database-backed model serialisation format for performance reasons) without requiring changes to the EGL templates.

The Web 2.0 MetaModelbrowser[6] is a web application that can visualise Ecore metamodels and their instances using a fixed tree-based interface that closely mimics the Eclipse-based EMF reflective tree editor. In contrast to MetaModelbrowser, the approach proposed in this paper allows developers to implement custom interfaces for displaying models to end users. Also, using the Graphviz extension our approach enables developers to also embed automatically-generated diagrams to their model browsing web applications.

5 Conclusions and Further Work

In this paper we have presented an approach for re-using a model-to-text transformation language in order to provide support for web-based model navigation and view extraction. Using this approach, non-technical stakeholders can have access to the latest versions of the models of the system from their browser, without needing to purchase and install additional software. Moreover, using such an approach, access control can be enforced if necessary.

With the advent of cloud computing, we believe that web-based model management is a promising field of study with significant potential for practical real-world impact. A few of the open issues that we have identified through this work include management of very large models, concurrent modification of models, caching, and access control. In the future, we will investigate related work in areas such as the management of very large databases and web-based technologies, and adapt best-of-breed approaches to solve the respective problems in the field of web-based model management.

Acknowledgements. The work in this paper was supported by the European Commission via the MADES and INESS projects, co-funded under the 7th Framework programme (grants #218575 (INESS), #248864 (MADES)). We would also like to thank Darren Clowes, Chris Holmes, Julian Johnson, Ray

[6] http://www.metamodelbrowser.org/

Dawson, and Steve Probets from BAE Systems and Loughborough University for their contributions to earlier work that led to the approach presented in this paper.

References

1. Darren Clowes, Dimitrios S. Kolovos, Chris Holmes, Louis Rose, Richard Paige, Julian Johnson, Ray Dawson, and Steve Probets. A Reflective Approach to Model Driven Web Engineering. In *Proc. 6th European Conference on Modelling Foundations and Applications (ECMFA)*, Paris, France, 2010 2010.
2. Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, Fiona A.C. Polack. The Epsilon Generation Language (EGL). In *Proc. European Conference in Model Driven Architecture (ECMDA)*, 2008.
3. Sven Efftinge. XPand Language Reference. http://www.eclipse.org/gmt/oaw/doc/4.1/r20_ xPandReference.pdf.
4. Jon Oldevik. MOFScript User Guide. http://www.eclipse.org/gmt/mofscript/doc/ MOFScript-User-Guide.pdf.
5. Eclipse Foundation. Epsilon Modeling GMT component. http://www.eclipse.org/gmt/epsilon.
6. Community Z Tools. http://czt.sourceforge.net.
7. Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige. *The Epsilon Book.* 2008. http://www.eclipse.org/gmt/epsilon/doc/book/.
8. Louis M. Rose. A text-generation language for Epsilon. Master's thesis, University of York, 2007.
9. OMG. Unified Modelling Language 2.2 Specification [online]. [Accessed 10 February 2011] Available at: http://www.omg.org/spec/UML/2.2/, 2007.