EuGENia: Taming EMF and GMF using Model Transformation

Dimitrios S. Kolovos[1], Louis M. Rose[1], Saad Bin Abid[2], Richard F. Paige[1], Fiona A.C Polack[1], and Goetz Botterweck[2]

[1] Department of Computer Science,
University of York, YO10 5DD, York, UK,
{dkolovos, louis, paige, fiona}@cs.york.ac.uk
[2] Lero - The Irish Software Engineering Research Centre,
Limerick, Ireland,
{saad.binabid, goetz.botterweck}@lero.ie

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 low level-interconnected models through a loosely-guided, labour-intensive and error-prone process. In this paper we demonstrate how the application of model transformation techniques can help with taming the complexity of GMF and EMF and deliver significant productivity and quality benefits. In particular we demonstrate EuGENia, a widely-used tool that adopts a single-sourcing approach and advanced model transformation techniques for automatically producing and maintaining the low-level models required by EMF and GMF. We evaluate EuGENia through automated testing and substantial feedback from researchers and practitioners within the Eclipse modelling community.

1 Introduction

The Eclipse Modelling Framework (EMF)[1] is a widely-used model management framework implemented atop the Eclipse software development platform. Over the last few years, Eclipse and EMF have become the de-facto standards in the MDE community and currently the majority of MDE tools (e.g. ATL, oAW, Kermeta, MOFScript, Epsilon) are seamlessly integrated with them. The Graphical Modelling Framework (GMF) is a powerful and widely-used framework for implementing graphical editors for EMF-based modelling languages.

Both EMF and GMF adopt a generative approach: starting from an Ecore[3] metamodel which specifies the abstract syntax of the modelling language, developers derive and maintain a set of more fine-grained, lower-level models that describe the graphical syntax of the language and various implementation options, and which can be then consumed by the EMF and GMF code generators

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

in order to produce the concrete artefacts (i.e. Java code and configuration files) that realize the editor. EMF and GMF are particularly powerful and flexible and provide a wide range of options for customizing almost every aspect of the generated editor. However, the trade-off for power and flexibility is increased complexity. As discussed in the industrial experience report presented by Wienands and Golm [2], 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, complex interconnected models. Increased complexity in conjunction with sub-optimal tool support for creating and maintaining the required low-level models make implementing a graphical editor with GMF a painful experience, particularly for inexperienced developers.

In this paper we demonstrate how model transformation can help with taming the complexity of GMF and EMF, by raising the level of abstraction, lowering the entrance barrier for new developers, and delivering significant productivity and quality benefits to the process of constructing graphical editors for modelling languages. In particular we demonstrate EuGENia, a mature and widely-used tool that adopts a single-sourcing approach based on metamodel annotation and model transformation techniques (both model-to-model and in-place model transformation) for producing and maintaining all the low-level models required by the EMF and GMF code generators in an automated fashion.

The rest of the paper is organized 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 evaluates the findings of this work and demonstrates the productivity and quality benefits delivered by model transformation in this practical problem. Section 5 provides an overview of related work, and Section 6 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 identify the labour-intensive, error-prone and maintenance-crippling 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 respective 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.

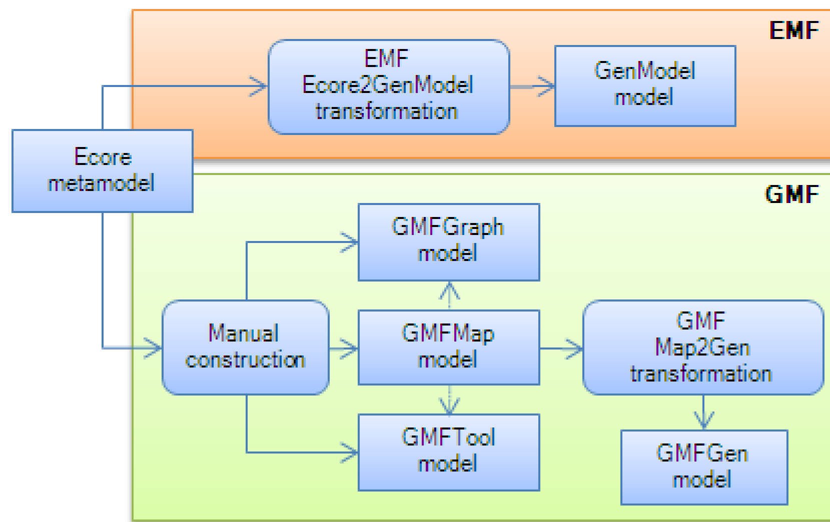

Fig. 1. EMF/GMF Process Overview

2.1 Specifying the Abstract Syntax and Generating Code using EMF

In this step, the developer needs to define the abstract syntax of the language using Ecore. Following that, the developer can invoke the built-in EMF transformation to transform the Ecore metamodel into a *GenModel*. A *GenModel* is a model which 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 optionally customized and fine-tuned manually. Finally, the GenModel is consumed by an EMF built-in code generator which produces all the necessary 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 respective GenModel without overwriting the user-defined customizations. However, the reconciler is only effective for simple changes in the Ecore metamodel; for more complex changes the GenModel needs to be regenerated and customized from scratch. This introduces a significant maintenance overhead as it's 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 again) in case this is needed in the future.

2.2 Specifying the Graphical Syntax and Generating Code with GMF

Once the metamodel has been defined and the respective EMF code has been generated, in order to implement a graphical editor for the language using GMF, the developer needs to construct three additional models. The *graph* model (*GMFGraph*) specifies the graphical elements (shapes, connections, labels, decorations etc.) involved in the editor, the *tooling* model (*GMFTool*) specifies the element creation tools that will be available in the palette of the editor, and 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.). The mapping model is then automatically transformed into an even more fine-grained generator model (*GMFGen*) which contains all the low-level information[4] that the GMF code generator needs in order to produce the concrete artefacts (Java code and configuration files) that realize the graphical editor.

In terms of automation, GMF provides a built-in wizard for automatically generating initial versions of 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[2] - and this is reasonable given how little can be generally inferred about the graphical syntax based on the abstract syntax alone. As a result, in practice these three models need to be hand-crafted using a set of very basic tree-based editors provided by GMF, and this is widely-recognized to be a laborious and error-prone process, particularly given the complexity of the metamodels these models conform to, and the low-level error messages that GMF produces if they are not configured in a valid way. Perhaps more challenging than constructing these GMF-specific models is maintaining them as, unlike in EMF, in GMF there is no reconciler that can update these models automatically (even for very simple changes) when the Ecore metamodel changes. Therefore, once customized in any way, these models need to be maintained manually.

As a result, implementing a graphical editor with EMF and GMF is a laborious and error prone task, particularly so for the inexperienced developer. Given that implementing a simple graphical editor is typically one of the first steps attempted by most of the newcomers in MDE[2], the risk of giving up or forming a negative impression about the quality of the MDE tool-chain from their interaction with GMF is considerable. Moreover, even for seasoned MDE developers[5], this predominately manual and repetitive process is clearly tedious.

3 EuGENia: Model Transformation to the Rescue

Having strongly criticised some aspects of GMF in the previous section, it is worth stressing again, that despite its weaknesses, GMF still is the most powerful, flexible and widely-used open-source graphical editor framework available

[4] Including loose bits of Java.

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

today and when tuned correctly it can achieve impressive results (the widely used IBM RSA UML modeller, as well as the open-source Topcased and Papyus modelling tools are all implemented atop GMF).

To shield developers from the complexity of GMF and address the highlighted challenges, in this work we adopt a single-sourcing approach in which all the additional information that is necessary for implementing a graphical editor is captured by embedding a number of respective high-level annotations in the Ecore metamodel. We then use automated model-to-model and in-place transformations in order to generate all the platform-specific models required by the EMF and GMF code generators in a consistent and repeatable manner. In this section we demonstrate an implementation of our approach in the context of the EuGENia tool [3] and highlight the benefits in terms of productivity, quality and maintainability that this approach delivers.

3.1 Generating GenModels with EuGENia

The first challenge highlighted in section 2 is to customize the EMF generator model (*GenModel*) produced automatically from an Ecore metamodel, and keep the two synchronized when the Ecore metamodel is subsequently modified with minimal effort. To address this challenge we embed GenModel-specific information in the Ecore model in the form of dedicated annotations. We have also implemented a model-to-model transformation (*Ecore2GenModel*) which can consume the annotated Ecore metamodel and create a GenModel, where beyond the main Ecore elements (classes, features etc.), their annotation values are also transformed into respective GenModel feature values. As an example, in Figure 2 beyond creating the *Simplem2* GenPackage from the *simplem2* EPackage, the value of the *emf.gen basePackage* annotation of the *simplem2* EPackage has also been copied into the *basePackage* attribute of the respective GenPackage[6].

For more complex customizations which require creating or deleting elements in the GenModel, 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 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. This is illustrated in Figure 3 and a concrete example of a polishing transformation is provided in Listing 1.3 of Section 4.

Using the built-in Ecore2GenModel transformation provided by EuGENia, and the user-defined polishing transformation - if one is required for the specific case, the GenModel needs no longer be maintained manually as it can be regenerated at any point from the Ecore metamodel. A screencast that demonstrates the Ecore2GenModel transformation in action is available at:
http://www.eclipse.org/gmt/epsilon/cinema/#eugenia-genmodel.

[6] The basePackage attribute specifies the base package under which all Java code will be generated.

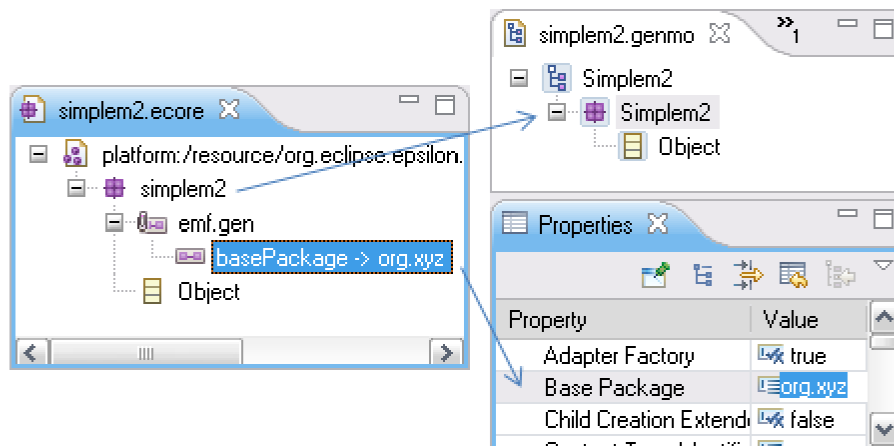

Fig. 2. Exemplar output of the Ecore2GenModel transformation

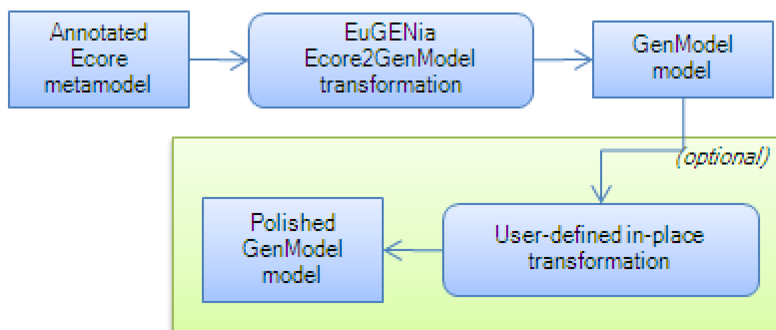

Fig. 3. The EuGENia Ecore2GenModel transformation workflow

3.2 Generating GMF-specific Models with EuGENia

In order 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 annotations and then use a model-to-model transformation (*Ecore2GMF*) in order to generate the 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. Moreover, as illustrated in Figure 4, consistently with the practice followed in the Ecore2GenModel transformation, the developer can contribute additional polishing transformation both for the Ecore2GMF and for the FixGMFGen transformations which can fine-tune the produced final models.

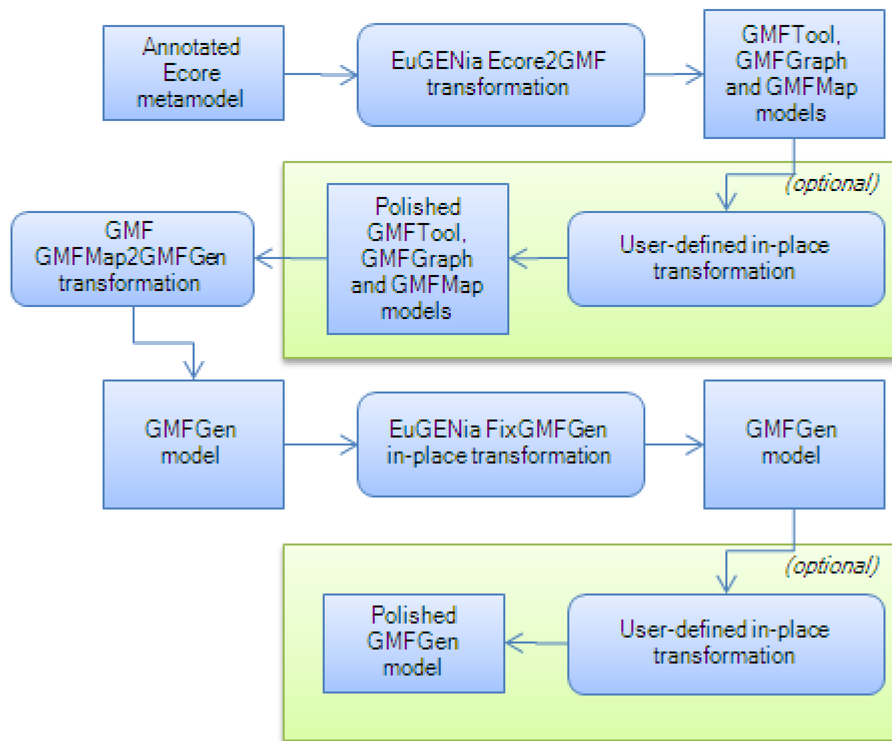

Fig. 4. The EuGENia Ecore2GMF and FixGMFGen transformation workflow

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. Section 4.1 provides a detailed example that demonstrates a substantial subset of the supported annotations and a complete list of all the annotations supported by EuGENia is available in [4]. It is worth stressing that the annotations supported by EuGENia are not a 1-1 mapping with the features of GMF (otherwise it would be just as complex). GMF features that are not covered by the annotations that EuGENia provides (e.g. setting the font of particular types of nodes) can be managed using the polishing transformation mechanism. A screencast that demonstrates the GMF-model generation part of EuGENia is available at
http://www.eclipse.org/gmt/epsilon/cinema/#Eugenia

3.3 Implementation Notes

All the transformations in EuGENia have been implemented using languages of the Epsilon platform [5]. More specifically, the built-in *Ecore2GenModel* trans-

formation has been implemented using the rule-based ETL[6] model-to-model transformation language, while the *Ecore2GMF* and *FixGMFGen* transformations have been implemented using the imperative EOL language[7]. The reason for implementing the Ecore2GMF transformation in an imperative language, as opposed to a rule-based language such as ETL, was its high complexity and the need for low-level control of the execution flow. In terms of size, the Ecore2GenModel transformation is 264 lines long, the Ecore2GMF one contains 1167 lines of code (including operation libraries), and the FixGMFGen one contains 91 lines of code.

It should be possible to implement these transformations using other M2M languages (e.g. ATL, QVT and Kermeta) as long as they satisfy the following requirements:

- They support managing more than one source and target models in the same transformation
- They support in-place as well as model-to-model transformation
- They support establishing and navigating cross-model references
- They support 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). This is particularly desirable in the Ecore2GenModel transformation which will otherwise to contain a big number of explicit annotation copying statements (76 only for EPackages).

4 Evaluation

In this section we demonstrate using EuGENia in a simple but representative graphical editor development case study. We also report on the feedback received by the community and the testing mechanisms we have used to evaluate the correctness of the transformations EuGENia provides. Finally, we identify the limitations of this approach.

4.1 Case Study

In this section we present a case study that demonstrates using EuGENia to implement a graphical editor for a simple component-connector language (which we call SCL for brevity) using EMF and GMF. First we need specify the abstract syntax of SCL using Ecore. As a brief description, an SCL model can contain named components, with each component owning a number of ports as well as other other 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 1.1.

Listing 1.1. The SCL Ecore metamodel in Emfatic

```
1  @namespace(uri="scl", prefix="scl")
2  package scl;
```

```
3
4  class Component {
5    attr String name;
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 }
```

In order to use EuGENia to derive the EMF/GMF-specific models which the respective generators need to generate the concrete artefacts that realize the graphical editor for SCL, we need to annotate the Ecore metamodel accordingly. Listing 1.2 demonstrates the annotated version of the metamodel. In particular, the annotations specify the following:

- Line 2: All generated source code should be put under the *org.eclipse.epsilon.eugenia.examples* Java package
- Line 5: The diagram element represents the top-level Component of the model
- Line 6: Each component is represented in the diagram as a light blue node that contains a label which reflects the value of its *name* attribute
- Line 9: Each Component has a compartment where sub-components can be placed
- Lines 15-16: Each Connector is represented as a link (association) between its *from* and *to* ports. The end that is attached to the *to* port is decorated with an arrow
- Line 23: Each Port is represented as an 15x15 icon-less circle, attached to the border of the component it belongs to (Line 11)
- Line 24: The label of a Port reflects the value of its *name* attribute and is located externally to the circle

Listing 1.2. The annotated SCL Ecore metamodel in Emfatic

```
1 @namespace(uri="scl", prefix="scl")
2 @emf.gen(basePackage="org.eclipse.epsilon.eugenia.examples")
3 package scl;
4
5 @gmf.diagram(foo="bar")
6 @gmf.node(label="name", color="219,238,253")
```

```
7  class Component {
8    attr String name;
9    @gmf.compartment(foo="bar")
10   val Component[*] subcomponents;
11   @gmf.affixed(foo="bar")
12   val Port[*] ports;
13 }
14
15 @gmf.link(source="from", target="to",
16   label="name", target.decoration="arrow")
17 class Connector {
18   attr String name;
19   ref Port#outgoing from;
20   ref Port#incoming to;
21 }
22
23 @gmf.node(figure="ellipse", size="15,15", label.icon="false",
24   label.placement="external", label="name")
25 class Port {
26   attr String name;
27   val Connector#from outgoing;
28   ref Connector#to incoming;
29 }
```

From this annotated metamodel, EuGENia can automatically generate the GMF editor that appears in Figure 5. While the generated editor is fully-functional, and decent in terms of appearance, we wish to further customize it to match our exact requirements (see Figure 6) .

To achieve this, we specify the polishing transformation of Listing 1.3 which performs the required changes to the GMFGraph model and place it in the predefined location (in a file named Ecore2GMF.eol in the same directory as SCL.ecore) so that EuGENia can discover and run it every time after the built-in Ecore2GMF transformation.

Listing 1.3. The polishing in-place transformation in EOL

```
1  // Add bold font to component label
2  var componentLabel = GmfGraph!Label.
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.
10  selectOne(r|r.name="ComponentSubcomponentsCompartmentFigure");
11 var lineBorder = new GmfGraph!LineBorder;
12 lineBorder.width = 1;
13 componentCompartment.backgroundColor =
14  createColor(255,255,255);
```

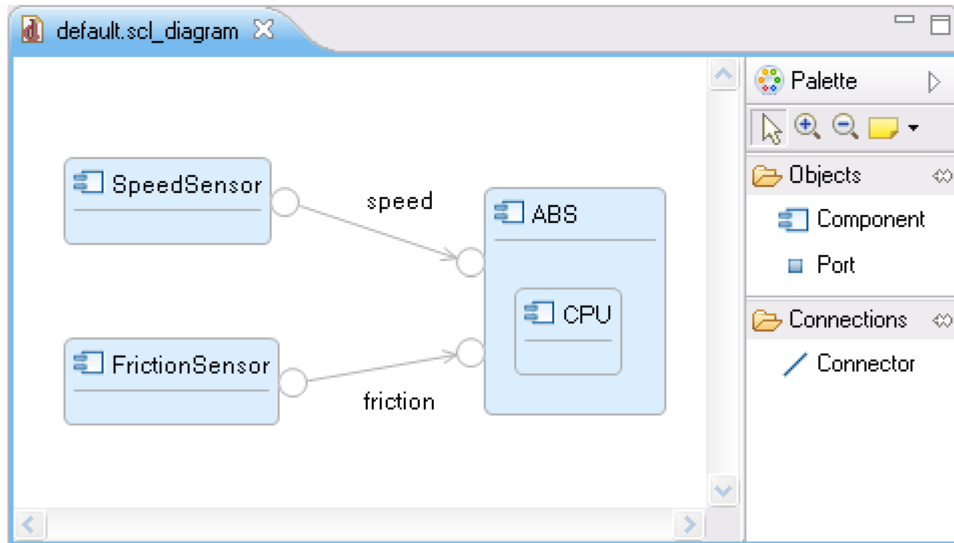

Fig. 5. The first version of the GMF SCL editor

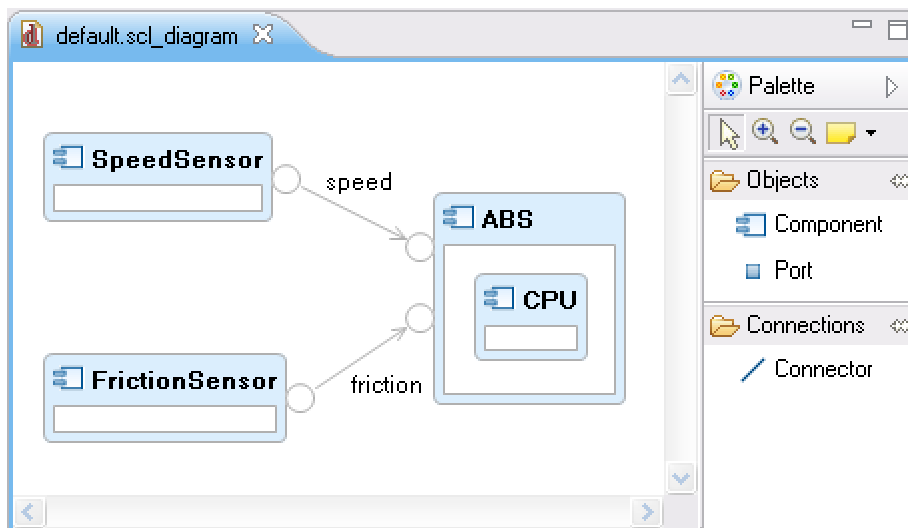

Fig. 6. The polished version of the GMF SCL editor

```
15  componentCompartment.border = lineBorder;
16
17  operation createColor(red : Integer, green : Integer,
18     blue : Integer) : GmfGraph!RGBColor {
19
20   var color = new GmfGraph!RGBColor;
```

```
21   color.red = red;
22   color.blue = blue;
23   color.green = green;
24   return color;
25 }
```

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, 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[7], 5 lines of Emfatic annotations result to 102 elements in the graph, tooling and mapping models.

Polishing transformations may not have similar productivity results in terms of the number of model elements they produce/modify (for example the polishing transformation in Listing 1.3 takes 25 lines of code to create 3 and modify 2 elements), however in our experience the effort spent for construcing them quickly pays off as graphical editor development is a highly iterative process[2].

4.2 Community Feedback

EuGENia is part of the Epsilon component of the Eclipse Modeling GMT project. Since it was first released in August 2008, it has been widely used in the Eclipse modelling community both by researchers (in research institutes such as Fraunhofer FOKUS, SINTEF and in several universities world-wide) and practitioners (in companies such as IBM, Siemens and WesternGeco). Evidence for this exists among the large number of posts in the Epsilon forum[8] which refer to EuGENia. A proposal for a long talk[9] on EuGENia has also been accepted and delivered in the predominately industrially-oriented Eclipse Summit Europe 2009.

4.3 Evaluating Correctness

To evaluate the correctness of the transformations provided by EuGENia and avoid regressions we rely on a growing test set that includes manually constructed input and output models for each transformation, as well as on the feedback of the community (through which several bugs have already been identified and fixed[10]).

[7] http://www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial/
[8] http://www.eclipse.org/gmt/epsilon/forum/
[9] http://www.eclipsecon.org/summiteurope2009/sessions?id=979
[10] http://bit.ly/bMOP6R

Moreover, to test whether the *Ecore2GenModel* transformation preserves the behaviour of the respective EMF built-in transformation it replaces, we executed the two transformations on a common set of 20 Ecore metamodels obtained from the EMFText Syntax Zoo[11]. Initial comparison results indicated that that the produced GenModels were almost identical - with the exception of a small number of non-critical attribute values. As a result, we have updated the Ecore2GenModel transformation accordingly, and the two transformations now produce identical results for our test-set.

4.4 Limitations

There are two main limitations in EuGENia. The first is that by embedding graphical-syntax- and implementation-related annotations into the Ecore metamodel, the metamodel is polluted with information which is irrelevant to its original purpose (abstract syntax definition). Feedback from the users of EuGENia indicates that this is perceived as a fair trade-off for the increased usability that having all information in a single physical file delivers. To address the pollution issue without sacrificing usability, we are experimenting with more modular concrete syntaxes for specifying Ecore metamodels. More specifically, we are investigating the definition of a textual syntax based on Emfatic [8], which allows developers to specify annotations in separate physical files and fuse them at runtime with the core abstract syntax elements using name-based correspondences. Another option that has been suggested by the community is to extract a standalone language out of the annotations provided by EuGENia - which we also consider as a potential direction for the evolution of the tool in the future.

The second limitation of EuGENia is that in order to compose polishing transformations for non-trivial customizations which are not supported by the built-in annotations, developers need to become familiar with both the in-place model transformation language (EOL) and the metamodels of the models they need to customize. Having said this, it is worth mentioning that extensive documentation as well as several concrete examples are publicly available for EOL. Regarding the need to familiarize with the metamodels of the EMF- and GMF-specific models, this can be achieved in an incremental manner having the models produced by EuGENia as a solid base for incremental exploration.

5 Related Work

Similarly to EuGENia, GmfGen [9] 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

[11] http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo

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 GMFs models is required to understand the way in which GmfGens templates are constructed. Instead, EuGENia focuses on abstracting away from the details of GMFs models.

In a broader context, a number of graphical modelling frameworks of similar functionality to GMF are available, most notably MetaEdit+ [10], GME [11], and XMF-Mosaic [12]. A detailed analysis of their features appears in [5]. In this work we have concentrated on GMF only as it is increasingly gaining momentum, mainly due to its open-source nature, its extensive set of features, and the immense success of the underlying EMF framework which is widely accepted as the de-facto for modelling in the Java and Eclipse communities.

An early version of the work presented in this paper was presented in a workshop paper [13]. Since this publication however, EuGENia has been extended significantly based on feedback from the community and additional features such as the Ecore2GenModel transformation, and the support for polishing transformations have been realized.

6 Conclusions and Further Work

In this paper we have presented EuGENia, a tool that employs metamodel annotations as well as 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. EuGENia has been well-received from the Eclipse modelling community and there is strong evidence that it is extensively used both in the industry and the academia.

While EuGENia already greatly improves the usability of GMF and lowers the entrance barrier for inexperienced developers, there is a significant amount of work that still remains to be done, including adding support for sub-diagrams, support for multiple (non hierarchical) diagrams in the same file, and advanced property editing - only to name a few. There are several ongoing efforts in the community which address some of these issues such as the MOSKitt project[12], the EEF project[13]. Our aim is to converge with these efforts and progressively extend EuGENia so that it provides built-in support for all these features in a usable and intuitive manner.

An additional interesting direction for further research is to target alternative graphical editor frameworks such as the upcoming Graphiti framework[14] from SAP, or web-based frameworks such as UMLCanvas[15].

[12] http://www.moskitt.org
[13] http://www.eclipse.org/modeling/emft/?project=eef#eef
[14] http://www.eclipse.org/proposals/graphiti/
[15] http://umlcanvas.org/

References

1. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks. *EMF: Eclipse Modelling Framework*. Eclipse Series. Addison-Wesley Professional, 2 edition, December 2008.
2. 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.
3. Epsilon Eclipse GMT Component. EuGENia. http://www.eclipse.org/gmt/epsilon/doc/eugenia.
4. Epsilon Eclipse GMT Component. EuGENia GMF Tutorial. http://www.eclipse.org/gmt/epsilon/doc/articles/eugenia-gmf-tutorial/.
5. Eclipse Foundation. Epsilon Modeling GMT component. http://www.eclipse.org/gmt/epsilon.
6. 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.
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. IBM alphaWorks. Emfatic Language for EMF Development, February 2005. http://www.alphaworks.ibm.com/tech/emfatic.
9. 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.
10. MetaCase. Meta-Edit+. http://www.metacase.com.
11. Generic Modeling Environment. http://www.isis.vanderbilt.edu/Projects/gme.
12. Xactium. XMF-Mosaic. http://www.xactium.com.
13. Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige, Fiona A.C. Polack. Raising the Level of Abstraction in the Development of GMF-based Graphical Model Editors. In *Proc. 3rd Workshop on Modeling in Software Engineering (MISE), ACM/IEEE International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009.