Novel Features in Languages of the Epsilon Model Management Platform

Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack
Department of Computer Science,
The University of York,
York, YO10 5DD, UK
{dkolovos, paige, fiona}@cs.york.ac.uk

ABSTRACT
We present a set of novel features that we have recently incorporated to model management languages of the Epsilon platform. We provide a detailed discussion on the usefulness of each feature and present motivating scenarios that highlight the benefits they deliver to programmatic model management.

1. INTRODUCTION
As Model-Driven Development (MDD) is increasingly used to construct complex systems, a growing number of model management languages are becoming available. For example, there are currently many languages for model-to-model transformation such as QVT [1], ETL, ATL [2], XText [3], Viatra2 [4], model-to-text transformation such as MOFScript [5], Xpand [3] and JET, and model validation such as OCL [6], EVL [7], and Check [3].

With the exception of the languages that belong to the Epsilon [8] and oAW [3] frameworks, contemporary model management languages are disconnected, i.e., each has been designed and implemented from scratch, often only conceptually reusing a subset of OCL. As discussed in [9], this leads to inconsistency and unnecessary diversity in the MDD toolkit as users need to learn and employ a number of languages with similar – but (unnecessarily) inconsistent – concrete syntaxes and features. Another issue identified in [9] is that by spending significant effort on re-implementing trivial features, such as model querying and navigation, designers of model management languages often misapply valuable resources which should instead be focused on developing novel, unsupported task-specific features. Furthermore, even in the situation where the designers of a language implement a novel feature, this is only supported in that particular language and the benefits cannot be used for other model management tasks.

To address those issues, as discussed in [8], Epsilon provides a layered architecture that enables designers of model management languages to reuse a significant amount of functionality when constructing a new model management language and concentrate on the task-specific features. Moreover, improvements and new features in the core language, EOL, are propagated to all task-specific languages atop it. In this paper we present a number of such features we have implemented in EOL, which due to the layered architecture of Epsilon are inherited by all other languages of the platform. Our aim is dual; to publicise the new and noteworthy features of Epsilon but also to demonstrate their importance so that similar frameworks and standards (such as QVT [1] and MOF2Text) might be influenced and progress to adopting similar mechanisms.

The rest of the paper is organized as follows. In Section 2 we present the mechanism that enables Epsilon languages to communicate with the underlying execution platform so that they can delegate computationally demanding and otherwise out-of-scope functionality to native objects. In Section 3, we stress the need for support for model transactions in model management languages and present how we have implemented this feature in Epsilon. In Section 4 we demonstrate the usefulness of features that enable programmatic user-interaction and illustrate the context-independent solution we have implemented in Epsilon. In Section 5 we motivate the need for profiling capabilities in model management languages and illustrate the different levels at which this is supported in Epsilon. In Section 6 we conclude and suggest directions for further work on the subject.

2. ACCESS TO NATIVE OBJECTS
As with all software languages, model management languages have a particular scope of applicability. For example, while a model-to-model transformation language should make it easy for the user to specify complex model-to-model transformations, it is not typically expected to be efficient at performing complex mathematical calculations, or to be able to invoke operating system services. Nevertheless, one can easily imagine scenarios where such tasks would be useful in the context of a model transformation. For instance, we have found complex mathematical calculations to be essential when expressing 3D graph layout algorithms as model transformations. Also, after code generation, it is often useful to compile the code using a call to the executable file of a suitable compiler, or even to perform tasks such as registering operating systems libraries and adding SQL queries directly to a relational database.

There are two alternatives for coping with such scenarios. The first approach is to enhance the model management language with such functionality, while the second is to delegate such tasks to constructs from the implementation platform. The first option would progressively require making the model management language of equivalent expressive power to a contemporary general-purpose programming language (e.g., Java, C#), which is not a realistic or useful option. Therefore, in Epsilon we have decided to take the second approach and allow the user to instantiate native objects of the underlying platform – Java for the current implementation – and invoke their methods using EOL. To achieve this, EOL adds a new...
Native type which represents an underlying Java class. Listing 1 demonstrates initializing and using an object of Native type.

Listing 1: Creating configuring and displaying a native Swing JFrame from within EOL
1 var frame := new Native("javax.swing.JFrame");
2 frame.setTitle := 'Opened using EOL';
3 frame.setBounds(0, 0, 100, 100);
4 frame.visible := true;

This EOL fragment instantiates a new Swing JFrame, sets its bounds and its title and makes it visible to the user. As discussed above, Epsilon can use any Java class as the implementation of a Native type. However, if the class also happens to implement the org.epsilon.eol.tools.ITool interface, the execution engine uses the setContext(EolContext context) method that the interface provides to enable the instantiated object to access the runtime of the Epsilon program in the context of which it is created. For example, the SchedulerTool (which can schedule a periodic execution of a particular parameter-less EOL operation), which is illustrated in Listing 2, demonstrates the usefulness of allowing a native object to access the runtime infrastructure directly.

Listing 2: Scheduling periodic execution of operations with SchedulerTool in EOL
1 var scheduler := new Native('org.epsilon.eol.tools.SchedulerTool');
2 -- Schedules the execution of the foo operation
3 scheduler.schedule('foo', 100, 50);
4 operation foo() {
5 'foo'.println();
6 }

3. MODEL TRANSACTIONS

Another novel feature provided by Epsilon is the ability to define model transactions; that is, atomic sequences of model modifications that either succeed or fail as a whole. Support for transactions is implemented in two layers. In the model connectivity layer (EMC), each model can define an instance of IModelTransactionSupport to manage transactions using the underlying mechanisms the specific modelling technology (e.g. EMF, MDR) provides. Moreover, the model repository that manages the models involved in a model management process acts as a façade offering high-level services that enable users to set-up, commit and rollback transactions on selected models. The transaction services EMC provides are used by the validation (EVL [7]) and the in-place model transformation (EWL [10]) languages of the platform that perform consistent in-place model updates.

As discussed in [7], when expressing validation constraints in EVL, engineers can also specify a number of fixes that the end-user can invoke to repair models that happen to violate these constraints. If a syntax or logical error exists in the fix part of the constraint, the model can be left in an inconsistent state. Therefore, before executing a fix part, the EVL runtime initiates a transaction which is only committed if all the statements of the fix have been executed without errors.

Similarly, as discussed in [10], each EWL wizard specifies a body that is responsible for performing an in-place model transformation (e.g., refactoring) on a set of model elements selected explicitly by a user. Again, a logical or syntax error in the body statements can compromise the validity or well-formedness of the model. Therefore, EWL wizard bodies are also executed in a transactional mode.

Moreover, EOL also provides an explicit transaction block, and an abort statement, which enables users to specify blocks of statements that should be executed in a transactional manner. We have found this feature to be particularly (mis)useful when producing random mutations of a given model. For example, in Listing 3 we create 100 random mutations of a model consisting of a number of processors. In each mutation we mark 10 processors as failed and then evaluate the availability of the resulting system. To achieve this without re-loading the original model every time, we initiate a transaction inside the for loop (line 5), perform the mutations, evaluate the availability of the system (line 19) and then explicitly roll-back the transaction (using the abort; statement in line 21) so that the model is restored to its initial state and is ready for the next mutation step.

Listing 3: Using transactions to mutate and evaluate models in EOL
1 var system := System.allInstances.first();
2 for (i in Sequence (1..100)) {
3 transaction {
4 var failedProcessors := Set;
5 while (failedProcessors.size() < 10) {
6 failedProcessors.add ((system.processors.random()));
7 }
8 for (processor in failedProcessors) {
9 processor.failed := true;
10 processor.moveTasksElseWhere();
11 }
12 system.evaluateAvailability();
13 abort;
14 }
15 }

4. CONTEXT-INDEPENDENT USER INTERACTION

Language designers often assume that model management tasks are only executed in a batch manner without human intervention. However, in our experience it is often useful for the user to provide feedback that can precisely drive the execution of a model management operation. At first glance, interaction with the user can be realized using the Native Object access feature of EOL discussed in Section 2.

However, model management operations can be executed in diverse runtime environments, and in each a different user-input method may be more appropriate. For example, when executed in the context of an IDE (such as Eclipse) visual dialogs are preferable, while when executed in the context of a server or from within an ANT build-file, a command-line user input interface is often more suitable. To abstract away from the different runtime environments and enable the user to specify user interaction statements uniformly and regardless of the runtime context, EOL provides the IUserInfo interface that can be realized in different ways according to the execution environment and attached to the runtime context via the EolContext.setUserInfo(IUserInfo userInfo) method. The IUserInfo specifies the following methods:

• confirm(message : String, default : Boolean) : Boolean
  Prompts the user to confirm if the condition described by the message holds.
prompt(message : String, default : String) : String: Prompts the user for a string in response to the message.

promptInteger(message : String, default : Integer) : Integer: Prompts the user for an Integer.

promptReal(message : String, default : Real) : Real: Prompts the user for a Real.

choose(options : Sequence, default : Any): Prompts the user to select one of the options.

All the methods of the IUserInput interface accept a default parameter. The purpose of this parameter is dual. First, it enables the designer of the model-management program to prompt the user with the most likely value as a default choice and secondly it enables a concrete implementation of the interface (UnattendedExecutionUserInput) which returns the default values without prompting the user at all. This can be used for unattended execution of Epsilon programs. Figures 1 and 2 demonstrate the interfaces through which input is required by the user when the exemplar Epsilon programs are executed (using an Eclipse-based and a command-line-based IUserInput implementation respectively).

We have found user-input facilities to be particularly useful in all model management tasks. Such facilities are essential for performing operations on live models, such as model validation and model refactoring, but they can also be useful in model comparison where marginal matching decisions can be delegated to the user, and also model transformation, where the user can interactively specify the elements that will be transformed into corresponding elements in the target model.

An example of the usefulness of user interaction in model transformation is illustrated in the ETL transformation presented in Listing 4. In this scenario, we need to transform only selected instances of the Tree metaclass in a Tree source model, conforming to the metamodel illustrated in Figure 3, to respective Nodes and Edges in a target Graph model that conforms to the metamodel of Figure 4. We achieve this by prompting the user in the guard part of the rule (lines 5, 6) to indicate for each Tree in the source model if they want it to be transformed or not.

Listing 4: Example of interactive model transformation using ETL

```java
rule Tree2Node
transform t : Tree!Tree
to n : Graph!Node {
  guard : UserInput.confirm('Transform tree ' + t.label + '?', true)
  n.label := t.label;
  var target : Graph!Node ::= t.parent;
  if (target.isDefined()) {
    var edge := new Graph!Edge;
    edge.source := n;
    edge.target := target;
  }
}
```

An example of the usefulness of user interaction in model comparison is displayed in the ECL program of Listing 5. In this case, to decide if two trees match we define a rule (Tree2Tree) that specifies that for two trees to match, they should have matching labels and parents. Instead of using exact matching for labels we use the compareTo operation defined in line 15. There, we use the Levenshtein [11] algorithm, to calculate a similarity for the two labels. If the similarity has a marginal value (between 0.4 and 0.6), in lines 24-26 we delegate the decision making for the particular pair to the user. Also, by providing result > 0.5 as the default value, in an unattended execution context similarities greater than 0.5 will be treated as positive matches.
Listing 5: Example of interactive string matching using ECL

```plaintext
pre |
 var simmetrics := new Native ('org.epsilon.ecl.tools.SimmetricsTool');
 |
rule Tree2Tree
 compare l : Left!Tree |
 with r : Right!Tree |
 compare : l.label.compareTo(r.label) and
 l.parent.matches(r.parent)
 |
operation String compareTo(other : String)
 : Boolean |
 var result := simmetrics.
 match(self, other, 'Levenshtein');
 if (result > 0.6) {
 return true;
 } else {
 if (result > 0.4) {
 return UserInput.confirm
 (self + ' ' matches ' + other + '?',
 result > 0.5);
 } else {
 return false;
 }
 |
```

5. EXECUTION PROFILING

First-order logic queries over models such as those supported by OCL and OCL-based languages (such as EOL) demonstrate the advantages of brevity and conciseness. However, this also makes it easy for the user to compose complex queries that are expensive to execute. This is particularly the case in state-changing languages such as EOL, where caching and reusing the results of complex queries automatically is not always an option. In the absence of a profiling mechanism, users typically encounter prolonged execution times which are particularly challenging to explain and attribute to specific parts of the code.

To address this issue, Epsilon provides integrated support for code profiling. This support is realized at two levels. At the top level, users can add @profile annotations to rules and operations in all Epsilon languages to specify that their execution time should be profiled. At a lower level, users can use an instance of the org.epsilon.eol.tools.ProfilerTool native type to specify blocks of statements that need to be profiled.

After the execution of a particular model management program (or an ANT build file that contains calls to Epsilon tasks) users can inspect the results of the profiling using a dedicated view in Eclipse. Through this view, a screenshot of which is displayed in Figure 5, users can inspect both aggregated and fine-grained data gathered during profiling. A more technical discussion on the Epsilon Profiling tools is available in [12].

We have found profiling to be particularly beneficial both from the end-user perspective and from the design/maintenance perspective as it has allowed us to identify bottlenecks in the implementations of our runtime engines which would otherwise be nearly impossible to locate. Therefore, we highly recommend the addition of similar facilities to other model management frameworks as well.

6. CONCLUSIONS AND FURTHER WORK

We have presented a set of novel features implemented in the context of the Epsilon Object Language, from which all languages of the Epsilon platform benefit due to Epsilon’s layered, reuse-oriented architecture. These features establish the foundations for achieving more complex and interesting model management scenarios such as interactive model transformation, comparison and model mutation which we have only briefly discussed here, but plan to explore in depth in the near future. Also, we are looking forward to identifying and implementing additional novel features in EOL from which all the model management of the platform will benefit.

7. ACKNOWLEDGEMENTS

The work in this paper is partially supported by the European Commission via the MODELPLEX project, co-funded under the “Information Society Technologies” Sixth Framework Programme (2006-2009).

8. REFERENCES


