Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Eugenia: Towards Disciplined and Automated Development of GMF-based Graphical Model Editors

Dimitrios S. Kolovos[1], Antonio García-Domínguez[2], Louis M. Rose[1], Richard F. Paige[1]

[1] University of York (UK), e-mail: {dkolovos, louis, paige}@cs.york.ac.uk
[2] University of Cádiz (Spain), e-mail: antonio.garciadominguez@uca.es

Received: date / Revised version: date

Abstract EMF and GMF are powerful frameworks for implementing tool support for modelling languages in Eclipse. However, with power comes complexity; implementing a graphical editor for a modelling language using EMF and GMF requires developers to hand craft and maintain several detailed interconnected models through a loosely guided, labour-intensive and error-prone process. We demonstrate how the application of metamodel annotation and model transformation techniques can help to manage the complexity of GMF and EMF and deliver significant productivity, quality, and maintainability benefits. We present Eugenia, an open-source tool that implements the proposed approach, illustrate its functionality with an example, evaluate it through an empirical study, and report on the community's response to the tool.

1 Introduction

Domain Specific Languages (DSLs) play an increasingly important role in Model Driven Engineering: in a recent survey of MDE practitioners, "almost 40%" of participants had used custom DSLs, 25% had used off-the-shelf DSLs, and only UML (used by 85% of participants) had been used more widely than DSLs [1]. Many tools exist for implementing DSLs, including the Eclipse Modelling Framework (EMF) [2], a robust, widely used, and flexible framework for constructing DSLs on top of the Eclipse software development platform. Over the last few years, Eclipse and EMF have become the *de facto* standards in the MDE community; the majority of MDE tools (e.g. ATL, Xtend/Xpand, Acceleo, QVT, Epsilon) have been either implemented directly on top of them, or are seamlessly integrated with them. Building on EMF, the Graphical Modelling Framework (GMF) is a robust framework that facilitates the development of complex diagram-based editors for EMF-based DSLs.

Both EMF and GMF adopt a generative approach. Starting from an Ecore[1] metamodel which specifies the abstract syntax of a (domain-specific) modelling language, developers derive and maintain a set of more fine-grained, lower-level models. These specify the graphical syntax and other implementation options, and which can be consumed by EMF and GMF code generators to produce the concrete artefacts (i.e. Java code and configuration files) that realise the editor. EMF and GMF are particularly powerful and flexible, providing customisation options for almost every aspect of the generated editor.

As is often the case, the price to be paid for power and flexibility is increased complexity. As discussed in the industrial experience report presented by Wienands and Golm [3], implementing a graphical editor for a modelling language using EMF and GMF is a loosely guided and error-prone process, mainly because it requires developers to hand craft and maintain a number of low-level, complicated and interconnected models. Like Wienands and Golm, we argue that the application of EMF and GMF to implement a DSL introduces significant *accidental complexity* into the development process: we demonstrate the way in which developing a graphical editor with GMF is suboptimal, particular with respect to the maintainability of the resulting editor and the productivity of the development process.

In this paper we report on a way in which we have applied model transformation to reduce the accidental complexity of GMF and EMF. We propose a combination of a single-sourcing approach and model transformation to both raise the level of abstraction at which DSL developers must work and to deliver significant productivity benefits to the process of constructing graphical editors for modelling languages. We demonstrate

Send offprint requests to:

[1] Ecore is the object-oriented metamodelling language of EMF.

our proposed approach through Eugenia, a mature and widely used tool that abstracts over and automatically generates the low-level models required by the EMF and GMF code generators.

This paper is an extension of earlier work presented at the 2010 edition of the MoDELS conference [4]. Compared to [4], this paper provides an extensive discussion on the transformations that implement the proposed approach, demonstrates recent extensions related to pre-transformation validation and code patching, summarises recent experiments investigating the productivity benefits of Eugenia, and reports on wider evaluation of Eugenia via feedback gathered from the community.

The paper is organised as follows. Section 2 outlines the process of developing a graphical editor using EMF/GMF and highlights the error-prone and labour-intensive steps. Following this, Section 3 demonstrates how we have used metamodel annotation, model-to-model and in-place model transformation to automate these steps in the context of Eugenia. Section 4 presents a concrete example that demonstrates the major features of Eugenia. Section 5 reports on a set of experiments used to evaluate the productivity and maintainability benefits delivered by model transformation in this practical problem, and on feedback from Eugenia's user community. Section 6 provides an overview of related work, and Section 7 concludes the paper and provides directions for further work on the subject.

2 Motivation

In this section we outline the process of implementing a graphical editor for a modelling language using EMF and GMF and we identify the labour-intensive, error-prone and maintenance-challenging steps it involves. Figure 1 provides a graphical overview of the process and the artefacts involved.

The first part of the process involves specifying the abstract syntax of the language using Ecore and generating the Java code from it (in two stages), using the EMF built-in code generator. The second part involves specifying the graphical syntax of the editor using a number of graphical syntax-specific GMF models (in three stages), and then using the GMF code generator to generate the concrete graphical editor.

2.1 Specifying the Abstract Syntax and Generating Code using EMF

In the first step of the process, the developer needs to define the abstract syntax (metamodel) of the language using Ecore. Following that, the developer can invoke a built-in EMF model-to-model transformation to transform the Ecore metamodel into an *EMF generator model* (*GenModel*). The produced *GenModel* captures lower-level information that specifies how the metamodel should be implemented in Java (e.g. the Java package under which the code will be generated, copyright information to be embedded in the generated files, whether certain UI elements will be generated or not, etc.). Once derived from the Ecore metamodel, a GenModel can be customised and fine-tuned manually. Finally, the GenModel is consumed by a built-in model-to-text transformation which produces all the necessary Java code and configuration files.

If the Ecore metamodel is subsequently modified, EMF provides a built-in reconciler that can detect changes in the metamodel and propagate them to the corresponding GenModel without overwriting the user-defined customisations. However, the reconciler is only effective for simple changes in the Ecore metamodel; for more complex changes (e.g. moving EClasses across different EPackages) the GenModel needs to be regenerated and customised from scratch. This introduces a significant maintenance overhead as it is not always clear to developers which changes in the metamodel can or cannot be reconciled automatically. Therefore, it is common practice to maintain documentation about all manual changes in a separate location (e.g. a text file) so that they can be reapplied (manually) if or when necessary.

2.2 Specifying Graphical Syntax and Generating Code with GMF

Once the Ecore metamodel has been defined and the EMF code has been generated, the developer needs to construct three additional GMF-specific models to implement a graphical editor for the language.

- The *graph* model (*GMFGraph*) specifies the graphical elements (shapes, connections, labels, decorations etc.) used in the editor;
- The *tooling* model (*GMFTool*) specifies the element creation tools that will be available in the palette of the editor;
- The *mapping* model (*GMFMap*) maps the graphical elements in the graph models and the creation tools in the tooling model with the abstract syntax elements of the Ecore metamodel (classes, attributes, references etc.).

Figures 2-4 provide simplified views of the metamodels of the three models discussed above as the complete metamodels are too large to illustrate in this paper (for instance, the GMF Graph metamodel consists of 81 meta-classes containing 140 structural features in total).

The mapping model is then automatically transformed into an even more fine-grained generator model (*GMFGen*) which contains all the low-level information that the GMF code generator needs in order to produce the concrete artefacts (Java code and configuration files) that realise the graphical editor.

In terms of automation, GMF provides a built-in wizard for automatically generating initial versions of

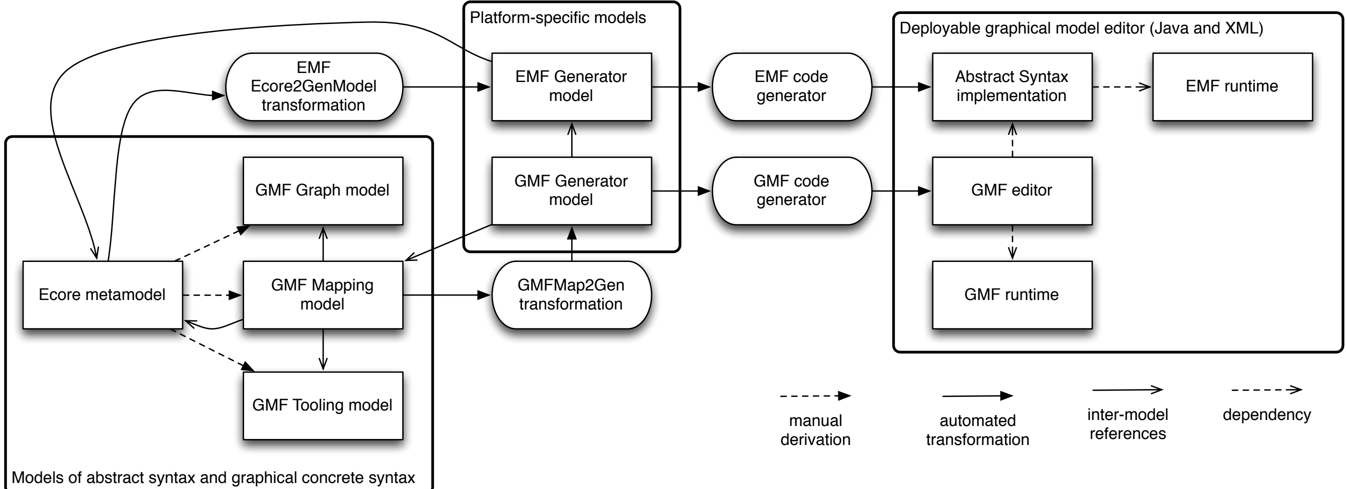

Fig. 1 EMF/GMF Graphical Editor Development Process Overview

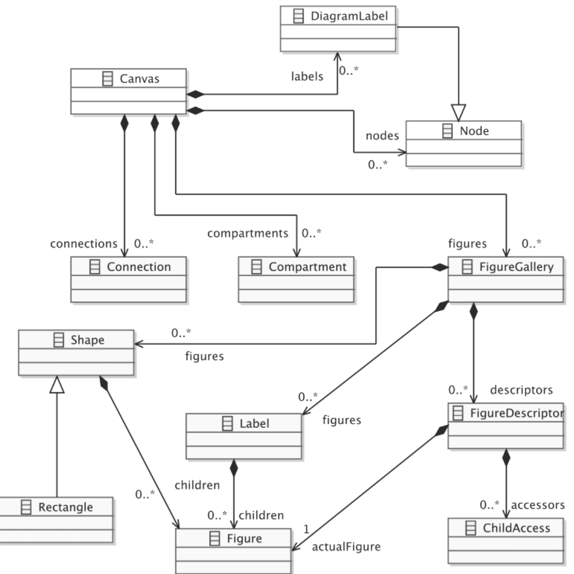

Fig. 2 Excerpt from the GMF Graph Metamodel

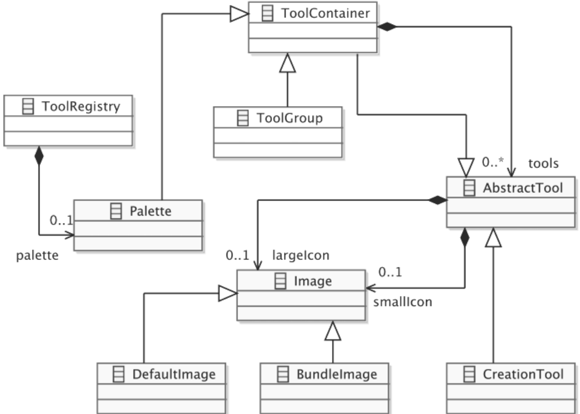

Fig. 3 Excerpt from the GMF Tooling Metamodel

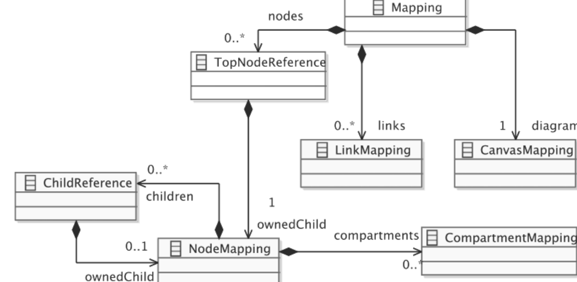

Fig. 4 Excerpt from the GMF Mapping Metamodel

the tooling, graph and mapping models from the Ecore metamodel itself. Unfortunately, in practice this wizard fails to yield useful results for anything beyond very simple metamodels [3] – and this is unsurprising given how little can in general be inferred about the graphical syntax based on the abstract syntax alone.

As a result, these three models need to be handcrafted using a set of very basic tree-based editors provided by GMF. This is liable to be a laborious and error-prone process, particularly given the complexity of the GMF metamodels, and the low-level error messages that GMF produces. Perhaps more challenging than constructing these GMF-specific models is maintaining them as, unlike with EMF, GMF does not provide a reconciler that can update these models automatically (even for very simple changes) when the Ecore metamodel changes. Therefore, once customised in any way, these models need to be maintained manually.

Arguably, implementing and maintaining a graphical editor with EMF and GMF is a laborious and error prone task, particularly so for inexperienced developers. Given that implementing a simple graphical editor is typically one of the first steps attempted by most of the newcomers in MDE [3], the risk of forming a negative impression about the effort and learning curve imposed by the MDE

tool-chain from their interaction with GMF is considerable. Moreover, even for seasoned MDE developers[2], this predominantly manual and repetitive process is clearly tedious, which is in a sense ironic: MDE has its strengths in automating repetitive processes. Arguably, this is a situation where MDE could "eat its own dog food."

3 Automation through Metamodel Annotation and Model Transformation

To shield developers from the complexity of GMF and address the highlighted challenges, in this work we propose a *single-sourcing* approach, in which additional information necessary for implementing a graphical editor is captured by embedding high-level annotations in the Ecore metamodel. We then use automated model-to-model and in-place transformations to generate, the platform-specific models required by the EMF and GMF code generators in a consistent and repeatable manner.

In this section we provide a detailed discussion on the steps of the proposed approach and highlight the productivity, quality and maintainability benefits it delivers. The proposed approach has been implemented in the context of the Eugenia open-source tool which, as discussed in Section 5, has received substantial positive feedback and validation from the Eclipse Modelling community and has several users across academia and industry. The reader can now either proceed with the remainder of the technical discussion of the proposed approach, or jump to Section 4 for a concrete example, and then return to this section.

3.1 Constructing and Maintaining EMF Generator Models

The first challenge highlighted in Section 2 relates to customising and maintaining the EMF Generator Model (GenModel) once it has been produced by the built-in EMF model-to-model transformation. As discussed, the existing change reconciler does not support propagating non-trivial structural changes made to the Ecore metamodel and therefore, for such changes the GenModel needs to be re-generated and customised from scratch.

To overcome this limitation, we propose capturing GenModel-specific information in the form of annotations attached to appropriate elements of the Ecore metamodel, and replacing the built-in EMF model-to-model transformation with a more sophisticated transformation (*Ecore2GenModel*) that can consume these annotations and propagate their values to the GenModel automatically. An overview of the proposed approach for constructing EMF generator models is illustrated in Figure 5.

[2] `http://voelterblog.blogspot.com/2009/06/gmf-is-still-awful.html`

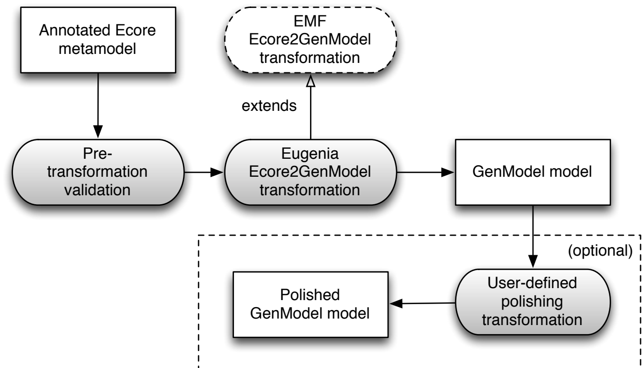

Fig. 5 The Eugenia Ecore2GenModel transformation workflow

The Ecore2GenModel transformation is illustrated at a conceptual level in Figure 6. Each transformation rule is represented as a titled box with two compartments for the source and target elements of the rule respectively. Arrows between elements represent references between them (for example, the EClass2GenClass rule generates one GenClass for each EClass, uses the copyAnnotations() operation, and the target GenClass refers back to the source EClass).

The concrete transformation, which has been implemented using the Epsilon Transformation Language (ETL) [5], replicates the behaviour of the built-in EMF transformation, but also adopts a name-matching reflective approach to copying Ecore annotation values to the respective GenModel model elements. This is demonstrated in the excerpt of the transformation illustrated in Listing 1: in the *EPackage2GenPackage* rule, after creating a *GenPackage* from a source *EPackage*, the rule invokes the *copyAnnotations* operation to copy any annotations attached to the *EPackage* to string-typed attributes of the respective *GenPackage* with matching names in a reflective manner[3].

For example, in the metamodel of Listing 2 – expressed using the Emfatic[4] textual notation for Ecore – we have added a GenModel-specific *emf.gen* annotation to the *simplem2* package[5] (Line 2) that specifies that the base package under which the Java implementation of the metamodel should be generated is *org.xyz.* As demonstrated in Figure 7, beyond creating the *Simplem2* GenPackage from the *simplem2* EPackage, the *GenModel* transformation has also copied the value of the *emf.gen basePackage* annotation of the *simplem2* EPackage, into the *basePackage* attribute to the respective GenPackage[6]. Similarly, every attribute of string/-boolean type in the GenModel can be controlled by an

[3] The transformation also supports copying Boolean annotations but the code for this has been omitted for conciseness.

[4] `http://www.eclipse.org/emfatic`

[5] simplem2 stands for *simple metamodel.*

[6] *@namespace* is a built-in annotation in Emfatic and is not processed by the transformation.

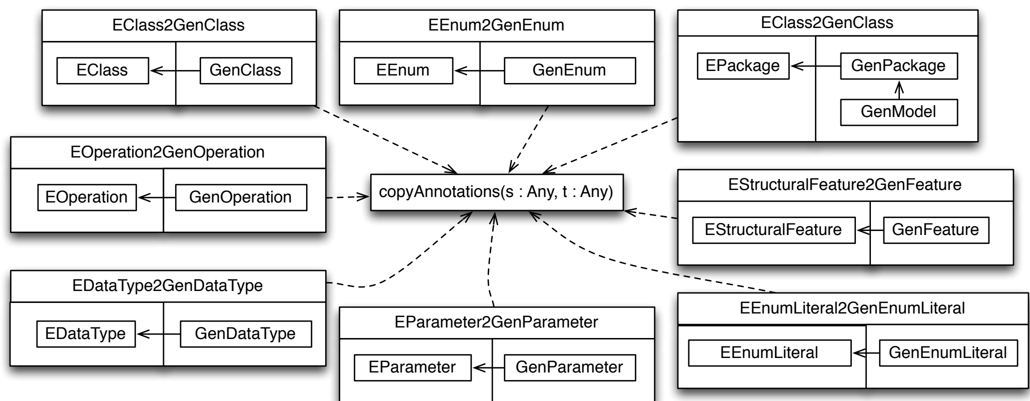

Fig. 6 Overview of the Ecore2GenModel transformation

```
1  rule EPackage2GenPackage
2    transform s : Ecore!EPackage
3    to t : GenModel!GenPackage {
4
5    genModel.genPackages.add(t);
6    t.ecorePackage = s;
7    t.disposableProviderFactory = s.getBooleanAnnotation("disposableProviderFactory", true);
8    t.prefix = s.getAnnotation("prefix", s.name.firstToUpperCase());
9    copyAnnotations(s, t);
10 }
11
12 operation copyAnnotations(source : Any, target : Any) {
13
14   for (stringFeature in target.eClass().eAllStructuralFeatures.
15     select(sf|sf.isOfStringType())) {
16
17     if (source.hasAnnotation(stringFeature.name)) {
18       var annotationValue = source.getAnnotation(stringFeature.name, "");
19       if (stringFeature.many) {
20         var parts = annotationValue.split(",").collect(s|s.trim());
21         target.eGet(stringFeature).addAll(parts);
22       }
23       else {
24         target.eSet(stringFeature, annotationValue);
25       }
26     }
27   }
28
29   ...
30
31 }
```

Listing 1 Excerpt from the Ecore2GenModel transformation

@emf.gen annotation with the same name, attached to its Ecore counterpart. For example, to set the value of the *propertyMultiLine* property of a GenFeature to *false*, an *@emf.gen(propertyMultiLine="false")* annotation can be attached to the respective EStructuralFeature in the Ecore metamodel.

This reflective approach is both concise (with fewer than 80 lines of transformation code the user can currently control 106 string and boolean attributes provided by elements of the GenModel) and future-proof as the transformation will support additional string/-

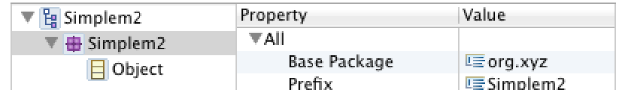

Fig. 7 Output of the Ecore2GenModel transformation applied to the Ecore metamodel of Listing 2

```
1 @namespace(uri="simplem2", prefix="")
2 @emf.gen(basePackage = "org.xyz")
3 package simplem2;
4
5 class Object {}
```

Listing 2 The simplem2 annotated metamodel in Emfatic

boolean properties that may be added in the future to the GenModel metamodel without modifications.

For more complex customisations which require creating and deleting elements in the GenModel or modifying their properties in a batch mode, Eugenia supports user-defined *polishing transformations*. In this context we use the term *polishing transformation* to describe a user-defined in-place model transformation - with a predefined file-name (*Ecore2GenModel.eol* in this case) and location relative to the Ecore metamodel - which is executed by Eugenia after the built-in Ecore2GenModel transformation and through which the developer can fine-tune the produced GenModel in a programmatic – and thus repeatable – manner.

For example, if the developer needs a multi-line attribute value editor for all single-valued string attributes in the metamodel, instead of annotating all of them as *@emf.gen(multiLineProperty = "true")*, they can define the following in-place transformation (in EOL) under a pre-defined filename next to their Ecore metamodel.

```
1 for (gf in GenModel!GenFeature.all) {
2   if (gf.ecoreFeature.isOfStringType() and
3     not gf.ecoreFeature.isMany()) {
4
5     gf.propertyMultiLine = true;
6   }
7 }
```

Listing 3 Sample GenModel polishing transformation

Using this approach, the GenModel can now be treated as a fully derived artefact, and as such, there is no need to edit/maintain it manually any more.

3.2 Generating GMF-specific Models

To automate the construction of the GMF-specific models, we follow a similar approach to the one outlined above: we annotate Ecore models with high-level GMF-specific information and then use a model-to-model transformation (*Ecore2GMF*) to generate the interwoven tooling, graph and mapping GMF models - all in one step. Once the mapping model has been transformed into a GMF generator model (*GMFGen*) using the built-in GMF transformation, Eugenia applies an in-place update transformation to it (*FixGMFGen*), as some of the graphical syntax configuration options (e.g. compartment layout) can only be specified in this model. Consistent with the practice followed in the Ecore2GenModel transformation, the developer can contribute additional polishing transformations for the *Ecore2GMF* and *FixGMFGen* transformations, to fine-tune the generated models. It should be stressed that polishing transformations have read-write access to all GMF-specific (tooling, graph, mapping and generator) models and as such, they can be used to make full use of the expressive power of GMF (i.e. any editor that can be implemented using pure GMF can also be implemented using Eugenia). Figure 8 illustrates this workflow. In addition to polishing transformations, the developer can optionally specify further, low-level customisation via *parametric patches* (Section 3.5), which are applied to systematically tailor the Java source code generated by GMF.

The GMF-specific annotations supported by Eugenia allow developers to specify a large proportion of the graphical syntax of the language including node shapes, feature-based and static labels, class- and reference-based associations (links), affixed and phantom nodes, compartments (with a free or a list-based layout), colours and borders. When we started developing Eugenia, we made a conscious decision to only support the most common elements of graphical syntaxes and leave any remaining aspects for users to customise through polishing transformations. Of course, our understanding of what *most common* means has evolved over time and based on the feedback of the user community since the first release of the tool, we have extended the number of supported annotations in an organic manner. Currently, built-in annotations can create instances of 25 of the 61 non-abstract EClasses in the GMF Graph metamodel, 13 of the 28 non-abstract EClasses in he GMF Mapping metamodel, and 6 of the 19 non-abstract EClasses in the GMF Tooling metamodel (as discussed above, polishing transformations can be used to create/configure instances of the remaining EClasses).

Section 4 provides a detailed example that demonstrates a substantial subset of the supported annotations. A complete list of all the annotations supported by Eugenia is available in Appendix A. In brief, Eugenia provides six categories of annotation:

1. ***@gmf.diagram*** annotations are used to specify diagram-wide settings, such as the type of the root model element, the file extension for the graphical editor, and whether to generate Eclipse plug-ins or a standalone Java application.
2. ***@gmf.node*** annotations are used to indicate which types in the abstract syntax will be represented as nodes (vertices) in the graphical syntax, and to specify the shape, colour, size, label, etc., of each node in the graphical syntax.
3. ***@gmf.link*** annotations are used to indicate which types in the abstract syntax will be represented as edges in the graphical syntax, and to specify the

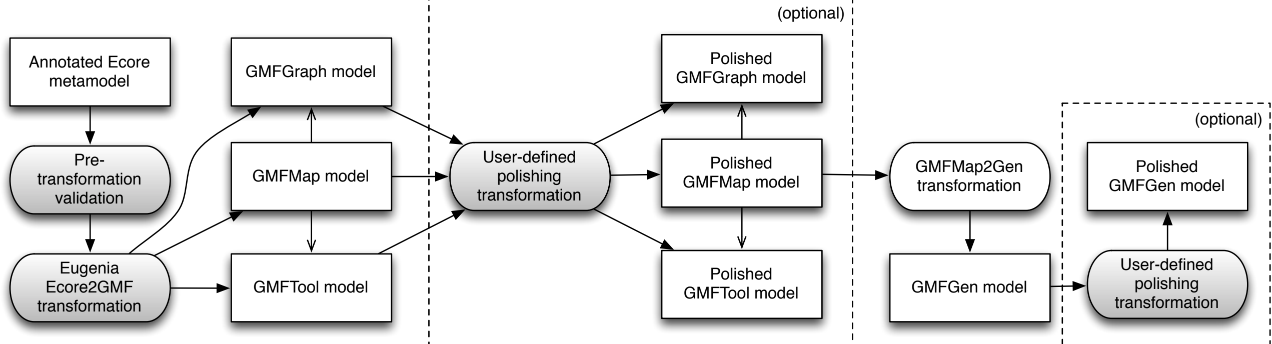

Fig. 8 The Eugenia Ecore2GMF and FixGMFGen transformation workflow

thickness, colour, style, arrowheads, labels, etc., of each edge in the graphical syntax.

4. ***@gmf.compartment*** annotations are used to indicate which nodes may be nested inside other nodes in the graphical syntax (e.g., attributes are nested inside classes in UML class diagram syntax).
5. ***@gmf.affixed*** annotations are used to indicate which nodes may be attached to the border of other nodes in the graphical syntax (e.g., boundary events are attached to the borders of activities in BPMN 2.0 syntax).
6. ***@gmf.label*** annotations are used to specify additional labels for a node in the graphical syntax.

3.3 The Ecore2GMF transformation

Having enumerated the annotations supported by Eugenia, in this section we present the organisation of the Ecore2GMF transformation into functional units. As the concrete transformation comprises more than 1200 lines of code, our intention here is to provide a high-level overview of each functional unit. However, even at this level of abstraction it is impossible not to refer to specific constructs in the involved metamodels. As such, this section is mostly targeted towards readers who already possess some understanding of the GMF Graph, Tooling, and Mapping metamodels.

It should be mentioned that as the GMF models do not provide any notion of inheritance, during the transformation process, the inheritance hierarchy of the input metamodel is logically flattened: EClasses inherit all the annotations (and can override annotation details) from their super-classes, and features (attributes, references, and operations) are logically copied to all sub-classes.

3.3.1 Transforming @gmf.diagram-annotated EClasses The transformation assumes that one EClass of the metamodel is annotated as @gmf.diagram (as such, the @gmf.diagram annotation is not inherited). As displayed in Figure 9, from this EClass, the transformation creates the scaffolding of the Graph, Mapping and Tooling models under which all other model elements produced in the transformation will be placed. More specifically, it creates a *Canvas* and a *FigureGallery* in the Graph model and a *ToolRegistry*, a *Palette* and two built-in *ToolGroup*s (one for nodes and one for connections) in the palette. In the Mapping model, the transformation creates a *CanvasMapping* that links back to the *EClass* and contains one *TopNodeReference* and *NodeMapping* for each sub-type of each of the containment references of the *EClass*.

3.3.2 Transforming @gmf.node-annotated EClasses As displayed in Figure 10, for each such EClass, the transformation creates interwoven elements in all three output models. More specifically, in the Graph model it creates a *Node* of an appropriate *Shape* and *MarginBorder* and - optionally a - *Label*. The shape and its background colour are controlled by the *figure* and *color* annotation details, while the properties of the border are controlled by the *border.** annotation details. The properties of the generated *Label* and *DiagramLabel* are controlled by the *label.** annotation details. For external labels (*label.position="external"*) an additional *FigureDescriptor* is produced. The transformation also creates a *CreationTool* and respective *DefaultImage*s in the tooling model (configured via the *tool.** annotation details) and brings everything together by generating a *NodeMapping* and a *FeatureLabelMapping* in the Mapping model.

3.3.3 Transforming @gmf.link-annotated EClasses and EReferences As illustrated in Figures 11 and 12, for @gmf.link-annotated EClasses and EReferences the transformation produces, in the Graph model, one *FigureDescriptor* for the link and (optionally) one for its label. It also produces a *Connection*, a *PolylineConnection* and the link-end *PolylineDecorations*, the shapes of which are controlled by the **.decoration* annotation details. In the Tooling model, it creates a *CreationTool* and its respective icons.

In the Mapping model, the transformation produces a *LinkMapping* that binds the EClass/EReference to the *Connection* in the Graph model and to the *CreationTool* in the Tooling model. Depending on whether the annotation is attached to an EClass or to an EReference,

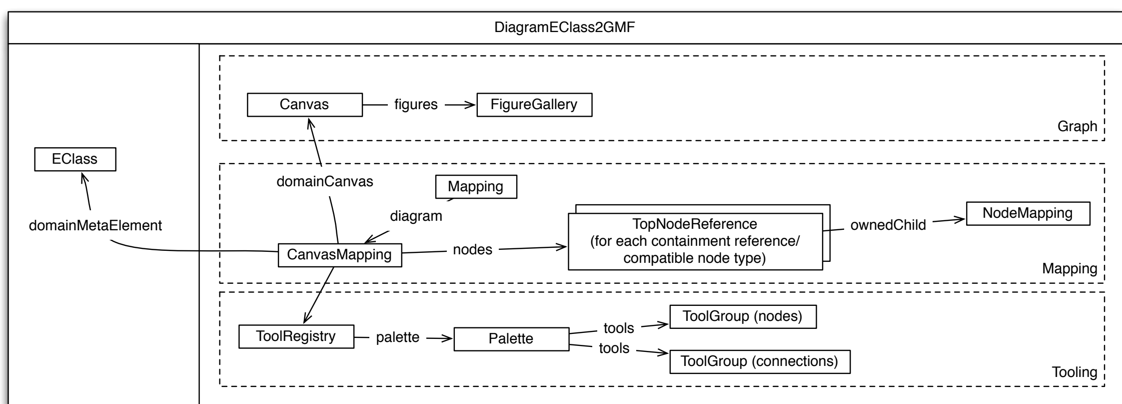

Fig. 9 Overview of the transformation of EClasses annotated as *gmf.diagram*

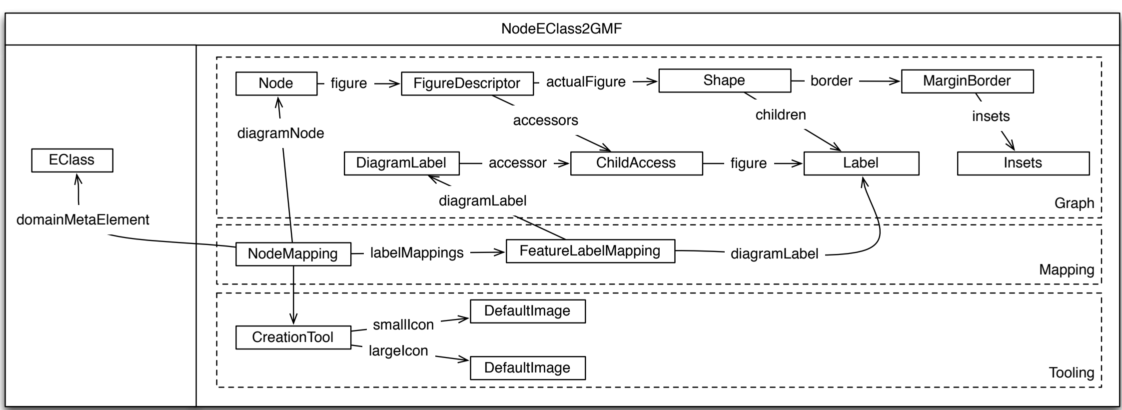

Fig. 10 Overview of the transformation of EClasses annotated as *gmf.node*

the transformation also produces a *FeatureLabelMapping* which links the generated label with the label-attributes of the EClass or a read-only *DesignLabelMapping*.

3.3.4 Transforming @gmf.compartment-annotated EReferences As illustrated in Figure 13, for each containment *EReference* annotated as *@gmf.compartment* in the flattened metamodel, the transformation produces a *Compartment* in the Graph model, and as many *ChildReferences* as the subclasses of the type of the EReference in the Mapping model.

3.4 Pre-Transformation Validation

In its earlier versions, the *Ecore2GMF* transformation would assume that the GMF-specific annotations attached to the input Ecore metamodel would conform to the constraints specified above and would fail with a runtime exception in case they did not. To remedy this limitation, since publishing an earlier version of this work [4], we have implemented a set of constraints (using EVL - the validation language of Epsilon), which guard the transformation against invalid input and provide understandable error messages to the user. For example, the metamodel of Listing 4 does not satisfy the EVL constraints of Listing 5, and as such the following error messages are produced and the transformation process is aborted:

– No polygon x/y coordinates provided for polygon figure Base (error).
– The value of @gmf.node(figure) of Hazard must be one of: rectangle, ellipse, rounded, svg, polygon or a fully-qualified Java class name (warning).

3.5 Post-transformation parametric patching

Not all aspects of the appearance and behaviour of a GMF editor can be controlled via the GMF models described in the previous sections, because the GMF code generator hardcodes many details of the generated Java code. Customisations that cannot be specified in the GMF models must instead be made by either altering or extending the GMF code generator, or by customising the Java source code generated by GMF. The former approach is limited by the developer's knowledge of the code generation language used by GMF (currently Xpand), the extensibility mechanisms provided by that code generation language, and the modularity of

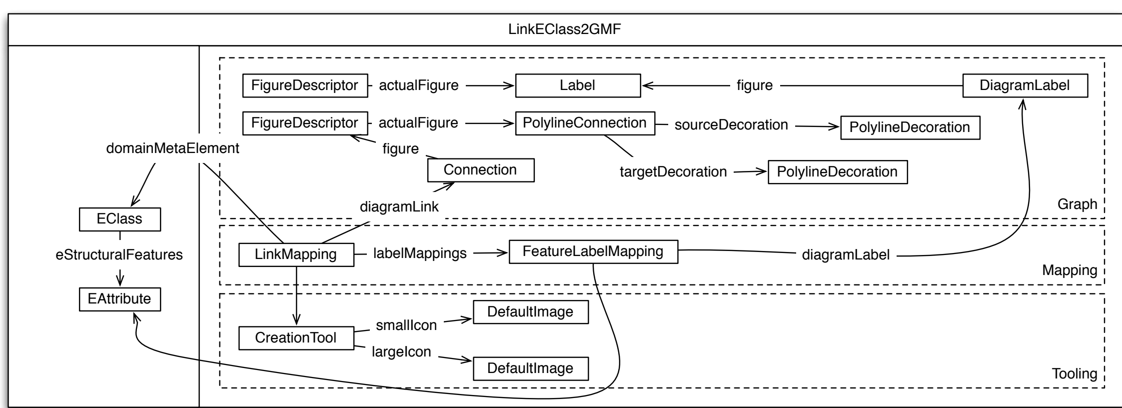

Fig. 11 Overview of the transformation of EClasses annotated as *gmf.link*

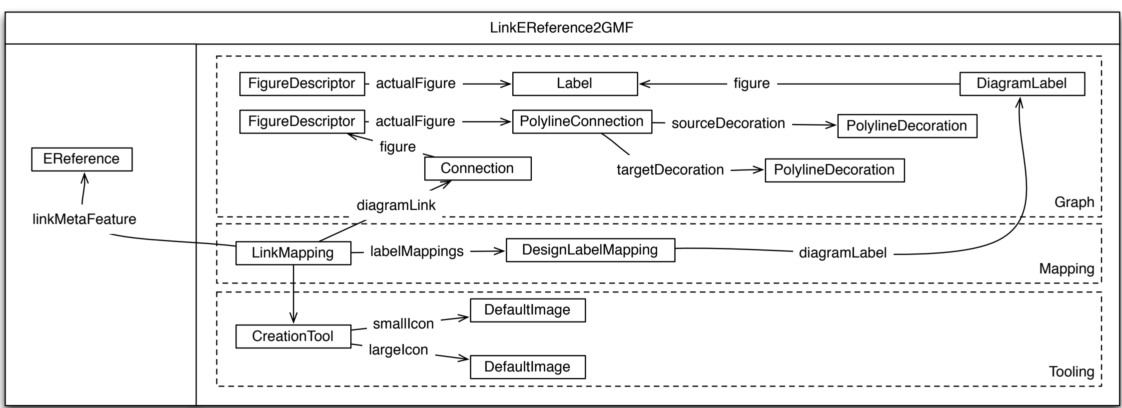

Fig. 12 Overview of the transformation of EReferences annotated as *gmf.link*

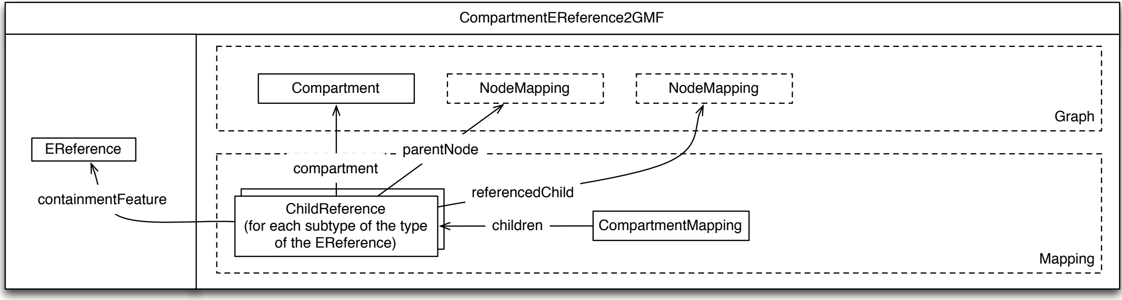

Fig. 13 Overview of the transformation of EReferences annotated as *gmf.compartment*

the GMF code generator. At present, using the aspect-oriented programming constructs of Xpand is likely the most maintainable approach to extending the GMF code generator[7] but this limits extension to the existing pointcuts of the GMF code generator. In other words, only customisations that have been anticipated by the developers of the GMF code generator are possible. The latter approach – altering the generated Java source code – requires no knowledge of Xpand or of the GMF code generator, but is not easily repeatable if performed manually. This section discusses the *parametric patching* capabilities of Eugenia, which we have added since publishing an earlier version of this work [4], and which allow a developer to automate the latter approach to tailoring the generated source code of a GMF editor.

With Eugenia, customisation of the Java source code generated by GMF involves the automated application of *patches*, deltas generated by source code management systems such as Subversion or Git. After invoking the GMF code generator on the GMFGen model, Eugenia will automatically apply any *.patch* files (conventionally located in a *patches* directory) to the generated Java

[7] `http://community.bonitasoft.com/blog/customize-your-gmf-editor-customizing-templates`

```
1 context EClass {
2
3  guard: self.isNode()
4
5  /* Checks that polygon nodes also provide polygon.x and polygon.y coordinates*/
6  constraint IsValidPolygonNode {
7
8   check : (self.getAnnotationValue("gmf.node", "figure") = "polygon") implies
9     (
10      self.getAnnotationValue("gmf.node", "polygon.x").isDefined() and
11      self.getAnnotationValue("gmf.node", "polygon.y").isDefined()
12     )
13
14   message : "No polygon x/y coordinates provided for polygon figure " + self.name
15
16  }
17 }
18
19 context EStringToStringMapEntry {
20
21  critique IsValidNodeFigure {
22
23   guard : self.is("gmf.node", "figure")
24
25   check {
26    var values = Sequence{"rectangle", "ellipse", "rounded", "svg", "polygon"};
27    return self.value.isWithinValuesOrLooksLikeJavaClassName(values);
28   }
29
30   message : "The value of " + self.toEmfatic() + " must be one of: " +
31        values.concat(", ") + " or a fully-qualified Java class name."
32  }
33 }
```

Listing 5 Excerpt from the set of pre-transformation constraints

source code. Because the *.patch* files used with Eugenia are generated by existing source code management systems, Eugenia reuses the Compare library of the Eclipse Platform to programmatically apply patches using a *PatchParser* to obtain IFilePatchResults (Listing 6). By automatically applying source code patches, Eugenia makes customisation of the generated Java source code a repeatable process, and hence simplifies the maintenance of altering generated code.

For similar modifications that need to be applied in several places in the generated source code, Eugenia supports *parametric patches*, deltas created by a patch generator defined by the developer. Instead of – or in addition to – any *.patch* files, the developer provides a patch generator (a model-to-text transformation, conventionally named *GeneratePatches.egx*) that specifies how one or more *.patch* files are to be generated by Eugenia at runtime. After invoking the GMF code generator, Eugenia executes the patch generator, and then applies any *.patch* files (generated or otherwise). A patch generator has access to all of the (polished) GMF and EMF models described in the previous sections, and hence can programatically generate multiple patches that apply the same change to different parts of the generated Java source code. Currently, Eugenia only supports patch generators that are specified in the Epsilon Generation Language (EGL), the model-to-text transformation language of the Epsilon platform, but the conceptual approach is not limited to EGL.

An overview of the parametric patching workflow is shown in Figure 14. A similar workflow could be applied to the generation of code from the EMFGen model, though we have not yet found a practical need for extending the EMF code generator in this manner.

An example of using parametric patching to customise a GMF editor is described in Section 4. In the context of the example, we also describe the way in which parametric patches can be specified in a manner that makes them reusable over different GMF editors.

3.6 Implementation Notes

The Eugenia transformations are implemented using the Epsilon platform [6]. More specifically, the built-in

```
package org.eclipse.epsilon.eugenia.patches;

public class Patcher {

  ...

  private void applyPartialPatch(IFile targetFile, IFilePatchResult result) {
    if (targetFile.exists()) {
      // Alter contents of existing file according to patch
      targetFile.setContents(result.getPatchedContents(), IFile.KEEP_HISTORY, null);
    } else {
      // Create new file from patch
      targetFile.create(result.getPatchedContents(), false, null);
    }

    // If the file has no content after the patch has been applied
    // assume that the file should be deleted.
    if (isEmpty(result.getPatchedContents())) {
      targetFile.delete(false, null);
    }
  }
}
```

Listing 6 Excerpt of the patch application code

```
package sample;

@gmf.diagram
class Map {
  val Hazard[*] hazards;
  val Base[*] bases;
}

@gmf.node(label="name",
    figure="triangle")
class Hazard {
  attr String name;
}

@gmf.node(label="name", figure="polygon")
class Base {
  attr String name;
}
```

Listing 4 Sample invalid Ecore metamodel in Emfatic

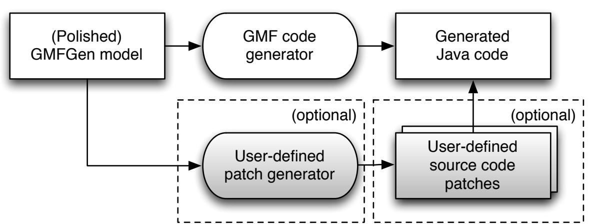

Fig. 14 The GMF code generation and parametric patching workflow

Ecore2GenModel transformation has been implemented using the rule-based ETL [5] model-to-model transformation language, while the *Ecore2GMF* and *FixGMFGen* transformations have been implemented using the imperative EOL language [7]. The *Ecore2GMF* transformation is implemented with an imperative – as opposed to a rule-based – language due to its high complexity and need for low-level control of the execution flow. Validation constraints have been specified using EVL and share code with the ETL and EOL transformations discussed above. In terms of size, the *Ecore2GenModel* transformation is 264 lines long, the *Ecore2GMF* transformation contains 1217 lines of code (including operation libraries), the FixGMFGen transformation is 91 long and the EVL constraints that guard the transformation span 353 lines of code.

These transformations could also be implemented using other M2M languages (e.g. ATL, QVT and Kermeta) as long as they provide the following capabilities:

- Managing more than one source and target model in the same transformation.
- In-place as well as model-to-model transformation.
- Establishing and navigating cross-model references.
- Reflective access to model elements (i.e. the ability to find a feature of a given element by name and get/set its value at runtime), which is particularly desirable in the *Ecore2GenModel* transformation that otherwise will contain many explicit annotation copying statements (76 for EPackages alone).

The transformations can be launched both manually and automatically from Eclipse. Eugenia extends

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="run-eugenia">
3   <target name="main">
4     <epsilon.eugenia src="metamodel.emf"/>
5   </target>
6 </project>
```

Listing 7 Sample project description for automatically invoking Eugenia on the Emfatic source file *metamodel.emf* through the Ant build system in Eclipse.

the Ant build system integrated in Eclipse with an additional task that can run the entire Eugenia workflow or only a part of it. This Ant task can be then integrated into the advanced workflows required by some developers, and can be invoked automatically by Eclipse when the annotated metamodel is changed. Listing 7 shows the simplest Ant build file needed to invoke Eugenia. The Ant task can be further configured to provide additional models for the user-defined polishing transformations or to start or stop the process at different steps than usual.

3.7 Scalability

Due to the annotation inheritance mechanism discussed above, for large metamodels, Eugenia annotations arguably scale up better than manually maintaining the respective GMF models. A potential point of concern is that for highly-customised editors, polishing transformations can become disproportionately large. In such a case, developers can leverage the modularity mechanisms and the mature tool support provided by EOL (e.g. syntax-aware editor, debugger) to manage these transformations efficiently.

4 Example

In this section we present an example that demonstrates using Eugenia for implementing the graphical editor of a Simple Component-connector Language (SCL) using EMF and GMF[8]. Firstly, we specify the abstract syntax of SCL using Ecore. Briefly, an SCL model contains named components, which contain any number of ports and subcomponents. Pairs of components can be linked through their ports. The Ecore metamodel of SCL, expressed in the Emfatic textual notation for Ecore is illustrated in Listing 8.

```
1 @namespace(uri="scl", prefix="scl")
2 package scl;
3 class Component {
4   attr String name;
5   attr String description;
6   val Component[*] subcomponents;
7   val Port[*] ports;
8 }
9
10 class Connector {
11   attr String name;
12   ref Port#outgoing from;
13   ref Port#incoming to;
14 }
15
16 class Port {
17   attr String name;
18   val Connector#from outgoing;
19   ref Connector#to incoming;
20 }
```

Listing 8 The SCL Ecore metamodel in Emfatic

[8] Additional examples are available on the Epsilon website: `http://eclipse.org/epsilon/doc/articles/#eugenia`

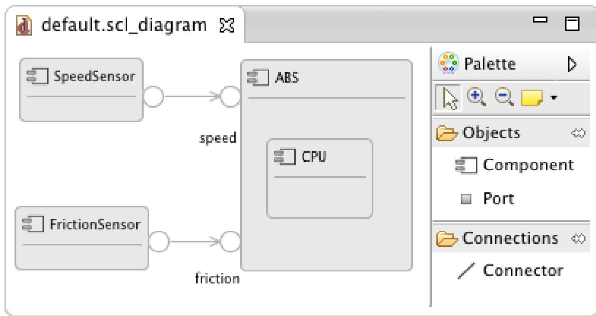

Fig. 15 The first version of the GMF SCL editor

For Eugenia to realise the graphical editor for SCL using EMF and GMF, we need to annotate the Ecore metamodel as shown in Listing 9. In particular, the annotations specify the following:

– Line 2: Source code should be generated in the *org.eclipse.epsilon.eugenia.examples* Java package.
– Line 5: Each diagram contains a top-level Component model element.
– Line 6: Each component is represented in the diagram as a light grey node labelled with the *name* of the component.
– Line 11: Each Component has a compartment in which sub-components are placed.
– Line 17: Each Connector is represented as a link (association) between its *from* and *to* ports. The end attached to the *to* port is decorated with an arrow.
– Lines 24, 25: Each Port is represented as a 15x15 icon-less circle, attached to the border of the component to which it belongs (Line 13).

From this annotated metamodel, Eugenia can automatically generate the GMF editor that appears in Figure 15. While the generated editor is fully-functional, we wish to further customise it to match our requirements (see Figure 16) .

To achieve this, we specify the polishing transformation shown in Listing 10 and place it in a predefined

```
1 @namespace(uri="scl", prefix="scl")
2 @emf.gen(basePackage="org.eclipse.epsilon.eugenia.examples")
3 package scl;
4
5 @gmf.diagram
6 @gmf.node(label="name", color="232,232,232")
7 class Component {
8   attr String name;
9   @emf.gen(propertyMultiline="true")
10  attr String description;
11  @gmf.compartment(layout="free")
12  val Component[*] subcomponents;
13  @gmf.affixed
14  val Port[*] ports;
15 }
16
17 @gmf.link(source="from", target="to", label="name", target.decoration="arrow")
18 class Connector {
19   attr String name;
20   ref Port#outgoing from;
21   ref Port#incoming to;
22 }
23
24 @gmf.node(figure="ellipse", size="15,15", label.icon="false",
25  label.placement="external", label="name")
26 class Port {
27   attr String name;
28   val Connector#from outgoing;
29   ref Connector#to incoming;
30 }
```
Listing 9 The annotated SCL Ecore metamodel in Emfatic

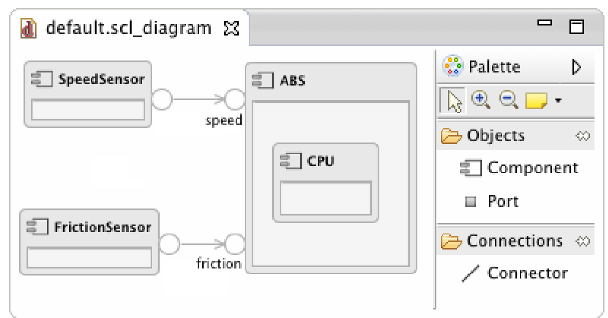

Fig. 16 The polished and patched version of the GMF SCL editor

location (a file named *Ecore2GMF.eol* in the same directory as *SCL.ecore*) so that Eugenia can locate and execute it it after the built-in Ecore2GMF transformation every time it is invoked. The polishing transformation fulfils almost all of our requirements, but cannot be used to adjust the position of the labels placed external to our *Port* construct (see, for example, the position of the *speed* and *friction* labels in Figure 15). Instead, we specify the patch generator shown in Listing 11 and also place it in a predefined location (a file named *GeneratePatches.egx*) in the same directory as *SCL.ecore* so that Eugenia can locate and execute it after invoking the GMF code generator. Our patch generator makes use of the patch template shown in Listing 12 to generate a *.patch* file for each node in our Ecore file which has a label placement set to external. Currently, only the *Port* construct of SCL uses GMF nodes with externally placed labels, but our patch generator can be reused for future iterations of SCL.

Specifying the graphical syntax information in the form of annotations in the SCL metamodel involved adding 7 lines of Emfatic code (excluding line-breaks for formatting reasons). From these 7 lines, 59 elements were produced by Eugenia in the graph, tooling and mapping models. The productivity benefits delivered by Eugenia increase alongside the size and complexity of the metamodel - mainly because the graph, mapping and tooling models do not support the notion of inheritance and therefore inheritance in the Ecore metamodel causes a significant amount of duplication in these models. For example, for the FileSystem metamodel[9], 5 lines of Emfatic annotations result in 102 elements in the graph, tooling and mapping models.

[9] http://www.eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial/

```
1 // Add bold font to component label
2 var componentLabel = GmfGraph!Label.all.
3   selectOne(l|l.name="ComponentLabelFigure");
4 componentLabel.font = new GmfGraph!BasicFont;
5 componentLabel.font.style = GmfGraph!FontStyle#BOLD;
6
7 //Set background color and border
8 //of the component compartment
9 var componentCompartment = GmfGraph!Rectangle.all.
10   selectOne(r|r.name="ComponentSubcomponentsCompartmentFigure");
11 var lineBorder = new GmfGraph!LineBorder;
12 lineBorder.width = 1;
13 componentCompartment.backgroundColor = createColor(245,245,245);
14 componentCompartment.border = lineBorder;
15
16 operation createColor(red : Integer, green : Integer,
17    blue : Integer) : GmfGraph!RGBColor {
18
19   var color = new GmfGraph!RGBColor;
20   color.red = red;
21   color.blue = blue;
22   color.green = green;
23   return color;
24 }
```

Listing 10 The polishing in-place transformation in EOL

```
1 // Imports the EClass#getLabelPlacement() operation from Eugenia
2 import "platform:/plugin/org.eclipse.epsilon.eugenia/transformations/ECoreUtil.eol";
3
4 rule FixExternalLabelMargins
5   // apply this rule to all EClasses where...
6   transform c : ECore!EClass {
7
8   // ... the EClass is annotated with @gmf.node(label.placement="external")
9   guard: c.getLabelPlacement() == "external"
10
11   // invoke the following EGL template on the EClass
12   template : "FixExternalLabelMargin.egl"
13
14   // make the source directory and name of the node available to the template
15   parameters : Map{ "srcDir" = getSourceDirectory(), "node" = c.name }
16
17   // and save the generated text to the following .patch file
18   target : "FixExternalLabelMarginsFor" + c.name + ".patch"
19 }
20
21 // Determine source directory from GMF Gen model
22 @cache
23 operation getSourceDirectory() {
24   var genEditor = GmfGen!GenEditorGenerator.all.first;
25   return genEditor.pluginDirectory.substring(1) + "/" +
26         genEditor.packageNamePrefix.replace("\\.", "/");
27 }
```

Listing 11 The patch generator in EGL

```
diff --git [%=srcDir%]/edit/parts/[%=node%]EditPart.java [%=srcDir%]/edit/parts/[%=node%]
    EditPart.java
index d0684d6..f162365 100644
--- [%=srcDir%]/edit/parts/[%=node%]EditPart.java
+++ [%=srcDir%]/edit/parts/[%=node%]EditPart.java
@@ -143,7 +143,7 @@
    if (borderItemEditPart instanceof [%=node%]NameEditPart) {
      BorderItemLocator locator = new BorderItemLocator(getMainFigure(),
          PositionConstants.SOUTH);
-     locator.setBorderItemOffset(new Dimension(-20, -20));
+     locator.setBorderItemOffset(new Dimension(-5, -5));
      borderItemContainer.add(borderItemEditPart.getFigure(), locator);
    } else {
      super.addBorderItem(borderItemContainer, borderItemEditPart);
```
Listing 12 The patch template for customising the margins of externally placed labels in EGL

Polishing transformations and parametric patching may not have similar productivity results in terms of the number of model elements they produce/modify. For example the polishing transformation in Listing 10 takes 25 lines of code to create 3 and modify 2 elements, and the patch generator and template in Listings 11 and 12 take 41 lines of code to customise 1 line of code per applicable node in the generated Java source. However, in our experience the effort spent for constructing them quickly pays off as graphical editor development is a highly iterative process [3] and because common polishing and patching code can be shared between projects.

5 Evaluation

To assess the extent to which Eugenia delivers the envisioned productivity benefits compared to pure GMF in a conclusive manner, ideally, a large-scale experiment would need to be performed. In such an experiment, productivity would be assessed by comparing the results of two substantially large groups of developers (to minimise skill-related bias), developing graphical editors for a wide range of DSLs (to avoid language-related bias), with and without Eugenia. As GMF targets a niche developer audience, recruiting a sufficiently large number of developers with substantial GMF experience to perform such an experiment in a meaningful way proved to be beyond our capacity. This is consistent with existing related literature (e.g. [8,9]), which also lacks evidence of experiments of this scale. To partially overcome this restriction, in this section, we i) discuss our observations from two small-scale experiments involving two developers, ii) report on our experiences in re-implementing an existing non-trivial graphical editor with Eugenia, ii) present a summary of the feedback collected from Eugenia's user community, and iv) discuss the testing mechanisms used to evaluate the correctness of Eugenia. Finally, we consider the limitations of the proposed approach.

5.1 Small-Scale Evaluation Experiments

Two small-scale experiments were performed to assess the productivity of developers and the maintainability of the produced artefacts using Eugenia versus the standard GMF tooling. Both experiments involved (the same) two volunteers (A, B), both of whom had substantial experience with Eclipse and EMF, but no experience with GMF.

In each experiment, the volunteers were asked to develop a GMF editor for a provided Ecore metamodel. In both experiments, developer A was asked to develop the editor with Eugenia while developer B was asked to develop the editor using standard GMF tooling.

Each experiment consisted of three tasks. In the first task, developers were asked to develop an editor for a provided metamodel that matched in its appearance a provided screenshot. For the next task they were asked to customise an aspect of the editor so that the editor matched a second screenshot. It is worth noting that the customisation requested in both cases was intentionally not supported by Eugenia annotations (i.e. the font weight of certain labels). For the final task, developers were asked to modify the Ecore metamodel (rename an EClass) and then evolve their GMF editor to reflect this change. Developers were asked to keep track of the time it took them to complete each task and record any challenges they faced.

Each experiment involved a 15-minute briefing session, a 2-hour development session and a 15-minute debriefing session where developers would report on their progress, on the time it took them to complete each task, and on any challenges they faced.

Our initial intention was to run only one experiment, however, during the debriefing session that followed the first experiment we identified that both developers had spent a significant amount of time learning how to use the respective tools, by reviewing documentation and trial-by-error. As such, we decided to perform a second experiment that was similar in structure to the first ex-

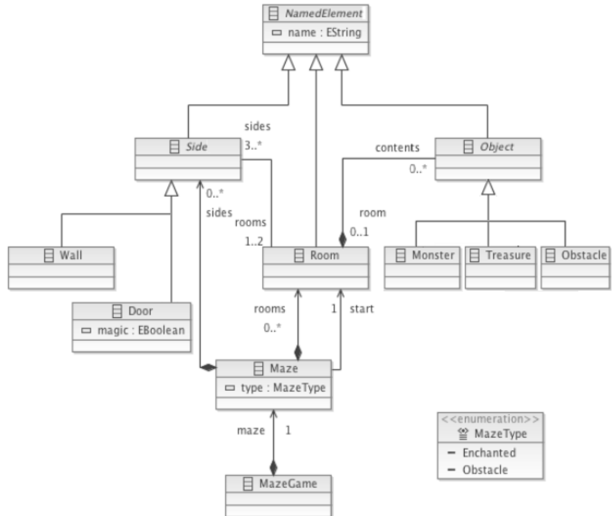

Fig. 17 Maze Metamodel

periment, but asked the developers to construct a graphical editor for a different DSL. Both developers performed significantly better in the second experiment than they did in the first.

5.1.1 Experiment 1 For the first task of this experiment, the two developers were asked to build a GMF editor for a maze game metamodel, which the authors had originally prepared for an MSc module on MDE. The metamodel appears in Figure 17 and the provided reference screenshot appears in Figure 18. For the next task of the experiment, the developers were asked to customise the editor by changing the font weight of room labels to bold. For the final task, developers were asked to rename the Room class of the metamodel to Level and update their editor accordingly. The following sections report on the performance of the two developers.

Developer A (Standard GMF tooling) Developer A completed only the first of the three tasks. Developing the first version of the editor took 100 minutes. The last 20 minutes were used to try and add composition to the diagram but no results were accomplished. As it was the first time that Developer A had used GMF, most of the time on the first task was spent trying to understand the functionality and limitations of the different wizards provided by GMF, and how to appropriately customise the models produced by the wizards.

Developer B (Eugenia) Developer B completed 2 of the 3 tasks within the 2-hour slot. In particular, developing the first version of the editor for the first task took 78 minutes, and polishing the appearance of the editor (the second task) took 41 minutes. Developer B commented that once he became familiar with the tool and created a first functional version of the editor which supported one or two meta-model concepts, completing the task by introducing the required metamodel annotations

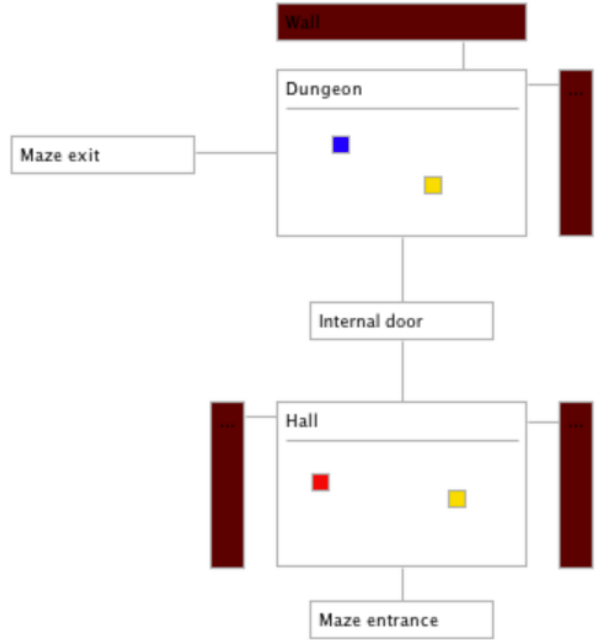

Fig. 18 Maze Reference Editor

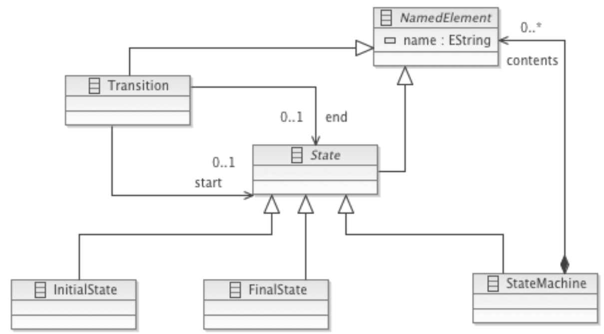

Fig. 19 State Machine Metamodel

was quite straightforward. Regarding task 2, Developer B commented that the most difficult part of the task was to identify the changes that needed to be performed on the GMF Map model; encoding them in EOL was then straightforward.

5.1.2 Experiment 2 For the first task of this experiment, the two developers were asked to develop a GMF editor for a state machine metamodel (Figure 19). The developers were provided with a reference screenshot (Figure 20) for their eventual graphical editor. As in the first experiment, Developer A worked with GMF and Developer B with Eugenia. In the second task of the experiment, the developers were asked to customise the editor by changing the background colour of the state machine compartment to light grey (Figure 21). In the last step of the experiment, developers were asked to rename the State class of the metamodel to Step and update their editor accordingly. The following sections report on the performance of the two developers.

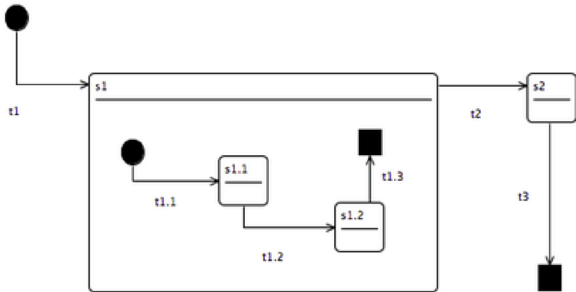

Fig. 20 State Machine Reference Editor

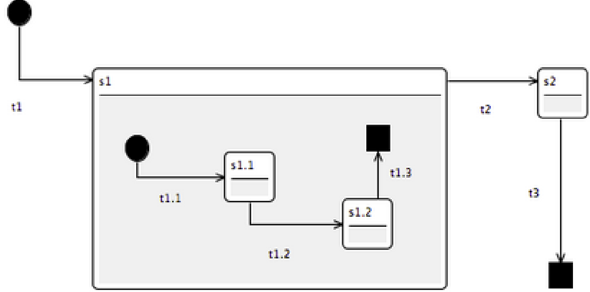

Fig. 21 Polished State Machine Reference Editor

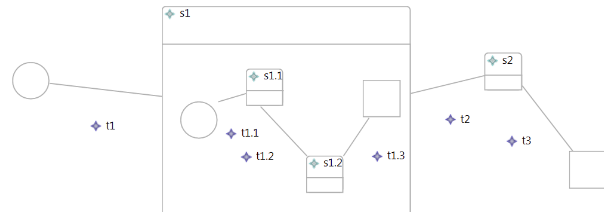

Fig. 22 Developer A's State Machine Editor

Developer A (Standard GMF tooling) Within the available time for the experiment, Developer A managed to produce a nearly complete editor for the provided metamodel with the following limitations: as demonstrated in Figure 22, the editor does not provide a black background colour for the initial and final states, and does not provide an arrowhead on the target of transitions. Developer A did not have time to attempt the second and third task.

Developer B (Eugenia) Developer B completed all three tasks within 45 minutes. In particular, he was able to construct the first version of the editor (the first task) within 25 minutes, customise the background colour of the state compartment (the second task) within 18 minutes, and co-evolve their editor to reflect the change requested in the third task within 3 minutes. Developer B reported that most of the time spent on the second task was used to find the way in which the GMF Graph model should be modified through the built-in editor, and only relatively little time was then used to express the same change as a polishing transformation in EOL code.

5.1.3 Discussion The results obtained through these small-scale experiments are in line with our informal experience that Eugenia can significantly simplify and speed up the editor development process, and make GMF accessible to inexperienced developers. However, as discussed at the beginning of Section 5, due to the small number of developers and the small number of DSLs involved in the experiment, the obtained results are not necessarily generalisable. To further assess whether Eugenia is a significant improvement compared to pure GMF, the following section discusses our observations from a re-implementation of an existing non-trivial graphical editor with Eugenia.

5.2 Implementation of a Simplified BPMN Editor

In [8], the authors performed a comparative evaluation of different graphical modelling frameworks by using them to implement graphical editors for the simplified BPMN metamodel illustrated in Figure 23. In this experiment we attempted to implement a graphical editor that conforms to the visual guidelines proposed in Figure 24 using Eugenia, and compare the development effort against that required by the authors of the paper to implement the same editor using pure GMF (25 man-days).

Our implementation consists of the Eugenia-annotated copy of the metamodel of Figure 23, seven SVG images, a concise polishing transformation and two custom Java classes[10].

5.2.1 Annotated BPMN metamodel The annotated metamodel illustrated in Listing 13, defines the bulk of the graphical BPMN syntax, using SVG images to represent non-trivial graphical elements such as gateways (lines 50, 54 and 68) and events (lines 61 and 66), and required 73 minutes to develop. However, we were unable to define the left-to-right layout of pools and lanes and the source decoration (white circle) of message flows (see Figure 24) using the built-in annotations provided by Eugenia.

5.2.2 Polishing Transformation To address these limitations, we had to use Java to define a custom layout for pools and lanes (24 lines of code), and a custom white circle figure (44 lines of code) to be used as the source decoration of message flows (see Figure 24). To integrate

[10] The complete source code is available under `http://dev.eclipse.org/svnroot/modeling/org.eclipse.epsilon/trunk/examples/org.eclipse.epsilon.eugenia.bpmn` and `http://dev.eclipse.org/svnroot/modeling/org.eclipse.epsilon/trunk/examples/org.eclipse.epsilon.eugenia.bpmn.diagram.custom`.

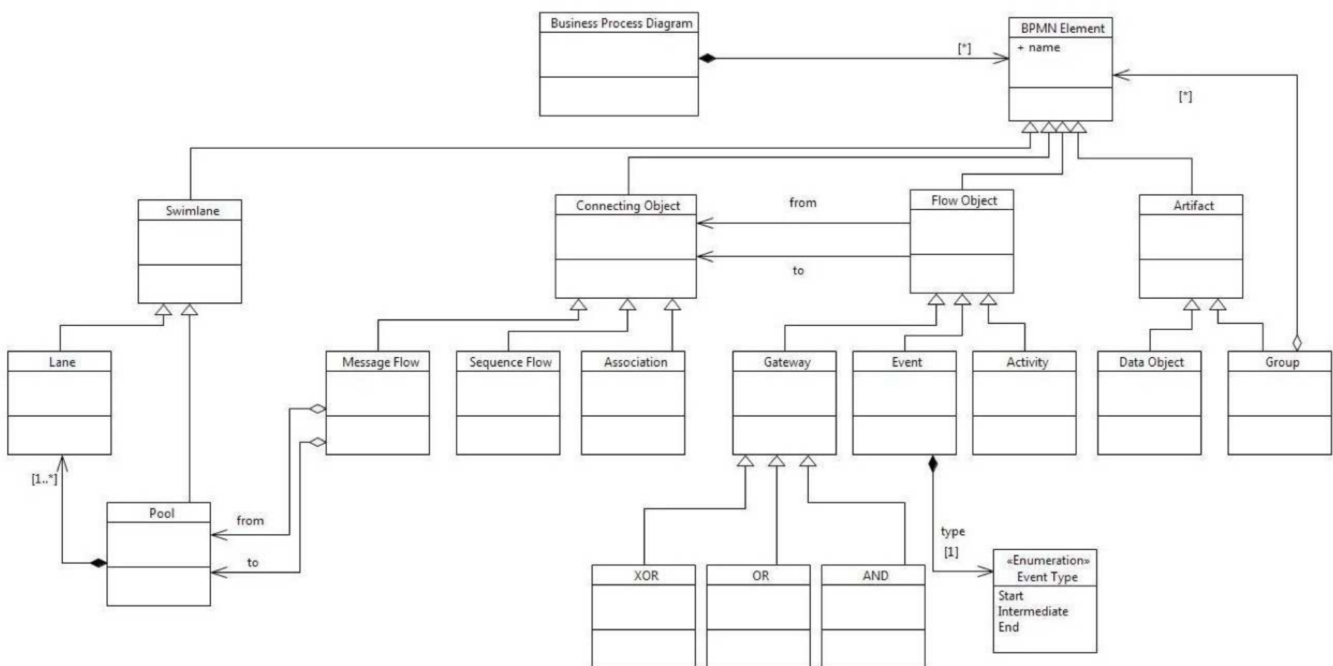

Fig. 23 Simplified BPMN metamodel from [8]

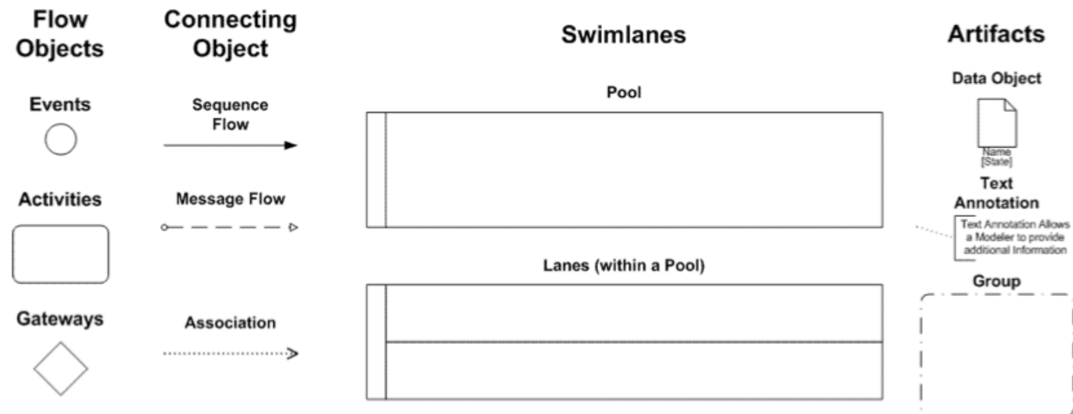

Fig. 24 Core set of BPMN elements from [8]

this custom code with the generated part of the editor, we had to write a brief polishing transformation illustrated in Listing 14. This required 92 minutes of additional effort.

5.2.3 Discussion Overall, implementing an editor for the simplified BPMN metamodel using Eugenia required 165 minutes (2.75 hours), which is a significant reduction from the 25 days measured by the authors of [8]. Similar to the set of experiments discussed in Section 5.1, comparing the two figures directly is not straightforward as performance may have been affected by the familiarity of the developers with the respective tools and the chosen subset of BPMN. The following section extends the discussion by reviewing feedback obtained from the user community of Eugenia.

5.3 Community Validation and Feedback

Since its first release, Eugenia has been used extensively by researchers and engineers both in academia and in industry. Evidence for this exists in the large number of posts in the Epsilon forum[11], and in several publications that discuss using Eugenia to develop graphical editors for domain specific languages across a number of domains.

Seehusen and Stølen [10] describe their experiences in constructing a GMF editor for a risk modelling language (CORAS) with and without Eugenia. When using GMF alone, Seehusen and Stølen report that the development of the GMF editor took longer than anticipated (about 3.5 man-months), partly due to the time taken to iter-

[11] `http://www.eclipse.org/epsilon/forum`

```
@namespace(uri="http://eclipse.org/eugenia/simplebpmn", prefix="sbpmn")
@gmf
package SimpleBPMN;

@gmf.diagram
class BusinessProcessDiagram {
 val BPMNElement[*] elements;
}

class BPMNElement {
 attr String name;
}

@gmf.node(label="name")
abstract class Swimlane extends BPMNElement {}

class Lane extends Swimlane {
 @gmf.compartment
 val FlowObject[*] flowObjects;
}

class Pool extends Swimlane {
 @gmf.compartment
 val Lane[*] lanes;
}

@gmf.link(label="name", source="from", target="to", color="0,0,0")
abstract class ConnectingObject extends BPMNElement {
 ref FlowObject from;
 ref FlowObject to;
}

@gmf.link(tool.name="Message Flow", style="dash", target.decoration="closedarrow")
class MessageFlow extends ConnectingObject {}

@gmf.link(tool.name="Sequence Flow", target.decoration="filledclosedarrow")
class SequenceFlow extends ConnectingObject {}

@gmf.link(style="dot", target.decoration="arrow")
class Association extends ConnectingObject {}

@gmf.node(label="name")
abstract class FlowObject extends BPMNElement {}

@gmf.node(figure="svg", margin="2", label.icon="false",
 label.placement="external", resizable="false")
abstract class Gateway extends FlowObject {}

@gmf.node(tool.name="XOR Gateway",
 svg.uri="platform:/plugin/org.eclipse.epsilon.eugenia.bpmn/svg/xor-gateway.svg")
class XOR extends Gateway {}

@gmf.node(tool.name="OR Gateway",
 svg.uri="platform:/plugin/org.eclipse.epsilon.eugenia.bpmn/svg/or-gateway.svg")
class OR extends Gateway {}

@gmf.node(tool.name="AND Gateway",
 svg.uri="platform:/plugin/org.eclipse.epsilon.eugenia.bpmn/svg/and-gateway.svg")
class AND extends Gateway {}
```

```
60 @gmf.node(tool.name="Start Event", figure="svg",
61  svg.uri="platform:/plugin/org.eclipse.epsilon.eugenia.bpmn/svg/start-event.svg",
62  label.icon="false", label.placement="external", resizable="false", margin="2")
63 class StartEvent extends FlowObject {}
64
65 @gmf.node(tool.name="Intermediate Event", figure="svg",
66  svg.uri="platform:/plugin/org.eclipse.epsilon.eugenia.bpmn/svg/intermediate-event.svg",
67  label.icon="false", label.placement="external", resizable="false", margin="2")
68 class IntermediateEvent extends FlowObject {}
69
70 @gmf.node(tool.name="End Event", figure="svg", label.icon="false",
71  label.placement="external", resizable="false", margin="2"
72  svg.uri="platform:/plugin/org.eclipse.epsilon.eugenia.bpmn/svg/end-event.svg")
73 class EndEvent extends FlowObject {}
74
75 class Activity extends FlowObject {}
76
77 @gmf.node(label="name")
78 abstract class Artifact extends BPMNElement {}
79
80 @gmf.node(tool.name="Data Object", figure="svg", label.icon="false",
81  label.placement="external", resizable="false", margin="2",
82  svg.uri="platform:/plugin/org.eclipse.epsilon.eugenia.bpmn/svg/data-object.svg")
83 class DataObject extends Artifact {}
84
85 @gmf.node(label.placement="external", label.icon="false", border.style="dash", margin="2")
86 class Group extends Artifact {
87   @gmf.compartment
88   val BPMNElement[*] elements;
89 }
```

Listing 13 Eugenia-annotated copy of the Simplified BMPN metamodel

atively develop the various EMF and GMF artefacts in Figure 1. After switching to Eugenia, the authors remark that "This [Eugenia] worked very well; it was very easy to learn and the execution of the transformation [that produced a GMF editor from the CORAS metamodel] was reduced to pressing a button as opposed to clicking through a series of dialogs". A primary concern with the GMF transformations were that they were found to be impractical due to their interactive nature, and that the single-sourcing approach of Eugenia "worked far better than the GMF dialog based approach". Seehusen and Stølen also report that their unfamiliarity with GMF (and related technologies) and that a lack of good GMF documentation were also contributing factors to the longer-than-anticipated development time of their GMF editor. It is possible that the GMF annotations provided by Eugenia increase the learnability of GMF and reduce the need for detailed documentation of low-level GMF features, but Seehusen and Stølen do not consider this in their evaluation.

Sun et al. [11] report and reflect on their experiences in developing a graphical DSL for distributed time-triggered systems. Sun et al. notes that Eugenia was chosen over GMF due to "previous experience with the complexities and inefficiencies of GMF" and that applying Eugenia "considerably sped up the creation of graphical DSL editors". Sun et al. focuses on the benefits of the single-sourcing approach employed by Eugenia, noting that "building and maintaining a modeling tool with GMF is by no means an easy task, which requires six individual models that are highly dependent on each and all need to be in sync with each other. The Eugenia tool essentially reduces development and maintenance down to one model and some optional, separate customization information".

Dayibaş and Oğutüzün [12] applied Eugenia to develop a graphical editor for feature-driven product derivation, and noted that Eugenia makes "editor development easier" as compared to GMF, which "requires lots of configurations [sic]".

Eugenia has also been used to develop graphical editors for DSLs in fields such as distributed time-triggered systems , software architectures [13–15], industrial robot control [16], mobile robotics[12], distributed application modelling [17], access control policies [18], and enterprise application integration [19].

[12] `https://code.google.com/p/rbcodegen` (Last accessed: May 2013)

```
1 var laneFigure = GmfGraph!RoundedRectangle.all.selectOne(r|r.name="LaneFigure");
2 var poolFigure = GmfGraph!RoundedRectangle.all.selectOne(r|r.name="PoolFigure");
3
4 poolFigure.setCustomLayout(
5   "org.eclipse.epsilon.eugenia.bpmn.diagram.custom.SwimlaneLayout");
6 laneFigure.setCustomLayout(
7   "org.eclipse.epsilon.eugenia.bpmn.diagram.custom.SwimlaneLayout");
8
9 var laneLabelFigure = GmfGraph!Label.all.selectOne(l|l.name="LaneLabelFigure");
10 var poolLabelFigure = GmfGraph!Label.all.selectOne(l|l.name="PoolLabelFigure");
11 laneLabelFigure.makeVertical();
12 poolLabelFigure.makeVertical();
13
14 var circleDecoration = new GmfGraph!CustomDecoration;
15 circleDecoration.qualifiedClassName =
16   "org.eclipse.epsilon.eugenia.bpmn.diagram.custom.CircleDecoration";
17 circleDecoration.name = "Circle";
18 GmfGraph!FigureGallery.all.first().figures.add(circleDecoration);
19 GmfGraph!PolylineConnection.all.
20   selectOne(pc|pc.name="MessageFlowFigure").sourceDecoration = circleDecoration;
21
22 operation GmfGraph!Layoutable setCustomLayout(class : String) {
23   var layout = new GmfGraph!CustomLayout;
24   layout.qualifiedClassName = class;
25   self.layout = layout;
26 }
27
28 operation GmfGraph!Label makeVertical() {
29   var labelFigure = self;
30   var labelFigureName = labelFigure.name;
31   var labelFigureText = labelFigure.text;
32   var figure = labelFigure.eContainer();
33   var labelFigureChildAccess =
34     GmfGraph!ChildAccess.all.selectOne(ca|ca.figure = labelFigure);
35   delete labelFigure;
36   labelFigure = new GmfGraph!VerticalLabel;
37   labelFigure.name = labelFigureName;
38   labelFigure.text = labelFigureText;
39   figure.children.add(0, labelFigure);
40   labelFigureChildAccess.figure = labelFigure;
41 }
```

Listing 14 Simplified BPMN editor polishing transformation

As a teaching tool, Eugenia has been used in lectures at universities in Kassel (Germany)[13], Madrid (Spain)[14] and Oslo (Norway)[15]. An introductory book recently written by the Spanish MDE research community includes a chapter on Eugenia [20]. Eugenia has been used to implement an arcade game[16] for a project that seeks to teach DSLs via educational videogames[17] at the University of Cadiz. It is also used at York (UK) for both MSc teaching on MDE[18], and to introduce high-school students to MDE [21].

[13] http://seblog.cs.uni-kassel.de/wp-content/uploads/2011/11/Uebung2.pdf (Last accessed: April 2013)

[14] http://astreo.ii.uam.es/~jlara/doctorado.2010/3_DSLs_tecnologias.pdf (Last accessed: April 2013)

[15] http://www.uio.no/studier/emner/matnat/ifi/INF5120/v11/undervisningsmateriale/Lecture4_ModelTransformation.pdf (Last accessed: April 2013)

[16] http://github.com/chelder86/ArcadeTongame (Last accessed: May 2013)

[17] http://wikis.uca.es/wikiPILI/index.php/Videojuegos_Educativos_DSL (Last accessed: May 2013)

[18] http://www.cs.york.ac.uk/postgraduate/modules/mode.html (Last accessed: February 2014)

5.4 Regression Testing

Eugenia provides a set of transformations on top of another model-driven workflow (GMF). Ideally, each aspect of the transformation should be individually tested with an appropriate set of input models and assertions on the resulting models. However, these tests would be brittle, as a new version of GMF with revised metamodels and transformations of its own could potentially break many of them. The tests would also potentially need to consider many model elements, as a single annotation in the Emfatic code could produce large changes in the GMF models.

As a compromise, we use regression testing for validating each new version of Eugenia, using the EUnit testing framework [22]. We run Eugenia against several annotated metamodels that provide a representative sample of all features in Eugenia, and ensure that there are no significant differences between the generated GMF models and a previously generated set of models which have been manually validated. Non-significant differences are normalised using the available options in EUnit and custom EOL code: for instance, some of the GMF models contain Java code, and differences due to whitespace in the code should not be treated as significant. However, if the significant differences are due to a change in GMF and not a bug in Eugenia, the newly generated models are manually validated and replace the old generated models.

5.5 Limitations

Eugenia currently demonstrates four notable limitations. Firstly, Eugenia annotations pollute metamodels with information irrelevant to their primary purpose (abstract syntax definition). While user feedback indicates that this is often an acceptable trade-off for the increased efficiency offered by the tool, to avoid metamodel pollution without sacrificing usability, we are experimenting with extracting a standalone text-based language from the annotations provided by Eugenia.

The second limitation of Eugenia is that it does not provide support for automating the process of developing graphical editors comprising of more than one type of diagrams. Developers can still use the standard GMF process for achieving this, but as this is a fragile and error-prone process it would benefit from additional automation.

A third limitation of Eugenia is that, to compose polishing transformations, developers need to become familiar with a particular transformation language (EOL). To overcome this limitation, future versions of Eugenia would benefit from modular support for custom transformation languages.

Finally, this annotation-based approach is arguably more suitable for languages in which the structure of the abstract and concrete (graphical) syntax are not radically different. For languages with divergent abstract and graphical syntaxes, or where substantial customisation is required, a substantial amount of polishing transformation code may need to be developed and maintained as discussed in Section 3.7.

6 Related Work

Similarly to Eugenia, GmfGen [23] also aims at simplifying the incremental development of GMF editors. The graph, mapping and tooling models depicted in Figure 1 typically contain some duplication of information. This duplication exasperates any inconsistency problems that may arise when changes are made to one of the models. GmfGen provides templates for generating the models needed to construct a GMF editor. The templates remove most of the duplication present in GMF models. However, GmfGen does not address the steep learning curve encountered when first using GMF to generate a visual editor. In fact, knowledge of GMF is required to understand the way in which the GmfGen templates are constructed. Instead, Eugenia focuses on abstracting away from GMF. In [24], the authors present a metamodel-annotation-based framework that postdates Eugenia and targets a home-grown graphical modelling framework instead of GMF.

Obeo Designer (OD) is a commercial product that builds on top of the GMF runtime but does not make use of the GMF code generation facilities. Obeo Designer replaces the 3 intermediate models of GMF (Graph, Tool, Mapping) with a single model that contains similar constructs and provides more elaborate — but still tree-based editors — to define this model. A strong advantage of OD over the default GMF tooling is that the former does not involve a code generation step and as such changes to the graphical syntax can be tested more quickly.

This work focuses on GMF as, despite its shortcomings, it is still one of the most powerful, flexible and widely-used open-source graphical editor frameworks available today. GMF has a large user community and when tuned appropriately it can achieve impressive results (the widely used IBM RSA UML[19] modeller, as well as the open-source Topcased[20] and Papyrus[21] modelling tools are all implemented atop GMF).

Beyond GMF there is a large number of open-source and commercial frameworks that provide comparable support for developing graphical editors, most notably MetaEdit+ [25], GME [26] (and its Eclipse-based GEMS branch), AToM^3 [27], Graphiti (and its Spray

[19] http://www-01.ibm.com/software/websphere
[20] `http://www.topcased.org/`
[21] `http://www.eclipse.org/papyrus/`

extension[22]). In our view, comparing these frameworks against Eugenia would quickly boil down to a comparison against GMF, which has been attempted before [8, 9]. This is beyond the scope of this paper, but a very interesting direction for future work.

7 Conclusions and Further Work

In this paper we have presented an approach that employs metamodel annotations, model-to-model and in-place model transformations to deliver productivity and consistency benefits to the process of developing graphical model editors with the EMF and GMF frameworks. The tool that implements the proposed approach (Eugenia) has been well-received from the Eclipse modelling community and there is strong evidence that it is extensively used by both researchers and practitioners.

While Eugenia already greatly improves the usability of GMF and lowers the entrance barrier for inexperienced developers, a significant amount of work remains, including support for: sub-diagrams, multiple (non-hierarchical) diagrams in the same file, and advanced property editing. Ongoing research seeks to addresses some of these issues in the MOSKitt[23] and EEF[24] projects. We aim to converge with these projects and progressively extend Eugenia to support, in a usable and intuitive manner, all of the features discussed above.

An additional interesting direction for further research is to target alternative graphical editor frameworks such as Graphiti, or web-based frameworks such as UMLCanvas[25].

Acknowledgements

Parts of this work were supported by the European Commission's 7th Framework Programme, through grant #611125 (MONDO). Other parts of this work were supported by the doctoral scholarship PU-EPIF-FPI-C 2010-065 from the University of Cádiz and by the MoD-SOA project (TIN2011-27242) of the National Research, Development and Innovation Program of the Spanish Ministry of Science and Innovation.

The authors would also like to thank Adolfo Sanchez-Barbudo Herrera and Horacio Hoyos Rodriguez for their help with the evaluation experiments discussed in Section 5.

[22] `https://code.google.com/a/eclipselabs.org/p/spray`
[23] `http://www.moskitt.org`
[24] `http://www.eclipse.org/modeling/emft/?project=eef\#eef`
[25] `http://umlcanvas.org/`

A List of Annotations Supported by Eugenia

The complete list of metamodel annotations currently supported by Eugenia is given below. Features of GMF that are not made available via annotations can be managed using the polishing transformation mechanism. An up-to-date reference guide to the annotations of Eugenia is available on the Epsilon website[26].

A.1 gmf.diagram

Denotes the root EClass for the editor. Only one (non-abstract) *EClass* must be annotated as *gmf.diagram*. The annotation accepts the following details.

- *diagram.extension* (optional): the file extension for the diagram file;
- *model.extension* (optional): the file extension for the domain (EMF) model;
- *onefile* (optional): specifies whether the domain model and the diagram should be stored in the same file;
- *rcp* (optional): specifies whether the editor is intended to be part of a Rich Client Platform product;
- *units* (optional): the units for the diagram (e.g. *Pixels*).

A.2 gmf.node

Applies to an *EClass* and denotes that its instances should appear on the diagram as nodes. The annotation accepts the following details.

- *figure* (optional): the figure that will represent the node. Can be set to *rectangle*, *ellipse*, *rounded* (default), *svg* (see *svg.uri*), *polygon* (see *polygon.x* and *polygon.y*) or to the fully qualified name of a Java class that implements the GMF *Figure* interface;
- *border.color* (optional): an RGB color (e.g. *255,0,0*) that will be set as the node's border color;
- *border.style* (optional): the style of the node's border. Can be set to *solid* (default), *dash* or *dot*;
- *border.width* (optional): an integer that specifies the width of the node's border;
- *color* (optional): an RGB color that specifies the node's background color;
- *label*: the name(s) of the *EAttribute*(s) of the *EClass*, the value(s) of which will be displayed as the label of the node. If *label.placement* is set to *none*, this detail is not required;
- *label.icon* (optional): if set to *true* (default) a small icon appears on the left of the label;
- *label.parser* (optional): indicates the unqualified name of the class that will parse the text entered by the user into the label;

[26] `http://www.eclipse.org/epsilon/doc/eugenia/`

– *label.edit.pattern* (optional): similar to *label.pattern*, but only for editing the label;
– *label.pattern* (optional): if more than one attributes are specified in the label, the format detail is necessary to specify how their values will be rendered in the label. The format follows the Java Message Format style (e.g. {*0*} *:* {*1*}). The same pattern is used for editing and viewing the label.
– *label.view.pattern* (optional): similar to *label.pattern*, but only for viewing the label;
– *label.placement* (optional): defines the placement of the label in relation to the node. Can be set to *internal*, *external* or *none* (in which case, no label will be shown);
– *label.text* (optional): defines the default text to be used when the *EAttribute*(s) in *label* are not set. By default, it is set to the name of the *EClass*;
– *label.readOnly* (optional): a value of *true* denotes that the label cannot be changed in the generated diagram editor;
– *margin* (optional): inset margin (5 units by default) for the node;
– *phantom* (optional): defines if the node is phantom (*true*/*false*). Phantom nodes are particularly useful in order to visualize containment references using links instead of spatial containment[27];
– *polygon.x* (when *figure* is set to *polygon*): list of space-separated integers with the X coordinates of the polygon used as figure;
– *polygon.y* (when *figure* is set to *polygon*): list of space-separated integers with the Y coordinates of the polygon used as figure;
– *resizable* (optional): a value of *false* disables all the resize handles for the node;
– *size* (optional): a GMF dimension that will be used as the node's preferred size (e.g. *10,5*);
– *svg.uri* (when *figure* is set to *svg*): URI of the *.svg* file to be used as figure for the node. For instance, *platform:/plugin/my.plugin/my.svg* will access the *my.svg* file in the *my.plugin* plugin.

A.3 gmf.link

Applies to *EClass*es that should appear on the diagram as links, and to non-containment *EReference*s.

A.4 gmf.link (for EClasses)

The annotation accepts the following details.

– *color* (optional): the RGB color of the link;
– *incoming* (optional): specifies whether the generated editor should allow links to be created from target to source. Defaults to *false*;
– *label* (optional): the names of the *EAttribute*s of the *EClass* the value of which will be displayed as the label of the link;
– *label.parser* (optional): indicates the unqualified name of the class that will parse the text entered by the user into the label;
– *label.text* (optional): defines the default text to be used when the *EAttribute*(s) in *label* are not set. By default, it is set to the name of the *EClass*;
– *source* : the source non-containment *EReference* of the link;
– *source.constraint* (optional): OCL assertion that should be checked by the graphical editor when creating a link. For instance, `self <> oppositeEnd` would forbid users for creating a link from a node to itself (a self-loop): *self* is the source of the link, and *oppositeEnd* is the target of the link;
– *source.decoration* (optional): the decoration of the source end of the link. Can be set to *none*, *arrow*, *rhomb*, *filledrhomb*, *square*, *filledsquare*, *closedarrow*, *filledclosedarrow*, or the fully qualified name of a Java class that implements the *org.eclipse.draw2d.RotatableDecoration* interface;
– *style* (optional): the style of the link (see *border.style* above);
– *target* : the target non-containment *EReference* of the link;
– *target.constraint* (optional): See *source.constraint*;
– *target.decoration* (optional): See *source.decoration*;
– *width* (optional): the width of the link.

A.5 gmf.link (for non-containment EReferences)

It accepts the following details:

– *color* (optional): the RGB color of the link;
– *label* (optional): The static text that will be displayed as the label of the link. If no label is specified, the name of the reference is displayed instead;
– *label.text* (optional): equivalent to *label* in this case;
– *source.decoration* (optional): See *source.decoration* above;
– *style* (optional): the style of the link (see *border.style* above);
– *target.decoration* (optional): As above;
– *width* (optional): the width of the link.

A.6 gmf.compartment (for containment EReferences)

Defines that the containment reference will create a compartment where model elements that conform to the type of the reference can be placed. It accepts the following details:

[27] For an example involving phantom nodes, the reader can refer to http://eclipse.org/epsilon/doc/articles/eugenia-phantom-nodes/

- *collapsible* (optional): When set to *false* it prevents the compartment from collapsing (default is *true*);
- *layout* (optional): The layout of the compartment. Can be set to free (default) or list.

A.7 gmf.affixed (for containment EReferences)

Defines that the containment reference will create nodes which are affixed to the edges of the containing node. An example demonstrating affixed references is illustrated in Section 5.

A.8 gmf.label (for EAttributes)

Defines additional labels for the containing *EClass*. These labels will be displayed underneath the default label for the containing *EClass*. It accepts the following details:

- *label.edit.pattern* (optional): like *label.pattern*, but only for editing the label;
- *label.parser* (optional): indicates the unqualified name of the class that will parse the text entered by the user into the label;
- *label.pattern* (optional): if more than one attributes are specified in the label, the format detail is necessary to show how their values will be rendered in the label. The format follows the Java Message Format style (e.g. *0 : 1*). The same pattern is used for editing and viewing the label;
- *label.readOnly* (optional): a value of *true* denotes that the label cannot be changed in the generated diagram editor;
- *label.text* (optional): defines the default text to be used when the attribute is not set;
- *label.view.pattern* (optional): similar to *label.pattern*, but only for viewing the label.

All *gmf.node* and *gmf.link* annotations also support the following details which can be used to define the appearance of the respective palette tools of the editor.

- *tool.description* (optional): the description of the creation tool.
- *tool.large.bundle* (optional): the bundle of the large icon of the creation tool.
- *tool.large.path* (optional): the path of the large icon of the creation tool.
- *tool.name* (optional): the name of the creation tool.
- *tool.small.bundle* (optional): the bundle of the small icon of the creation tool.
- *tool.small.path* (optional): the path of the small icon of the creation tool.

References

1. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proc. ICSE*, pages 471–480. ACM, 2011.
2. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks. *EMF: Eclipse Modelling Framework*. Eclipse Series. Addison-Wesley Professional, second edition, December 2008.
3. Christoph Wienands, Michael Golm. Anatomy of a Visual Domain-Specific Language Project in an Industrial Context. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 453–467, Denver, Colorado, USA, 2009.
4. Dimitrios S. Kolovos, Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck. Taming EMF and GMF using model transformation. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 211–225, Berlin, Heidelberg, 2010. Springer-Verlag.
5. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. The Epsilon Transformation Language. In *Proc. 1st International Conference on Model Transformation (ICMT)*, Zurich, Switzerland, July 2008.
6. Eclipse Foundation. Epsilon Modeling GMT component. http://www.eclipse.org/gmt/epsilon.
7. 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.
8. Amine El Kouhen, Cedric Dumoulin, Sébastien Gerard, and Pierre Boulet. Evaluation of Modeling Tools Adaptation. Technical report, Laboratoire d'Intégration des Systèmes et des Technologies - LIST , LIFL - DART - LIFL - DART , Laboratoire d'Informatique Fondamentale de Lille - LIFL , LIFL - DART/Émeraude, 2012.
9. Nick Baetens. Comparing graphical DSL editors: AToM3, GMF, MetaEdit+. Technical report, University of Antwerp, 2011.
10. Fredrik Seehusen and Ketil Stølen. An Evaluation of the Graphical Modeling Framework (GMF) Based on the Development of the CORAS Tool. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin Heidelberg, 2011.
11. Yu Sun, Christoph Wienands, Meik Felser. Applying Model-Driven Design and Development to Distributed Time-Triggered Systems. In *Proc. 2nd International Conference on Engineering and Meta-Engineering*, 2011.
12. O. Dayibas and H. Oguztuzun. Kutulu: A domain-specific language for feature-driven product derivation. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 105–110, 2012.
13. E. Demirli and B. Tekinerdogan. Save: Software architecture environment for modeling views. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 355–358, 2011.
14. Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. Developing next generation ADLs through MDE techniques. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 85–94, New York, NY, USA, 2010. ACM.
15. C. Pena and J. Villalobos. An MDE approach to design enterprise architecture viewpoints. In *Commerce and Enterprise Computing (CEC), 2010 IEEE 12th Conference on*, pages 80–87, 2010.
16. Yu Sun, Jeff Gray, Karlheinz Bulheller, and Nicolaus Baillou. A model-driven approach to support engineering changes in industrial robotics software. In RobertB. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*, pages 368–382. Springer Berlin Heidelberg, 2012.
17. A. Noguero and I. Calvo. FTT-Modeler: A support tool for FTT-CORBA. In *Information Systems and Technologies (CISTI), 2012 7th Iberian Conference on*, pages 1–6, 2012.
18. J. Calvillo, I. Román, and L.M. Roa. Empowering citizens with access control mechanisms to their personal health resources. *International Journal of Medical Informatics*, 82(1):58 – 72, 2013.
19. Rafael Z. Frantz, Antonia M. Reina Quintero, and Rafael Corchuelo. A domain specific language to design enterprise application integration solutions. *International Journal of Cooperative Information Systems*, 20(02):143–176, 2011.
20. J. García, F. O. García, V. Pelechano, A. Vallecillo, J. M. Vara, and C. Vicente-Chicote, editors. *Desarrollo de software dirigido por modelos: conceptos, métodos y herramientas*. Ra-Ma, 2013.
21. J. R. Williams, S. Poulding, L. M. Rose, R. F. Paige, and F. A. C. Polack. Identifying desirable game character behaviours through the application of evolutionary algorithms to model-driven engineering metamodels. volume 6956, pages 112–126. Springer Berlin / Heidelberg, 2011.
22. Antonio García-Domínguez, Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, and Inmaculada Medina-Bulo. EUnit: a unit testing framework for model management tasks. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 395–409, Berlin, Heidelberg, 2011. Springer-Verlag.
23. Enrico Schnepel. GenGMF: Efficient editor development for large meta models using the Graphical Modelling Framework. In *Proc. Special Interest Group on Model-Driven Software Engineering (SIG-MDSE)*, 2008.
24. S. Temate, L. Broto, A. Tchana, and D. Hagimont. A high level approach for generating model's graphical editors. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 743–749, 2011.
25. MetaCase. Meta-Edit+. http://www.metacase.com.
26. Generic Modeling Environment. http://www.isis.vanderbilt.edu/Projects/gme.
27. Juan De Lara, Hans Vangheluwe. Using AToM3 as a Meta-CASE Tool. In *Proc. 4th International Conference on Enterprise Information Systems*, pages 642–649, Ciudad Real - Spain, April 2002.