

A Framework for Composing Modular and Interoperable Model Management Tasks

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

Department of Computer Science, University of York
{dkolovos, paige, fiona}@cs.york.ac.uk

Abstract. The ability to compose modular and interoperable model management tasks into automated workflows is essential for the widespread use of Model Driven Development. Driven by this need, we present a framework that enables developers to define, execute and profile such workflows. The framework is based on the widely-used ANT tool and Epsilon, a platform of integrated task-specific model management languages. We present the design and implementation of the framework and demonstrate its practicality and usefulness through a concrete example.

1 Introduction

A Model Driven Development (MDD) process typically involves both model management tasks, such as model validation, transformation and code generation, and classical software development tasks such as code compilation and deployment, each performed with appropriate tool support. Moreover, these tasks are seldom standalone; they often depend on the outcome of the execution of preceding tasks. Thus, for complex processes, coordinating the execution of lengthy sequences of tasks manually is counter-productive, uninteresting and error prone.

In this paper, we present a framework for composing and coordinating model management tasks with classical software development tasks into coherent workflows. We have built our technical solution atop ANT [1], a robust and widely used tool for composing classical software development tasks, and Epsilon [2], a platform of task-specific languages for model management that addresses a wide range of tasks including model transformation, validation, comparison, merging and text generation.

The rest of the paper is organized as follows. In Section 2 we provide a short discussion on the aims and structure of Epsilon. In Section 3 we briefly discuss ANT and outline its extensibility mechanisms that enable us to implement our solution atop it. In Section 4 we outline the integration challenges between the two tools. In Section 5 we present design and implementation details of our technical solution that consists of a core framework and a number of ANT tasks that support different model management activities. To demonstrate the practicality and usefulness of our approach, in Section 6 we provide a concrete example. In Section 7 we evaluate our solution by presenting its advantages and shortcomings. In Section 8 we provide an overview of related work and compare with our

approach, and in Section 9 we conclude and discuss on directions for further research on the subject.

2 The Epsilon Model Management Platform

Epsilon [2], is a platform of integrated task-specific model management languages, and a component of the Eclipse Generative Modeling Technologies (GMT) research incubator project. Epsilon provides languages for direct manipulation of models (EOL) [3], model merging (EML) [4], model comparison (ECL) [5], model-to-model transformation (ETL), model validation (EVL) [6] and model-to-text transformation (EGL). All the languages of the platform build on a common OCL-based model navigation and manipulation language (EOL) and a common runtime environment, and are therefore highly interoperable. For example, an *operation* defined using EOL can be reused (imported) as-is by the model-to-model and the model-to-text transformation languages. Moreover, due to the common runtime environment, modules of different languages can exchange variables with each other, a feature which is discussed in detail later. With regard to supported modelling technologies, the architecture of Epsilon allows users to manage models of different technologies such as MDR and EMF models and XML documents and even implement support for additional formats.

3 The ANT Tool

ANT is a robust and widely-used tool for composing automated workflows from small reusable activities. An ANT *project* consists of a number of *tasks* grouped in the form of *targets*. Each ANT task is responsible for a distinct activity and can either succeed or fail. Each task also depends on the outcome of a list of preceding tasks. Exemplar activities implemented by ANT tasks include filesystem management, compiler invocation, execution of SQL statements etc. In terms of concrete syntax, ANT provides an XML-based syntax for defining workflows, for which visual editors are also available. In Listing 1.1, an exemplar ANT project that compiles a set of Java files is illustrated.

Listing 1.1. Compiling Java classes using the `javac` task

```
<javac srcdir="${src}"
  destdir="${build}"
  classpath="dependencies.jar"
  debug="on"
  source="1.4"/>
```

3.1 The Extensibility Mechanism of ANT

Binding between the XML tags describing the tasks and the actual implementations of the tasks is implemented using a light-weight mechanism. First, the tag (in the example of Listing 1.1, `javac`) is resolved to a Java class that implements the `org.apache.ant.Task` interface (in the case of `javac`, the class is

org.apache.tools.ant.taskdefs.Javac) via a configuration file. Then, the attributes of the tasks (e.g. *srcdir*) are set using the reflective features that Java provides. Finally, the *execute()* method of the task is invoked to perform the actual job.

This lightweight and straightforward way of defining tasks has rendered ANT particularly popular in the Java development community and currently there is a large number of tasks contributed by ANT users, ranging from invoking tools such as code generators and XSLT processors, to emulating logical control flow structures such as *if* conditions and *while* loops.

4 Integration Challenges

Having outlined Epsilon and ANT, in this section we discuss the challenges that arise when attempting to integrate the two tools and justify the need for an additional layer between them.

As discussed in Section 2, each model management task operates on a number of models. Since models are typically serialized in the file-system, before a task is executed the respective models must be parsed and loaded into memory. In the absence of a more elaborate framework, each model management task must take responsibility for parsing and storing the models it operates on. However, in most workflows more than one task operates on the same models sequentially; parsing/storing the same models many times in the context of a workflow is an expensive operation both time and memory-wise, particularly as the size of models increases. Therefore, providing a facility that eliminates the need to parse/store models repeatedly in the context of the same workflow is essential.

Another challenge is that of inter-task communication. In the absence of a communication framework that allows model management tasks to exchange information with each other, it is often the case that more than one tasks end up performing the same (potentially expensive) queries. By contrast, a communication framework would enable time and resource intensive calculations to be performed once and their results to be communicated to all interested subsequent tasks.

5 Extending ANT with Support for Epsilon Model Management Tasks

Having discussed ANT, Epsilon and the challenges their integration poses, we now present the technical solution we have implemented to enable developers to invoke model management tasks in the context of ANT workflows. Our solution is composed of a core framework that addresses the challenges discussed in Section 4, and a set of specific tasks, each implementing a distinct model management activity. Figure 1 illustrates the design of the core framework.

5.1 The Core Framework

The *EpsilonTask* task An ANT task can access the project in which it is defined via the *Task.getProject()* method. To facilitate sharing of arbitrary infor-

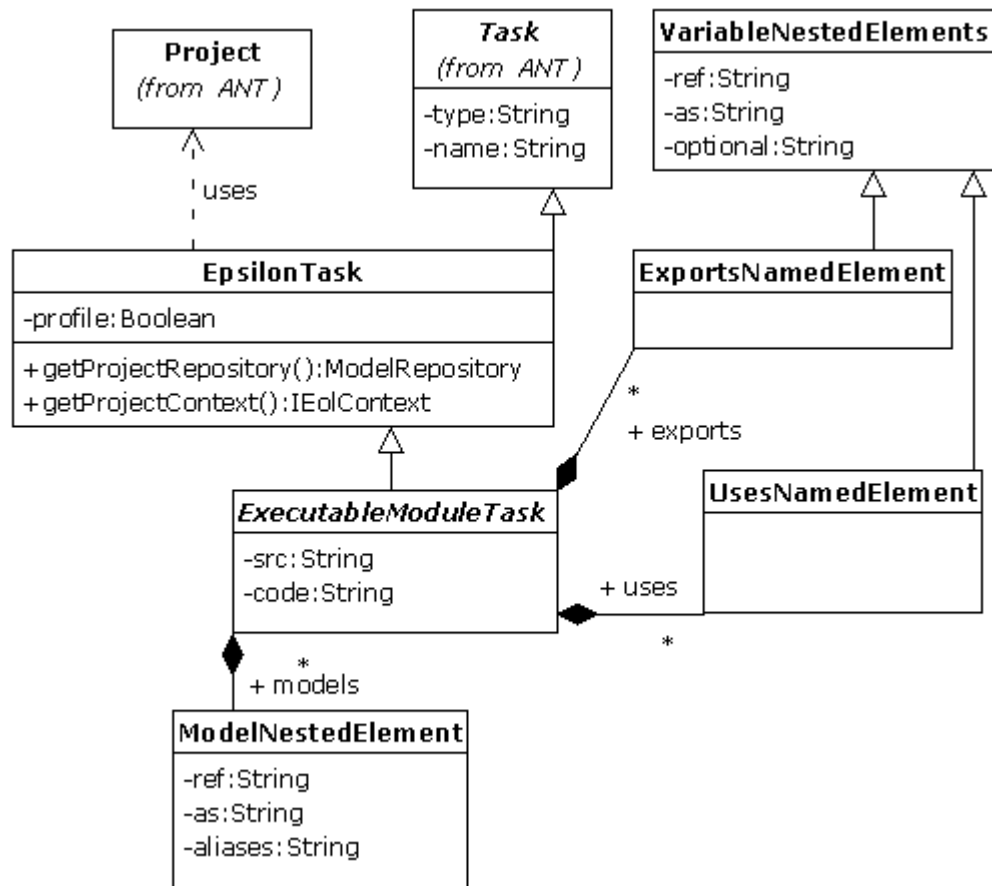


Fig. 1. Core Framework

mation between tasks, ANT projects provide two convenience methods, namely *addReference(String key, Object ref)* and *getReference(String key) : Object*. The former is used to add key-value pairs, that are then accessible using the latter from other tasks of the project.

To avoid loading models multiple times and to enable on-the-fly management of models from different Epsilon tasks without needing to store and re-load the models after each task, we have added a reference to a project model repository to the current ANT project using the *addReference* method discussed above. Therefore, all the subclasses of the abstract *EpsilonTask* can invoke the *getProjectRepository()* method to access the project model repository. Also, to provide a variable sharing mechanism that enables inter-task communication, we have used the same technique to add a shared context that all Epsilon tasks can access via the *getProjectContext()* method.

Model Loading Task The *LoadModelTask* (*epsilon.loadModel*) loads a model from an arbitrary location (e.g. file-system, database) and adds it to the project repository so that subsequent Epsilon tasks can query or modify it. Since Epsilon supports many modelling technologies (e.g. EMF, MDR, XML), the *LoadModelTask* defines only three generic attributes. The *name* attribute specifies the name of the model in the project repository. The *type* attribute specifies the modelling technology in which the model is captured and is used to resolve the technology-specific model loading functionality. Finally, the *aliases* attribute defines a comma-separated list of alternative names by which the model can be accessed in the model repository. The rest of the information needed to load a model is implementation-specific and is therefore provided through *parameter* nested elements, each one defining a pair of *name-value* attributes.

Model Storing Task The *StoreModelTask* (*epsilon.storeModel*) stores a model residing in the project repository. The *StoreModelTask* defines two attributes. The *name* attribute specifies the name of the model to be stored and the *target* attribute specifies the location where the model will be stored. The *target* attribute is optional and if it is not defined, the model is stored in the location from which it was originally loaded.

The Abstract Executable Module Task This abstract task is the basis of all the model management tasks presented in Section 5.2. Its aim is to encapsulate the commonalities of Epsilon tasks in order to reduce duplication among them. In Epsilon, specifications of model management tasks are organized in executable modules. While modules can in principle be stored anywhere (e.g. in a database or in a remote location), in the case of the workflow we assume that they are stored in the local file-system. Thus, this abstract task defines an *src* attribute that specifies the path of the source file in which the Epsilon module is stored. It also defines the following nested elements:

0..n model nested elements Through the *model* nested elements, each task can define which of the models, loaded in the project repository it needs to access. Each *model* element defines three attributes. The *ref* attribute specifies the name of the model that the task needs to access, the *as* attribute defines the name by which the model will be accessible in the context of the task, and the *aliases* defines a comma-delimited sequence of alternative names for the model in the context of the task.

0..n parameter nested elements The *parameter* nested elements enable users to pass String parameters to tasks. Each *parameter* element defines a *name* and a *value* attribute. Before executing the module, each *parameter* element is transformed into a String variable with the respective name and value which is then made accessible to the module.

0..n exports nested elements To facilitate low-level integration between different Epsilon tasks, each task can export a number of variables to the project context, so that subsequent tasks can access them later. Each *export* nested element defines the three attributes. The *ref* attribute specifies the name of the variable to be exported, the *as* string attribute defines the name by which the variable is stored in the project context and the *optional* boolean attribute specifies whether the variable is mandatory. If *optional* is set to *false* and the module does not specify such a variable, an ANT *BuildException* is raised.

0..n uses nested elements The *uses* nested elements enable tasks to import variables exported by previous Epsilon tasks. Each use element supports three attributes. The *ref* attribute specifies the name of the variable to be used. If there is no variable with this name in the project context, the ANT project properties are queried. This enables Epsilon modules to access ANT parameters (e.g. provided using command-line arguments). The *as* attribute specifies the name by which the variable is accessible in the context of the task. Finally, the *optional* boolean parameter specifies if the variable must exist in the project context.

5.2 Model Management Tasks

Having discussed the core framework, in this section we present the model management tasks that we have implemented atop it, using languages of the Epsilon platform. As the languages involved have already been discussed in detail elsewhere, in the following paragraphs we present each language briefly and provide references to external resources for additional details on their syntax and semantics.

Model Validation Task To validate models, we have implemented the *EvlTask* (*epsilon.evl*) that executes an Epsilon Validation Language (EVL) module. Compared with OCL, EVL offers a number of additional features such as distinguishing between constraints and critiques, support for meaningful error messages, and support for capturing user input and repairing identified inconsistencies in a semi-automatic manner. A complete discussion on EVL is provided in [6]. The result of the validation process is stored in an internal trace, which can be exported using the *exportValidationTrace* attribute so that it can be accessed by subsequent tasks using the *exports-uses* mechanism as discussed in Section 5.1. The *epsilon.evl* task also supports the *failOnErrors* and *failOnWarnings* boolean attributes that specify if an ANT *BuildException* should be raised in case of unsatisfied constraints or critiques, respectively.

Model-to-Model Transformation Task To support model to model transformations, we have implemented the *EtlTask* (*epsilon.etl*) that executes an Epsilon Transformation Language (ETL) [7] module. Similarly to the validation task, the internal trace that is established during the execution of the ETL module can be exported to the project context using the *exportTransformationTrace* attribute.

Model Comparison Task To compare models, we have implemented the *EclTask* (*epsilon.ecl*) that executes an Epsilon Comparison Language (ECL) module. ECL [5] is a rule-based language that supports matching and conformance checking between models of arbitrary metamodels. Similarly to EVL and ETL, the result of the comparison is stored in an internal trace, which can be exported using the *exportMatchTrace* attribute so that it can be queried by subsequent tasks using the *exports-uses* mechanism.

Model Merging Task To merge models we have implemented the *EmlTask* (*epsilon.eml*) that executes an Epsilon Merging Language (EML) module. EML [4], is a rule-based language for model merging that builds atop the transformation language (ETL). In EML, model merging is performed in two steps. In the first step, elements from the input models are compared either manually or using ECL match rules. Then, elements that match are (optionally) merged using EML merge-rules and elements that don't match are (optionally) transformed using ETL transform-rules.

The *epsilon.eml* task provides the *importMatchTrace* attribute for importing a match-trace calculated in another (typically ECL) task of the workflow and the *exportMergeTrace* and *exportTransformationTrace* attributes that enable it to export the traces established during its execution.

Model-to-Text Transformation Task To support model to text transformations, we have implemented the *EglTask* (*epsilon.egl*) that executes an Epsilon Generation Language (EGL) module. EGL [8] is a template language, similar to XPand [9] and MOFScript [10], suitable for producing text-based artefacts by combining static text and model-based information. In addition to the attributes defined by *ExecutableModuleTask*, *EglTask* also defines a *target* attribute that defines the path of the file where the generated text will be stored.

Generic Model Manipulation Task To provide support for model management tasks that do not fall into a main pattern (rule-based transformation, validation etc.) we have implemented the *EolTask* (*epsilon.eol*), that is used to execute an EOL module. EOL [3] is an OCL-based imperative language that facilitates a procedural approach to querying and manipulating models. The task does not define any additional attributes to those provided by its ancestor *ExecutableModuleTask*.

6 Example

This section presents a concrete example that demonstrates the proposed solution. In this example, we need to merge two models that conform to the minimal SimpleOO metamodel presented in Figure 2. The process involves three interdependent steps.

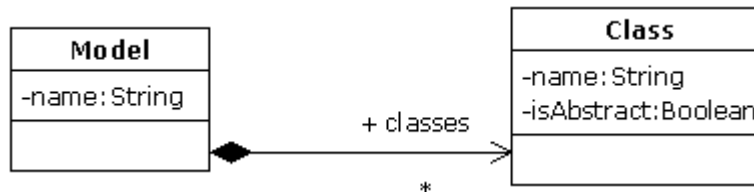


Fig. 2. The SimpleOO metamodel

In the first step, we need to establish correspondences between the two models. More specifically we need to specify when two elements match so that they can later on be merged instead of copied twice. This is achieved by the ECL module of Listing 1.2.

Listing 1.2. The CompareSimpleOO.ecl module

```

1 rule Models
2   match l : Left!Model with r : Right!Model {
3     compare : true
4   }
5
6 rule Class
7   match l : Left!Class with r : Right!Class {
8     compare : l.name = r.name
9   }
  
```

Once we have established the correspondences between the two models, we need to ensure that the models are consistent with each other. In this case, we need to ensure that matching classes are both either abstract or non-abstract. We check this property using the EVL module of Listing 1.3, which utilises (in line 12) the trace (matchTrace) established during the previous ECL model comparison task.

Listing 1.3. The ValidateMatchingClasses.evl validation module

```

1 context SimpleOO!Class {
2   constraint BothAbstractOrNot {
3     guard : self.getMatching().isDefined()
4     check : self.getMatching().isAbstract = self.isAbstract
5     message : 'Inconsistent value in feature "abstract" '
6               + 'of class ' + self.name
7   }
8 }
9
10 @cached
11 operation Any getMatching() : Any {
12   var match := matchTrace.matches.selectOne
13               (m|m.left = self or m.right = self);
14   if (not match.isDefined()) { return null; }
15   if (match.left = self) { return match.right; }
16   else { return match.left; }
17 }
  
```

Finally, if the two models are found to be consistent, we can merge them on the correspondences identified during the model comparison task using the EML module of Listing 1.4.

Listing 1.4. The MergeSimpleOO.eml module

```
1 rule MergeModel
2   merge l : Left!Model with r : Right!Model into t : Target!Model {
3     t.name := l.name + ' and ' + r.name;
4     t.contents := l.contents + r.contents;
5   }
6 rule MergeClass
7   merge l : Left!Class with r : Right!Class into t : Target!Class {
8     t.name := l.name;
9     t.isAbstract := l.isAbstract;
10  }
11 rule CopyModel
12   transform s : Source!Model to t : Target!Model {
13     t.contents := s.contents;
14   }
15 rule CopyClass
16   transform s : Source!Class to t : Target!Class {
17     t.name := s.name;
18     t.isAbstract := s.isAbstract;
19   }
```

The workflow of Listing 1.5 integrates and coordinates the comparison, validation and merging tasks demonstrated above. In line 1, the *compose* target is defined as the default target of the workflow. The *compose* target in line 3 specifies that it depends on the *loadModels*, *compare*, *validate* and *merge* tasks. If one of the tasks fails for some reason (e.g. if the validation task determines that the models are not consistent), the remainder tasks are not executed.

In line 5 The *loadModel* target uses the *epsilon.loadModel* task three times to load the involved models (*Left*, *Right* and *Target*) in the project model repository. In line 21, the *compare* target uses the *epsilon.ecl* task to execute the ECL module of Listing 1.2. In line 22, the task specifies that it exports the match trace it has established to the common project context as a variable named *matchTrace*. In line 28, the *validate* target uses the *epsilon.evl* task to execute the EVL module of Listing 1.3. In line 32, the task specifies that it needs to access the *matchTrace* variable that the previous ECL task exported to the project context. Finally, in line 36, the *merge* target uses the *epsilon.eml* task to execute the EML module of listing 1.4. In line 37 the task defines that it will use the *matchTrace* as the correspondence trace on which it will merge the two models.

Listing 1.5. The ANT workflow that integrates the three tasks

```
1 <project default="compose">
2
3   <target name="compose" depends="loadModels,compare,validate,merge">
4
5     <target name="loadModels">
6       <epsilon.loadModel name="Left" type="EMF">
7         <parameter name="modelFile" file="Left.model"/>
8         <parameter name="metamodelUri" value="SimpleOO"/>
9       </epsilon.loadModel>
10      <epsilon.loadModel name="Right" type="EMF">
11        <parameter name="modelFile" file="Right.model"/>
12        <parameter name="metamodelUri" value="SimpleOO"/>
13      </epsilon.loadModel>
14      <epsilon.loadModel name="Target" type="EMF">
15        <parameter name="modelFile" file="Target.model"/>
16        <parameter name="metamodelUri" value="SimpleOO"/>
17        <parameter name="storeOnDisposal" value="true"/>
18      </epsilon.loadModel>
19    </target>
20  </target>
21 </project>
```

```

18 </epsilon.loadModel>
19 </target>
20
21 <target name="compare">
22 <epsilon.ecl src="CompareSimple00.ecl" exportmatchtrace="matchTrace">
23 <model ref="Left" alias="Source"/>
24 <model ref="Right" alias="Source"/>
25 </epsilon.ecl>
26 </target>
27
28 <target name="validate">
29 <epsilon.evl src="ValidateMatchingClasses.evl">
30 <model ref="Left" alias="Simple00"/>
31 <model ref="Right" alias="Simple00"/>
32 <uses ref="matchTrace"/>
33 </epsilon.evl>
34 </target>
35
36 <target name="merge">
37 <epsilon.eml src="MergeSimple00.eml" usematchtrace="matchTrace">
38 <model ref="Left" alias="Source"/>
39 <model ref="Right" alias="Source"/>
40 <model ref="Target"/>
41 </epsilon.eml>
42 </target>
43 </project>

```

7 Evaluation

By building on the robust and widely-used ANT tool, the proposed solution can be used to define complex workflows that include both model-management and classical software development tasks. Given the wide-spread acceptance of ANT, our approach should be straightforward to use by most contemporary developers. Also, it benefits from the wealth of existing tool support for ANT such as the built-in Eclipse ANT editing and execution tools, visualization tools such as VizAnt¹, and analysis tools such as AntUtility².

From a functionality perspective, the proposed solution covers a wide range of model management tasks and provides features such as a generic model loading and sharing mechanism, an inter-task communication facility, precise error reporting and task and code profiling.

On the other hand, by building our solution atop ANT we have willingly made a compromise on the level of control the user has on defining the workflow process. While ANT provides tasks that can emulate loops and control flow branches, it is not a programming language; therefore, workflows with particularly complex execution logic can be challenging - if indeed they are even possible - to define. Also, building atop ANT adds an additional execution cost introduced by the loosely-coupled (and thus costly) task discovery mechanism ANT features. Finally, while the framework makes it possible to integrate with model management tools and languages other than Epsilon, this may be challenging in practice mainly because each tool uses a different internal model representation scheme for accessing and modifying the contents of models.

¹ <http://vizant.sourceforge.net/>

² <https://antutility.dev.java.net/>

8 Related Work

In this section we discuss on other approaches that address the problem of constructing workflows of model management tasks and compare them with our solution.

OpenArchitectureWare (oAW) [11] defines a proprietary framework for coordinating model management tasks. The oAW workflow language [12] is very similar to ANT and thus almost everything that can be achieved with ANT can also be achieved with it. However, implementing support for tasks such as code compilation, deployment, file management etc. - which do not come out of the box with oAW - would require significant - and most importantly duplicated - effort. oAW partly overcomes this issue by providing an ANT task that can execute oAW workflows. Nevertheless, this approach prohibits individual model management (oAW) tasks from depending on non-model-management (ANT) tasks (only entire oAW workflows can depend on other ANT tasks). Moreover, the range of supported model management tasks in our solution is different as oAW does not support model comparison and merging tasks and Epsilon does not support text-to-model transformation tasks.

Our choice to build our solution atop ANT was influenced by AM3 [13] which also provides ANT tasks for loading, storing and transforming models using the Atlas Transformation Language (ATL) [14]. ATL can be used to implement various model management tasks such as model validation and text generation. The main difference with our approach is that instead of using a single language for all tasks, for each task we use a language that is tailored to the task's specific requirements.

ModelBus [15] proposes an approach based on web services to support integration of model management tasks. Its advantages over our approach is that, since it is based on Web services, it can be used to integrate tools running in different machines across a network. However, ModelBus requires all models to be expressed in XMI (even if they are natively stored in a non-XMI format) while our solution allows users to manage models of diverse modelling technologies and formats.

9 Conclusions and Further Work

In this paper we have presented a framework for composing model management tasks expressed using languages of the Epsilon platform into automated workflows. Our solution builds on the robust and widely used ANT tool and thus, enables developers to add model management tasks to their workflows with little effort and complexity. Compared with existing approaches, our solution supports a wider range of model management tasks including model transformation, validation, comparison, merging and text generation and provides enhanced interoperability between the composed tasks due to the common infrastructure on which Epsilon languages have been built on. In the future we plan to provide support for tasks implemented by model management languages beyond Epsilon.

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).

References

1. Steve Holzner. *Ant: The Definitive Guide, Second Edition*. O’Reilly, April 2005. ISBN 0-596-00609-8.
2. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon). <http://www.eclipse.org/gmt/epsilon>.
3. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.
4. 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.
5. 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.
6. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis*, 2007.
7. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. The Epsilon Transformation Language. In *Proc. 1st International Conference on Model Transformation*, Zurich, Switzerland, July 2008. to appear.
8. Louis M. Rose. The Epsilon Generation Language (EGL). MEng Thesis, Department of Computer Science, The University of York, 2008.
9. Sven Efftinge. XPand Language Reference. http://www.eclipse.org/gmt/oaw/doc/4.1/r20_xPandReference.pdf.
10. Jon Oldevik. MOFScript User Guide. <http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf>.
11. openArchitectureWare, Official Web-Site. <http://www.openarchitectureware.org/>.
12. Sven Efftinge, Markus Voelter. OpenArchitectureWare 4.1 Workflow Engine Reference. web resource. http://www.eclipse.org/gmt/oaw/doc/4.1/r05_workflowReference.pdf.
13. F Allilaire, J Bezivin, H Bruneliere, and F Jouault. Global Model Management In Eclipse GMT/AM3. In *Proc. Eclipse Technology eXchange workshop (eTX), ECOOP*, Nantes, France, 2006.
14. 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.
15. Model Driven Development Integration Project. ModelBus. <http://www.eclipse.org/mddi>.