

# Seamless Navigation of Heterogeneous EMF models in Epsilon

Dimitrios S. Kolovos, Nicholas Drivalos,  
Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science,  
University of York,  
York, YO10 5DD, UK.  
{dkolovos, nikos, paige, fiona}@cs.york.ac.uk

**Abstract.** Model weaving is a particularly useful technique for a number of scenarios in an MDE process such as model traceability, decomposition and annotation. In this paper we propose a technical solution that allows model management languages of the Epsilon GMT component to seamlessly navigate woven models by exploiting the information captured in the respective weaving models.

## 1 Introduction

Model weaving [1] is the technique of establishing well-defined links between elements belonging to different models (called *woven* models), and has been shown to be useful in a number of MDE scenarios including inter-model traceability, model decomposition and annotation. The problem of establishing links between different models in EMF has been satisfactorily solved as EMF natively supports cross-model references, and other solutions such as GMT AMW are also available. What has not been investigated so far is how to exploit weaving models for seamlessly navigating across woven models in model management languages (e.g. OCL, M2M and M2T languages).

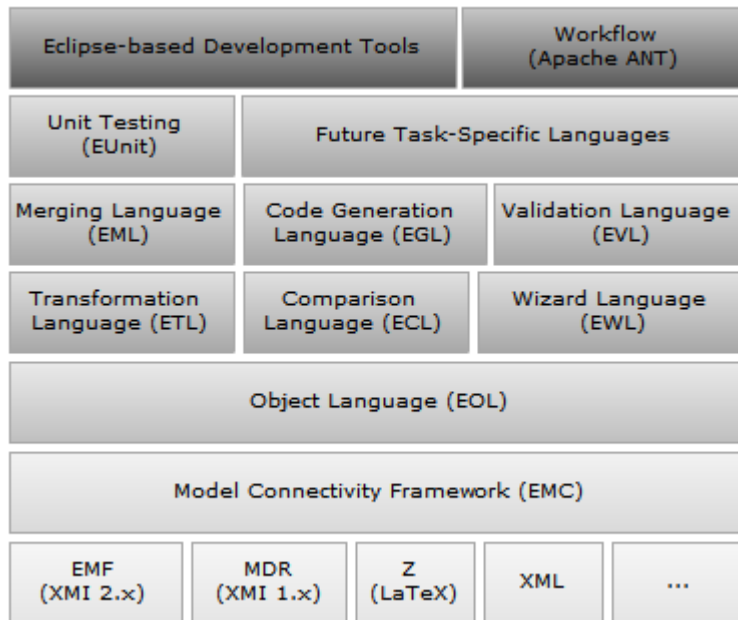
In this paper we present a technical solution implemented in the context of the Epsilon GMT component which allows users of languages built atop Epsilon to use weaving models in order to seamlessly navigate across the woven models. The rest of the paper is organized as follows. In section 2 we provide a brief description of Epsilon. In section 3 we present a motivating example. In section 4 we provide insights of the model connectivity and property navigation framework that underpins Epsilon and in section 5 we demonstrate how this framework has been used to provide support for seamlessly navigating across woven models. Finally, section 6 concludes this work and provides directions for further research and experimentation on the topic.

## 2 Epsilon

Epsilon is a component of the Eclipse GMT project [2] that provides infrastructure for implementing uniform and interoperable model management languages.

It can be used to manage models of diverse metamodels and technologies. At the core of Epsilon is the Epsilon Object Language (EOL) [3], 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 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 is used to provide a uniform interface for models of different modelling technologies. Currently, EMC drivers have been implemented to support EMF [4] (XMI 2.x), MDR [5] (XMI 1.x), Z [6] specifications in LaTeX using CZT [7] and XML. Also, to enable users to compose workflows that involve a number of individual model management tasks, Epsilon provides ANT [8] tasks and an inter-task communication framework discussed in detail in [9].



**Fig. 1.** Overview of the architecture of Epsilon

### 3 Motivation

To motivate the discussion we present a simple case where we need to annotate a UML 2 activity model with performance-related information for simulation purposes, and we do not wish to achieve this using the UML2 profiles mechanism. To achieve this we need to define a new weaving metamodel<sup>1</sup> as displayed in Listing 1.1.

**Listing 1.1.** The `UmlActivityPerformance` weaving metamodel expressed using the Emfatic textual syntax for ECore

```
@namespace(uri="UmlActivityPerformance", prefix="")
package UmlActivityPerformance;

import "http://www.eclipse.org/uml2/2.1.0/UML";

class ActivityPerformance {
    val ActivityPerformanceElement[*] contents;
}

abstract class ActivityPerformanceElement {}

class ExecutableNodePerformance extends ActivityPerformanceElement {
    ref uml.ExecutableNode executableNode;
    attr Integer cpuTime = 0;
}

class ControlFlowProbability extends ActivityPerformanceElement {
    ref uml.ControlFlow controlFlow;
    attr Float probability = 1;
}
```

The `UmlActivityPerformance` metamodel includes the container `ActivityPerformance` EClass, the `ExecutableNodePerformance` EClass, instances of which can capture the CPU time that a UML `ExecutableNode` will take to execute, and the `ControlFlowProbability` EClass, instances of which can capture the probability of a control flow happening (when a node has multiple alternative outgoing control flows).

Having defined the `UmlActivityPerformance` metamodel, we can construct two models (named `UML` and `Performance`) that capture a UML activity model and the related performance information respectively. When navigating the two models programmatically using EOL, to retrieve and print the `cpuTime` of a given instance of `ExecutableNode` (named `exNode`) we need to compose a query like the following:

---

<sup>1</sup> Although the metamodel of Listing 1.1 does not follow the technical implementation proposed in [1], it is still considered to be a weaving metamodel in the broad sense, as its instances link elements from different models.

**Listing 1.2.** Printing the `cpuTime` of the `exNode` `ExecutableNode`

```
Performance!ExecutionNodePerformance.allInstances.selectOne(  
  enp|enp.node = exNode).cpuTime.println();
```

Since *exNode* does not hold a reference to the respective *ExecutableNodePerformance* instance that captures its related *cpuTime*, we need to iterate all instances of *ExecutableNodePerformance* in the *Performance* model, find one which has its *node* set to *exNode* and return the value of its *cpuTime* feature. In fact, we may also need to check if such an *ExecutableNodePerformance* exists at all before attempting to retrieve its *cpuTime* if a respective constraint does not enforce completeness.

In this paper we present how we have automated this task so that developers of EOL - and languages built atop it - programs can now use the following simple syntax to achieve the same navigation, without needing to modify the base metamodels (e.g. the UML metamodel in this case).

**Listing 1.3.** Printing the `cpuTime` of `exNode` with a simpler syntax

```
exNode.cpuTime.println();
```

## 4 Model Connectivity and Property Navigation in EOL

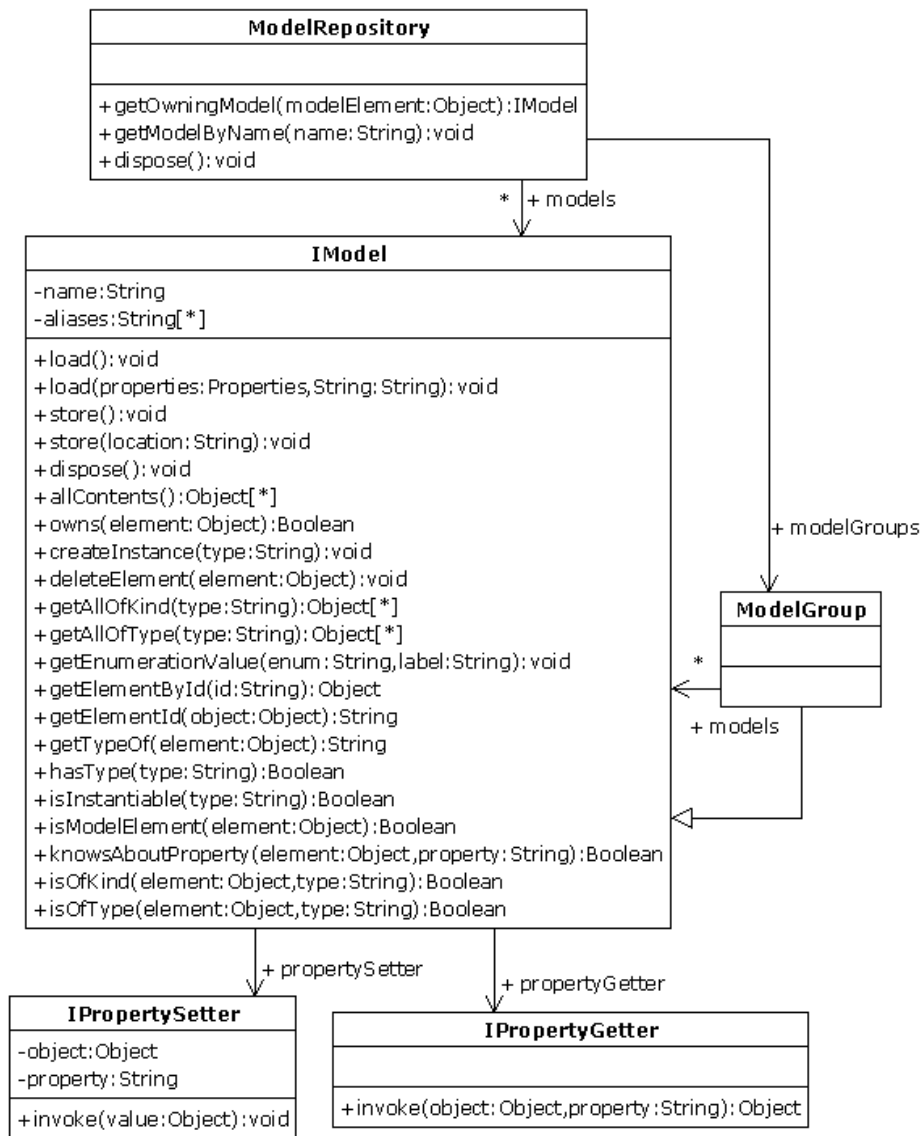
At runtime an EOL module is associated with a runtime context. The context, among others, contains a model repository which as displayed in Figure 2 contains a collection of named models that implement the *IModel* interface and which are available at runtime.

Of particular interest is the *knowsAboutProperty(Object instance, String property)* method which returns if the model can get/set the value of the property of a specific instance. If a model responds positively, it must provide an instance of *IPropertyGetter/Setter* to which this task can be delegated. Therefore, when the runtime needs to access the value of a property *y* of an object *x*, it iterates all models in the model repository and asks each one if it knows about this property of the object. If one responds positively, it asks for the respective *IPropertyGetter/Setter* and invokes it.

This approach to getting/setting property values delivers a number of benefits. In terms of memory usage, it avoids using wrappers around native model elements (*EObjects* in EMF, *RefObjects* in MDR etc) thus keeping the memory footprint low. In terms of flexibility, elements of the same native type (e.g. *EObject*) can be treated in very different ways according to the model in which they are contained. This has been demonstrated in [10] where an implementation of *IModel* is used to enable accessing M0-level *models* in EMF.

The mainstream implementation of *IModel* that can handle EMF models is called *EmfModel*. *EmfModel.knowsAboutProperty(Object instance, String property)* returns true if all of the following are satisfied:

- instance is an `EObject`
- instance belongs to *getAllContents()* of the underlying Resource



**Fig. 2.** The Model Connectivity Architecture of Epsilon

- the EClass of instance (*instance.eClass()*) has a structural feature with that name

Therefore, in our example - with the implementation of *EmfModel* - both the *UML* and the *Performance* models would return *false*; *UML* because *ExecutableNode* does not have a feature called *cpuTime*, and *Performance* because *exNode* does not belong to the result of *getAllContents()* of its *Resource*.

## 5 Introducing the EmfBridgeModel Specialization

To achieve the envisioned syntax (*exNode.cpuTime*) we have implemented a specialization of *EmfModel*, called *EmfBridgeModel*. The metamodel of an *EmfBridgeModel* can use the *bridge* and *bridge.end* annotations for *EClasses* and *EReferences* respectively to denote that particular *EClasses* and *EReferences* should be treated as navigation bridges and bridge ends respectively.

In *EmfBridgeModel* the *knowsAboutProperty(Object instance, String property)* method returns true if:

- *instance* is an *EObject*
- *instance* belongs to *getAllContents()* of the underlying *Resource*
- **or** there exists an *EClass* in the metamodel of the *Resource* that is annotated as *bridge*, and which has the following properties
  - it contains an *EReference* annotated as *bridge.end* and its type which is compatible with the *EClass* of *instance*
  - it has an *EStructuralFeature* with the name of the *property*

Listing 1.4 demonstrates the annotated version of the *UMLActivityPerformance* metamodel originally displayed in Listing 1.1. In this annotated version *ExecutableNodePerformance* and *ControlFlowProbability* have been annotated with the *bridge* annotation while *executableNode* and *controlFlow* have been annotated with the *bridge.end* annotation.

**Listing 1.4.** The annotated UMLActivityPerformance weaving

```
@namespace(uri="umlactivityperformance", prefix="")
package umlactivityperformance;

import "http://www.eclipse.org/uml2/2.1.0/UML";

class ActivityPerformance {
    val ActivityPerformanceElement[*] contents;
}

abstract class ActivityPerformanceElement {}

@bridge
class ExecutableNodePerformance extends ActivityPerformanceElement {
    @bridge.end
    ref uml.ExecutableNode executableNode;
    attr Integer cpuTime = 0;
}

@bridge
class ControlFlowProbability extends ActivityPerformanceElement {
    @bridge.end
    ref uml.ControlFlow controlFlow;
    attr Float probability = 1;
}
```

Therefore, when *exNode.cpuTime* is now evaluated, the runtime first asks the *UML* model if it knows about this object/property combination and - for the same reasons discussed earlier - it responds negatively. However, when the *Performance* model is now asked, it introspects the *UmlActivityPerformance* EPackage and discovers that the *ExecutableNodePerformance* EClass is annotated as *bridge* and also contains a *bridge.end* EReference (executableNode) typed as *uml.ExecutableNode*, as well as an attribute that matches the required name (*cpuTime*) and therefore it responds positively. Once the property getter is invoked, if it finds an instance of *ExecutableNodePerformance* the *node* feature of which is set to *exNode*, it returns the value of its *cpuTime* attribute; otherwise it returns the default value of the *cpuTime* attribute (0).

Due to the hierarchical organization of languages in Epsilon, which was discussed in section 2, this feature is inherited automatically to the rest of the languages of the platform and can therefore be used in model management tasks such as model validation, transformation and text generation without any additional effort.

## 6 Conclusions and Further Work

The aim of this work was to enable seamless navigation between heterogeneous linked EMF models using EOL. The preliminary results are encouraging as they have proved that the model connectivity framework that underpins EOL - and Epsilon in general - is flexible enough to achieve this without modifications. However, in its current state the implementation of *EmfBridgeModel* performs repeated look-ups in the underlying EMF resource, which - particularly for large models - are quite time-consuming. We are currently experimenting with establishing and maintaining a reverse index structure to speed up this process and expect to report back on this in the near future.

## 7 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).

## References

1. Marcos Didonet Del Fabro, Jean Bezivin, Frederic Jouault, Erwan Breton, Guillaume Gueltas. AMW: A Generic Model Weaver. In *Proceedings of IDM05*, 2005.
2. Eclipse GMT - Generative Modeling Technology, Official Web-Site. <http://www.eclipse.org/gmt>.
3. 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.
4. Eclipse Foundation. Eclipse Modelling Framework. <http://www.eclipse.org/emf>.

5. Sun Microsystems. Meta Data Repository. <http://mdr.netbeans.org>.
6. Jim Woodcock and Jim Davies. *Using Z : Specification, Refinement, and Proof*. Prentice Hall, March 1996.
7. Community Z Tools. <http://czt.sourceforge.net>.
8. The Apache Ant Project. <http://ant.apache.org>.
9. 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.
10. Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. Aligning OCL with Domain-Specific Languages to Support Instance-Level Model Queries. *Electronic Communications of the EASST*, 2007.