Towards Automatic Generation of UML Profile Graphical Editors for Papyrus

Athanasios Zolotas[1], Ran Wei[1], Simos Gerasimou[1]
Horacio Hoyos Rodriguez[1], Dimitrios S. Kolovos[1] and Richard F. Paige[1]

Department of Computer Science, University of York, York, United Kingdom
{thanos.zolotas, ran.wei, simos.gerasimou, dimitris.kolovos,
richard.paige}@york.ac.uk, horacio_hoyos_rodriguez@ieee.org

Abstract. We present an approach for defining the abstract and concrete syntax of UML profiles and their equivalent Papyrus graphical editors using annotated Ecore metamodels, driven by automated model-to-model and model-to-text transformations. We compare our approach against manual UML profile specification and implementation using Archimate, a non-trivial enterprise modelling language, and we demonstrate the substantial productivity and maintainability benefits it delivers.

1 Introduction

The Unified Modeling Language (UML) [10] is the *de facto* standard for software and systems modelling. Since version 2.0, UML has offered a domain-specific extensibility mechanism, *Profiles* [5], which allows users to add new concepts to the modelling language in the form of *Stereotypes*. Each stereotype extends a core UML concept and includes extra information that is missing from that base concept. With profiles, UML offers a way for users to build domain-specific modelling languages (DSML) by re-using and extending UML concepts, thus lowering the entry barrier to DSML engineering by building on engineer familiarity with UML and UML tools (a detailed comparison of using UML profiles versus domain-specific modelling technology such as [2, 17] is beyond the scope of this paper).

Papyrus [16] is a leading open-source UML modelling tool and after a decade in development, it is developing a critical mass for wider adoption in industry as means of (1) escaping proprietary UML tooling lock-in, (2) leveraging the MBSE-related developments in the Eclipse modelling ecosystem enabling automated management of UML models, and (3) enabling multi-paradigm modelling using a combination of UML and EMF-based DSLs. Papyrus offers support for the development of UML profiles; however, this is a manual, tedious and error-prone process [23], and as such it makes the development of graphical editors that are based on such profiles difficult and expensive.

In this paper, we automate the process of developing UML profiles and graphical editors for Papyrus. We propose an approach, called AMIGO, supported by a prototype Eclipse plugin, where annotated Ecore metamodels are used to generate fully-fledged UML profiles and distributable Papyrus graphical editors.

We evaluate the effectiveness of our approach for the automatic generation of a non-trivial enterprise modelling language (Archimate). Furthermore, we apply our approach on several other DSMLs of varying size and complexity [22], demonstrating its generality and applicability.

2 Background and Motivation

In this section we outline the process for defining a UML profile and supporting model editing facilities in Papyrus. We highlight labour-intensive and error prone activities that motivate the need of automatic generation of those artefacts.

2.1 UML Profile

In order to create a new UML Profile, developers need to create a new UML model and add new elements of type *Stereotype, Property, Association, etc.* to create the desired stereotypes, their properties and their relationships. Papyrus offers, among other choices, that of creating the UML profile via a *Profile Diagram*. Users can drag-and-drop elements from the palette to construct the profile. The properties of each element (e.g., multiplicity, navigability, etc.) can be then set using the properties view. In a profile, each stereotype needs to extend a UML concept (hereby referred to as *base element* or *meta-element*). Thus, users also need to import these meta-elements and add the appropriate extension links with the stereotypes. The process of creating the profile can be repetitive and labour-intensive, and depends on the size of the profile.

One of the limitations of UML Profiles in Papyrus is that links between stereotypes can be displayed as *edges* only if they extend a *Connector* meta-element. These connector stereotypes do not hold any information about the stereotypes that they can connect. Users need to define OCL constraints to validate if source and target nodes are of the desired type and if the navigation of the edges is in the correct direction. These constraints can be rather long and need to be manually written and customised for *each* edge stereotype. This can also be a labour-intensive and error-prone process.

2.2 Distributable Custom Graphical Editor

At this point, the created profile can be applied on UML diagrams. Users select a UML element (e.g., Class) and manually apply the stereotype. A stereotype can only be applied to the UML element that was defined as its base element. This task might be problematic as users need to remember the base elements for each domain-specific stereotype. To address this recurring concern, Papyrus offers at least three possible options for creating a custom palette which allows users to create base UML elements and apply selected stereotypes on them in a single step. The first option involves customisation through a user interface which has to be done manually everytime a new diagram is created and it is not distributable. The other two options require the manual definition of palette

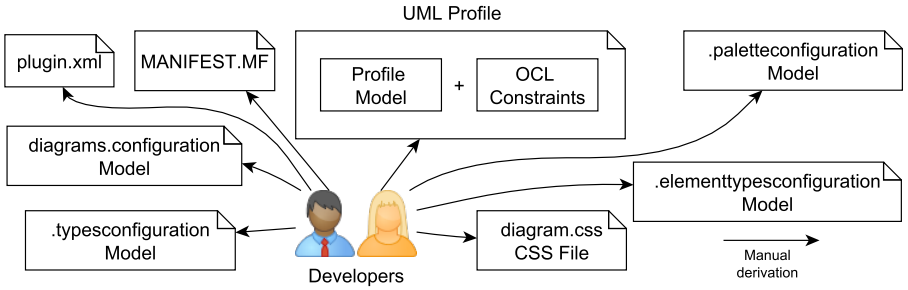

Fig. 1: All the artefacts users need to write to develop a distributable editor.

configuration files that are automatically loaded every time the profile is applied on a diagram. Although the first is simpler and requires the definition of a single XML file, it is not encouraged as it is based on a deprecated framework and does not allow the full use of Papyrus functionality.

The definition of custom shapes for the instantiated stereotypes is another common requirement. Scalable Vector Graphics (SVG) shapes can be bound to stereotypes at the profile creation process. However, to make these shapes visible, users must set the visibility of the shape of *each* element to true. Even if this is an acceptable trade-off, another drawback is that by default the shape overlaps with the default shape of the base meta-element. Users can hide the default shapes by writing CSS rules. The rules can be written once but need to be loaded each time *manually* on every diagram that is created.

To create a distributable graphical editor that has diagrams tailored for the profile and to avoid all the aforementioned drawbacks, users need to create several models and files. Figure 1 shows all the files needed to be created *manually*. In this work, we propose an approach that uses a single-source input to automate this *labour-intensive*, *repetitive* and *error-prone* process. This work is motivated by the increasing interest among our industrial partners on exploring the viability of Papyrus as a long-term open-source replacement for proprietary UML tools. While Papyrus provides comprehensive profile support, the technical complexity if high due to the multitude of interconnected artefacts required (see Figure 1), which can be a significant hurdle for newcomers. We aim to lower the entry barrier for new adopters and help them achieve a working (but somewhat restricted) solution with minimal effort.

3 Proposed Approach

We propose AMIGO, an automatic approach in which information like stereotypes that should be instantiated in the profile, structural (e.g., references) and graphical information (e.g., shapes) are captured as high-level annotations in an Ecore metamodel, and is transformed into Papyrus-specific artefacts using automated M2M and M2T transformations. Figure 2 shows an overview of AMIGO.

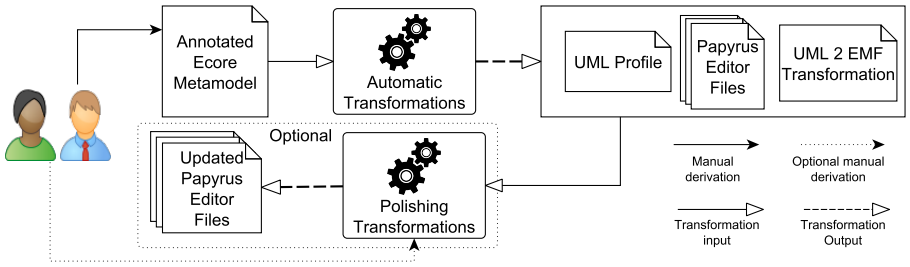

Fig. 2: An overview of the proposed approach

All EClasses in the Ecore metamodel are automatically transformed into stereotypes. Annotated EReferences can also create stereotypes. Developers use the annotations listed below to specify the required graphical syntax of the stereotype (i.e., if it should be represented as a *node* or as an *edge* on the diagram). A detailed list of all valid annotation properties is given in the Appendix.

(1) **@Diagram** annotations define diagram-specific information like the name and the icon of the diagram type. This annotation is always placed at the top package of the Ecore metamodel.
(2) **@Node** annotations are used for stereotypes that should be instantiated as nodes in the diagrams. The UML meta-element that this stereotype extends is provided through the *base* property, while the SVG shape and the icon in the palette are specified through the *shape* and *icon* properties, respectively.
(3) **@Edge** annotations are used for stereotypes that should be instantiated as edges and it can be applied to both EClasses and EReferences. The base UML element is provided through the *base* property. The icon in the palette is also passed as property along with the desired style of the line.

The annotation of the ECore metamodel is the only manual process required in our approach. The annotated Ecore metamodels are then consumed by M2M and M2T transformations shown in Figure 4 and described in detail in Section 4. The transformations are written in the Epsilon Transformation Language (ETL) [13] and the Epsilon Generation Language (EGL) [18] but in principle, any other M2M and M2T language could be used. The automated workflow of transformations produces the UML profile with the appropriate OCL constraints and all the configuration models/files needed by Papyrus. In addition, an M2M transformation, that can be used to transform the UML models back to EMF models that conform to the original Ecore metamodel, is also generated. Thus, model management programs already developed to run against models conforming to the EMF metamodel can be re-used.

AMIGO provides the option to execute *polishing transformations* that allow fine-tuning of the generated artefacts. In the following section, the proposed approach is explained via a running example.

```
1  @namespace(uri="sdpl",prefix="sdpl")
2  @Diagram(name="SDPL", icon="sdpl.png")
3  package Process;
4
5  @Node(base="Class", shape="step.svg",
       icon="step.png")
6  class Step {
7    attr String stepId;
8    ref Step[1] next;
9  }
10 @Node(base="Class", shape="tool.svg",
       icon="tool.png")
11 class Tool {
12   attr String name;
13 }
14 @Node(base="Class", shape="per.svg", icon
       ="per.png")
15 class Person {
16   attr String name;
17   @Edge(base="Association", icon="line.
         png")
18   ref Tool[*] familiarWith;
19 }
20 @Edge(base="Association", icon="line.png"
       , source="src", target="tar")
21 class Role {
22   attr String name;
23   ref Step[1] src;
24   ref Person[1] tar;
25 }
```

Listing 1.1: The annotated ECore metamodel of SDPL.

3.1 Running Example

We use AMIGO to define and generate the UML profile and the Papyrus graphical editor for a Simple Development Processes Language (SDPL). We start by defining ECore metamodel using Emfatic (see Listing 1.1). A process defined in SDPL consists of *Steps*, *Tools* and *Persons*. Each person is familiar with certain tools and has different *Role*s, while each step refers to the next step using the *next* reference. To generate the UML profile and the Papyrus graphical editor, we add the following concrete syntax-related annotations shown in Listing 1.1. The produced by AMIGO SDPL Papyrus editor is presented in Figure 3.

- **Line 2:** The name and the icon that should be used in Papyrus menus are defined using the *name* and *icon* properties of the *@Diagram* annotation.
- **Lines 5, 10 & 14:** The *@Node* annotation is used to define that these types should be stereotypes that will be represented as nodes on the diagram. The *base* parameter defines the UML meta-element the stereotype should extend. The shape and the palette icon are given using the *shape* and *icon* details.
- **Line 17 & 20:** The *familiarWith* EReference and the *Role* EClass are extending the meta-element *Association* of UML. These stereotypes should be shown as links in the diagrams. In contrast with the *familiarWith EReference*, the types the *Roles* edge should be able to connect are not known and need to be specified as properties of the annotation (i.e., *source="src"* and *target="tar"*). This denotes that the source/target nodes of this connector are mapped to the values of the *src*/*tar* EReferences, respectively.
- **NB Line 8:** The *next* EReference is not required to be displayed as an edge on the diagram thus it is not annotated with *@Edge*.

3.2 Polishing Transformations

The generated editor is fully functional but it can be further customised to fit custom user needs. In this example the labels should be in red font. This can be achieved by manually amending the generated CSS file. However, the CSS file will be automatically overridden if the user regenerates the editor. To avoid

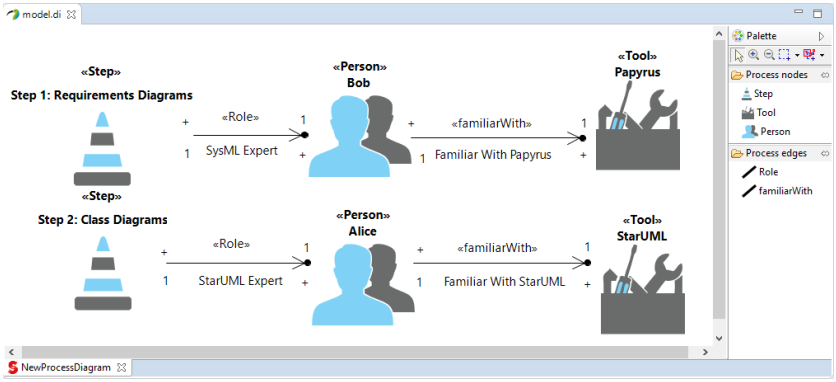

Fig. 3: The SDPL editor for Papyrus generated using AMIGO.

this, the user can use CSS polishing transformation (#6b in Figure 4) shown in Listing 1.2. Every time the profile and editor generation is executed, the polishing transformation will amend the original CSS file with the information shown in Listing 1.3.

```
var allNodeStereotypes = Source!EClass.all().select(c|c.getEAnnotation("Node").isDefined());
for (stereo in allNodeStereotypes) {%}
[appliedStereotypes~=[%=stereo.name%]][% if (hasMore){%} , [%}}
%] {
  fontColor:red;
}
```

Listing 1.2: A CSS polishing transformation.

```
[appliedStereotypes~=Step],[appliedStereotypes~=Tool],[appliedStereotypes~=Person]{
  fontColor:red;
}
```

Listing 1.3: The output that is amended in the original CSS file.

4 Implementation

This section discusses the implementation of the proposed approach. Figure 4 shows the transformations workflow. As the transformations consist of about 1K lines of code, we will describe them omitting low level technical details[1]. Every step in the process, except that of polishing transformations described in Section 4.8 is fully automated, as the only required manual step in AMIGO, is that of annotating the ECore metamodel.

4.1 EMF to UML Profile Generation (#1)

This transformation consists of two rules: the first creates one stereotype for each EClass in the metamodel and the second creates a stereotype for EReferences annotated as @Edge. The source model of this transformation is the annotated Ecore metamodel and the target model is a UML profile model.

[1] The code and instructions are available at `http://www.zolotas.net/AMIGO`

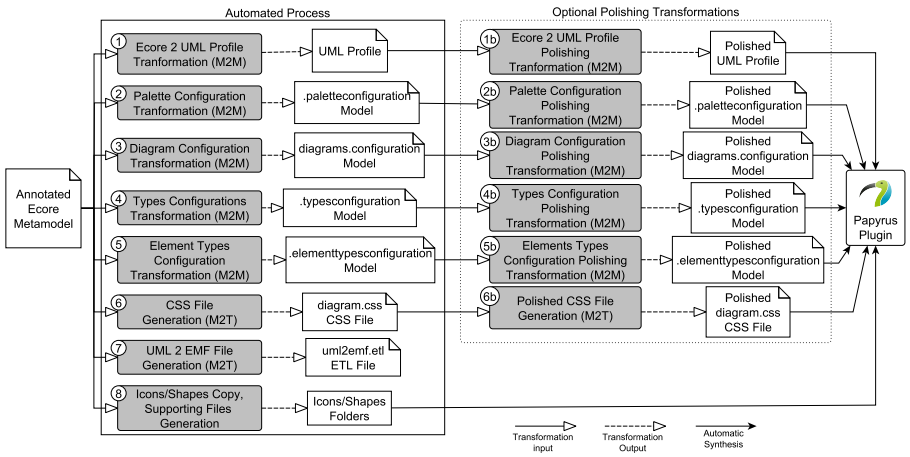

Fig. 4: An overview of the transformation workflow.

When all stereotypes are created, a number of post-transformation operations are executed to (1) create the generalisation relationships between the stereotypes, (2) add the references/containment relationships between the stereotypes, (3) create the extension with the UML base meta-element and (4) generate and add the needed OCL constraints for each edge:

(1) For each of the superclasses of an EClass in the metamodel we create a *Generalisation* UML element. The generalisation element is added to the stereotype created for this specific EClass and refers via the *generalization* reference to the stereotype that was created for the superclass.
(2) For each reference (ref or val) in the metamodel a new *Property* UML element is created and added to the stereotype that represents the EClass. A new *Association* UML element should also be created and added to the stereotype. The name and the multiplicities are also set.
(3) By default the stereotypes extend the *Class* base element unless a different value is passed in the *base* property of the @Node/@Edge annotation. In this post-transformation operation the necessary *Import Metaclass* element and *Extension* reference are created and attached to the stereotype.
(4) In the last operation, the OCL constraints are created for each stereotype that will be represented as an edge on the diagram. Two *Constraint* and two *OpaqueExpression* elements are created for each edge stereotype that check the two necessary constraints.

4.2 Constraints

To illustrate the OCL constraints, we provide a partial view of the SDPL UML profile in Figure 5[2].

[2] The attributes of the stereotypes are omitted for simplicity.

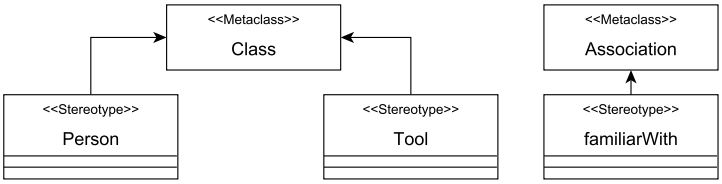

Fig. 5: Example UML profile for SDPL.

In Figure 3, the *familiarWith* association is used to connect *Person Alice* with *Tool StarUML*. However, the *familiarWith* stereotype can be applied to any *Association*, and not strictly to *Associations* which connect *Person* and *Tool* stereotyped elements. Therefore, constraints are needed to check two aspects:

- **End Types**: the elements that a *familiarWith* association connects have *Person* and *Tool* stereotypes applied;
- **Navigability**: the *familiarWith* association *starts* from an element stereotyped as *Person* and *points* to an element stereotyped as *Tool*.

End Types In listing 1.4, line 1 accesses the types that *familiarWith* connects. Lines 2 and 3 check if the types that *familiarWith* connects are of type that either has stereotype *Person* or *Tool*.

```
1 let classes = self.base_Association.endType→selectByKind(UML::Class) in
2   classes→exists(c|c.extension_Person→notEmpty()) and
3     classes→exists(c|c.extension_Tool→notEmpty())
```

Listing 1.4: The End Types constraint in OCL.

Navigability In Listing 1.5, in lines 2 and 3, we obtain the member ends that *familiarWith* connects with. If these ends are obtained successfully (line 4), we check that the *personEnd* (connecting element stereotyped as *Person*) is not navigable (line 5) and the *toolEnd* (connecting element stereotyped as *Tool*) is navigable (line 6). Therefore, we are checking that a *familiarWith* association can only go from *Person* to *Tool*.

```
1 let memberEnds=self.base_Association.memberEnd in
2 let toolEnd=memberEnds→select(type.oclIsKindOf(UML::Class) and type.oclAsType(UML::
      Class).extension_Tool→notEmpty()),
3 personEnd=memberEnds→select(type.oclIsKindOf(UML::Class) and type.oclAsType(UML::
      Class).extension_Person→notEmpty()) in
4 if personEnd→notEmpty() and toolEnd→notEmpty() then
5 personEnd→first().isNavigable() = false and
6 toolEnd→first().isNavigable() = true
7 else false endif
```

Listing 1.5: The Navigability constraint in OCL.

We use the *End Types* and *Navigability* constraints as templates with dynamic sections, where the specific stereotype names are inserted dynamically.

4.3 Palette Generation (#2)

This transformation is responsible for creating a model (file .paletteconfiguration) that configures the custom palette for the diagram. The model conforms to

the *PaletteConfiguration* metamodel that ships with Papyrus. The transformation creates a new *PaletteConfiguration* element and adds two new *DrawerConfiguration* elements that represent the two different tool compartments in our palette (i.e., nodes and edges). For each element in the Ecore source annotated as @Node/@Edge, a new *ToolConfiguration* element is created and added to the nodes/edges drawer respectively. An *IconDescriptor* element is also added to the ToolConfiguration pointing to the path of the icon for that tool.

4.4 Diagram Configuration (#3, #4 & #5)

Firstly, in order for Papyrus to offer the choice of creating new custom diagrams for the generated profile via its menus, a *Viewpoint Configuration* needs to be created. This configuration hides the default palette and attaches the custom one created before. It is also responsible for binding the generated CSS stylesheet file (see transformation #6) to the diagram. Transformation #3 creates a new model that conforms to the *Configuration* metamodel that ships with Papyrus and stores this new *Viewpoint Configuration* element.

The second artefact that needs to be created is the types configuration model (i.e., .typesconfiguration file) that conforms to the *Element Types Configuration* metamodel provided by Papyrus. This is achieved through transformation #4. This model is responsible for binding the types of the drawn elements to stereotypes. For each stereotype a new *Specialization Type Configuration* element is created and a unique *id* is created in the format "ProfileName.StereotypeName" (e.g., "SDPL.Step"). The value of the *Hint* attribute is set to the qualified name of the meta-element that this type specialises (e.g., "UML::Class"). A new *Stereotype Application Matcher Configuration* element is also created that holds the qualified name of the stereotype that should be applied to the drawn element. Binding is performed by creating a new *Apply Stereotype Advice Configuration* element that points to the equivalent stereotype application matcher configuration element created before. Having this file, when an element of a specific type is drawn the appropriate stereotype is applied automatically.

The last model (i.e., the .elementtypesconfiguration file) is one that conforms to the *Element Types Configuration* metamodel. This is done by transformation #5. As all the stereotypes created extend a UML meta-element, this model is responsible for specializing the meta-element *shapes* to the custom ones created by the profile. Thus, for each stereotype, a new *Specialization Type Configuration* element is created. This element points to the two elements that it specializes: the specialization type configuration created in transformation #4 via its id (e.g., "Process.Step") and the shape of the UML meta-element that this element specializes via its URI (e.g., "org.eclipse.papyrus.umldi.Class_Shape").

4.5 Stylesheet Generation (#6)

In Papyrus, the look and feel of diagram elements can be customised using CSS. Each node on a diagram has a set of compartments where the attributes, the shape, etc. appear. Initially, we create a CSS rule to hide all their compartments and another rule to enable the compartment that holds the shape. The latter

rule also hides the default shape inherited from the meta-element the stereotype extends. Then, for each stereotype that appears as a node, a CSS rule is generated to place the SVG figure in the shape compartment. The URI of the SVG file is passed in the *svgFile* property available in CSS. Finally, we generate the CSS rules for each edge, e.g., if a *lineStyle* parameter is set, then the *style* property for that Edge stereotype is set to the value of the lineStyle parameter (e.g., "solid").

4.6 UML to EMF Transformation Generation (#7)

This M2T transformation generates the ETL file that can be used to transform the UML models created in Papyrus and conform to the UML Profile, back to EMF models that conform to the source Ecore metamodel. One rule is generated for each of the stereotypes that transforms the elements having this stereotype applied to them back to the appropriate type of the Ecore metamodel. Each stereotype has the same attributes and references as the original EClass thus, this EGL script also generates the statements in each rule that populate the attributes and the references. An example of an auto-generated rule is shown in Listing 1.6. This rule transforms elements stereotyped as "Person" in the UML model to elements of type "Person" in an EMF model which conforms to the Ecore metamodel presented in Listing 1.1.

```
1 rule PersonUML2PersonEMF
2   transform s: UMLProcess!Person
3   to t: EMFProcess!Person {
4     t.name = s.name;
5     t.age = s.age;
6     t.familiarWith ::= s.familiarWith;
7   }
```

Listing 1.6: Example of an auto-generated ETL rule.

4.7 Icons, Shapes and Supporting Files (#8)

The approach creates a Papyrus plugin, thus the "MANIFEST.MF", the "plugin.xml" and the "build.properties" files are created. The first includes the required bundles while defines the necessary extensions for Papyrus to register the UML profile and create the diagrams. The third points the project to the locations of the "MANIFEST.MF" and "plugin.xml" files. Finally, two files necessary for Papyrus to construct the UML profile model, (namely "model.profile.di" and "model.profile.notation") are generated.

4.8 Polishing Transformations (#1b - #6b)

For each of the transformations #1 - #6, users are able to define polishing transformations that complement those included in our implementation. After each built-in transformation is executed, the workflow looks to find a transformation with the same file name which is executed against the Ecore metamodel and updates the already created output model of the original transformation.

5 Evaluation

AMIGO is evaluated by firstly, applying it to generate a Papyrus editor for the non-trivial Archimate UML profile [11,12]. The Adocus Archimate for Papyrus[3] is an open-source tool that includes a profile for Archimate and the appropriate editors for Papyrus. We can compare the proportion of the tool that AMIGO is able to generate automatically, the number of polishing transformations that the user needs to write to complete the missing parts and finally, identify the aspects of the editor that our approach is not able to generate. As a result we can measure the *efficiency* of AMIGO in generating profiles/editors against an existing relatively large profile/editor.

Secondly, we assess the *completeness* of our approach by applying it on five other metamodels collected as part of the work presented in [22]. This way, the approach is tested to check if it can successfully generate profiles and editors for a wide variety of scenarios.

5.1 Efficiency

The Archimate for Papyrus tool offers five kind of diagrams (i.e., Application, Business, Implementation and Migration, Motivation and Technology diagrams). Each of the diagrams uses different stereotypes from the Archimate profile. Thus, in this scenario we need to create the 5 Ecore metamodels and annotate those EClasses/EReferences that need to appear as nodes or edges on the diagrams to generate the profiles and the editor.

AMIGO successfully generated the Papyrus editor for Archimate, however, some special features that the Archimate for Papyrus tool offers need further work. For example, the tool offers a third drawer in the palette for some diagrams that is called "Common" and includes two tools (i.e., "Grouping" and "Comment"). In order to be able to implement such missing features, we need to write the extra polishing transformations. For brevity, we will not go into details on the content of the polishing transformations for this specific example.

Table 1, summarises the lines of code we had to write to generate the editors using AMIGO versus the lines of code (LOC) the authors of the Archimate for Papyrus had to write. Since all the artefacts except the CSS file are models, we provide in parenthesis the number of model elements users have to instantiate. For the polishing transformations of our approach we only provide the LOC metric as the models are instantiated automatically by executing the transformation scripts and not manually. Our approach requires about 91% less handwritten LOC to produce the basic diagrams and about 86% less code to produce the polished editor that matches the original Archimate for Papyrus editor. In terms of model elements, we need to manually instantiate about 63% less model elements (668 vs. 1828) for the basic editor. Our approach offers an editor that matches the original Archimate for Papyrus tool but also atop that the ETL transformation and the OCL constraints.

[3] `https://github.com/Adocus/ArchiMate-for-Papyrus`

Table 1: Lines of manually written code (and model elements is parenthesis) of each file for creating a Papyrus UML profile and editor for ArchiMate.

	AMIGO			**Archimate for Papyrus**
File	**Handwritten**	**Handwritten (Polishing)**	**Total**	**Total Handwritten**
ECore	436 (668)	0	436 (668)	0
Profile	0	0	0	1867 (1089)
Palette Configuration	0	24	24	1305 (323)
Element Types Configuration	0	11	11	237 (61)
Types Configuration	0	10	10	788 (327)
Diagram Configuration	0	0	0	58 (28)
CSS	0	195	195	537
Total	**436 (668)**	**240**	**676 (668)**	**4792 (1828)**

Table 2: The metamodels used to evaluate the completeness of AMIGO.

Name	**#Types (#Nodes/#Edges)**	**Name**	**#Types (#Nodes/#Edges)**
Professor	5 (4/5)	**Ant Scripts**	11 (6/4)
Zoo	8 (6/4)	**Cobol**	13 (12/14)
Usecase	9 (4/4)	**Wordpress**	20 (19/18)
Conference	9 (7/6)	**BibTeX**	21 (16/2)
Bugzilla	9 (7/6)	**Archimate**	57 (44/11)

5.2 Completeness

In addition, we tested the proposed approach with nine more Ecore metamodels from different domains. The names and their size (in terms of types) are given in Table 2. Next to the size, in parenthesis, the number of types that should be transformed so they can be instantiated as nodes/edges is also provided. The approach was able to produce the profiles and the editors for *all* the metamodels, demonstrating that it can be used to generate the desired artifacts for a wide spectrum of domains.

5.3 Threats to Validity

There were a few minor features of the original Archimate for Papyrus tool that our approach could not support. Most of them are related to custom menu entries and wizards. For those to be created developer needs to extend the "plugin.xml" file. In addition, the line decoration shapes of stereotypes that extend the aggregation base element (i.e., diamond) can only be applied dynamically by

running Java code that will update the property each time the stereotype is applied. Our default and polishing transformations are not able to generate those features automatically; these should be implemented manually. For that reason, we *excluded* these lines of code needed by Archimate for Papyrus to implement these features from the data provided in Table 1 to have a fair comparison.

6 Related Work

Over the past years, several UML profiles have been standardised by the OMG (e.g., MARTE [9], SySML [4]) and are now included in most common UML tools (e.g., Papyrus [16]). A list of recently published UML profiles is available in [17]. Irrespective of the way these UML profiles were developed, either following ad-hoc processes or based on guidelines for designing well-structured UML profiles [5,19], they required substantial designer effort. Our approach, subject to the concerns raised in Section 5, automates the process of generating such profiles and reduces significantly the designer-driven effort for specifying, designing and validating UML Papyrus profiles and editors.

Relevant to our work is research introducing methodologies for the automatic generation of UML profiles from an Ecore-based metamodel. The work in [15] proposes a partially automated approach for generating UML profiles using a set of specific design patterns. However, this approach requires the manual definition of an initial UML profile skeleton, which is typically a tedious and error-prone task [23]. The methodology introduced in [7,8] facilitates the derivation of a UML profile using a DSML as input. The methodology requires the manual definition of an intermediate metamodel that captures the abstract syntax. Despite the potential of these approaches, they usually involve non-trivial human-driven tasks, e.g., a UML profile skeleton [15] or an intermediate metamodel [7,8]. In contrast, our approach builds on top of standard Ecore metamodels (which are usually available in MBSE). Furthermore, our approach supports the customisation of UML profiles and the corresponding Papyrus editor.

Our work also subsumes research that focuses on bridging the gap between MOF-based metamodels (e.g., Ecore) and UML profiles. In [1], the authors propose a methodology that consumes a UML profile and its corresponding Ecore metamodel, and uses M2M transformation and model weaving to transform UML models to Ecore models, and vice versa. The methodology proposed in [23] simplifies the specification of mappings between a profile and its corresponding Ecore metamodel using a dedicated bridging language. Along the same path, the approach in [6] employs an integration metamodel to facilitate the interchange of modelling information between Ecore-based models and UML models. Compared to this research, AMIGO automatically generates UML profiles (like [23] and [6]), but requires only a single annotated Ecore metamodel and does not need any mediator languages [23] or integration metamodels [6]. Also, the transformation of models from UML profiles to Ecore is only a small part of our generic approach (Section 4.6) that generates not only a fully-fledged UML profile but also a distributable custom graphical editor.

In terms of graphical modelling, our approach is related to EuGENia [14], which transforms annotated Ecore metamodels to GMF (Graphical Modelling Framework) models (i.e. GMF graph definition model, tooling model, mapping model and generation model) to automatically generate graphical model editors. Sirius [20], a tool based also on GMF, enables users to define a diagram definition model and use this model to generate at a graphical editor. Unlike EuGENia, Sirius does not require additions the original Ecore metamodel.

7 Conclusions and Future Work

In this paper we presented AMIGO, an MDE-based approach that uses annotated Ecore metamodels to automatically generate UML profiles and supporting distributable Papyrus editors. We evaluated AMIGO using Adocus Archimate for Papyrus and five other metamodels from [21] showing that AMIGO reduces significantly the effort required to develop these artifacts. Our future plans for AMIGO involve providing better support for compartments since although in the current version users can create compartments using the available compartment relationships in UML (e.g., Package-Class, etc.), the visual result is not appealing. More specifically, the compartment where containing elements are placed is distinct and lies above the compartment that hosts the shape. As a result, the contained elements are drawn above the custom shape and not inside it. Also, we will extend AMIGO with support for the automatic generation of OCL constraints for opposite references and more connectors. We also plan to support the execution of validation scripts against the Ecore file to check that the annotation provided in the Ecore file are correct, and if not, to produce meaningful error messages. Finally, we plan to execute usability tests with real users to evaluate AMIGO against the native Papyrus approach.

Acknowledgments. This work was partially supported by Innovate UK and the UK aerospace industry through the SECT-AIR project, by the EU through the DEIS project (#732242) and by the Defence Science and Technology Laboratory through the project "Technical Obsolescence Management Strategies for Safety-Related Software for Airborne Systems".

References

1. Abouzahra, A., Bézivin, J., Del Fabro, M.D., Jouault, F.: A practical approach to bridging domain specific languages with UML profiles. In: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA. vol. 5 (2005)
2. Bergmayr, A., Grossniklaus, M., Wimmer, M., Kappel, G.: JUMP—From Java Annotations to UML Profiles, pp. 552–568 (2014)
3. Erickson, J., Siau, K.: Theoretical and practical complexity of modeling methods. Communications of the ACM 50(8), 46–51 (2007)
4. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to SysML: the systems modeling language (2014)
5. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to UML profiles. UML and Model Engineering 2 (2004)
6. Giachetti, G., Marin, B., Pastor, O.: Using uml profiles to interchange dsml and uml models. In: Third International Conference on Research Challenges in Information Science. pp. 385–394 (2009)
7. Giachetti, G., Marín, B., Pastor, O.: Using UML as a Domain-Specific Modeling Language: A Proposal for Automatic Generation of UML Profiles, pp. 110–124 (2009)
8. Giachetti, G., Valverde, F., Pastor, O.: Improving Automatic UML2 Profile Generation for MDA Industrial Development, pp. 113–122 (2008)
9. Object Management Group: Modeling And Analysis Of Real-Time Embedded Systems. ONLINE (2011), http://www.omg.org/spec/MARTE/1.1/
10. Object Management Group: Unified Modeling Language. `http://www.omg.org/spec/UML/` (June 2015)
11. Haren, V.: Archimate 2.0 specification (2012)
12. Iacob, M.E., Jonkers, H., Lankhorst, M.M., Proper, H.A.: ArchiMate 1.0 Specification. Zaltbommel: Van Haren Publishing (2009)
13. Kolovos, D., Paige, R., Polack, F.: The Epsilon Transformation Language. Theory and Practice of Model Transformations pp. 46–60 (2008)
14. Kolovos, D.S., García-Domínguez, A., Rose, L.M., Paige, R.F.: Eugenia: towards disciplined and automated development of gmf-based graphical model editors. Software & Systems Modeling pp. 1–27 (2015)
15. Lagarde, F., Espinoza, H., Terrier, F., André, C., Gérard, S.: Leveraging Patterns on Domain Models to Improve UML Profile Definition, pp. 116–130 (2008)
16. Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F.: Papyrus uml: an open source toolset for mda. In: Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09). pp. 1–4 (2009)
17. Pardillo, J.: A Systematic Review on the Definition of UML Profiles, pp. 407–422 (2010)
18. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: The Epsilon Generation Language. In: Model Driven Architecture–Foundations and Applications. pp. 1–16. Springer (2008)
19. Selic, B.: A systematic approach to domain-specific language design using uml. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07). pp. 2–9 (2007)
20. Viyović, V., Maksimović, M., Perisić, B.: Sirius: A rapid development of dsm graphical editor. In: IEEE 18th International Conference on Intelligent Engineering Systems INES 2014. pp. 233–238. IEEE (2014)

21. Williams, J.R.: A Novel Representation for Search-Based Model-Driven Engineering. Ph.D. thesis, University of York (2013)
22. Williams, J.R., Zolotas, A., Matragkas, N.D., Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: What do metamodels really look like? EESSMOD@ MoDELS 1078, 55–60 (2013)
23. Wimmer, M.: A semi-automatic approach for bridging dsmls with uml. International Journal of Web Information Systems 5(3), 372–404 (2009)

A Annotations and Parameters

The following are all the currently supported parameters for the annotations.

A.1 @Diagram

- name: The name of the created diagrams as it appears on the diagram creation menus of Papyrus. [required]
- icon: The icon that will appear next to the name on the diagram creation menus of Papyrus. [optional]

A.2 @Node

- base: The name of the UML meta-element that this stereotype should extend. [required]
- shape: The shape that should be used to represent the node on the diagram. [required]
- icon: The icon that will appear next to the name of the stereotype in the custom palette. [optional]

A.3 @Edge

- base: The name of the UML meta-element that this stereotype should extend. [required]
- icon: The icon that will appear next to the name of the stereotype in the custom palette. [optional]
- lineStyle: The style of the line (possible values: solid, dashed, dotted, hidden, double). [optional]
- source (*for EClasses only*): The name of the EReference of the EClass that denotes the type of the source node for the edge. [required]
- target (*for EClasses only*): The name of the EReference of the EClass that denotes the type of the target node for the edge. [required]