Eclipse Development Tools for Epsilon

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

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

Abstract. Epsilon is a platform that provides the necessary infrastructure for developing task-specific languages to support model management tasks such as transformation, merging, comparison and validation. Currently, a number of task-specific languages are implemented atop Epsilon. Each language is supported by an execution engine and a set of development tools for the Eclipse platform, that enable developers to compose, execute and debug specifications in a user-friendly and practical environment. In this paper, we provide an overview of the Epsilon infrastructure and the languages build atop it and demonstrate the supporting development tools.

1 Introduction

Model Driven Development aims at raising the level of abstraction, at which software is developed, by promoting models (as opposed to code) into first-class artefacts [3]. To achieve this, MDD requires a set of Model Engineering (ME) facilities that enable developers to manage their models in an automated and efficient manner. Typical model management tasks include validation, transformation, comparison, merging, text generation, reverse engineering and version reconciliation. As discussed in [7], all those activities share common requirements for features such as model navigation (and possibly modification), but each also demonstrates unique patterns and abstractions. To address the increasing need for automated model management, a number of task-specific languages such as OCL [13] for validation, QVT [12] and ATL [9] for transformation and MOFScript [2] for text generation have been proposed.

As discussed in [7], the majority of contemporary model-management languages conceptually build on a subset of OCL for navigating models, but in practice implement everything from scratch. This inevitably causes a number of practical problems:

1. Developing a language for a new task (e.g. model merging, model comparison) is particularly time and resource-intensive.
2. There is duplication of effort across the implementation of tool-support for different languages (e.g. to support OCL navigation features or different modelling frameworks such as MDR and EMF).
3. Because significant time and effort are consumed on re-implementing trivial features, important task-specific features are left under-developed (e.g. rule-inheritance in transformation languages).

4. Each language defines its own development tools, thus making the user experience inconsistent across different languages (e.g. different interfaces to loading models, inconsistent error-messages).

To address these issues, the Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon) [4] provides a layered architecture that facilitates extensive reuse both at the language definition and the tool-support level. The rest of the paper is organized as follows. In Section 2, we discuss the architecture of Epsilon and provide a brief overview of the task-specific languages implemented so far. In Section 3, we present the architecture and features of the Eclipse-based Epsilon Development Tools. Finally, in Section 4, we conclude and discuss our plans for extending and improving the tool-support for languages of the Epsilon platform.

2 Overview of Epsilon

Epsilon is a platform for developing task-specific model management languages. It has been designed and implemented at the University of York, and recently joined the Generative Modeling Technology (GMT) [1] Eclipse sub-project. The purpose of Epsilon is to address the problems discussed in Section 1 by providing infrastructure, on which task-specific languages and supporting tools can be developed with minimal effort and overlap. In this section, we discuss on this infrastructure as well as on the task-specific languages that have been developed atop it so far. A high-level view of the architecture of Epsilon is illustrated in Figure 1.

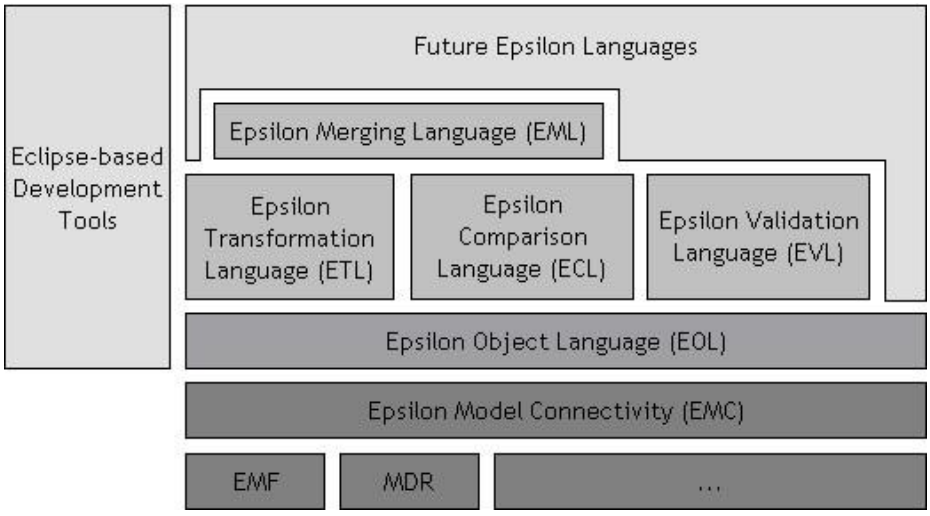

Fig. 1. Architecture of Epsilon

2.1 Infrastructure

The infrastructure of Epsilon consists of the Epsilon Model Connectivity (EMC) layer and the Epsilon Object Language (EOL).

2.1.1 Epsilon Model Connectivity (EMC) There are currently many technologies for representing and serializing models. The most prominent examples are the Meta-Data Repository (MDR) [14], an open source product from Sun Microsystems that implements MOF 1.4 and XMI 1.2, and the Eclipse Modeling Framework (EMF) [8] that implements MOF 2.0 and XMI 2.0. Moreover, there are other types of structured artefacts, such as WSDL descriptors, BPML and OWL documents, which can be perceived and managed as models. To embrace such diverse representations in a uniform manner, in Epsilon we have introduced the *Epsilon Model Connectivity (EMC)* layer[1]. EMC provides a set of interfaces (driver specifications) that can be implemented to bridge a specific modelling technology with the Epsilon languages. So far, we have implemented EMC *drivers* that support EMF, MDR and XML models.

2.1.2 Epsilon Object Language (EOL) EOL [7] is an action language that builds on the navigational mechanisms of OCL, but also supports useful features such as statement sequencing, multiple model access, conventional programming constructs (e.g. for and while loops). While EOL can be used as a standalone language for programmatic model management, its purpose is to be re-used by task-specific languages. This is achieved through grammar-inheritance and the extensible architecture of the EOL execution engine. Detailed discussion on the semantics, the abstract and concrete syntax of EOL as well as a detailed comparison with OCL can be found in [7].

2.2 Task-Specific Languages

Based on EMC and EOL, we have developed a number of task-specific model management languages. Our aim is to investigate the particularities of each task and propose useful abstractions and mechanisms, accompanied by concrete and practical examples. This can advance the state-of-the-art and provide useful feedback for the evolution of current standards (e.g. QVT, MOF2Text) and hopefully trigger future standardization efforts (e.g. on model merging). In the sequel, we present the task-specific languages currently implemented in Epsilon and provide a short discussion and an example of the concrete syntax of each language. For extensive discussion on specific languages, readers can refer to the respective publications.

2.2.1 Epsilon Comparison Language (ECL) ECL is a rule-based language tailored to the task of model comparison. An ECL specification consists of *match-rules* that can compare pairs of elements of the input models. Each *match-rule* comprises of a *compare* and a *conform* part that decide if the elements under comparison match and conform

[1] EMC has been inspired by the Java DataBase Connectivity (JDBC) framework that offers a uniform abstraction layer for diverse databases

with each other. The bodies of the *compare* and *conform* parts of match rules are expressed in EOL as displayed in the exemplar rule of Listing 1.1. Detailed discussion on ECL can be found in [6].

Listing 1.1. Example of an ECL match-rule for comparing a UML Association End with a Relational database Relationship (taken from [6])

```
rule OneToManyAssociationEnd2Relationship
 match ae : UML!AssociationEnd
 with rel : Database!Relationship {

 guard {
  return ae.isMultiple() and
   ae.getOpposite().isMultiple() = false;
 }

 compare {
  return ae.participant.matches(rel.child.owner)
  and ae.getOpposite().participant.
     matches(rel.parent.owner);
 }
}
```

2.2.2 Epsilon Transformation Language (ETL) ETL is a rule-based model transformation language. An ETL module consists of *transform-rules* that can translate elements of the source models into elements in the target models. Similarly to ECL *match-rules*, each *transform-rule* has a declarative signature and an imperative body expressed in EOL. An exemplar ETL *transform-rule* is demonstrated in Listing 1.2.

Listing 1.2. Example of an ETL transform-rule for translating a KM3 [11] class into a UML class

```
rule Class2Class
      transform s : KM3!Class
      to t : UML!Class
      extends ModelElement2ModelElement {

      t.isAbstract := s.isAbstract;

      for (s in s.supertypes){
            def g : new UML!Generalization;
            g.namespace := t.namespace;
            g.parent := s.equivalent();
            g.child := t;
      }

}
```

2.2.3 Epsilon Merging Language (EML) EML addresses the problem of merging models of common or diverse metamodels and technologies. As discussed in [5], EML reuses the concept of *match-rules* from ECL and *transform-rules* from ETL and adds the task-specific concept of *merge-rules* to allow developers compose merging specifications in a structured and comprehensive manner.

Listing 1.3. Simple EML merging specification for merging PIMs with PDMs into PSMs (taken from [5])

```
rule PimTypeMapping
 match pimType : PIM!Class
 with mapping : PDM!PrimitiveTypeMapping {

 guard {
  return pimType.stereotype.exists(s|s.name = 'primitive');
 }

 compare {
  return pimType.name = mapping.independent;
 }

}

auto rule PimTypeToPsmType
 merge pimType : PIM!Class
 with mapping : PDM!PrimitiveTypeMapping
 into psmType : PSM!Class {

 psmType.name := mapping.specific;

}
```

2.2.4 Epsilon Validation Language (EVL) EVL is the latest addition to the family of Epsilon languages and aims at improving the way with which validity constraints on models are expressed. To achieve that, we are experimenting with the concepts of *depenedent* constraints, meaningful error messages and the separation of critical and non-critical constraints (warnings), a distinction also discussed in [10]. An example of the concrete syntax of EVL appears in Listing 1.4. As this is work in progress, there is no published detailed discussion on EVL yet.

Listing 1.4. A constraint and a warning for validation of UML models expressed in EVL

```
context Class {
 constraint HasName :
  self.name <> ''
```

```
  fail : 'The name of a Class in package ' +
    self.namespace.name + ' is empty'

  warning NameStartsWithCapital assumes HasName {
    def firstChar : String;
    firstChar := self.name.toCharSequence().first();
    return firstChar = firstChar.toUpperCase();
  }

  fail : 'The name of Class ' + self.name +
    ' should start with a capital letter'
}
```

3 The Epsilon Development Tools

To be of practical use, each language in the Epsilon platform is supported by a set of development tools: an editor, an outline view and launch configuration interfaces and delegates. In this section, we provide an overview of these facilities in Epsilon and discuss how they are reused/specialized for task-specific languages.

3.1 Editors and Outline Views

To efficiently support editing specifications in task-specific languages, language-specific editors aware of the specialities of the concrete syntax of each language are required. Moreover, to enable users locate and correct syntactical errors in-place markers that can highlight the lines that contain the error(s) are considered particularly useful. To maximize reuse, we have introduced an abstract $AbstractModuleEditor$ class that extends the built-in $TextEditor$ Eclipse editor and provides basic syntax highlighting services for keywords, comments, strings and numbers that apply to all Epsilon languages. Editors of individual languages (e.g. $EclEditor$ and $EtlEditor$) extend this abstract editor to provide their specific keywords (e.g. $compare$ and $transform$ respectively). Similarly, we have introduced a $ModuleContentOutlinePage$ that implements a basic outline view providing services such as linking with editors and sorting (Figure 2, point 5). To specialize for a task-specific language, little customization (specifying icons and labels) is required. As an example, in Figure 2, the editor (1) and outline view (4), that provides a compact view of the transform-rules of an ETL module, are illustrated. Points (2) and (3) show how design-time errors are presented with the use of markers both in the editor and the *problems* view of Eclipse.

3.2 Launch Configuration Interface

Since languages in the Epsilon platform are about managing models, a major task when creating a launch configuration is to define the models it operates on. To achieve reuse and uniformity, we have implemented separate tabs and dialogs, each being able to configure models of a particular technology (e.g. EMF, MDR). These tabs are reused in the

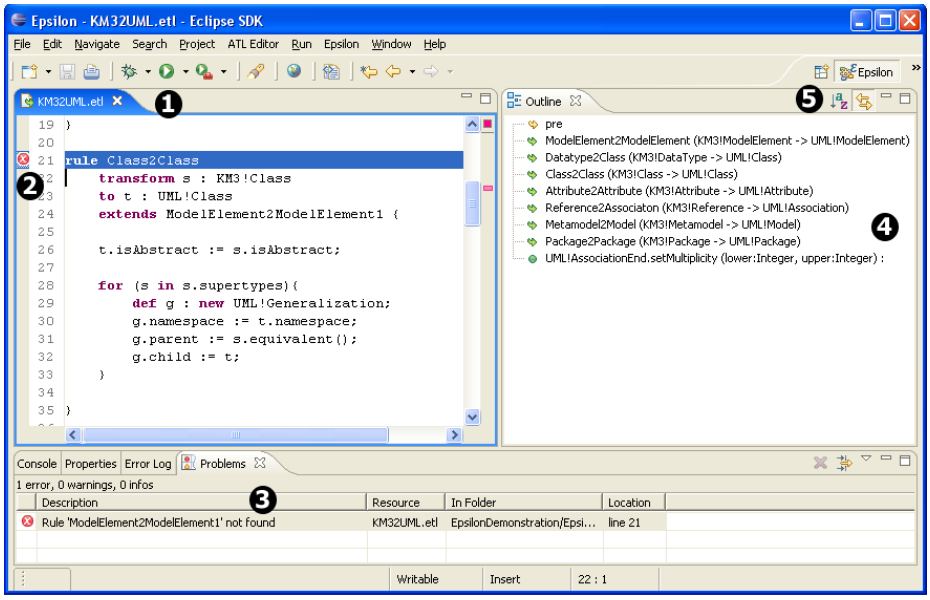

Fig. 2. ETL Editor and Outline View

launch configuration interfaces of all Epsilon languages. In Figure 3, we demonstrate the MDR and EMF model configuration tabs (1) and the dialog through which the name and extent (2), as well as the locations of the model and metamodel file (3), of an MDR model can be specified.

3.3 The Epsilon Console

To provide the user with feedback at run-time, we have implemented a console dedicated to Epsilon languages. Users can send textual messages to the Epsilon console via the built-in $print()$ and $err()$ EOL operations. Moreover, as demonstrated in Figure 4, when a run-time error occurs during the execution of a module, an appropriate message (1) and a hyperlink (2) are displayed in the console to inform the user about the error. Clicking on the hyperling navigates the user to the actual source of the error (3).

4 Conclusions and Further Work

With the infrastructure provided by Epsilon we can design and implement task-specific model management languages and supporting tools for Eclipse in a structured manner with minimal effort and overlap, and enhanced reuse.

Epsilon and the languages that build atop it are under active development and testing. For the future, there are two significant enhancements we are considering implementing; first, we anticipate that Epsilon languages could benefit from the existence of language-specific debuggers for Eclipse. This requirement was conceived early in

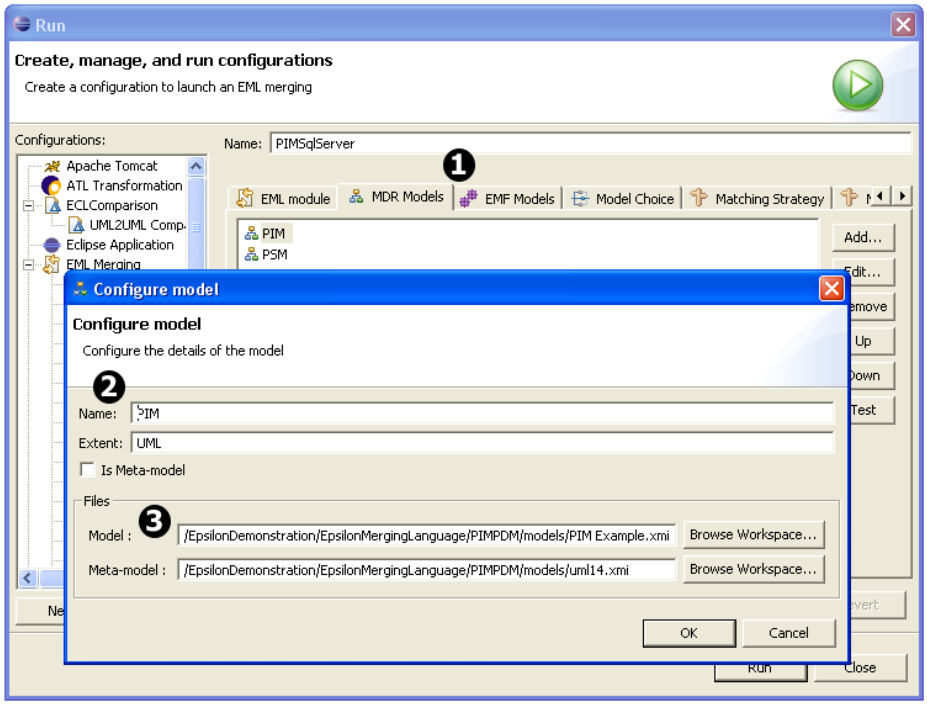

Fig. 3. EML Launch Configuration Interface

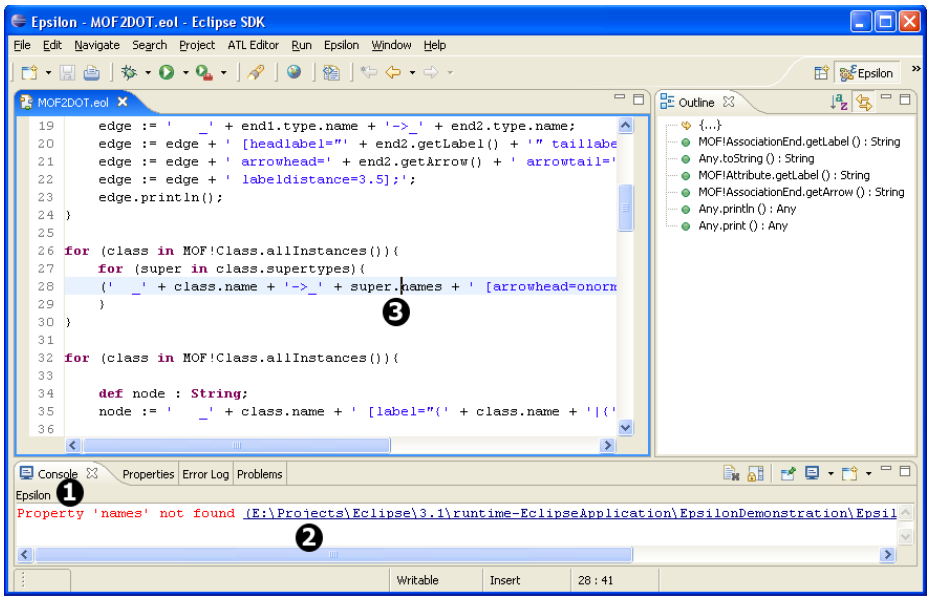

Fig. 4. Epsilon Console

the design and development process and therefore, the necessary slots already exist in the execution engines. What remains is to get acquainted with the technicalities of the Eclipse debugging framework and implement this feature in practice. Another feature we are considering is to implement static type checking, so that more errors can be captured before the execution of a module (currently, type-checking is performed at run-time). Finally, we are particularly interested in identifying new model management tasks that can potentially benefit from the construction of task-dedicated languages.

5 Acknowledgements

The work in this paper was supported by the European Commission via the MODELWARE project. The MODELWARE project is co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006).

References

1. Eclipse GMT - Generative Modeling Technology, Official Web-Site. http://www.eclipse.org/gmt.
2. MOFScript. Official Web-Site: http://www.modelbased.net/mofscript/.
3. Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20, 2003.
4. Dimitrios S. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon), Official Web-Site. http://www.cs.york.ac.uk/~dkolovos/epsilon.
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. (to appear).
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), ICSE 2006*, pages 13 – 20, Shanghai, China, 2006. ACM Press.
7. 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.
8. Eclipse.org. Eclipse Modelling Framework. http://www.eclipse.org/emf.
9. 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.
10. Frédéric Jouault, Jean Bezívin. Using ATL for Checking Models. In *Proc. International Workshop on Graph and Model Transformation (GraMoT)*, Tallinn, Estonia, September 2005.
11. Fréderic Jouault, Jean Bezívin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, LNCS, pages 171–185, Bologna, Italy, 2006.
12. Object Management Group. MOF QVT Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf.

13. Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf.
14. Sun Microsystems. Meta Data Repository. http://mdr.netbeans.org.