The Epsilon Object Language (EOL)

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. Model-Driven Development requires *model management* languages and tools for supporting model operations such as editing, consistency checking, and transformation. At the core of these model management techniques is a set of facilities for model *navigation* and *modification*. A subset of the Object Constraint Language can be used for some of these tasks, but it has limitations as a general-purpose language to be used in a variety of model management tasks. We present the metamodel independent Epsilon Object Language (EOL) which builds on OCL. EOL can be used both as a standalone generic model management language or as infrastructure on which task-specific languages can be built. We describe how it has been used to construct a selection of languages, such as model merging, comparison, and text generation languages.

1 Introduction

Increasingly, software-intensive systems are constructed using Model-Driven Development (MDD). For MDD approaches, such as the Model-Driven Architecture (MDA) [25], to be used successfully, two key technologies are required.

- Standardised modelling and metamodelling languages, which are rich and expressive enough to capture domain-independent and domain-specific concerns. MDA relies on languages that are based on the Meta-Object Facility (MOF) [24].
- *Model management features*. Effective model management requires a set of languages and tools for manipulating models in *automated* ways [2]. A toolset for model management might include model editors (e.g., UML diagram tools, or tools for domain-specific languages such as Microsoft's domain-specific language tools for Visual Studio [7]), transformation engines (e.g., ATL [3]), model version control, consistency checking engines, and model merging engines.

In this paper, we present a new model management language, with prototype tool support: the Epsilon Object Language (EOL). EOL has evolved from careful analysis of existing model management frameworks and languages (discussed in the next section), particularly the Object Constraint Language (OCL). The novelties with EOL are its technology agnosticism, as it can be used to manage models from diverse technologies such as MOF, EMF and XML, its metamodel independence, since it is not bound to a specific metamodel, and the fact that it can be used as both a generic model management language and as the basis for defining task-specific model management languages. We describe how EOL has been used for the latter purpose in Section 4.

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 128–142, 2006.

The paper is organised as follows. We establish terminology and review related work on model management, and identify the need for a common infrastructure language for core model management operations, namely model navigation, modification and multiple model access. We argue that OCL is insufficient as such an infrastructure language, and then present EOL. We show how EOL can be used as a standalone language for model navigation and modification, and then describe how EOL has been used to derive a selection of model management languages, e.g., a model comparison, a model merging and a text generation language.

2 Background and Motivation

We take a very general view of MDD in this paper: a *model* is a description of phenomena of interest; thus, a model is represented using textual or graphical languages. Examples of models include UML models, XML schemas, or web documents.

A variety of *operations* on models can be provided by model management systems. These operations can be classified in the same way as database management system operations:

- *create* new models and model elements that conform to a metamodel;
- *read* or *query* models, e.g., to project out information of interest to specific stakeholders. Specific examples of queries include boolean queries to determine whether two or more models are mutually consistent, and queries to select a subset of modelling elements satisfying a particular property.
- *update* models, e.g., changing the properties of a model, adding elements to a model. A specific example of an update operation is *model merging*.
- *delete* models and model elements.

2.1 Model Management Frameworks and Languages

The Meta-Object Facility is a standard model management framework from the OMG [24]. It is a metamodelling language that provides core facilities for defining modelling languages. There is limited tool support for MOF 2.0 at present, though there are tools for earlier versions, e.g., UML2MOF and the MetaData repository under NetBeans for MOF 1.4 [21].

Possibly the most well-known and widely used framework for implementing model management is the Eclipse Modelling Framework (EMF) [17]. EMF is a model engineering extension for Eclipse, and enriches it with model manipulation capabilities via a model handling API. EMF provides support for operations such as creation and deletion of model elements, property assignment, and navigation.

An exemplar of a model management framework mainly built atop EMF is the Atlas Model Management Architecture (AMMA) [2]. AMMA is a general-purpose framework for model management, and is based on ATL. AMMA provides a virtual machine and infrastructural tool support for combining model transformations, model compositions, resource management into an open model management framework.

XMF-Mosaic is a standalone meta-programming environment from Xactium [32] that can be used for model management. It is based on a dialect of MOF and an

executable dialect of OCL, and provides built-in support for defining model transformations. It is not yet clear whether the infrastructure in XMF-Mosaic is sufficiently flexible and extensible to define other model management operations, e.g., model composition. These operations are not yet supported in the tool, to the best of our knowledge.

Perhaps of greatest similarity to the framework proposed in this paper is the work on Kermeta [18]. Kermeta is a metamodelling language, compliant with the EMOF component of MOF 2.0, which provides an action language for specifying behaviour. Kermeta is intended to be an imperative language for implementing *executable* metamodels; as such, it is general-purpose and can be used directly for implementing metamodels for transformation languages, action languages, etc.

2.2 Transformations and Compositions

Transformations are sets of rules describing how models that conform to a metamodel are to be expressed in models that conform to a second (not necessarily different) metamodel. Specialised transformation tools are beginning to become available. Of note is ATL, which is inspired by (but does not entirely conform to) the MOF 2.0 QVT standard for transformations on MOF-defined languages [11]. The XMF-Mosaic environment can be used to implement (much of) the QVT standard, and bases its transformation rules on an executable language called XMap. Since XMF-Mosaic is a general-purpose meta-programming environment, the transformations are not dependent on any metamodel. Patrascoiu has proposed the YATL transformation language as part of the Kent Modelling Framework (KMF) [28], and uses a subset of OCL for model navigation.

Model *compositions* involve merging or integrating two or more models to produce a consistent single model. Model composition is founded on theory from database schema integration [29]. One of the first prototypes of a model composition framework is the Atlas Model Weaver (AMW), which is part of the AMMA model management framework [20]. The intent of AMW is to allow compositions of two models or two metamodels via *weaving* sessions, which are based on specific weaving metamodels. AMW has been shown to have general applicability to data management and software engineering [20].

2.3 Model Consistency Checking

An essential model management operation is *model consistency checking*, which involves determining whether information contained in two or more models contains contradictions. Model consistency is recognised as one of the most important qualities sought in model management [12,14]. Two types of consistency are intra-model and inter-model consistency [12]. To achieve intra-model consistency, a model must comply with its meta-model. Moreover, in multi-view modelling languages (such as UML [27]), views of a model must not contradict each other [33]. Inter-model consistency, on the other hand, is about maintaining a set of models in a state where they are consistent with each other. Substantial work has been carried out on checking intra-model consistency, e.g., based on evaluating OCL constraints [9,10]. However, OCL is inherently limited to specifying constraints in the context of a single model and cannot be used as-is for expressing consistency rules across different models. This is of importance when checking consistency between different versions of a model. Intra-model consistency has been achieved via construction [23], and by analysis [23,31]. Generic

consistency checking approaches, e.g., based on XML [13] may be at the inappropriate level of abstraction for defining meaningful consistency constraints on models.

2.4 Common Requirements

Among the generic and specific model management tools and frameworks discussed above (e.g., OCL [26] for inter-model consistency checking, QVT [11], Kermeta, YATL [28], ATL [3]), we can clearly identify a need for a *common infrastructure* for model management that provides three key facilities in languages and tools, as illustrated in Figure 1.

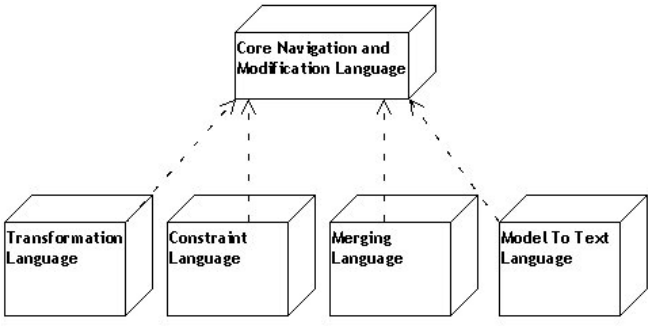

Fig. 1. Model Management Languages Requirements

The first characteristic is the ability to *navigate* models to extract elements of interest. The second characteristic is the ability to *modify* models to implement model management operations for change, e.g., to add, update and delete elements. Finally, multiple models must be concurrently accessible to support cross-model operations such as transformation, merging and inter-model consistency checking.

2.5 Limitations of OCL for Model Management

OCL provides model navigation facilities for UML and MOF models, and it is an OMG standard. The majority of contemporary model management languages and tools use a subset of OCL for navigation and expressing constraints. However, there are a number of limitations with using OCL as the basis for model management:

- The subset of OCL used for navigation and expression varies among different model management frameworks; incompatibilities can easily arise between, e.g., a transformation language using a subset of OCL, and a model-to-text language using a slightly different subset of OCL. There is *no standard OCL core* that can be reused for model navigation and building new task-specific languages.
- By design, OCL does not support model modification capabilities. In particular, it cannot be used to create, update, or delete model elements, nor can it update attribute or reference values. However model modification features are essential to

most model management tasks. Consequently, each new model management language has to implement its own model modification features, adding unnecessary diversity between languages and duplicating effort.
- OCL does not support statement sequencing; this must be encoded using nested and quantified expressions, leading to complex statements which are difficult to understand and maintain. This makes it difficult to use a model management framework in batch mode, and to express complex navigations.
- OCL can only refer to a single model at a time. This is particularly problematic for tasks such as inter-model consistency checking, transformation, and merging. It is particularly challenging in the case where some models have been constructed manually and others automatically. OCL needs to be supplemented with other languages and tools for inter-model consistency checking [22].
- OCL provides two operators for model navigation (the '.' and → operators). This adds unnecessary diversity to navigation expressions [19].

However, OCL has a significant user base and its navigation mechanisms are efficient, platform independent, and allow expression of complex queries. What is needed is a flexible, metamodel-independent model navigation language that builds on OCL but also addresses the aforementioned limitations. Such a language could play the role of a common infrastructure language for model management tasks and is essential to provide integrated support for diverse (domain specific) modelling languages.

In the next section we introduce the Epsilon Object Language as a language that contributes to filling this gap.

3 The Epsilon Object Language

The *Epsilon Object Language* (EOL) is the result of efforts to reuse the navigational mechanisms of OCL while adding support for other language features like multiple model access, statement sequencing, simple programming idioms and model modification capabilities. We are using EOL as a core language upon which we are developing a family of task-specific model management languages such as transformation, code generation, merging/integration and consistency checking languages. We call this family of languages the *Extensible Platform for Specification of Integrated Languages for mOdel maNagement* (Epsilon).

We now present an overview of EOL focusing on its differences from OCL, its abstract and concrete syntax, and some examples. In the next section we briefly describe the use of EOL in deriving task-specific model management languages.

3.1 Features

EOL reuses a significant part of OCL, including model querying operations such as the `select()`, `collect()` and `iterate()`. Moreover, it uses a similar syntax for defining variables: `def <name> : <type>;` and has an identical type system. In the sequel we describe the additional features of EOL in relation to OCL.

3.1.1 Access to Multiple Models. To support multiple model access in EOL, each model has a unique identifier (name). Access to a specific meta-class of a model is performed via the `!` operator. For example, if `UML` is a UML 1.4 model, `UML!Class` will return the `Class` meta-class reference and `UML!Class.allInstances()` will return all the instances of the `Class` meta-class that are contained in `UML`. If there are conflicting meta-class names, the full path, e.g., `UML!Foundation::Core::Class` can be used. The `!` operator is inspired by ATL [3]. However, in ATL, instead of the model name, the name of the metamodel is used as identifier. This is not appropriate for EOL, which must accommodate multiple models of a metamodel.

3.1.2 Statement Sequencing and Grouping. Sequencing and grouping statements allow developers to disentangle complicated, nested queries, potentially making them easier to read and debug. Statements in EOL can be sequenced using the `;` and grouped using the `{` and `}` delimiters.

3.1.3 Uniformity of Invocation. Providing two operators for invoking operations and accessing model element features ($\rightarrow$ and '.') adds unnecessary diversity to OCL expressions. In EOL, we use the dot operator as a uniform navigation and invocation operator. The arrow operator can still be used to facilitate compatibility with the syntax that OCL developers are familiar with, or to resolve potential conflicts with built-in EOL operations. For instance, EOL provides a built-in `print()` operation that displays a `String` representation of the object to which it is applied. However, a meta-class may itself define a `print()` operation. In that case, the arrow operator will invoke the built-in operation while the dot operator will invoke the metamodel-defined operation.

3.1.4 Model Modification. A core requirement of OCL as a constraint language is to preserve the state of models by performing read-only operations [26]. Therefore, OCL expressions cannot create, update or delete model elements. While such operations are not required for expressing constraints, for most model management languages this feature is essential. Therefore, in EOL we have introduced the `:=` operator, which performs assignments of values to variables and model element features: e.g., `class.name := 'SomeClass'`. Moreover, EOL extends the built-in collection types (Bag, $Sequence$, Set, $OrderedSet$) with operations (such as the $add(Any)$ and $remove(Any)$ operations) that can modify the contents of the collection to which they are applied. Regarding element creation and deletion, EOL supports the new keyword and the $newInstance()$ operation for creating new model elements as well as the $delete()$ operation for deleting model elements from a model.

3.1.5 Debugging and Error Reporting. For debugging and error reporting, it is essential that the user can send text to predefined output streams. While nearly all programming languages support this feature, OCL currently lacks such a mechanism. In EOL we have introduced the `print()` and `err()` built-in operations that send a $String$ representation of the object that they apply to, to the standard output and error stream respectively. Reporting operations return the object to which they are applied, to facilitate integration of debugging messages without changing the structure of

a program. For instance the statement `a.owner := b.owner.print();` prints a *String* representation of `b.owner` and returns `b.owner` so that it can be assigned to `a.owner`. To facilitate meaningful messages, the EOL engine supports pluggable *pretty printers* that can print *String* representations of model elements in a readable way.

3.1.6 Reusability. EOL allows users to define operations that apply to elements of a specific meta-class (similar to OCL helpers). Such operations can be used not only from EOL programs but also from any EOL-derived languages programs as well. Moreover, operations can be grouped in different physical files and be imported on demand through the `import` statements. An operation that checks if a `UML!ModelElement` has a specific stereotype is displayed in Listing 1.1.

Listing 1.1. EOL Operation example

```
operation UML!ModelElement hasStereotype(name : String) : Boolean
{
  return self.stereotype.exists(st:UML!Stereotype|st.name = name);
}
```

Having outlined the basic features of EOL, we now present its abstract syntax and a short worked example that demonstrates its concrete syntax.

3.2 EOL Abstract Syntax

Figure 2 presents a snapshot of the core part of the abstract syntax of the language. The example that follows, demonstrating parts of the concrete syntax of EOL, clarifies the missing parts of the abstract syntax (e.g., the syntax for logical expressions).

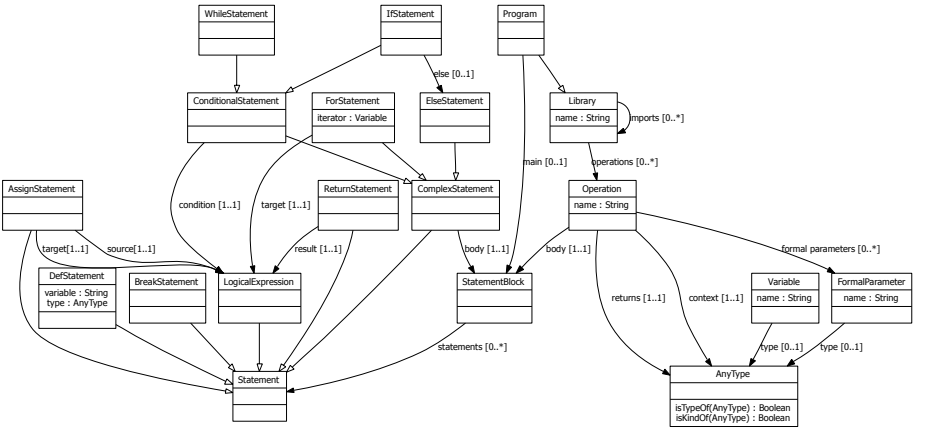

Fig. 2. Snapshot of EOL Abstract Syntax

3.3 EOL Concrete Syntax Example

In this section, we demonstrate a scenario and a working solution using EOL as a stand-alone model management language.

3.3.1 Scenario. When designing class diagrams in UML, it is common practice to mark attributes as private and to provide public getter and setter operations. A common naming convention commonly is that for an attribute named `attr` of type `AttrType` the setter and getter operations should have signatures `getAttr():AttrType` and `setAttr(attr:AttrType)` respectively.

Adding getter and setter operations is mechanical and can benefit from automation. In fact, some UML tools provide built-in wizards for converting public attributes into triplets of private attributes, setters and getters. However, those wizards are tool-specific and often pertain only to a particular extent. EOL allows a generic, tool-independent (requiring only that the modelling tool can serialise models, e.g., in XMI) solution to be defined.

3.3.2 Solution. In Listing 1.2, we demonstrate a user-defined EOL program that runs on any UML 1.4 model and performs the desired addition of getters and setters.

Listing 1.2. EOL Program

```
1  for ( attribute in UML!Attribute.allInstances()) {
2    if ( attribute . visibility = UML!VisibilityKind#vk_public){
3      attribute . visibility := UML!VisibilityKind#vk_private;
4      attribute . createGetter ();
5      if ( attribute . changeability =
6        UML!ChangeableKind#ck_changeable){
7        attribute . createSetter ();
8      }
9    }
10 }
11
12 operation UML!Attribute createSetter () {
13   def setter : new UML!Operation;
14    setter .name := 'set' + self .name.firstToUpperCase ();
15    setter . visibility := UML!VisibilityKind#vk_public;
16    setter .concurrency := UML!CallConcurrencyKind#cck_sequential;
17
18   def valueParam : new UML!Parameter;
19   valueParam.name := self .name;
20   valueParam.type := self .type;
21   valueParam.kind := UML!ParameterDirectionKind#pdk_in;
22    setter .parameter.add(valueParam);
23
24   self .owner. feature .add( setter );
25 }
26
```

```
27  operation UML!Attribute createGetter () {
28    def getter : new UML!Operation;
29    getter.name := 'get' + self.name.firstToUpperCase();
30    getter.visibility := UML!VisibilityKind#vk_public;
31    getter.concurrency := UML!CallConcurrencyKind#cck_sequential;
32
33    def returnParam : new UML!Parameter;
34    returnParam.type := self.type;
35    returnParam.kind := UML!ParameterDirectionKind#pdk_return;
36    getter.parameter.add(returnParam);
37
38    self.owner.feature.add(getter);
39  }
```

Lines 1-10 constitute the body of the EOL program. It iterates over each attribute of the UML model, changing its visibility to private. If the attribute is changeable, both setter and getter operations are created, otherwise only a getter operation is created.

Lines 12-25 define the `createSetter` EOL operation. Line 12 declares that the operation applies to elements of the type `UML!Attribute`. In lines 13-16, a new `UML!Operation` is created using the new keyword with its name set according to the naming convention discussed above. Its visibility and concurrency are set. In lines 18-22, the parameter of the operation is defined, its type and kind are set and it is added to the formal parameters of the operation. Finally, in line 24, the setter is added to the features of the class that owns the attribute.

Lines 27-39 define the `createGetter` operation that creates a getter `UML!Operation` for an attribute, in a similar way to the `createSetter` discussed above.

As proof of concept, we execute this EOL program using the model displayed in Figure 3 (left) as input. In this model, all of the attributes are public and changeable except for the `registrationNumber` of class `Student` that is read-only (*frozen* according to UML terminology). The target, refactored model is shown in Figure 3 (right).

All public attributes of the source model have been converted to private in the target, and setters and getters have been added for all except for `registrationNumber`, for which only a getter has been added.

This example demonstrates using EOL as a standalone language, showing key parts of EOL's concrete syntax. The program of Listing 1.2 shows a minimal functionality. A more complete program would include looking for existing setters and getters before creating, as well as handling static, derived and multi-valued attributes.

3.4 EOL Tool Support

We have implemented an EOL engine using a modular architecture that allows us to plug-in virtually any type of structured model. In its current implementation, we have

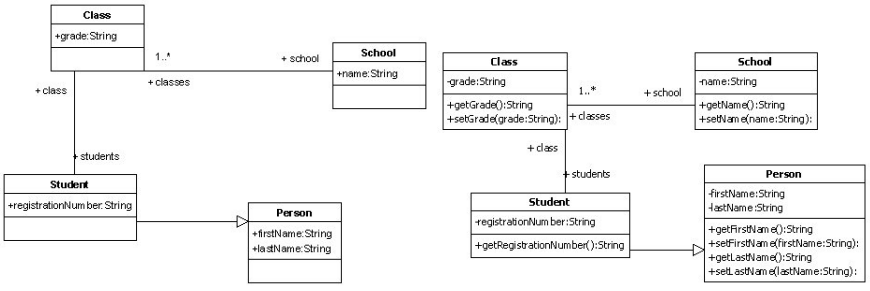

Fig. 3. UML models for EOL example

developed full support for Meta-Object Facility (MOF) models using the MetaData Repository [30], EMF models and XML documents using JDOM [6]. We are also experimenting with models from the Microsoft DSL Toolkit [7]. Since the EOL engine treats all models identically through a layer of abstraction we call `EolModel`, it is feasible to access and manage models from different platforms in the same program. For instance, we have developed EOL programs that check XML documents against MOF models and vice versa.

We have developed a set of plug-ins for Eclipse [4] (editor, perspectives, wizards, and launching configuration) that allow developers to use the language in real problems. The plug-ins and the source for the examples can be found at [15].

For EOL-based languages, the architecture of both the execution engine and the Eclipse plug-ins is designed to facilitate reusability, as described in the next section.

4 Building Task-Specific Model Management Languages

In Section 3.3 we presented EOL as a standalone language. However, a primary motivation for developing EOL is to embed it in a family of task-specific languages for model management. In this section, we briefly describe several task-specific languages we have constructed: the Epsilon Merging Language (EML), the Epsilon Comparison Language (ECL), and the Epsilon Generation Language (EGL) for generating text (e.g. code and documentation) from models.

4.1 Epsilon Comparison Language (ECL)

ECL is a metamodel-independent model comparison language built atop EOL. It is used to express rules that compare a pair of models. The results of comparisons can then be used in, e.g., a merging process. An ECL specification consists of match rules that apply to the elements of the models; these rules include compare and conform parts. The matching process classifies elements into those that match and conform, those that do not, those that match *or* conform, and those to which no match rule has applied. Further discussion about the rationale of this classification approach is presented in [16]. The results can then be processed in a variety of ways. The classification is made accessible

through the API of the ECL engine that executed the specification; it can be processed according to the needs and capabilities of the environment. For example, it can be used to visually highlight the elements in the source models, or to provide text messages to the user pointing at the sources of inconsistencies; the type of post-processing depends on the application domain and the needs of the users of ECL.

A very simple and partial ECL example allows us to illustrate both ECL and its relationship to EOL. Listing 1.3 contains an ECL program containing rules for comparing elements from two models: the first of a simplified class diagram metamodel, and the second a simplified relational database metamodel (we omit the metamodels themselves, but they are typical).

Listing 1.3. ECL Specification

```
−− Match classes against tables rule Class2Table
  match class : CD!Class
  with table : DB!Table {

  compare {
    return class .name = table .name;
  }
  conform {
    return table .columns. exists (
      c : DB!PrimaryKey|c.name = class.name + 'ID');
  }
}
```

The `Class2Table` rule is executed for each pair of instances of `CD!Class` and `DB!Table` in the source models. In its compare part, it checks that the class has the same name as its comparable table. When this condition is met, the two entities are considered to be semantically equivalent and the match rule can proceed to executing its conform part. There, it checks if the table has a primary key named after the name of the class suffixed with `ID`. Thus, the compare part of a rule identifies a small amount of contextual information necessary to carry out deeper semantic checking in the conform part. In general, whether to check constraints in the compare or the conform part of a rule is application dependent. A similar rule can be written to match attributes against database columns, but we omit this due to space restrictions.

4.2 Epsilon Merging Language (EML)

EML is a metamodel-independent language for expressing model merging operations. It is built atop EOL: model navigation expressions and model modification operations used within the merging process are written directly in EOL.

EML is rule based. It allows specification of different kinds of rules for expressing model merges. Rules in EML are either *match* rules (identical to those of ECL), *merge* rules, or *transform* rules. In the example presented in Listing 1.4, the match rule on Classes returns true iff two classes (the left and right class) are both abstract or both concrete, and their names and namespaces match. The merge rule on Classes produces,

in the merged model, a class which has the name and namespace of the left class, and all features of both left and right.

Listing 1.4. Match and merge rules in EML based on EOL

```
rule MatchClasses {
    match l: Left!Class with r: Right!Class

    compare {
      return l.name = r.name and
             l.namespace.matches(r.namespace);
    }
    conform { return l.isAbstract = r.isAbstract; }
  }

  rule MergeClasses {
    merge l: Left!Class with r: Right!Class
    into m: Merged!Class

    m.name := l.name;
    m.namespace := l.namespace.equivalent();
    m.feature := l.feature.includeAll(r.feature).equivalent();
  }
```

EML provides more features than we can describe here. It includes rule inheritance, exception handling, and transformation capabilities. An EML development tool, which builds on the EOL tool, has been developed and is available at [15].

4.3 Epsilon Generation Language (EGL)

A third task-specific language that we have recently been developing is the Epsilon Generation Language (EGL), which targets the problem of model-to-text mapping, similar to MOFScript[8]. Unlike MOFScript, EGL is once again built on top of EOL, and uses EOL to provide model navigation and modification facilities. An example EGL specification is in Listing 1.5. The delimeters `[%` and `%]` separate EOL code from static text (similar to what is done in MOFScript).

Listing 1.5. Example EGL specification built atop EOL

```
[%for (class in UML!Class.allInstances()) { %]
public class [%=class.name%] {
  [%for (att in class.feature.select(a:UML!Attribute|true)){ %]
    private [%=att.type.name%] [%=att.name%];
  [%}%]
}
[%}%]
```

In the example, an EGL program iterates across all UML classes and outputs target Java code, wherein each UML class is mapped to a Java class, and each UML attribute

is mapped to a private Java attribute. The EGL development tool transforms this specification into a pure EOL program, which can then be executed against a UML model.

4.3.1 Reuse of EOL Tools in Task-Specific Languages. The details of the implementation process of ECL exemplifies the effort that is saved by using EOL as an infrastructure language for the development of task-specific model management languages.

The abstract syntax (grammar) of ECL contains only a small number of task-specific elements (*MatchRule*, *Conform* and *Compare*), and depends substantially on EOL (*Statement Block*, *Formal Parameter*, *Library*) for its navigational and computational characteristics. This, together with the flexible architecture of the EOL engine, makes the implementation of the ECL engine straightforward. More specifically, the ECL parser required 121 lines of ANTLR [1] grammar specification and 6910 lines of Java code, with 4883 of them being generated automatically from the ANTLR grammar, leaving only 2027 lines of hand-written code.

5 Conclusions and Further Work

In this paper we have presented the Epsilon Object Language, a metamodel independent model management language. Its novelties include its support for model modification, additional conventional programming constructs, and the ability to access multiple models, e.g., for model comparison. While EOL can be used as a standalone language for model management, its primary purpose is to be embedded into higher-level task-specific languages of the Epsilon platform. The architecture of the EOL execution engine helps to achieves this. As proof of concept, we have designed and implemented three task-specific languages built on EOL: a merging language, comparison language, and model-to-text language. Implementing these languages and reusing the EOL infrastructure was predominantly straightforward.

We are in a continual process of attempting to align with the OCL 2.0 standard in those aspects of EOL that are not characterized by fundamentally different design decisions (such as the ability to modify models).

Our plans for the near future include releasing the EOL execution engine and plugins as part of the Eclipse GMT [5] project, and to develop comprehensive documentation of the internals of the architecture of the EOL execution engine. This will allow external developers to use the EOL infrastructure to build custom task-specific languages. As well, we are aligning the merging language with the approach taken by Atlas Model Weaver, to produce weaving models, enabling reuse.

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). Information included in this document reflects only the authors views. The European Commission is not liable for any use that may be made of the information contained herein. We thank the Atlas and AMMA teams at INRIA for their help.

References

1. ANTLR: ANother Tool For Language Recognition, Official Web-Site. http://www.antlr.org.
2. Atlas Model Management Architecture. http://www.sciences.univ-nantes.fr/lina/ atl/AMMAROOT/.
3. Atlas Transformation Language, official web-site. http://www.sciences.univ-nantes.fr/lina/atl/.
4. Eclipse Foundation, Official Web-Site. http://www.eclipse.org.
5. Eclipse GMT - Generative Model Transformer, Official Web-Site. http://www.eclipse.org/gmt.
6. JDOM Official Web-Site. http://www.jdom.org.
7. Microsoft Domain Specific Languages Framework, Official Web-Site. http://msdn.microsoft.com/ vstudio/ teamsystem/ workshop/ DSLTools/ default.aspx.
8. MOFScript. Official Web-Site: http://www.modelbased.net/mofscript/.
9. OCLE: Object Constraint Language Environment, official web-site. http://lci.cs.ubbcluj.ro/ocle/.
10. Octopus: OCL Tool for Precise Uml Specifications, official web-site. http://www.klasse.nl/ocl/octopus-intro.html.
11. QVT Partners Official Web-Site. http://qvtp.org/.
12. Bogumila Hnatkowska, Zbigniew Huzar, Ludwik Kuzniarz and Lech Tuzinkiewicz. A systematic approach to consistency within UML based software development process. In *Consistency Problems in UML-based Software Development Workshop*, pages 16–29, 2002.
13. Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein and Erns Ellmer. Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, 2003.
14. Dan Chiorean, Mihai Pasca, Adrian Carcu, Christian Botiza, Sorin Moldovan. Ensuring UML models consistency using the OCL Environment. In *Sixth International Conference on the Unified Modelling Language - the Language and its applications*, 2003.
15. 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.
16. 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)*, 2006. http://www.cs.york.ac.uk/~dkolovos/publications/GaMMa02-kolovos.pdf.
17. Eclipse.org. Eclipse Modelling Framework. http://www.eclipse.org/emf.
18. F. Chauvel and F. Fleurey. Kermeta Language Overview. http://www.kermeta.org.
19. Mandana Vaziri and Daniel Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. Response to Object Management Group's Request for Information on UML 2.0 , December 1999. http://www.omg.org/docs/ad/99-12-05.pdf.
20. Marcos Didonet Del Fabro, Jean Bezivin, Frederic Jouault, Erwan Breton, Guillaume Gueltas. AMW: A Generic Model Weaver. *Proceedings of IDM05*, 2005.
21. Martin Matula. NetBeans UML Profile for MOF. http://mdr.netbeans.org/uml2mof/.
22. MODELWARE Partners. D1.5: Model Consistency Rules, 2005. http://www.modelware-ist.org.
23. Monique Snoeck, Cindy Michiels and Guido Dedene. Consistency by Construction: The Case of MERODE. In *International Workshop on Conceptual Modeling Quality*, 2003.
24. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
25. Object Management Group. Model Driven Architecture, official web-site.

26. Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf.
27. Object Management Group. UML official web-site. http://www.uml.org.
28. Octavian Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.
29. Rachel A. Pottinger and Philip A. Bernstein. Merging Models Based on Given Correspondences. Technical Report UW-CSE-03-02-03, University of Washington, 2003.
30. Sun Microsystems. Meta Data Repository. http://mdr.netbeans.org.
31. Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds. Maintaining Consistency between UML Models with Description Logic Tools. In *Sixth International Conference on the Unified Modelling Language - the Language and its applications, Workshop on Consistency Problems in UML-based Software Development II*, 2003.
32. Xactium. XMF-Mosaic. http://www.xactium.com.
33. Zhiming Liu, He Jifeng, Xiaoshan Li, Yifeng Chen. Consistency and Refinement of UML Models. In *Consistency Problems in UML-based Software Development Workshop III*, 2004.