The Epsilon Generation Language

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

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

Abstract. We present the Epsilon Generation Language (EGL), a model-to-text (M2T) transformation language that is a component in a model management tool chain. The distinctive features of EGL are described, in particular its novel design which inherits a number of language concepts and logical features from a base model navigation and modification language. The value of being able to use a M2T language as part of an extensible model management tool chain is outlined in a case study, and EGL is compared to other M2T languages.

1 Introduction

For Model-Driven Development to be applicable in the large, and to complex systems, mature and powerful *model management* tools and languages must be available. Such tools and languages are beginning to emerge, e.g., model-to-model (M2M) transformation tools such as ATL [8] and VIATRA [19], workflow architectures such as oAW [17], and model-to-text (M2T) transformation tools such as MOFScript [15] and XPand [17].

Whilst there are some mature model management tools, most such tools are stand-alone, or are loosely integrated through their ability to manipulate and manage the same kind of models, for instance via Eclipse EMF. (An exception is oAW, which supports model management workflows). These limitations mean that development of new tools often entails substantial effort, with few opportunities for reuse of language constructs and tools [13]. However, model management tasks have many common requirements (e.g., the need to be able to traverse models), share common concepts (e.g., the ability to query models) and have a common logic. There is substantial value, for developers and users, in integrating model management tools, to share features and facilitate construction of support for new model management tasks. Integrated tools improve our ability to provide rich automated support for model management in the large.

M2T transformation is an important model management task with a number of applications, including model serialisation (enabling model interchange); code and documentation generation; and model visualisation and exploration. In 2005, the OMG [9] recognised the lack of a standardised approach to performing M2T transformation with its M2T language RFP [16]. Various MDD tool vendors have developed M2T languages, including JET [5], XPand and MOFScript. None of these M2T languages has been built from other model management languages, and none directly exploits existing M2M support – the new languages have been developed either from scratch (e.g., JET and MOFScript), or as a component that can be applied within a modelling workflow (e.g., XPand).

Our approach is to create tools that are components in an *extensible* and *integrated* model management tool chain. This paper introduces the *Epsilon Generation Language (EGL)*, a language for specifying and performing M2T transformations. We describe EGL's basic and distinctive features, with particular emphasis on the advantages of building EGL as part of an extensible, integrated platform for model management; we illustrate the minimalist derivation needed, from the existing EOL [13] that supports model navigation and modification.

We start with an overview of key concerns for M2T transformation tools. In Section 3, we discuss the features and tool support provided by EGL. We discuss EGL's unique features, and explain its development from EOL, focusing on how EOL's design has been reused in EGL. In Section 4, a case study demonstrates the use of EGL to perform model visualisation. In Section 5, we compare EGL to other M2T transformation tools. Finally, in Section 6, we discuss future work.

2 Background

In this section, we briefly outline the key concerns of an effective M2T transformation solution. We also briefly describe the Epsilon model management platform, and its support for building new languages and tools.

2.1 Concerns of Model-to-Text Transformation

There are four key concerns in any M2T transformation solution.

Repeatability. M2T transformations may need to be repeatable, so that changes made to models percolate through to generated text. However, repeated invocation of transformations may need to respect hand-written changes that have been made to generated artefacts [16].

Traceability. After performing a M2T transformation, it should be possible to determine the elements of the source model from which a portion of the text has been produced. Such traceability is particularly valuable when debugging a model or when auditing the development process.

Readability. An M2T solution must maintain readability aspects, such as layout and indentation.

Flexibility. M2T transformations, like M2M, need to be flexible; one approach is to support parameterised transformation definitions [10].

2.2 The Epsilon Platform

Epsilon, the Extensible Platform for Specification of Integrated Languages for mOdel maNagement [11], is a suite of tools and domain-specific languages for model-driven development. Epsilon comprises a number of integrated model management languages, based upon a common infrastructure, for performing tasks such as model merging, model transformation and intermodel consistency checking [12]. Whilst many model management languages are bound to a particular subset of modelling technologies, limiting their applicability [14], Epsilon is metamodel-agnostic – models written in any modelling language can be manipulated by Epsilon's model management languages. (Epsilon currently supports models implemented using EMF, MOF 1.4, pure XML, or CZT.)

Figure 1 illustrates the various components of Epsilon.

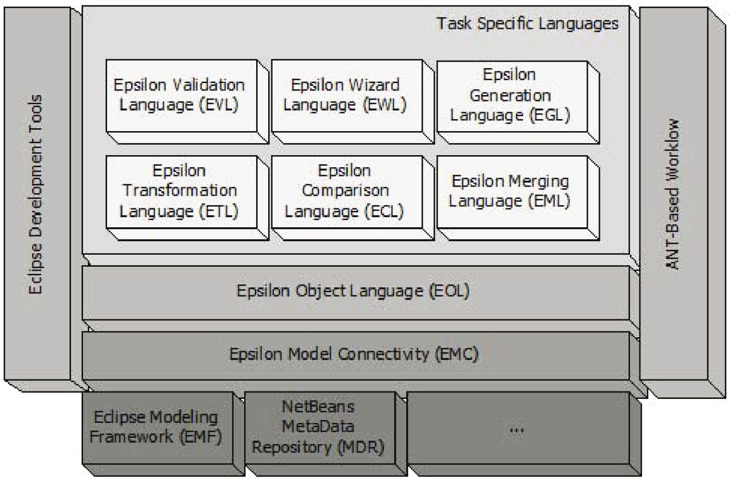

Task Specific Languages
Epsilon Validation Language (EVL)
Epsilon Wizard Language (EWL)
Epsilon Generation Language (EGL)
Epsilon Transformation Language (ETL)
Epsilon Comparison Language (ECL)
Epsilon Merging Language (EML)
Epsilon Object Language (EOL)
Epsilon Model Connectivity (EMC)
Eclipse Development Tools
ANT-Based Workflow
Eclipse Modeling Framework (EMF)
NetBeans MetaData Repository (MDR)
...

Fig. 1. The architecture of Epsilon

The design of Epsilon promotes reuse when building task-specific model management languages and tools. Each individual Epsilon language (e.g., ETL, ECL, EGL) can be reused wholesale in the production of new languages. Ideally, the developer of a new language only has to design language concepts and logic that do not already exist in Epsilon languages.

EGL follows this principle, and inherits concepts and logic from Epsilon's base language, EOL, as described in Section 3. First, we outline EOL's features, particularly as they pertain to EGL.

2.2.1 The Epsilon Object Language. The core of the platform is the Epsilon Object Language (EOL) [13]. EOL's scope is similar to that of OCL. However, EOL boasts an extended feature set, which includes the ability to update models, conditional and loop statements, statement sequencing, and access to standard output and error streams. Every Epsilon language re-uses EOL, so improvements to this object language enhance the entire platform.

A recent enhancement to EOL is the provision of constructs for profiling. EOL also allows developers to delegate computationally intensive tasks to extension points, where the task can be authored in Java. This now allows developers using any Epsilon language to monitor and fine-tune performance – in EGL, this allows fine-tuning of M2T transformations.

EOL itself provides the most basic M2T transformation facilities, because every EOL type provides `print` and `println` methods, which append a textual representation of the instance to the default output stream. However, native EOL is insufficient for M2T in the large – transformation specifications littered with explicit print statements become unreadable, and EOL alone does not support the sorts of features, specific to M2T transformation, which address the concerns identified in Section 2.1.

3 The Epsilon Generation Language (EGL)

EGL provides a language for M2T in the large. EGL is a model-driven template-based code generator, built atop Epsilon, and re-using all of EOL. In this section, we discuss the design of EGL and its construction from existing Epsilon tools.

3.1 Abstract Syntax

Figure 2 depicts the abstract syntax of EGL's core functionality.

In common with other template-based code generators, EGL defines *sections*, from which templates may be constructed. Static sections delimit sections whose contents appear verbatim in the generated text. Dynamic sections contain executable code that can be used to control the generated text.

In its dynamic sections, EGL re-uses EOL's mechanisms for structuring program control flow, performing model inspection and navigation, and defining custom operations. EGL provides an EOL object, `out`, for use within dynamic

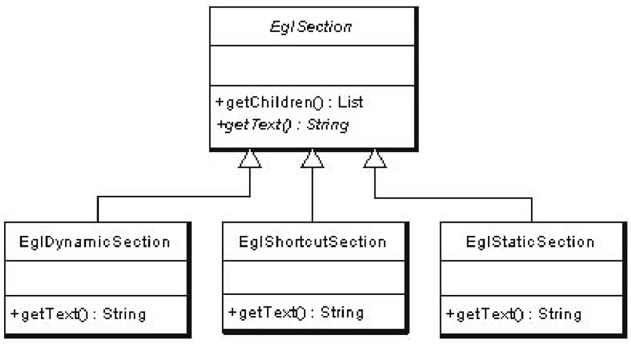


Fig. 2. The abstract syntax of EGL's core

sections. This can be used to perform operations on the generated text, such as appending and removing strings and specifying the type of text to be generated.

EGL also provides syntax for defining *dynamic output* sections, which provide a convenient shorthand for outputting text from within dynamic sections. Similar syntax is often provided by template-based code generators.

3.2 Concrete Syntax

The concrete syntax of EGL mirrors the style of other template-based code generation languages. The tag pair `[% %]` is used to delimit a dynamic section. Any text not enclosed in such a tag pair is contained in a static section. Listing 1.1 illustrates the use of dynamic and static sections to form a basic EGL template.

Listing 1.1. A basic EGL template

```
[% for (i in Sequence{1..5}) { %]
i is [%=i%]
[% } %]
```

The `[%=expr%]` construct is shorthand for `[% out.print(expr); %]`, which appends `expr` to the output generated by the transformation. Note that the `out` keyword also provides `println(Object)` and `chop(Integer)` methods, which can be used to construct text with linefeeds, and to remove the specified number of characters from the end of the generated text.

EGL exploits EOL's model querying capabilities to output text from models specified as input to transformations. For example, the EGL template depicted in Listing 1.2 may be used to generate text from a model that conforms to a metamodel that describes an object-oriented system.

Listing 1.2. Generating the name of each Class contained in an input model

```
[% for (class in Class.allInstances) { %]
[%=class.name%]
[% } %]
```

3.3 Parsing and Preprocessing

EGL provides a parser which generates an abstract syntax tree comprising static, dynamic and dynamic output nodes for a given template. A preprocessor then translates each section into corresponding EOL: static and dynamic output sections generate `out.print()` statements. Dynamic sections are already specified in EOL, and require no translation.

Consider the EGL depicted in Listing 1.1. The preprocessor produces the EOL shown in Listing 1.3 – the `[% %]` and `[%= %]` tag pairs have been removed, and the text to be output is translated into `out.print()` statements.

Listing 1.3. Resulting EOL generated by the preprocessor

```
for (i in Sequence{1..5}) {
  out.print('i is ');
  out.print(i);
  out.print('\r\n');
}
```

When comparing Listings 1.1 and 1.3, it can be seen that the template-based syntax is more concise, while the preprocessed syntax is arguably more readable. For templates where there is more dynamic than static text, such as the one depicted in Listing 1.1, a template-based syntax is often less readable. However, this loss of readability is somewhat mitigated by EGL's developer tools, which are discussed in Section 3.8. By contrast, for templates that exhibit more static than dynamic text, a template-based syntax is often more readable than its preprocessed equivalent.

3.4 Deriving EGL from EOL

In designing functionality specific to M2T transformation, one option was to enrich the existing EOL syntax with keywords such as `print`, `contentType` and `merge`. However, EOL underpins all Epsilon languages, and the additional keywords were needed only for M2T. Furthermore, the refactorings needed to support the new keywords affect many components – the lexer, parser, execution context and execution engine – complicating maintenance and use by other developers. Instead, we define a minimal syntax for EGL, allowing easy implementation of an EGL execution engine as a simple preprocessor for EOL.

The EGL execution engine augments the default context used by EOL during execution with two read-only, global variables: `out` (Section 3.2) and `TemplateFactory` (Section 3.5). The `out` object defines methods for performing operations specific to M2T translation, and the `TemplateFactory` object provides methods for loading other templates. The implementation for the latter was extended, late in the EGL development, to provide support for accessing templates from a file-system – a trivial extension that caused no migration problems for existing EGL templates, due to the way in which EGL extends EOL.

3.5 Co-ordination

In the large, M2T transformations need to be able to not only generate text, but also files, which are then used downstream as development artefacts. An M2T tool must provide the language constructs for producing files and manipulating the local file system. Often, this requires that the destination, as well as the contents, be dynamically defined at a transformation's execution time [6].

The EGL co-ordination engine supplies mechanisms for generating text directly to files. The design encourages decoupling of generated text from output destinations. The `Template` data-type is provided to allow nested execution of

M2T transformations, and operations on instances of this data-type facilitate the generation of text directly to file. A factory object, `TemplateFactory`, is provided to simplify the creation of `Template` objects. In Listing 1.4, these objects are used in an EGL template that loads the the EGL template in Listing 1.2 from the file, ClassNames.egl, and writes out to disk the text generated by executing ClassNames.egl.

Listing 1.4. Storing the name of each Class to disk

```
[%
 var t : Template := TemplateFactory.load('ClassNames.egl');
 t.process();
 t.store('Output.txt');
%]
```

This approach to co-ordination allows EGL to be used to generate one or more files from a single input model. Moreover, EGL's co-ordination engine facilitates the specification of platform-specific details (the destination of any files being generated) separately from the platform-independent details (the contents of any files being generated). The approach is compared to that in other M2T transformation tools in Section 5.

3.6 Merge Engine

EGL provides language constructs that allow M2T transformations to designate regions of generated text as *protected*. The contents of protected regions are preserved every time a M2T transformation generates text to the same destination.

Protected regions are specified by the `preserve(String, String, String, Boolean, String)` method on the `out` keyword – based on the `PROTECT` construct of the XPand language [18]. The first two parameters define the comment delimiters of the target language. The other parameters provide the name, enable-state and content of the protected region, as illustrated in Listing 1.5.

Listing 1.5. Protected region declaration using the preserve method

```
[%=out.preserve('/*', '*/', 'anId', true,
             'System.out.println(foo);')
%]
```

A protected region declaration may have many lines, and use many EGL variables in the contents definition. To enhance readability, EGL provides two additional methods on the `out` keyword: `startPreserve(String, String, String, Boolean)` and `stopPreserve`. Listing 1.6 uses these to generate a protected region equivalent to that in Listing 1.5.

Listing 1.6. Protected region declaration

```
[%=out.startPreserve('/*', '*/', 'anId', true)%]
System.out.println(foo);
[%=out.stopPreserve()%]
```

Because an EGL template may contain many protected regions, EGL also provides a separate method to set the target language generated by the current template, `setContentType(String)`. By default, EGL recognises Java, HTML, Visual Basic, Perl and EGL as valid content types. An alternative configuration file can be used to specify further content types. Following a call to `setContentType`, the first two arguments to the `preserve` and `startPreserve` methods can be omitted, as shown in Listing 1.7.

Listing 1.7. Setting the content type

```
[% out.setContentType('Java'); %]
[%=out.preserve('anId', true, 'System.out.println(foo);')%]
```

Because some languages define more than one style of comment delimiter, EGL allows mixed use of the styles for `preserve` and `startPreserve` methods.

Once a content type has been specified, a protected region may be declared entirely from a static section, using the syntax in Listing 1.8.

Listing 1.8. Declaring a protected region from within a static section

```
[% out.setContentType('Java'); %]
// protected region anId [on|off] begin
System.out.println(foo);
// protected region anId end
```

When a template that defines one or more protected regions is processed by the EGL execution engine, the target output destinations are interrogated and existing contents of any protected regions are preserved. If either the output generated by from the template or the existing contents of the target output destination contains protected regions, a merging process is invoked. Table 1 shows the default behaviour of EGL's merge engine.

3.7 Readability and Traceability

Conscientious developers apply various *conventions* to produce readable code. EGL encourages template developers to prioritise the readability of templates over the text that they generate. Like XPand [18], EGL provides a number of text post-processors – or *beautifiers* – that can be executed on output of

Table 1. EGL's default merging behaviour

Protected Region Status		**Contents taken from**
Generated	**Existing**	
On	On	Existing
On	Off	Generated
On	Absent	Generated
Off	On	Existing
Off	Off	Generated
Off	Absent	Generated
Absent	On	Neither (causes a warning)
Absent	Off	Neither (causes a warning)

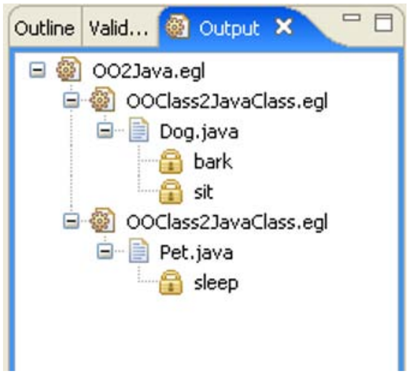


Fig. 3. Sample output from the traceability API

transformations to improve readability. Currently, beautifiers are invoked via Epsilon's extensions to Apache Ant [1], an XML-based build tool for Java.

EGL also provides a traceability API, as a debugging aid, and to support auditing of the M2T transformation process. This API facilitates exploration of the templates executed, files affected and protected regions processed during a transformation. Figure 3 shows sample output from the traceability API after execution of an EGL M2T transformation to generate Java code from an instance of an OO metamodel.

The beautification interface is minimal, in order to allow re-use of existing code formatting algorithms. Consequently, there is presently no traceability support for beautified text. However, due to the coarse-grained approach employed by EGL's traceability API, this has little impact: clicking on a beautified protected region in the traceability view might not highlight the correct line in the editor.

3.8 Tool Support

The Epsilon platform provides development tools for the Eclipse development environment [4]. Re-use of Eclipse APIs allows Epsilon's development tooling to incorporate a large number of features with minimal effort. Furthermore, the flexibility of the plug-in architecture of Eclipse enhances modular authoring of development tools for Epsilon.

In addition to the traceability view shown in Figure 3, EGL includes an Eclipse editor and an outline view. In order to aid template readability, these tools provide syntax highlighting and a structural overview for EGL templates, respectively. Through its integration in the Epsilon perspective, EGL provides an Eclipse workbench configuration that is tailored for use with Epsilon's development tools.

EGL, like other Epsilon languages, provides an Apache Ant [1] task definition, to facilitate invocation of model-management activies from within a build script.

4 Case Study

In this section, we demonstrate EGL's capabilities and design with a case study. The example scenario requires analysis of the architecture and performance characteristics of a number of *system*s. Distinct metamodels are used to describe the way in which systems may be constructed and their response times.

The architecture metamodel, Figure 4, defines a *system* to comprise a number of *service*s. A *Workflow* describes the combination of services needed to perform a complex task. The components of an example system are given in Table 2. In the system, the SearchForProperty workflow comprises the LookupDatabase, FilterUnsafeHouse and DisplayResults services; the BuyHouse workflow comprises SearchForProperty, and services, SelectProperty and PrintHouseDetails.

The metamodel defining system performance characteristics is shown in Figure 5. Each performance model comprises a number of *service implementation*s, which have a name and a response time. The name attribute of the service implementation meta-class must correspond to the name attribute of the service meta-class in the architectural metamodel. The response time of workflows are

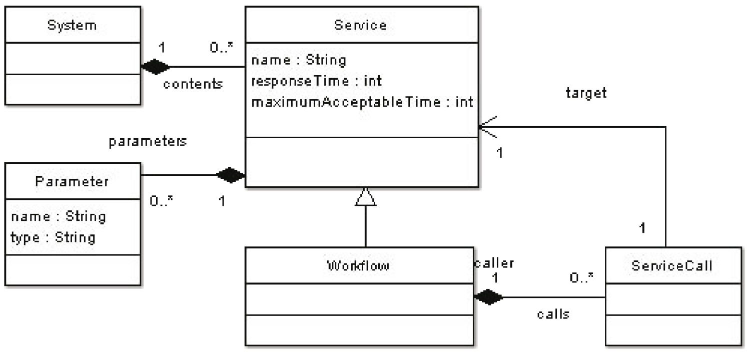


Fig. 4. The Service architecture metamodel

Table 2. An example instance of the Service metamodel

Type	**Name**	**Max Response Time**
Service	SelectProperty	3
Service	FilterUnsafeHouses	4
Service	DisplayResults	5
Service	PrintHouseDetails	6
Service	LookupDatabase	35
Workflow	SearchForProperty	
Workflow	BuyHouse	30

not included in the performance metamodel; these are derived from the response times of the services that make up the workflow.

Tooling was implemented using EGL and other Epsilon languages that allow the performance characteristics of a system to be calculated and visualised. Firstly, the Epsilon Comparison Language (ECL) is used to determine whether an instance of the architectural metamodel and an instance of the performance metamodel are compatible. A performance model, `p`, was deemed to be compatible with an architectural model, `a`, if, for each service implementation in `p`, there existed a service in `a` with `name` equal to the service implementation `service`.

Where instances have been shown compatible, the two are merged using the Epsilon Merging Language (EML). This creates an instance of the architectural metamodel that also contains response times. This allows the response times of each service and workflow to be compared with their maximum acceptable response times. The Epsilon Validation Language (EVL) is used to enforce this constraint and report any non-conformance.

Finally, EGL is used to produce a visualisation of the resulting model. The code, given in Listing 1.9, generates a table with a row for each service and workflow in the system, highlighting the performance characteristics of each. Example output is given in Figure 6. Of particular interest are the use of EOL's declarative functions on *collections* (a feature that EOL re-uses from OCL), which provide a concise means for expressing complex model inspections. For example, the use of `collect` on line 73 allows the total response time of a workflow to be calculated without explicit iteration.

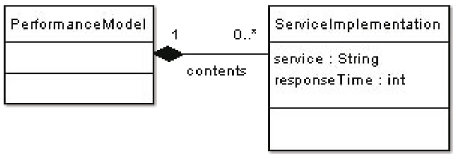
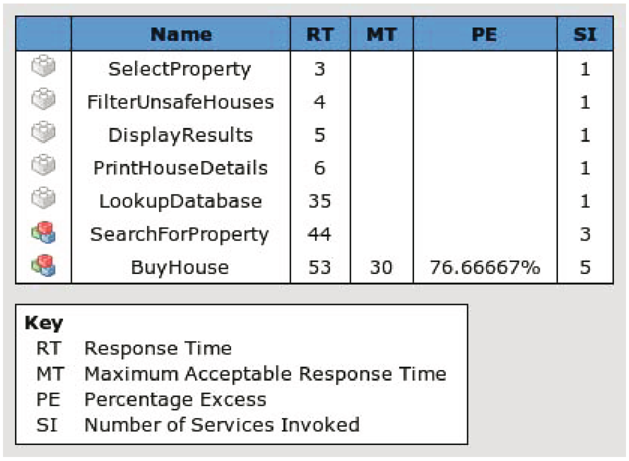


Fig. 5. The ServicePerformance metamodel

Listing 1.9. The EGL code used to generate the visualisation (HTML for the table key is omitted)

```
<html>
<head>
    <title>Service Model Visualisation</title>
    <link title=‘‘default’’ rel=‘‘stylesheet’’ type=‘‘text/css’’
 href=‘‘Viz.css’’/>
</head>
<body>

<table>
    <tr>
        <th>&nbsp;</th>
        <th>Name</th>
        <th>RT</th>
        <th>MT</th>
        <th>PE</th>
        <th>SI</th>
    </tr>
[%
  services = Service.allInstances();
  for (service in services.sortBy(s | s.getResponseTime())) {
%]
    <tr>
        <td>[%=service.getImage()%]</td>
        <td>[%=service.name%]</td>
        <td>[%=service.getResponseTime()%]</td>
        <td>[%=service.maxAcceptableTime.toString()%]</td>
        <td>[%=service.percentageExcess().toString()%]</td>
        <td>[%=service.numberOfCalls()%]</td>
    </tr>
[% } %]
 </table>

</body>
</html>

[%
    operation Any toString() : String {
        if (self.isDefined()) {
            return self;
        } else {
            return ‘&nbsp;’;
        }
    }

    operation Service percentageExcess() : String {
        if (self.maxAcceptableTime.isDefined() and
            self.maxAcceptableTime > 0) {

            var percentage := 100 * self.getResponseTime() /
                                self.maxAcceptableTime;
            return (percentage − 100) + ’%’;
        }
    }

    operation Service numberOfCalls() : Integer { return 1; }

    operation Workflow numberOfCalls() : Integer {
        return self.calls.collect(c|c.target.numberOfCalls())
               .sum().ceiling();
    }

    operation Service getImage() : String {
        return ‘<img src=‘‘service.eps’’ alt=‘‘Service’’ />’;
    }

    operation Workflow getImage() : String {
        return '<img src="workflow.eps" alt="Workflow" />';
    }

    operation Service getResponseTime() : Integer {
        return self.responseTime;
    }

    operation Workflow getResponseTime() : Integer {
        return self.calls
                      .collect(c|c.target.getResponseTime())
                      .sum();
    }

    operation Any getResponseTime() : Integer { return 0; }
%]
```



	Name	RT	MT	PE	SI
⬜	SelectProperty	3			1
⬜	FilterUnsafeHouses	4			1
⬜	DisplayResults	5			1
⬜	PrintHouseDetails	6			1
⬜	LookupDatabase	35			1
🟥	SearchForProperty	44			3
🟥	BuyHouse	53	30	76.66667%	5

Key

RT	Response Time
MT	Maximum Acceptable Response Time
PE	Percentage Excess
SI	Number of Services Invoked

Fig. 6. Example output from the model visualisation phase

5 Related Work

5.1 JET 2.0

JET [5] is perhaps the most popular code-generation framework available for the Eclipse platform. JET's dynamic sections employ custom XML tags in order to describe control flow. Attributes of these tags may include XPath [3] path expressions to support model interrogation. Developers may also include dynamic sections, written in Java, in JET templates.

Out of the box, JET can perform transformations only upon XML- and EMF-based models, and, unlike EGL, does not provide support for models implemented using MOF 1.4 or CZT. Furthermore, while XPath provides a concise means for specifying navigation of tree structures, it lacks some of the expressiveness of the OCL-like constructs for navigating collections (e.g. select, reject, forAll) that Epsilon provides through EOL.

JET provides support for active code generation, via the `c:userRegion` and `c:initialCode` constructs. This is slightly more flexible than EGL, where text used in EGL protected region markers has to conform to a simple grammar. However, this slight loss of flexibility enables EGL to provide constructs to simplify protected region demarcation and thus to reduce duplication in templates.

5.2 MOFScript

MOFScript [15] has influenced the OMG's MOF-based M2T transformation RFP. MOFScript's declarative style of syntax has similarities to the syntax style of M2M transformation languages such as ATL. The declarative approach allows transformation rules to use sophisticated mechanisms for abstraction and code re-use, such as inheritance. However, EGL provides much the same scope for reducing duplication of code using an imperative syntax plus facilities for code modularisation. Although the way in which abstraction and re-use is achieved is slightly less succinct in the imperative approach, the resulting templates are more readable.

A key problem with MOFScript is that it encourages transformations with tight coupling of destination and content: the MOFScript file type allows transformations to write generated text (content) directly to disk. This means that modification is needed to use the same transformation to generate content to a different type of destination (a socket, an HTTP stream, etc.). The EGL style encourages developers to separate destination and content of generated text, by restricting direct access to output destinations from within templates, as discussed in Section 3.5.

Unlike JET, MOFScript provides some OCL-like constructs for traversing and interrogating data structures (forEach and select keywords). MOFScript also provides a prototype implementation for aspect-oriented programming constructs, allowing transformations to be woven together at compile-time.

5.3 XPand

The openArchitectureWare platform, oAW [17], provides open-source, model-driven software development tools. It includes a M2T language, XPand [18], with a declarative template syntax. XPand meets many of the requirements outlined in Section 2.1. For instance, the language supports active code generation via the `PROTECT` construct, and provides beautifiers to enhance readability of both templates and generated text. However, like MOFScript, XPand encourages transformations to couple destination and content, which limits re-usability.

Unlike the other M2T languages considered, XPand templates can be invoked as part of a workflow, using oAW's proprietary workflow definition language. By contrast, Epsilon utilises Apache Ant to define workflows, which encourages reuse of existing tools, such as AntUtility [7], for profiling, and the Nurflugel AntScript Visualizer [2] for visualisation.

6 Conclusions and Further Work

In this paper, we have presented the Epsilon Generation Language, a template-based M2T transformation language for the Epsilon platform. Through its derivation from EOL, EGL provides features specific to the M2T transformation domain as well as having direct access to general model-management support, and to future enhancements to EOL and the Epsilon platform. Furthermore, by deriving EGL from EOL, we have been able to simplify the language design activity of EGL development, and re-use the EOL execution engine.

We are now working on combining EGL with Epsilon's languages for model comparison and transformation (ECL and ETL), to support incremental code generation. This would allow users to reflect in code all changes made to source models, by applying a minimal number of transformations. We are also investigating an alternative approach, using change information derived by model editing tools to perform impact analysis; by adding keywords to EGL this would allow direct checking of staleness of model elements.

An alignment of EGL with a web-server has a number of potentially interesting applications. The case study in Section 3.5 shows the potential of using EGL as a component in a web-based model repository. Extending this idea, the Epsilon languages could provide a scaffold for developing web-based applications: suppose the domain objects of such an application were encoded as EMF-compliant models – Epsilon's transformation language could be used to describe suitable transformations on the domain model (such as adding, editing or removing an instance); the Epsilon Validation Language could check that models remain valid and consistent subsequent to domain model transactions; and EGL could be used to produce HTML for viewing domain objects.

Information on EGL, Epsilon and associated languages is available from the Epsilon GMT project website, `http://www.eclipse.org/gmt/epsilon`.

Acknowledgement. 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. Apache. The Apache Ant Project (2007), `http://ant.apache.org/`
2. Bullard, D.: Ant Script Visualizer (2005), `http://www.nurflugel.com/webstart/AntScriptVisualizer/`
3. World Wide Web Consortium. XML Path Language (XPath) Version 1.0 (1999), `http://www.w3.org/TR/xpath`
4. The Eclipse Foundation. Eclipse - an open development platform (2007), `http://www.eclipse.org`
5. The Eclipse Foundation. JET, part of Eclipse's Model To Text (M2T) component (2007), `http://www.eclipse.org/modeling/m2t/?project=jet#jet`
6. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, New York (2003)

7. David Green. Ant Utility (2007), https://antutility.dev.java.net/
8. ATLAS Group. Atlas Transformation Language Project Website (2007), http://www.eclipse.org/m2m/atl/
9. The Object Management Group. OMG Official Website (2007), http://www.omg.org
10. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc, Boston (2003)
11. Kolovos, D.S.: Extensible Platform for Specification of Integrated Languages for mOdel maNagement Project Website (2007), http://www.eclipse.org/gmt/epsilon
12. Kolovos, D.S., Paige, R.F., Polack, F.: Epsilon Development Tools for Eclipse. In: Eclipse Summit 2006, Esslingen, Germany (October 2006)
13. Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
14. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: A Short Introduction to Epsilon (2007), http://www-users.cs.york.ac.uk/~dkolovos/epsilon/Epsilon.ppt
15. Oldevik, J., Neple, T., Grønmo, R., Aagedal, J.Ø., Berre, A.-J.: Toward standardised model to text transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 239–253. Springer, Heidelberg (2005)
16. OMG. MOF Model to Text Transformation Language RFP (2005), http://www.omg.org/docs/ad/04-04-07.pdf
17. openArchitectureWare. openArchitectureWare Project Website (2007), http://www.eclipse.org/gmt/oaw/
18. openArchitectureWare. XPand Language Reference (2007), http://www.eclipse.org/gmt/oaw/doc/4.1/r20_xPandReference.pdf
19. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program 68(3), 187–207 (2007)