JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2007

Vol. 6, No. 9, Special Issue. TOOLS EUROPE 2007, October 2007

Update Transformations in the Small with the Epsilon Wizard Language

Dimitrios S. Kolovos
Richard F. Paige
Fiona A.C. Polack
Louis M. Rose

Department of Computer Science, University of York, United Kingdom

We present the Epsilon Wizard Language (EWL), a tool-supported language for specifying and executing automated update transformations in the small based on existing model elements and input from the user. We discuss on EWL's requirements and relevant design decisions, as well as the infrastructure upon which the language has been developed. We also provide concrete working examples to demonstrate how EWL can be used to automate the process of constructing and refactoring models.

1 INTRODUCTION

Enhancing consistency and productivity in software development are the most important promises of Model Driven Development. The role of Model Transformation for delivering these benefits is so important that it has been characterized the *heart and soul of Model-Driven Development* [1]. There are two types of transformations: mapping and update transformations [2]. Mapping transformations typically transform a source model into a target model expressed in a different modelling language by creating zero or more model elements in the target model for each model element of the source model. A prominent example of mapping transformations is the popular Class to Database scenario found in [3]. By contrast, update transformations perform in-place modifications of a model. They can be further classified into two subcategories: transformations in the small and in the large. Update transformations in the large apply to sets of model elements calculated using well-defined rules in a batch manner. On the other hand, update transformations in the small are applied in a user-driven manner on model elements that have been explicitly selected by the user.

In the context of tool-support for model transformation, a large number of task-specific languages for mapping transformations have been proposed, including QVT [4], ATL [3] and VIATRA2 [5]. Moreover, it has been shown that such languages can also be used to achieve update transformations in the large (e.g. the *refinement* mode of ATL). By contrast, the field of update transformations in the small remains relatively underdeveloped.

In this paper we present the Epsilon Wizard Language (EWL), a language that supports update transformations in the small for diverse modelling languages. EWL provides novel features such as support for arbitrary modelling technologies, user input and flexible guards in transformation rules.

The rest of the paper is organized as follows. In Section 2 we discuss the process of constructing and refactoring models and show that it can benefit substantially from additional automation beyond that provided by the modelling tools themselves. In Section 3 we discuss the concept of update transformations in the small and outline the specific requirements of the task. Through this discussion we elaborate a set of desirable characteristics for a language that can effectively support update transformations in the small. In Section 4 we present the prototype Epsilon Wizard Language (EWL), with tool support, which is implemented atop the Epsilon platform [6] and tailored specifically to support update transformations in the small. To show the practicality and usefulness of our approach, in Section 5 we provide concrete examples implemented with EWL. In Section 6 we discuss related work and in Section 7 we conclude and provide directions for further research on the subject.

2 CONSTRUCTING AND REFACTORING MODELS

Constructing and refactoring models is undoubtedly a mentally intensive process. However, during modelling, reoccurring patterns of model update activities typically appear. As an example, when renaming a class in a UML class diagram, the modeller also needs to manually update the names of association ends that link to the renamed class. Thus, when renaming a class from `Chapter` to `Section`, all associations ends that point to the class and are named `chapter` or `chapters` should be also renamed to `section` and `sections` respectively. As another example, when the modeller needs to refactor a UML class into a singleton [7], they need to go through a number of well-defined steps such as attaching a stereotype (`<<singleton>>`), defining a static `instance` attribute and adding a static `getInstance()` method that returns the unique instance of the singleton.

It is generally accepted that performing repetitive tasks manually is both time-consuming and error-prone [8]. On the other hand, failing to complete such tasks correctly and precisely compromises the consistency, and thus the quality, of the models. In Model-Driven Development, this is particularly important since models are increasingly used to automatically produce (parts of) working systems.

Automating the Construction and Refactoring Process

Some modelling tools provide built-in transformations (*wizards*) for automating common repetitive tasks. However, according to the architecture of the designed system and the specific problem domain, additional repetitive tasks typically appear, which cannot be addressed by the pre-conceived built-in wizards of a modelling tool.

To address the automation problem in its general case, users must be able to easily define update transformations (wizards) that are tailored to their specific needs.

To an extent, this can be achieved via the extensible architecture that state-of-the-art tools often provide and which enables users to add functionality to the tool via scripts or application code using the implementation language of the tool. Nevertheless, as discussed in [9], the majority of modelling tools have a proprietary API through which they expose an edited model, and therefore such scripts and extensions are not portable to other tools. Moreover, scripting languages and third-generation languages such as Java and C++ are not particularly suitable for model navigation and modification [9].

Existing languages for mapping transformations, such as QVT and ATL, cannot also be used as-is for this purpose. The main reason is that such languages have been designed to operate in a batch manner without human involvement in the process. By contrast, as discussed above, the task of constructing and refactoring models is inherently user-driven. The option of extending/customizing them is discussed in the sequel.

3 UPDATE TRANSFORMATIONS IN THE SMALL

Update transformations are actions that automatically create, update or delete model elements based on a selection of existing elements in the model and information obtained otherwise (e.g. through user input), in a user-driven fashion. In this paper we refer to such actions as *wizards* instead of *rules* to reduce confusion between them and rules of languages for mapping transformations such as ATL and QVT. In the following sections we elaborate the desirable characteristics of wizards in an informal way.

Structure of Wizards

In its simplest form, a wizard only needs to define the actions it will perform when it is applied to a selection of model elements. The structure of such a wizard that transforms a UML class into a *singleton* is shown using pseudo-code in Listing 1.

Listing 1: The simplest form of a wizard for refactoring a class into a singleton

```
do :
  attach the singleton stereotype
  create the instance attribute
  create the getInstance method
```

Since not all wizards apply to all types of elements in the model, each wizard needs to specify the types of elements to which it applies. For example, the wizard of Listing 1, which automatically transforms a class into a singleton, applies only when the selected model element is a class. The simplest approach to ensuring that the wizard will only be applied on classes is to enclose its body in an `if` condition as shown in Listing 2.

Listing 2: The wizard of Listing 1 enhanced with an *if* condition

```
do :
  if (selected element is a class) {
    attach the singleton stereotype
    create the instance attribute
    create the getInstance method
  }
```

A more modular approach is to separate this condition from the body of the wizard. This is shown in Listing 3 where the condition of the wizard is specified as a separate `guard` stating that the wizard applies only to elements of type Class. The latter is preferable since it enables filtering out wizards that are not applicable to the current selection of elements by evaluating only their `guard` parts and rejecting those that return `false`. Thus, at any time, the user can be provided with only the wizards that are applicable to the current selection of elements. Filtering out irrelevant wizards reduces confusion and enhances usability, particularly as the list of specified wizards grows.

Listing 3: The wizard of Listing 2 with an explicit *guard* instead of the *if* condition

```
guard : selected element is a class
do :
  attach the singleton stereotype
  create the instance attribute
  create the getInstance method
```

To enhance usability, a wizard also needs to define a short human-readable description of its functionality. To achieve this we add another field named `title`. There are two options for defining the title of a wizard: the first is to use a static string and the second to use a dynamic expression. The latter is preferable since it enables definition of context-aware titles.

Listing 4: The wizard of Listing 3 enhanced with a *title* part

```
guard : selected element is a class
title : Convert class <class-name> into a singleton
do :
  attach the singleton stereotype
  create the instance attribute
  create the getInstance method
```

Capabilities of Wizards

The `guard` and `title` parts of a wizard need to be expressed using a language that provides model querying and navigation facilities. Moreover, the `do` part also requires model modification capabilities to implement the transformation. To achieve complex transformations, it is essential that the user can provide additional information. For instance, to implement a wizard that addresses the class renaming scenario discussed in Section 2, the information provided by the selected class does not suffice; the user must also provide the new name of the class. Therefore, a suitable language must also provide mechanisms for capturing user input.

4 THE EPSILON WIZARD LANGUAGE (EWL)

In the previous section we used an example to elaborate the desired structure and capabilities of a language for supporting update transformations in the small effectively. In this section, we present such a language, the Epsilon Wizard Language (EWL), and discuss the infrastructure on which it has been implemented as well as its abstract syntax and execution semantics.

Infrastructure

As discussed in Section 3, a language for update transformations in the small must provide model querying, navigation and modification facilities. To achieve this in the context of EWL with minimal effort and duplication, we have implemented the language atop Epsilon, a platform of task-specific languages for model management.

The Epsilon Platform

Experience has shown that each model management task is best supported by a task-specific language. In this context, many different languages have been proposed for tasks such as model transformation, merging, validation, comparison and code generation. While each language has its own task-specific features, they all provide mechanisms for model querying, navigation and modification. Moreover, languages for tasks such as comparison, merging and transformation also need to support access to more than one models - of potentially different metamodels and modelling technologies - concurrently.

Epsilon consolidates these common facilities in a base language, the Epsilon Object Language (EOL) [9], and the Epsilon Model Connectivity (EMC) layer that new task-specific languages can reuse. EOL supports the powerful navigation and querying mechanisms of OCL but also provides features such as statement sequencing, model modification, operation polymorphism, multiple model access, user input/output facilities and enhanced modularity (using the `import` statement [9]). The EMC on the other hand is a layer that abstracts over the different modelling frameworks and enables languages built atop it to manage models of different modelling technologies uniformly. So far, EMC drivers for MDR[1], EMF[2] and XML models have been developed and in the future we plan to develop drivers for the Microsoft DSL framework [10] and for Java source code through the Spoon [11] project.

The architecture of Epsilon facilitates implementing task-specific languages with minimal replication. Apart from the wizard language discussed here, four more task-specific languages have been developed atop EOL and EMC: the Epsilon Comparison Language [12], the Epsilon Transformation Language [13], the Epsilon Merging Language (EML) [14] (built atop ETL and ECL), and the Epsilon Validation Language (EVL) [13]. Figure 4 provides a graphical overview of the architecture of Epsilon.

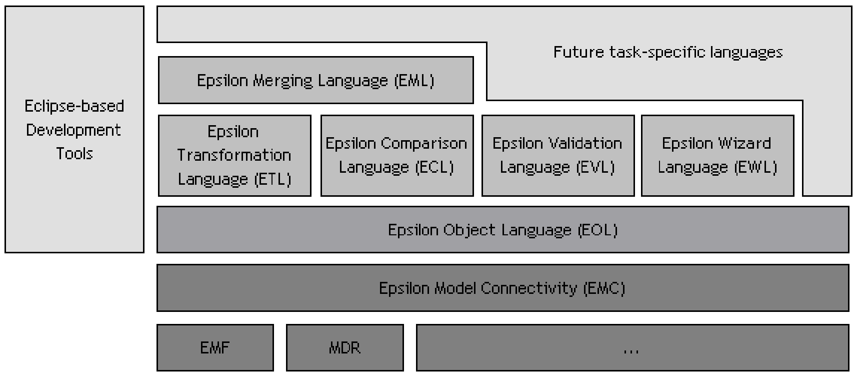

Eclipse-based Development Tools
Future task-specific languages
Epsilon Merging Language (EML)
Epsilon Transformation Language (ETL)
Epsilon Comparison Language (ECL)
Epsilon Validation Language (EVL)
Epsilon Wizard Language (EWL)
Epsilon Object Language (EOL)
Epsilon Model Connectivity (EMC)
EMF
MDR
...

Figure 1: Overview of the architecture of Epsilon

Alternative Implementation Options

An alternative to implementing EWL atop Epsilon would be to extend the syntax of a mapping transformation language such as QVT or ATL by adding the `title` construct to rules, and implement a user-driven rule-execution scheme. From a technical perspective this would be very challenging to achieve as neither language has been designed with extensibility as a primary aim. Especially for QVT, a publicly available implementation of the full standard is still missing, and therefore it is practically impossible to build atop it. On the other hand, Epsilon provides a more suitable infrastructure for implementing EWL since it has been built with the precise aim of supporting the development of task-specific languages.

Abstract Syntax of EWL

Since EWL is built atop Epsilon, its abstract and concrete syntax need only to define the concepts that are relevant to the task it addresses; they can reuse lower-level constructs from EOL. The basic concept of the EWL abstract syntax is a

[1]http://mdr.netbeans.org
[2]http://www.eclipse.org/emf

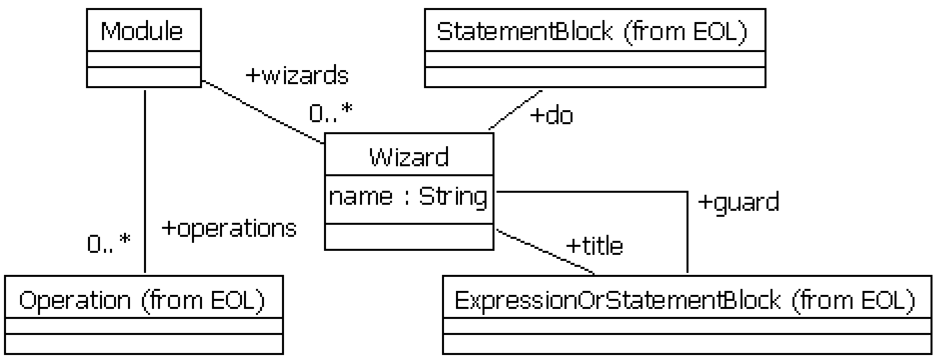


Figure 2: EWL Abstract Syntax

`Wizard`. A wizard defines a `name`, a `guard` part, a `title` part and a *do* part. Wizards are organized in `Modules`. The `name` of a wizard acts as an identifier and must be unique in the context of a module. The `guard` and `title` parts of a wizard are of type `ExpressionOrStatementBlock`, inherited from EOL. An `ExpressionOrStatementBlock` is either a single EOL expression or a block of EOL statements that include one or more `return` statements. This concept allows users to express simple declarative calculations as single expressions and complex calculations as blocks of imperative statements. The usefulness of this construct is further discussed in the case study presented in Section 5. Finally, the `do` part of the wizard is a block of EOL statements that specify the effects of the wizard when applied to a compatible selection of model elements. A graphical overview of the abstract syntax of EWL is provided in Figure 2.

Executing EWL Wizards in ArgoUML

The process of executing EWL wizards is inherently user-driven and as such it depends on the environment in which it is used. In this work, we have integrated the execution engine of EWL with the ArgoUML[3] open-source UML modelling tool[4]. Here we should stress that EWL does not depend in any way on ArgoUML (or UML in general) and can be integrated as an extension to any Java-based modelling tool such as the Topcased AADL and SAM or the GMF ECore graphical modellers.

To achieve integration, we have implemented an EMC compatible driver for the internal model of ArgoUML so that EWL can query and modify it. Since ArgoUML has been developed following a Model-View-Controller architecture, changes in the internal model are reflected to the user interface automatically. To integrate with the user-interface of ArgoUML, we have added a panel at the right side of the tool (point 1 in Figure 3) where users can view and execute the applicable wizards. The specifications of wizards are contained in a file (`wizards.ewl`) located in the

[3]http://argouml.tigris.org

[4]The extended version of ArgoUML is available at http://www-users.cs.york.ac.uk/∼dkolovos/software.php

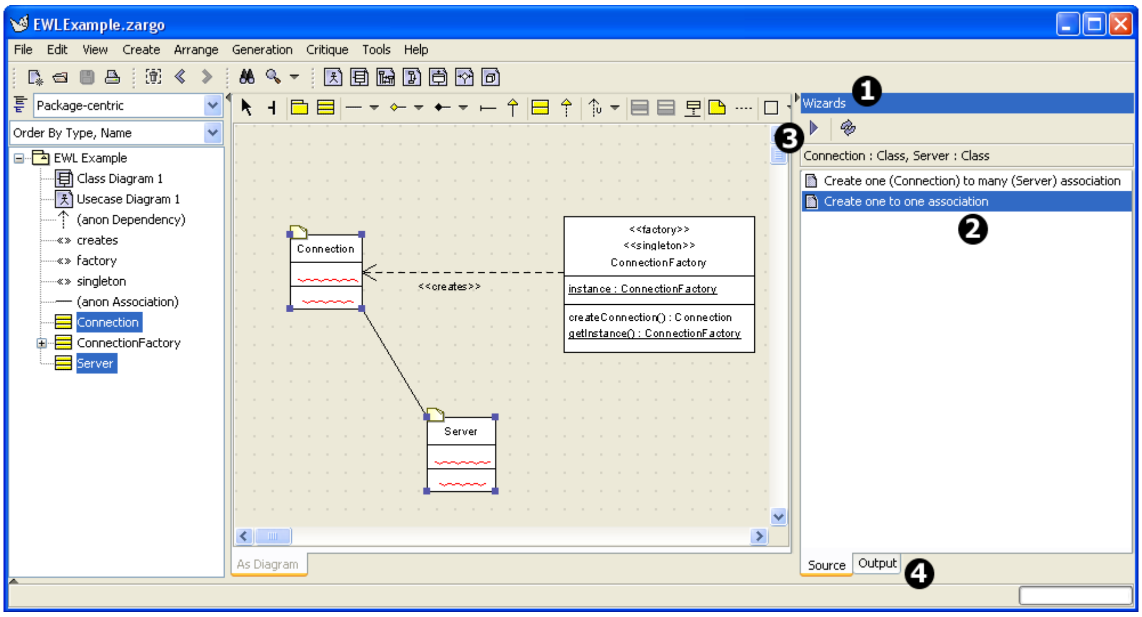


Figure 3: Screenshot of the extended ArgoUML with support for EWL Wizards

installation directory of ArgoUML.

With regard to the execution process, each time the selection of model elements changes (i.e. the user selects or deselects a model element), the guards of all wizards are evaluated. If the guard of a wizard is satisfied, the `title` part is also evaluated and the wizard is added to the list of *applicable* wizards (point 2 in Figure 3). Then the user can select a wizard and press the *execute* button (point 3 in Figure 3) to execute its *do* part. In all parts of the wizard, the elements selected by the user can be referenced using the built-in (`self`) variable. Finally, any output or runtime errors produced during the execution of any part of the wizard are displayed in the output tab (point 4 in Figure 3).

Evaluating the `guard` of every wizard whenever the selection of model elements changes can be quite time-consuming when many wizards are defined. Therefore, we are considering adopting a more scalable approach for future versions of the tool. An option is to evaluate the `guards` of wizards only when the user right-clicks on the selected elements, and, thereafter display applicable wizards in a pop-up menu.

5 CASE STUDY

In this section we present three concrete examples of EWL wizards for refactoring UML 1.4 models. Our aim is not to provide complete implementations that address all the sub-cases of each scenario but to provide enhanced understanding of the concrete syntax, the features and the capabilities of EWL to the reader. Moreover, we should stress again that although the examples in this case study are based on UML models, EWL can be used to capture wizards for diverse modelling languages and technologies as discussed in Section 4.

Case 1: Converting a Class into a Singleton

The singleton pattern [7] is applied when there is a class for which only one instance can exist at a time. In terms of UML, a singleton is a class stereotyped with the `<<singleton>>` stereotype, and it defines a static attribute named `instance` which holds the value of the unique instance. It also defines a static `getInstance()` operation that returns that unique instance. Wizard `ClassToSingleton`, presented in Listing 5, simplifies the process of converting a class into a singleton by adding the proper stereotype, attribute and operation to it automatically.

Listing 5: Implementation of the ClassToSingleton Wizard

```
wizard ClassToSingleton {

  -- The wizard applies when a class is selected
  guard : self.isTypeOf(Class)

  title : 'Convert ' + self.name + ' to a singleton'

  do {
    -- Create the getInstance() operation
    var gi : new Operation;
    gi.owner := self;
    gi.name := 'getInstance';
    gi.visibility := VisibilityKind#vk_public;
    gi.ownerScope := ScopeKind#sk_classifier;

    -- Create the return parameter of the operation
    var ret : new Parameter;
    ret.type := self;
    ret.kind := ParameterDirectionKind#pdk_return;
    gi.parameter := Sequence{ret};

    -- Create the instance field
    var ins : new Attribute;
    ins.name := 'instance';
    ins.type := self;
    ins.visibility := VisibilityKind#vk_private;
    ins.ownerScope := ScopeKind#sk_classifier;
    ins.owner := self;

    -- Attach the <<singleton>> stereotype
    self.attachStereotype('singleton');
  }
}

-- Attaches a stereotype with the specified name
-- to the Model Element on which it is invoked
operation ModelElement attachStereotype(stereotypeName : String) {
    var stereotype : Stereotype;
39
40     -- Try to find an existing stereotype with this name
41     stereotype := Stereotype.allInstances.select(s|s.name = stereotypeName).first();
42
43     -- If there is no existing stereotype
44     -- with that name, create one
45     if (not stereotype.isDefined()){
46       stereotype := Stereotype.createInstance();
47       stereotype.name := stereotypeName;
48       stereotype.namespace := self.namespace;
49     }
50
51     -- Attach the stereotype to the model element
52     self.stereotype.add(stereotype);
53 }
```

The *guard* part of the wizard specifies that it is only applicable when the selection is a single UML class. The *title* part specifies a context-aware title that informs the user of the functionality of the wizard and the *do* part implements the functionality by adding the *getInstance* operation (lines 10-14), the *instance* attribute (lines 23-28) and the $<<$ *singleton* $>>$ stereotype (line 31).

The stereotype is added via a call to the `attachStereotype()` operation. Attaching a stereotype is a very common action when refactoring UML models, particularly where UML profiles are involved, and therefore to avoid duplication we have specified this reusable operation that checks for an existing stereotype, creates it if it does not already exists, and attaches it to the model element on which it is invoked. In practice this, and other common operations on UML elements, are located in a library of reusable operations (`uml.lib`) which we can import using the EOL `import` statement from different EWL modules.

An extended version of this wizard could also check for existing association ends that link to the class and for which the upper-bound of their multiplicity is greater than one and either disallow the wizard from executing on such classes (in the *guard* part) or update the upper-bound of their multiplicities to one (in the *do* part). However, in this section our aim is not to implement complete wizards that address all sub-cases but to provide a better understanding of the concrete syntax and the features of EWL. This principle also applies to the examples presented in the sequel.

Case 2: Renaming a Class

The most widely used convention for naming attributes and association ends of a given class is to use a lower-case version of the name of the class as the name of the attribute or the association end. For instance, the two ends of a one-to-many association that links classes `Book` and `Chapter` are most likely to be named `book` and `chapters` respectively. When renaming a class (e.g. from `Chapter` to `Section`) the user must then manually traverse the model to find all attributes and association ends of this type and update their names (i.e. from `chapter` or `bookChapter` to `section` and `bookSection` respectively). This can be a daunting process especially in the context of large models. Wizard `RenameClass` presented in Listing 6 automates this process.

Listing 6: Implementation of the RenameClass Wizard

```
wizard RenameClass {

  -- The wizard applies when a Class is selected
  guard : self.isKindOf(Class)

  title : 'Rename class ' + self.name

  do {
    var newName : String;

    -- Prompt the user for the new name of the class
    newName := UserInput.prompt('New name for class ' + self.name);
    if (newName.isDefined()) {
      var affectedElements : Sequence;

      -- Collect the AssociationEnds and Attributes
      -- that are affected by the rename
      affectedElements.addAll(
        AssociationEnd.allInstances.select(ae|ae.participant = self));
      affectedElements.addAll(
        Attribute.allInstances.select(a|a.type = self));

      var oldNameToLower : String;
      oldNameToLower := self.name.firstToLowerCase();
      var newNameToLower : String;
      newNameToLower := newName.firstToLowerCase();

      -- Update the names of the affected AssociationEnds
      -- and Attributes
      for (ae in affectedElements) {
          ae.replaceInName(oldNameToLower, newNameToLower);
          ae.replaceInName(self.name, newName);
      }
      self.name := newName;
    }
  }

}

-- Renames the ModelElement on which it is invoked
operation ModelElement replaceInName(oldString : String, newString : String) {
  if (oldString.isSubstringOf(self.name)) {
    -- Calculate the new name
    var newName : String;
    newName := self.name.replace(oldString, newString);

    -- Prompt the user for confirmation of the rename
48      if (UserInput.confirm('Rename ' + self.name + ' to ' + newName + '?')) {
49        -- Perform the rename
50        self.name := newName;
51      }
52    }
53  }
```

As with the `ClassToSingleton` wizard, the `guard` part of `RenameClass` specifies that the wizard is applicable only when the selection is a simple class and the *title* provides a context-aware description of the functionality of the wizard.

As discussed in Section 3, the information provided by the selected class itself does not suffice in the case of renaming since the new name of the class is not specified anywhere in the existing model. In EWL, and in all languages that build on EOL, user input can be obtained using the built-in `UserInput` facility. Thus, in line 12 the user is prompted for the new name of the class using the `UserInput.prompt()` operation. Then, all the association ends and attributes that refer to the class are collected in the `affectedElements` sequence (lines 14-21). Using the `replaceInName` operation (lines 31 and 32), the name of each one is examined for a substring of the upper-case or the lower-case version of the old name of the class. In case the check returns true, the user is prompted to confirm (line 48) that they want the feature to be renamed. This further highlights the importance of user input for implementing update transformations with fine-grained user control.

Case 3: Moving Model Elements into a Different Package

A common refactoring when modelling in UML is to move model elements, particularly Classes, between different packages. When moving a pair of classes from one package to another, the associations that connect them must also be moved in the target package. To automate this process, in Listing 7 we present the `MoveToPackage` wizard.

Listing 7: Implementation of the MoveToPackage Wizard

```
wizard MoveToPackage {

  -- The wizard applies when a Collection of
  -- elements, including at least one Package
  -- is selected
  guard {
    var moveTo : Package;
    if (self.isKindOf(Collection)) {
      moveTo := self.select(e|e.isKindOf(Package)).last();
    }
    return moveTo.isDefined();
  }

  title : 'Move ' + (self.size() - 1) + ' elements to ' + moveTo.name

  do {
```

```
17      -- Move the selected Model Elements to the
18      -- target package
19      for (me in self.excluding(moveTo)) {
20        me.namespace := moveTo;
21      }
22
23      -- Move the Associations connecting any
24      -- selected Classes to the target package
25      for (a in Association.allInstances) {
26        if (a.connection.forAll(c|self.includes(c.participant))){
27          a.namespace := moveTo;
28        }
29      }
30    }
31
32  }
```

The wizard applies when more than one element is selected and at least one of the elements is a `Package`. If more than one package is selected, the last one is considered as the target package to which the rest of the selected elements will be moved. This is specified in the `guard` part of the wizard.

To reduce user confusion over the package to which the elements will be moved, the name of the target package appears in the title of the wizard. This example shows the value of our choice to express the title as a dynamically calculated expression (as opposed to a static string). It is worth noting that in the `title` part of the wizard (line 14), the `moveTo` variable declared in the `guard` (line 7) is referenced. Through experimenting with a number of wizards, we have found that in complex wizards duplicate calculations need to be performed in the `guard`, `title` and `do` parts of the wizard. To eliminate this duplication we have extended the scope of variables defined in the `guard` part so that they are also accessible from the `title` and *do* part of the wizard.

6 RELATED WORK

In [15], an approach to expressing model refactorings as update transformations is described. There, similarly to our approach, update transformations are expressed as guarded actions. The main difference with our approach is that it does not consider user input. To our view this is somewhat limiting as user input has been shown to be necessary for achieving certain kinds of transformations such as the class renaming example presented in Section 5. Moreover, it is not clear that the language can be used to capture wizards with complex applicability criteria, such as the one presented in Section 5, since the examples presented there address wizards that apply to a single model element only.

In [16], the authors propose a graphical approach to defining update transformations targeting the Eclipse Modeling Framework. The transformations can then be interpreted using the AGG graph transformation engine [17] or compiled to Java.

The transformation language proposed in this tool [16] is rather simple and in absence of a proper query language (e.g. OCL), it is not clear how it can be used for complex scenarios such as the one demonstrated in Listing 7. In [18], a set of refactorings on UML models is proposed and specified using OCL pre and post conditions. The paper refers only to specifications and not implementations, and there is no mechanism for executing them to perform the refactorings automatically.

7 CONCLUSIONS AND FURTHER WORK

In this paper we have presented the Epsilon Wizard Language (EWL), a language tailored to specifying executable update transformations in the small. Compared with existing approaches, EWL provides a number of novel features such as support for capturing user input, enhanced modularity and flexible guards. We have presented tool support for EWL in the context of ArgoUML and demonstrated working examples of wizards specified using the language.

To extend our tool-support for EWL beyond ArgoUML and enable developers to use the language to specify wizards for Domain Specific Languages (DSLs), we are working towards integrating EWL with the Eclipse Graphical Modeling Framework (GMF), a framework for developing graphical editors for DSLs. Epsilon already provides support for EMF which is the underlying modelling framework of GMF, so aligning with GMF is straightforward and requires only the implementation of the necessary user interface components.

8 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] Shane Sendall and Wojtek Kozaczynski. Model Transformation the Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, September/October 2003.

[2] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA ’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[3] 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.

[4] Object Management Group. MOF QVT Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf.

[5] Andras Balogh, Daniel Varro. Advanced model transformation language constructs in the VIATRA2 framework. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287, New York, NY, USA, 2006. ACM Press.

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

[7] Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall PTR, 3rd edition, October 2004.

[8] Jack Herrington. *Code Generation in Action.* Manning, 2003. ISBN: 1930110979.

[9] 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.

[10] Microsoft Domain Specific Languages Framework, Official Web-Site. http://msdn.microsoft.com/ vstudio/ teamsystem/ workshop/ DSLTools/ default.aspx.

[11] Renaud Pawlak, Carlos Noguera, Nicholas Petitprez. Spoon: Program Analysis and Transformation in Java. Technical Report 5901, INRIA, May 2006.

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

[13] Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. Eclipse Development Tools for Epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, Esslingen, Germany, October 2006.

[14] 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.

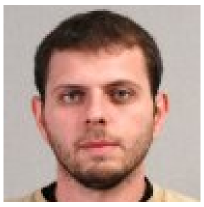
[15] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In Perdita Stevens and Jon Whittle and Grady Booch, editor, *Proc. UML 2003 - The Unified Modeling Language, 6th International Conference*, volume 2863 of *LNCS*, pages 159–174. Springer-Verlag, 2003.

[16] E. Biermann, K. Ehrig, C. Khler, G. Kuhns, G. Taentzer, and E. Weiss. EMF Model Refactoring based on Graph Transformation Concepts. In *Proc. Third International Workshop on Software Evolution through Transformations (SE-Tra'06)*, volume 3, Natal, Brazil, September 2006.

[17] Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In *Proc. Applications of Graph Transformations with Industrial Relevance, AGTIVE*, page 481, September 1999.

[18] Gerson Sunyé, Damien Pollet, Yves Le Traon, Jean-Marc Jézéquel. Refactoring UML Models. In *Proc. Int'l Conf. UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 134–148. Springer-Verlag, 2001.

ABOUT THE AUTHORS



Dimitrios S. Kolovos is a PhD student and Research Associate at the Department of Computer Science at The University of York, United Kingdom. He can be reached at dkolovos@cs.york.ac.uk. See also http://www-users.cs.york.ac.uk/∼dkolovos.

Richard F. Paige is a Lecturer at the Department of Computer Science at The University of York, United Kingdom. He can be reached at paige@cs.york.ac.uk. See also http://www-users.cs.york.ac.uk/∼paige.

Fiona A.C. Polack is a Senior Lecturer at the Department of Computer Science at The University of York, United Kingdom. She can be reached at fiona@cs.york.ac.uk. See also http://www-users.cs.york.ac.uk/∼fiona.

Louis M. Rose is an MEng student at the Department of Computer Science at The University of York, United Kingdom. He can be reached at lmr109@cs.york.ac.uk.