Noname manuscript No.
(will be inserted by the editor)

Automatic Generation of UML Profile Specific Graphical Editors for Papyrus

Ran Wei[1], **Athanasios Zolotas**[1], **Horacio Hoyos Rodriguez**[1], **Simos Gerasimou**[1], **Dimitrios S. Kolovos**[1], **Richard F. Paige**[1,2]

[1] Department of Computer Science, University of York, York, United Kingdom
e-mail: {ran.wei, thanos.zolotas, horacio.hoyos, simos.gerasimou, dimitris.kolovos, richard.paige}@york.ac.uk
[2] Department of Computer Science, McMaster University, Hamilton, Canada
e-mail: paigeri@mcmaster.ca

3rd March, 2019

Abstract UML profiles offer a way for developers to build domain-specific modelling languages by re-using and extending UML concepts. Eclipse Papyrus is a powerful open-source UML modelling tool which supports UML profiling. However, with power comes complexity; implementing non-trivial UML profiles typically requires the developers to hand craft and maintain several detailed interconnected models through a loosely guided, labour-intensive and error-prone process. We demonstrate how metamodel annotations and model transformation techniques can help to manage the complexity of Papyrus in the creation of dedicated UML profile editors. We present Jorvik, an open-source tool that implements the proposed approach. We illustrate its functionality with examples, and we evaluate our approach by comparing it against manual UML profile specification and implementation using a non-trivial enterprise modelling language (Archimate) as a case study. We also perform a user study in which developers are asked to produce identical editors using both Papyrus and Jorvik demontrating the substantial productivity and maintainability benefits Jorvik delivers.

1 Introduction

The Unified Modelling Language (UML) [17] is the *de facto* standard for object-oriented software and systems modelling. It offers a broad range of abstractions that can be used to express different views of a system, including Class, Use Case, State, Collaboration and Sequence diagrams. Since version 2.0, UML offers an extension and customisation mechanism named *UML Profiling* [11]. UML profiling enables the users to derive Domain-

Specific Languages (DSL) from UML's set of general language concepts. An important advantage of this approach to DSL design is that it allows the reuse of existing UML tools and it supports widely available UML expertise. The basic premise of profiles is that all domain specific concepts are derived as extensions or refinements of existing UML concepts (called UML *meta-element*s). These extensions are called *Stereotype*s. A *Stereotype* definition must be consistent with the abstract syntax and semantics of standard UML *meta-element*s it extends. Consequently, a profile-based model can be created and manipulated by any tool that supports standard UML. Moreover, the concepts underlying profile specialisations of existing UML concepts enables users with UML knowledge to adapt to the approach more easily.

Although Domain-Specific Modelling Languages and tools that support them, like Sirius [39] or Eugenia [24], are becoming more popular, UML is still widely used in model-based software engineering (MBSE) [9]. As a result, alternative ways to define Domain-Specific Languages using dedicated metamodels and textual/graphical editors are available to the users [2, 33].

Papyrus [30] is a leading open-source UML modelling tool developed under the Eclipse Foundation and driven by the PolarSys Initiative and the Papyrus Industry Consortium, which are spearheaded by large high-technology companies such as Airbus, Thales, Saab and Ericsson. After more than a decade in development, Papyrus is close to 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 (e.g. model validation and model-to-model (M2M) and model-to-text (M2T) transformation languages), and 3) enabling multi-paradigm modelling using a combination of UML and EMF-based DSLs. OMG-compliant UML profiles, like SysML [10] and MARTE [15] offer implementations for Papyrus. As highlighted in the recent systematic survey on execution of UML models and UML profiles [5], Papyrus is the most widely used tool for developing UML profiles. However, the ability of Papyrus to support non-trivial UML profiles, altogether with the effort and learning curve related to developing such profiles are recurring concerns. As reported in [42], the manual definition of new UML profiles is typically a tedious, time-consuming and error-prone process.

Will
Is this still the case?

Thanos
I was told that this is not going to be the case, but I don't know if they have made any progress. You might need to check online or in the forum.

Papyrus also supports the creation of profile-specific graphical editors which enables the users to define their own creation palettes, custom styles and related artefacts based on UML profiles. However, the process of creating profile-specific graphical editors is typically difficult because it requires a high level of modelling expertise and it can also be a repetitive and error-prone process.

In this paper, we simplify and automate the process of developing distributable UML profile-specific Papyrus editors. We present *Jorvik*, an approach supported by an Eclipse Plug-in, which enables the use of annotated Ecore metamodels to capture the abstract and graphical syntax of UML pro-

files at a high-level of abstraction, and are then automatically transformed to UML profiles, and artefacts that contribute to distributable Papyrus graphical editors based on the UML profiles.

We evaluate the completeness and efficiency of Jorvik for the automatic generation of a non-trivial enterprise modelling language (Archimate) and its corresponding Papyrus editor against an equivalent developer-driven UML profile and its Papyrus editor. We then apply our approach on several other DSMLs of varying size and complexity [41], to demonstrate its generality and wide applicability. Furthermore, we evaluate the productivity boost of our approach in a user study where developers are asked to created two UML profiles and their Papyrus graphical editors manually and with Jorvik. We report the findings in our study and we report the efficiency of Jorvik based on the results.

This paper significantly extends the prototype approach for automated generation of UML Profile Graphical Editors for Papyrus from our conference paper [44] in the following ways:

1. *We perform a major refactoring of Jorvik to adapt to the new underlying structure of Papyrus 3.0 and above.* – Since Papyrus 3.0, its underlying metamodels have changed, and the process for creating a Papyrus editor has changed significantly. We therefore re-implemented 70% of our work to adapt to the changes.
2. *We enhance considerably the experimental evaluation by conducting user experiments to compare Jorvik with the default Papyrus approach.* – Significant time has been spent on studying the efficiency of our approach, compared to the manual approach.
3. *We develop a validation script that checks the annotated ECore metamodels given as input to our approach.*
4. *We support more styling properties for diagrams created with our approach.*
5. *We explore the feasibility of having nested elements in Papyrus graphical editors.*

Thanos: @Horacio. not sure how to write this properly. Could you please fix this?

The rest of the paper is organised as follows. In Section 2 we motivate the need of the proposed approach. In Section 3 we describe the proposed approach and the process of the proposed approach. In Section 4 we discuss in detail the artefacts needed to create a working Papyrus editor, and the implementation details of automatically generating these artefacts. In Section 5 we present the evaluation conducted. In Section 6, we discuss the related works. Finally, in Section 7 we conclude the paper and highlight the plans for future work.

2 Background and Motivation

In this section we outline the process for defining a UML profile and the supporting model editing facilities for the UML profile in Papyrus. This process

typically involves the manual creation of a number of inter-related models and configuration files. We highlight labour-intensive and error prone activities involved in the process that motivate the need of automatic generation of these artefacts.

2.1 UML Profile

A UML Profile in Papyrus is an EMF model that conforms to the UML Ecore metamodel. In order to create a new UML Profile, developers need to create instances of *Stereotype, Property, Association*, etc. to create the elements of their domain specific modeling languages, and their properties and relationships among the elements.

Papyrus offers, among other choices, the mechanism of creating UML profiles using a *Profile Diagram*. Users can uses the palette provided in the UML Profile Diagram editor to create all elements required to form a UML profile. The properties of each element (e.g., data types of properties, multiplicity navigability, etc.) can then be set using the properties view. In a profile, each *Stereotype* needs to extend a UML concept (hereby referred to as *meta-element*). Thus, users need to define which *meta-elements* their *Stereotype*s extend. This is achieved by importing the meta-elements and by adding appropriate extension links between their *Stereotypes* and the *meta-elements* by using the tool provided in the palette. The process of creating a UML profile can be repetitive and labour-intensive, depending on the size of the profile. Having created a profile, users can then apply it to a UML model. Users typically create instances of UML meta-elements (e.g. UML::Class) and apply their *Stereotype*s defined in their UML profiles. For example, if a *Stereotype* extends the "UML::Class" meta-element, users can apply it to selected instances of UML:Class in their models. In this sense, the users are creating instances of the elements they define in their DSLs.

One of the limitations of UML Profiles in Papyrus is that links between stereotypes can be instantiated as edges in a diagram only if they extend a *Connector* meta-element (e.g., UML::Association). For example, if "Stereotype A" refers to "Stereotype B" via an "A_to_B" reference, then in order to be able to draw this connection as an edge on the diagram, "A_to_B" should be created as a separate *Stereotype*. These connector *Stereotype*s do not hold any information about the *Stereotype*s that they can connect, so users need to define such restrictions by manually writing OCL constraints to validate at least two things: 1) if the source and target nodes are of the correct type and 2) if the connector is in the correct direction (e.g. for "A_to_B", the edge should lead from *Stereotype* A to *Stereotype* B). These constraints can be rather long and need to be manually written and customised for *each* edge *Stereotype*. This, as illustrated in Section 4.3, can also be a labour-intensive and error-prone process.

2.2 Distributable Custom Graphical Editor

With the UML profile created, users can apply it to UML diagrams. Users select a UML element (e.g., an instance of UML::Class) and manually apply a *Stereotype* in the UML profile they define. A *Stereotype* can only be applied to instances of the meta-elements they extend. For example, a *Stereotype* that extends the UML::Package meta-element in the profile cannot be applied to an instance of UML::Class. This task is obviously labour-intensive and repetitive. In addition, the users typically need to remember the meta-element that each *Stereotype* extends in their UML profile.

To address this recurring concern, Papyrus offers at least three possible options for creating a custom palette which allows users to create UML elements and apply selected *Stereotype*s on them in a single step. Papyrus provides the facility to customise the palette for an open editor. In the first option, the users can make use of the customisation facility to create their own palettes, and then specify what *Stereotype*s should be instantiated for the creation tools in the palette. Although this is an easy-to-use approach, it must be done manually every time when a new diagram is created. In addition, it cannot be shared in case the profile needs to be distributed to collaborators. The second option involves the manual definition of an XML-based palette configuration file, which is automatically loaded every time the profile is applied to a diagram. This option however, is discouraged by Papyrus as it does not allow the full use of Papyrus functionality. Furthermore, this option is based on a deprecated framework, and its use is not encouraged. The third option is to create a UML profile editor, which includes the manual creation of a number of inter-related models and artefacts, including a palette configuration model. Although this option provides a whole solution to create a UML profile editor, in the paper we illustrate that this option is a labour-intensive and error prone process.

The definition of custom shapes for the instantiated *Stereotype*s is another common requirement. In Papyrus, Scalable Vector Graphics (SVG) shapes can be bound to *Stereotype*s during the profile creation process. However, to make these shapes visible, users need to set the visibility of the shape of *each Stereotype* to true. Although this is an acceptable trade-off, the users typically need to hide the default shapes by writing custom style rules in a Cascading Style Sheet (CSS), as by default the SVG shape bound to a *Stereotype* overlaps with the default shape of the base meta-element. The CSS 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 a number of inter-related models and files, and to implement extension points in an Eclipse plug-in project. To create a working UML profile editor in Papyrus, the users typically need to create:

- A Element Types Configuration model;
- A Palette Configuration model;

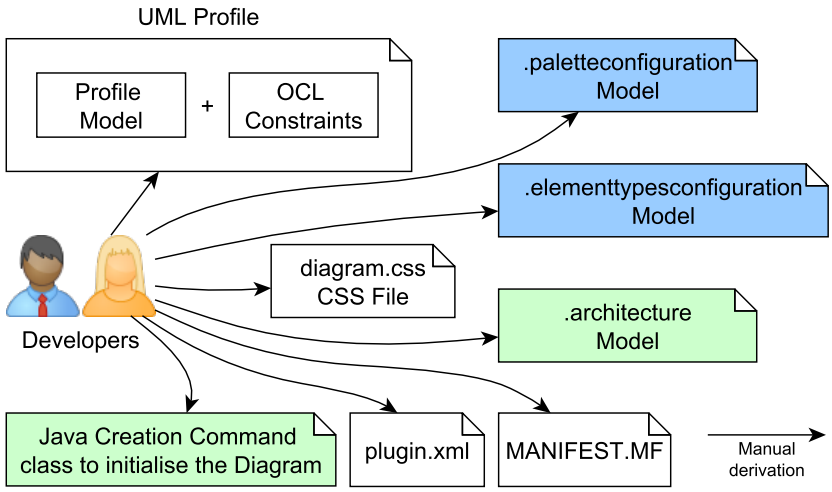

Fig. 1: Models/files developers need to write manually to develop a fully functional distributable Papyrus profile editor for Papyrus 3.0+.

– A Cascading Style Sheet for customised styles;
– A Java Creation Command class to initialise the diagram;
– An Architecture model to describe the architecture of the editor;
– Plug-in related files for extensions and dependencies.

Figure 1 shows all the artefacts needed to be created for having a distributable Papyrus UML profile editor for Papyrus 3.0+[1].

Will mention this in the evaluation

Through our experiment (see Section 5), we found out that it is almost impossible to create these models without any working examples. The documentation of Papyrus provide limited useful insight in this matter. In addition, it is also a labour-intensive and error-prone process to migrate editors created based on Papyrus 2.0 to Papyrus 3.0+. Developers typically need to create new Palette Configuration and Element Types Configuration models, as well as the new concepts such as the Architecture model.

UML profile editors created using approaches in Figure **??** and Figure 1 both have custom palette, with which the users are able to create elements in their DSL (UML meta-elements with *Stereotype*s defined in the UML profile applied automatically to them). Detailed discussions about the artefacts needed in order to create a UML profile editor are provided in Section 4. A few hundred lines of code need to be written while tedious Plug-in creation and repetitive model creation tasks should be done.

Will To reference to evaluation

[1] Metamodels for Element Types Configuration, Palette Configuration have been changed since our previous work (rendered in blue). The Architecture model and the Java Creation Command class are new concepts introduced in Papyrus 3.0 since our previous work (rendered in green).

This *labour-intensive*, *repetitive* and *error-prone* process could be automated. In this paper, we present our tool - Jorvik, which uses a single-source input to automatically generate a UML profile and models/files mentioned above for a UML profile editor for Papyrus.

3 Proposed Approach

In this section we present *Jorvik*, an automatic generation tool for Papyrus UML profiles and their profile-specific editors. Using Jorvik, we minimise the labour intensive and repetitive tasks required in the manual creation process. Through Jorvik, developers can define the abstract syntax and the concrete (graphical) syntax of their DSLs (the UML profile) in the form of an annotated Ecore metamodel. The annotations can be used to specify the *Stereotype*s to be created in the UML profile, and what meta-elements they extend. With the annotations, the developers can also specify the graphical syntax and related information (e.g. shapes for the *Stereotype*s and the icons for the creation tools in the palette). The annotated Ecore metamodel is then processed by Jorvik and transformed into a UML profile and related Papyrus-specific models/files needed for a standalone Papyrus editor. This is done hrough a series of model management operations like model validation, model-to-model (M2M) and model-to-text (M2T) transformations. An overview of our proposed approach is presented in Figure 2. We discuss the steps involved in the process in detail in the remainder of this section, followed by a running example. Technical details about the different steps of the proposed approach are provided in Section 4.

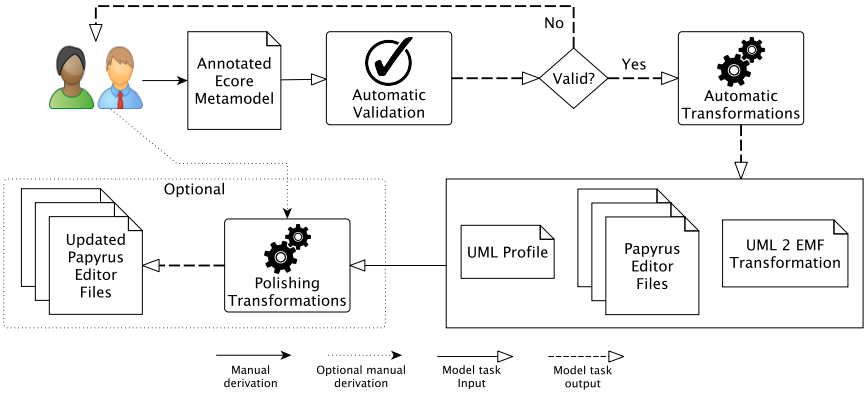

Fig. 2: An overview of the proposed approach

The first step is to create an annotated Ecore metamodel to define the intended DSL. In order to do this, a number of annotation keywords are defined for the users to use:

1. **@Diagram** annotations are used to define diagram-specific information like the name and the icon of the diagram. This annotation can be applied to EPackages and should always be applied to the EPackage at the top level.
2. **@Node** annotations are used to indicate the *Stereotype*s that should be instantiated as nodes in the diagrams. The UML meta-element that this *Stereotype* extends is provided through the *base* property. The SVG shape to be used on the canvas and the icon of the specific element in the palette are specified through the *shape* and *icon* properties, respectively.
3. **@Edge** annotations are used to denote *Stereotype*s that should be instantiated as edges in the diagrams. The UML meta-element extended by this stereotype is provided through the *base* property. The icon of the specific element in the palette is also passed as property along with the desired style of the line. This annotation can be applied either to an EClass or to an EReference.

In the metamodel, all annotated *EClass*es are automatically transformed into *Stereotypes*. *Stereotype*s are also created from annotated EReferences (more about this below). Depending on the required graphical syntax of the *Stereotype* (i.e., if it should be represented as a *node* or as an *edge* on the diagram), developers need to use the appropriate annotation on EClasses/EReferences. A detailed list of all valid annotation properties is provided in Appendix A.

With the annotated Ecore metamodel in place, the next step is to check the validity of the annotations in order to proceed with the generation process. Therefore, a custom-made model validation script written in the Epsilon Validation Language (EVL) [25] (see Figure 2) is used. The validation rules (which we describe in detail in Section 4) check if the annotations provided and their attributes match the expected ones by Jorvik (e.g., each annotated element includes a reference to the UML base element it extends). If the validation fails, feedback is provided to users to fix the problems detected. Otherwise, the annotated Ecore metamodels are consumed by a workflow of M2M and M2T transformations illustrated in Figure 4 and described in detail in Section 4. The transformations are written in the Epsilon Transformation Language (ETL) [23] and the Epsilon Generation Language (EGL) [35]. In principle, any other model validation, M2M and M2T language could be used (e.g., ATL [22]). The transformations produce the UML profile with the appropriate OCL constraints and all the configuration models and files needed by Papyrus. In addition, an M2M transformation is also generated that can be used later by developers to transform the UML models that conform to the generated UML profile, back to EMF models that conform to the Ecore metamodel used as source. Model management programs already developed to run against model conforming to the EMF metamodel can be re-used.

Jorvick offers the option of *polishing transformations*, where developers are able to write their own (optional) transformations to fine-tune the gen-

erated artefacts produced by the built-in transformations. In the following section, the proposed approach is explained via a running example.

```
@namespace(uri="sdpl",prefix="sdpl")
@Diagram(name="SDPL", icon="ic/sdpl.png")
package Process;

@Node(base="Class", shape="sh/step.svg", icon="icons/step.png")
class Step {
  attr String stepId;
  ref Step[1] next;
}
@Node(base="Class", shape="sh/tool.svg", icon="icons/tool.png", bold="
    true")
class Tool {
  attr String name;
  attr int version;
}
@Node(base="Class", shape="sh/per.svg", icon="icons/per.png")
class Person {
  attr String name;
  attr int age;
  @Edge(base="Association", icon="icons/line.png", fontHeight="15")
  ref Tool[*] familiarWith;
}
@Edge(base="Association", icon="icons/line.png", source="src", target="
    tar")
class Role {
  attr String name;
  ref Step[1] src;
  ref Person[1] tar;
}
```

Listing 1: The annotated Emfatic code that defines the metamodel of SDPL and can be used to generate the UML profile and the associated Papyrus editor.

3.1 Running Example

In this example we define a DSL - the Simple Development Process Language (SDPL) - in an annotated Ecore metamodel and we generate the corresponding UML profile and Papyrus graphical editor using Jorvik. We start by defining the abstract syntax of the DSL using Emfatic[2], a textual notation for Ecore, which is shown in Listing 1 (ignore the annotations at

[2] `https://www.eclipse.org/emfatic/`

this point). A process defined in SDPL consists of *Steps*, *Tools* and *Persons* participating in it. Each *Person* is familiar with certain tools and can have different *Role*s in steps of a process, while each step refers to the next step that follows using the *next* reference.

In order to generate the UML profile and the Papyrus graphical editor we need to add the following concrete syntax-related annotations shown in Listing 1:

- **Line 2:** The name and the icon that should be used by Papyrus in the custom diagrams menus are defined using the *name* and *icon* properties of the *@Diagram* annotation.
- **Lines 5, 10 & 15:** The *@Node* annotation is used to define that the three concepts (i.e., Steps, Roles and Tools) should be *Stereotype*s in the UML profile that will be represented as nodes on the diagram. The *base* parameter is used to define which UML meta-element the *Stereotype* should extend (i.e., *Class*[3] in the *Stereotype*s of this example[4]). The shape on the canvas and the icon in the palette for each *Stereotype* are given using the *shape* and *icon* annotation details. We also change the font style by setting the bold details to true (see line 10).
- **Line 19 & 22** The EReference *familiarWith* and the *Role* EClass are added in the profile as *Stereotype*s that extend the meta-element *Association* of UML (UML::Association). These *Stereotype*s should be represented as links in the diagrams and thus are annotated with the *@Edge* string. In contrast to the *familiarWith EReference*, the types the *Roles* edge should be able to connect are not know 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 EReferences: *src* and *tar* respectively. We also set the font height to 15 for the labels of the *familiarWith* edges (see line 19).
- **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*. However, it will be a property of the *Step Stereotype* in the generated profile, so it can be set in the model (but it will not be displayed on the diagram).

The automated M2M and M2T transformations discussed above are then executed on the Ecore file and the produced SDPL Papyrus editor is presented in Figure 3.

[3] Although profiles are meant to accommodate "natural" extensions to UML artefacts, sometimes is the case that generic meta-elements are used instead. For example, in the Archimate for Papyrus tool presented in Section 5.1, the Use Case meta-element is used as base for the *Business Service Stereotype*, or Class as the base for the *Business Role Stereotype*. Thus, the selection of the most appropriate base element is left for the user.

[4] *Class* is in fact *UML::Class*, we omit the "UML::" prefix to improve clarity.

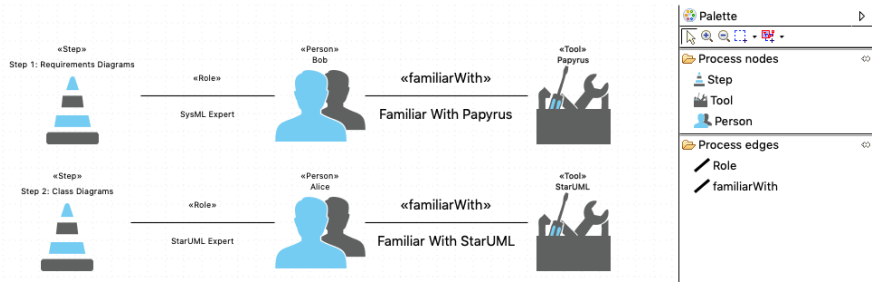

Fig. 3: The SDPL editor for Papyrus where two steps in the software development process are defined and responsible persons are attached to them along with the tools they are specialized on.

```
Label[type=StereotypeLabel]{
   visible : false ;
}

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

Listing 2: The polishing transformation for the CSS file generation that sets the visibility of the names of the nodes to true.

3.2 Polishing Transformations

The generated editor is fully functional but it can be further customised to fit users' custom needs. For example, by default, our automatic transformations dictate the diagram, through the CSS file to show the stereotype name applied to each node. However, in this example we want to hide the stereotype names and display labels 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 profile and the editor in the future (e.g., because of a change in the metamodel). To avoid this, the user can use the – optional – CSS generation polishing transformation (#5b in Figure 4) shown in Listing 2. Every time the profile and editor generation is executed, the polishing transformation will be called and will set the visibility of the stereotypes to false automatically.

The EGL template in Listing 2 generates a CSS rule in lines 1-3 that sets the visibility property of the stereotypes' labels to false. It stores all the elements in the Ecore metamodel that are annotated as @Node (line 6) in a collection and iterates though them in lines 7-10. For each of the node stereotypes, it generates the static text "[appliedStereotypes =" followed by the name of each stereotype and a comma. At the end it prints the curly brackets (lines 10 and 12) and the text "fontColor:red;" in line 11. The resulted output that is amended automatically in the original CSS file by the polishing transformation is the one shown in Listing 3.

```
1 Label[type=StereotypeLabel]{
2     visible:false  ;
3 }
4
5 [appliedStereotypes~=Step],
6 [appliedStereotypes~=Tool],
7 [appliedStereotypes~=Person] {
8   fontColor:red ;
9 }
```

Listing 3: The output that is amended in the original CSS file using the CSS polishing transformation of Listing 2

4 Implementation

In this section, we discuss the technical implementation of Jorvik which underpins the process of our approach discussed in the previous section.

Figure 4 shows the transformations workflow. Each transformation is identified by a number (#2 – #9 in Figure 4) for easier reference. In addition to the transformations, a pre-transformation validation script (#1 in Figure 4) is executed to verify the correctness of the annotations and provide useful feedback to the users if there is anything wrong. Moreover, supporting files needed for the creation of the Papyrus plugin are also generated while icons and shapes are placed next to the annotated metamodel (#8 in Figure 4). As the transformations consists of about 1000 lines of code we will describe them omitting low level technical details[5].

4.1 Pre-transformation Validation (#1)

To check the correctness of the annotated ECore metamodel, a model validation program is firstly executed against the metamodel. The program

[5] Full implementations and instructions are available at `https://github.com/wrwei/Jorvik`

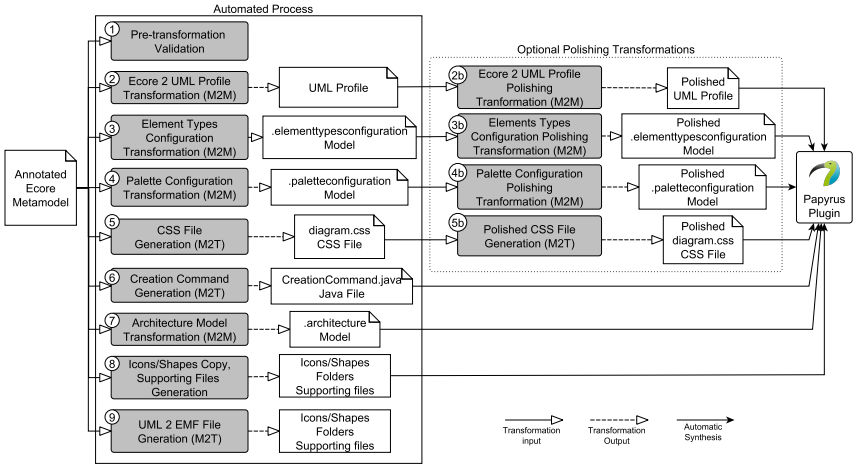

Fig. 4: An overview of the transformation workflow

is written using EVL and consists of several rules that check if the annotated ECore elements have all the necessary information (e.g., a UML base class) and if the values provided are correct (e.g., font size for labels is a positive integer). Listing 4 presents an example of a rule written in EVL which checks if the value provided for the "bold" details in a @Node annotation is correct (i.e., true or false). More specifically, in lines 2–3 a guard is applied to check that the rule is only applied to elements that have the @Node annotation attached to them (line 2) while tests the rule only on those annotations that have the "bold" detail defined (line 3). In lines 4–6, the condition to check is provided, which is that the value of the "bold" detail is either *true* or *false*. Finally, in lines 7–8 a message is displayed to the users in case the rule is violated providing information regarding the acceptable values and the name of the element(s) that violated this rule.

```
1 constraint NodeAnnotationBoldValueIsCorrect {
2     guard : self.getEAnnotation("Node").isDefined() and
3             self.getEAnnotation("Node").details.get("bold").isDefined()
4     check : self.getEAnnotation("Node").details.get("bold").equals("true")
5             or
6             self.getEAnnotation("Node").details.get("bold").equals("false")
7     message : "Bold value for element " + self.name + " is not correct.
8             Possible values are: true or false."
9 }
```

Listing 4: Example rule that checks that the values provided to the "bold" styling detail for @Node annotations is correct.

Table 1 enumerates all the rules included in the pre-transformation validation step along with their descriptions and the conditions that are

#	Rule Description	Condition Checked
1	There is exactly one Diagram annotation	The number of the Diagram annotations is 1
2	Diagram annotation has a name detail	The name detail is defined
3	Diagram annotation has acceptable details provided	There are no other details provided rather than name and/or icon
4	Node/Edge annotations have base class set	Any string is provided
5	Edge annotations of EClasses have source/target defined	The source/target details are defined
6	An acceptable lineStyle value for Edge annotations is provided	The value is dashed, solid, dotted, hidden or double
7	An acceptable bold value for Node/Edge annotations is provided	The value is either true or false
8	An acceptable fontHeight value for Node/Edge annotations is provided	The value is a positive integer
9	Node annotations have acceptable details provided	There are no other details provided rather than base, fontHeight and/or bold
10	Edge annotations have acceptable details provided	There are no other details provided rather than base, source, target, fontHeight, bold, and/or lineStyle

Table 1: The list of the rules checked for the annotated ECore metamodel.

checked. The implementation of the rules in EVL can be found in the paper's webpage[6].

4.2 EMF to UML Profile Generation (#2)

The EMF to UML Profile Generation executes a model-to-model transformation written in ETL. The source model of this transformation is the annotated Ecore metamodel (e.g., Listing 1) and the target model is a UML profile model. This transformation consists of two main rules, one that creates a *Stereotype* for each EClass element of the metamodel and a second that creates a *Stereotype* for each EReference annotated as @Edge.

- **rule eclass2stereotype:** This transformation rule transforms each EClass element in the annotated Ecore metamodel to an element of type *Stereotype* in the target UML model. All attributes of each EClass are also transformed across to the created *Stereotype*.
- **rule reference2stereotype:** This rule creates a new *Stereotype* with the same name in the UML profile model for each of the *EReferences*

[6] `https://github.com/wrwei/Jorvik`

that are annotated as @Edge in the Ecore metamodel. No attributes are added to the stereotype as EReferences do not support attributes in Ecore[7].

When all stereotypes are created, a number of post-transformation operations are executed to 1) create the generalisation relationships between the *Stereotype*s, 2) add the references/containment relationships between the *Stereotype*s, 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 superclass 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 is also created and added to the *Stereotype*. The name and the multiplicities are also set.
3) By default the *Stereotype*s 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. The OCL constraints are explained in details in the section that follows.

4.3 OCL Constraints

To illustrate the OCL constraints, we provide a partial view of the SDPL UML profile in Figure 5[8].

In Figure 5, there are three *Stereotype*s. *Person* and *Tool* extend meta-element *UML::Class*, and they correspond to classes *Person* and *Tool* in the metamodel shown in Listing 1. *Stereotype familiarWith*, which extends meta-element *UML::Association*, corresponds to the reference *familiarWith* in the *Person* class in Listing 1.

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

[7] It is the users' responsibilities to make sure that they handle name collisions for EReferences themselves (for that EClasses may have EReferences with the same name).

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

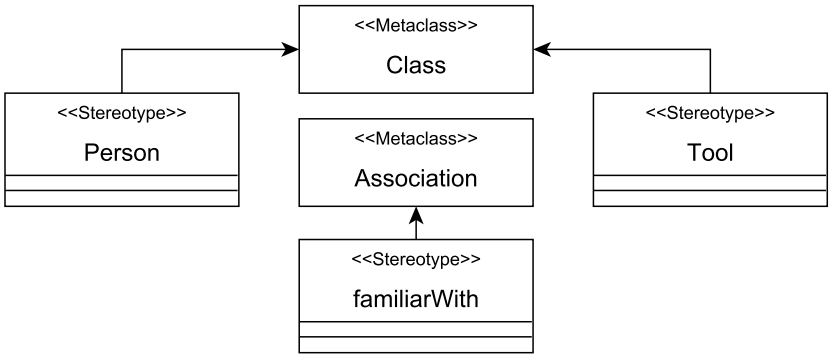

Fig. 5: Example UML profile for SDPL showing Person, Tool and the familiarWith association

- **End Types**: one of the elements a *familiarWith* association connects to, has the *Person* steretotype applied to it while the other has the *Tool* stereotypes applied to it;
- **Navigability**: the *familiarWith* association starts from an element stereotyped as *Person* and points to an element stereotyped as *Tool*.

4.3.1 End Types Listing 5 shows the OCL code for the *End Types* constraint. Line 1 accesses the types (instances of UML::Class that have *Stereotype*s defined in the profile applied to them) that *familiarWith* connects. Lines 3 and 4 check if the types that *familiarWith* connects are of type that either has *Stereotype Person* or *Tool*. In this way, if a *familiarWith* association connects two types that are not *Person* or *Tool*, the constraint fails.

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

Listing 5: The End Types constraint in OCL

4.3.2 Navigability Listing 6 shows the OCL code for the *Navigability* constraint. In this case, we are interested in checking the *isNavigable* property of each end. Thus, 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* and not the other way around.

```
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
8 false
9 endif
```

Listing 6: The Navigability constraint in OCL

We need to highlight, that currently, opposite references are not supported; plans for future work are outlined in Section 7.

With these two constraints implemented, we are able to automatically generate OCL constraints for *Stereotype*s that extend *UML::Association*. We use the *End Types* and *Navigability* constraints as templates with dynamic sections (where the specific *Stereotype* names are inserted dynamically, e.g., *Person* and *Tool*). So far we have only explored constraints for *Stereotype*s that extend *UML::Association*. The constraint templates for *Stereotype*s that extend other UML relationships need to be developed separately as the means to access source/target elements of the relationship are different.

4.4 Element Types Configuration Transformation(#3)

Apart from the UML profile, Papyrus graphical editors require an Element Types Configuration model, which associates the *Stereotype*s defined in the UML profile with their abstract syntax (the meta-elements in UML they extend) and their concrete syntax (the graphical notations of the meta-elements in UML they extend).

This transformation is responsible for creating an Element Types Configuration model (of extension .elementtypesconfiguration) that contains type specialisation information for the *Stereotype*s in the UML profile. For each *Stereotype*, two *SpecializationTypeConfiguration*s are created, one links the *Stereotype* to its UML meta-element, another links the *Stereotype* to the concrete syntax of its UML meta-element. For example, a *Stereotype* that extends UML::Class needs to specialise the UML::Class *MetamodelTypeConfiguration* defined in the UML Element Types Configuration model; and it needs to specialise the Class Shape *SpecializationTypeConfiguration* defined in the UML Diagram Element Types Configuration model[9].

[9] Both models reside in the Papyrus Plug-in *org.eclipse.papyrus.uml.service.types*

In addition to *SpecializationTypeConfiguration*s, for each *Stereotype* an *ApplyStereotypeAdviceConfiguration* needs to be created, which associates the *SpecializationTypeConfiguration* to the actual *Stereotype* in the UML profile.

4.5 Palette Configuration Transformation (#4)

This transformation is responsible for creating a Palette Configuration model (of extension .paletteconfiguration) that configures the contents of the custom palette for the diagram in Papyrus. 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., one for the tools that create nodes and one for those creating edges). For each element in the Ecore metamodel annotated as @Node/@Edge a new *ToolConfiguration* element is created and added to the nodes/edges drawer respectively. The kind of *ToolConfiguration* are decided based on the @Node/@Edge annotation. For nodes, the kind is *CreationTool* while for edges, the kind is *ConnectionTool*. An *IconDescriptor* element is also created and added to the *ToolConfiguration* pointing to the path of the icon for that tool (this is the path passed as argument to the *icon* property of the @Node/@Edge annotation). Finally, for each *ToolConfiguration*, they need to refer to the element types they conform to, which are defined in the Element Types Configuration model transformed in Step #3. In this way, when an element is created in the diagram using the palette, behind the scene, Papyrus is able to locate the *Stereotype* and infer the UML syntax it specialises.

4.6 CSS File Generation (#5)

As stated above, in Papyrus, the look and feel of diagram elements can be customised using CSS. In this transformation we generate the CSS style rules that define the appearance of nodes and edges in diagrams. Each node on a diagram has a set of compartments where the attributes, the shape, etc. appear. Initially, for all nodes that will appear on the diagram, 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 that 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. For elements of type *Stereotype*, the assignment of the SVG shapes to the *Stereotype*s is achieved by assigning the path of the SVG file to 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 stereotype is set to the value of the lineStyle parameter (e.g., “solid”, “dashed”, etc.).

4.7 Creation Command Generation (#6)

In order for Papyrus to create a diagram, it requires the initialisation of the diagram. The Creation Command is a Java class which is responsible for initialising Papyrus diagrams. The Creation Command class is needed since Papyrus 3.0+. The rationale for the creation command is that it creates a UML model from the UMLFactory as the root element of the diagram. The minimal requirement for diagram initialisations are:

- UML primitive types: the primitive types need to be imported to the diagram in order for the users to reference to them;
- UML profiles: the standard UML profile and the user defined UML profile need to be applied in order to initialise the diagram.

In order to apply the user defined UML profiles, they need to use the pathmap defined in their plug-ins, which is explained in #8. The Java class is generated using a model-to-text transformation written in EGL.

4.8 Architecture Model Generation (#7)

Papyrus adopted the notion of an Architecture model in order to describe the architecture of the graphical editors since Papyrus 3.0+. This transformation synthesises an Architecture model using a model creation program using the Epsilon Object Language (EOL) [26]. The program needs 1) the annotated Ecore metamodel for the DSL, 2) the Element Types Configuration model, 3) the Palette Configuration model, 4) the UML metamodel, 5) the Creation Command Java class and 6) the CSS custom style sheet as inputs. In particular, in the Architecture model, the generation program creates:

- An *ArchitectureDomain* which represents the domain of the DSL;
- A number of *Concern*s to describe the concerns of the domain;
- A number of *Stakeholder*s involved in the domain;
- An *ArchitectureDescriptionLanguage* to describe the architecture, which consists of a number of *Viewpoint*s, a number of *PapyrusDiagram*s. The Element Types Configuration and the Creation Command class are needed in the *ArchitectureDescriptionLanguage*. The Palette Configuration model and the CSS sheets are needed in *PapyrusDiagram*s.

The Architecture model then needs to be registered and acts as the entry point to all the models/files for a Papyrus editor, which is done in Step #9.

4.9 Icons, Shapes and Supporting Files (#8)

Jorvik supports the generation of the UML profile and the infrastructure for the Papyrus diagram in either a new Eclipse plug-in project or in the same

plug-in project where the annotated Ecore metamodel resides. In both scenarios, the "MANIFEST.MF", the "build.properties" and the "plugin.xml" files are created (or overridden respectively). The "MANIFEST.MF" file includes a list of all required bundles/Plug-ins. The "build .properties" file guides the build process of the Eclipse plug-in. The "plugin.xml" file includes all the necessary extensions for Papyrus to be able to register the UML profile and create the diagrams (e.g., extensions that point to the Architecture model). For the creation of a Papyrus editor, in the "plugin.xml", three extension points need to be implemented:

- *org.eclipse.emf.ecore.uri_mapping*, in which the users create a mapping between the path of the folder that hold their UML profile, and a PATHMAP, which they can reference in the files/models they create;
- *org.eclipse.papyrus.uml.extensionpoints.UMLProfile*, which points to the location of the UML profile that the users define;
- *org.eclipse.papyrus.infra.architecture.models*, which points to the location of the Architecture model that the users define in Step #7.

For the scenario where the Papyrus plug-in is created as a new project, the shapes (SVG files) and the icons (PNG files) are copied to the newly created Plug-in project. For the scenario where the UML profile and editor is generated in the same project in which the Ecore source file resides, the shapes and icons files do not need to be copied as they already exist.

Finally, two files that only consist of the XML and the XMI header (namely "*.profile.di" and "*.profile.notation") are generated. These files are necessary for Papyrus to construct the UML profile model[10].

4.10 UML to EMF Transformation Generation (#9)

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 generated by our approach, back to EMF models that conform to the source Ecore metamodel given as input to the approach. One rule is generated for each of the stereotypes that transforms them back to the appropriate type of the Ecore metamodel. Each stereotype has the same attributes and references as the original EClass, therefore, this EGL script also generates the statements in each rule that populate the attributes and the references of the newly created instance of each EClass with the equivalent values of the UML model. An example of an auto-generated rule is shown in Listing 7. 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 rule PersonUML2PersonEMF
```

[10] The diagram layout information requires coordinates related information in the diagram, therefore it is not related to this work

```
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 7: Example of an auto-generated ETL rule that transforms elements stereotyped as "Person" in the UML model to elements of type "Person" in an EMF model.

ETL provides the ::= operator for rule resolution. The ETL engine keeps transformation traces which links source elements to target elements of the transformation. When ::= is used, the ETL execution engine inspects the established transformation traces and invokes the applicable rules (if necessary) to calculate the counterparts of the source elements contained in the collection.

In our example (line 6 in Listing 7), the expression "s.familiarWith" returns a collection of *UMLProcess!Tool*s (denoted by *ct*). By using "::=", the ETL engine will look for the rules that transform *UMLProcess!Tool* to *EMFProcess!Tool* and invoke the rules if necessary (if the source elements have not been transformed, as shown in the transformation trace) and put the transformed elements into sub-collections (denoted by *sc*). After the ETL engine goes through all the elements in *ct*, the sub-collections *sc*s are returned (flattened to a single collection if more than one) and are added to "t.familiarWith".

4.11 Polishing Transformations (#2b - #5b)

For transformations #2- #5, 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 next to the Ecore metamodel. If a file with the same name exists, it is executed against the Ecore metamodel and targets the already created output model of the original transformation. The execution of the polishing transformation is set *not* to overwrite the target model but to refine it instead. Table 2 shows the names that each polishing transformation is expected to have.

Table 2: Polishing Transformations File Names

Transformation ID	**Required File Name**
#2	emf2umlProfile.etl
#3	elementTypesConfigurationsM2M.etl
#4	paletteConfigurationM2M.etl
#5	cssFileGeneration.egl

5 Evaluation

In this section we evaluate Jorvik in three different ways. In the first evaluation, we apply Jorvik to generate a Papyrus editor for the non-trivial Archimate UML profile [19,21]. We use the Adocus Archimate for Papyrus[11] (an open-source tool that includes a profile for Archimate and the appropriate editors for Papyrus) for reference. We compare the proportion of the tool that Jorvik is able to generate automatically, check 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 to measure the *efficiency* of Jorvik in generating profiles/editors against an existing relatively large profile/editor.

In the second evaluation, we assess the completeness of Jorvik by applying it to a number of metamodels collected as part of the work presented in [41]. This way, Jorvik is tested to check if it can successfully generate profiles and editors for a wide variety of scenarios.

In the third evaluation, we conduct a user experiment in which we asked software engineers to build Papyrus editors for a two metamodels. We first ask the engineers to create the profiles and editors manually, and then ask them to create the same profiles and editors using Jorvick. We measure the time and report problems encountered during the experiment and we compare the results.

5.1 Efficiency

The Archimate for Papyrus tool offers five kinds of diagrams (i.e., Application, Business, Implementation and Migration, Motivation and Technology diagrams). Each of the diagrams uses different stereotypes from the Archimate profile. In this scenario, we create five Ecore metamodels and annotate the elements that need to appear as nodes/edges in the diagrams. We then generate the editors for each kind of the Archimate diagram. At this point, five fully functional editors are generated that can be used to create each of the five types of diagrams that the Archimate for Papyrus tool also supports.

However, our generated editors do not offer some special features that the Archimate for Papyrus tool offers. For example, Archimate for Papyrus offers a third drawer in the palette for some diagrams that is called "Common" and includes two tools (named "Grouping" and "Comment"). Another feature that is not supported by our default transformations is the fact that in Archimate for Papyrus, users are able to have the elements represented either by their shapes or by a coloured rectangle depending on the CSS class applied to them. We also find out that Archimate for Papyrus also organises the creation of the *Junction* (which is a node that acts as a junction for edges) node in the relations drawer in the palette. In order to be able to implement such missing features, we need to write the extra polishing

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

transformations. We do not go into details on the content of the polishing transformations for this specific example[12].

In our previous work [44]

Table 3 summarises the lines of code we have to write manually to generate each file needed by Papyrus to create the 5 basic diagrams (column "Jorvik" / "Handwritten") and the number of lines of the polishing transformations that were needed to produce the enhanced editor (column "Jorvik" / "Handwritten (Polishing)"). The totals are given in column "Jorvik" / "Total". For easier comparison, the total lines of code the authors of the Archimate for Papyrus had to write manually are provided in column "Archimate for Papyrus" / "Total Handwritten"[13].

As one can see from the numbers, our approach requires about 90% less handwritten code to produce the basic diagrams and about 85% less code to produce the polished editor that matches the original Archimate for Papyrus editor. Our approach offers an editor that matches the original Archimate for Papyrus tool but also atop that the ETL transformation that allows the transformation of the created UML models back to EMF. The generation of OCL constraints is also an extra feature that our approach offers. In this scenario, however, the generation of constraints could not be assessed as the tool that we are comparing with (i.e., Archimate for Papyrus) allows any edge to connect any two types of nodes. We leave this evaluation as part of our future work.

5.1.1 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 3 to have a fair comparison.

[12] The generated Plug-ins for Archimate and the polishing transformations are available from `https://github.com/wrwei/Jorvik`

[13] The produced plugins generated with Jorvik for the Archimate profile can be downloaded from `https://github.com/wrwei/Jorvik`

	Jorvik (pre Papyrus 3.0)			Jorvik (post Papyrus 3.0)			Archimate for Papyrus (Pre Papyrus 3.0)
File	**Hand-written**	**Hand-written (Polishing)**	**Total**	**Hand-written**	**Hand-written (Polishing)**	**Total**	**Total Hand-written**
ECore	436 (668)	0	436 (668)	436 (668)	0	436 (668)	0
Profile	0	0	0	0	0	0	1867 (1089)
Element Types Configuration	0	11	11	0	0	0	237 (61)
Palette Configuration	0	50	50	0	50	50	1305 (323)
CSS	0	195	195	0	195	195	537
Creation Command				0	0	0	
Architecture Model				0	0	0	
Types Configuration	0	10	10				788 (327)
Diagram Configuration	0	0	0				58 (28)
Total	**436**	**266**	**702**	**436**	**245**	**681**	**4792 (1828)**

Table 3: Lines of manually written code of each file for creating a Papyrus UML profile and editor for ArchiMate.

5.2 Completeness

In addition to the generation of the Archimate profile/editors, we tested the proposed approach with nine more Ecore metamodels from different domains. The names of the metamodels (including Archimate) and their size (in terms of types) are given in Table 4. 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.

Table 4: The names and sizes of the ten metamodels against which the approach was evaluated to test completeness

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)

As illustrated in Table 4, the metamodels varied in size, from small profiles (having 5 stereotypes) to large profiles (up to 57 stereotypes). 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. The time needed for the generation varied from miliseconds up to a few seconds. In the future, we plan to assess further the scalability of our approach using larger metamodels.

5.3 Experiment

We have argued that using Jorvik provides significant gains in productivity when building custom editors for Papyrus. The experiment was designed to ascertain how significant the gains can be. The time taken to build a custom editor was recorded for both the Jorvik and the non-automated approach by novice and intermediate UML and Profiles users. The results suggest that regardless of the user expertise, the gains in productivity can be as high as 10 fold.

5.3.1 Method Three participants completed all conditions of this study: two where novice and one intermediate users. Participants are all computer science graduates with Masters Degree in CS too. All participants were naive to the purpose of the experiment.

Are we using Sina as participant or not?

Table 5: Tasks and times for the non-automated approach

Task	Total Time (m)	Basic Time (m)	Essential Time (m)
UML Profile	60	40	20
Element Types Configuration model	60	40	20
Palette Configuration model	60	40	20
Cascading Style Sheet	30	20	10

5.4 Materials

All participants used a computer with the Eclipse IDE available with the Papyrus and Jorvik plugins installed. The computer has internet access. Two questionnaires where used. One, before the experiment, so participants could self asses their expertise in UML, UML profiles and Papyrus. The second one, after the experiment, to record how participants perceived the difficulty of completing each of the tasks. The second questionnaire uses a 10-point Likert response scale to measure their confidence for completing each of the tasks. Questionnairs are available in Appendix **??**

5.5 Procedure

Participants run the experiments individually. We identified eight main tasks required to build a custom editor for a Papyrus profile in the non-automated approach. For the Jorvik approach two main tasks are required. Each task was given a total allotted time for completion. Further, this time was divided into two sections: basic and essential. In the basic section participants are expected to complete the task only using their knowledge and any information available on-line or elsewhere. In the essential section, essential information[14] on how to complete the task is given to the participants.

Participants are asked to complete each of the tasks individually and the total time to complete the task is recorded. When a participant expresses that he/she has completed the task the timer is paused and an assessment of their progress is made. If the assets they have developed will not be sufficient to have a functional editor they are asked to continue working and the timer is resumed. After the basic time is up, if the participant has not completed the task, the timer is paused and the essential information (in the form of instructions written in paper) is provided. The timer is resumed and participants are allowed to continue working.

paper or digital?

Since each task depends on the previous being completed, at the beginning of each task, the participants are given a partial solution that contains complete and correct assets for all the previous tasks. The experiment was carried out in a single day in two sessions. Tasks 1-4 are scheduled in the morning and tasks 4-8 in the afternoon with a 1 hour break for lunch.

[14] The essential information is provided as a set of step-by-step instructions.

Table 6: Tasks and times for the Jorvik approach

Task	Total Time (m)	Basic Time (m)	Essential Time (m)
Profile Metamodel	60	40	20
Run Jorvik	60	40	20

Papyrus

Task	Time Given Default (Essential)	Total Time Web	Total Time FTA	Correctness Web	Correctness FTA	Participant	Remarks
UML Profile	40m (20m)	25m 30s	53m 23s*	●	●	#1	Notes.....
		22m 30s	51m 23s*	◐	○	#2	Notes.....
ElementTypes...	40m (20m)	...	...	...	...	...	...
		...	...	...	...	...	...
...	...	...	...	...	...	...	...
		...	...	...	...	...	...
...	...	...	...	...	...	...	...
		...	...	...	...	...	...
...	...	...	...	...	...	...	...
		...	...	...	...	...	...
...	...	...	...	...	...	...	...
		...	...	...	...	...	...
...	...	...	...	...	...	...	...
		...	...	...	...	...	...
Total Time	... (...)	...	...				

Jorvik

EMF + Annotations	30m (10m)	25m 30s	33m 23s*	●	●	#1	Notes.....
		22m 30s	21m 23s*	◐	○	#2	Notes.....

Legend

● : Correct	◐ : Partially correct	○: Incorrect	* : Essential Information Given

Table 7: Evaluation of Requirement Specification Languages as shown in [**?**]

Total allotted time for completion is based on timing the tasks while done by an expert on Papyrus and creating custom editors for profiles. The time given to the participants was double the time taken by the expert. The reason for this choice was to allow non-experts enough time and also limit the total duration of the experiment to prevent participant exhaustion. The basic/essential times ratio was picked to be 2:1 arbitrarily, but with the aim to give participants the opportunity to complete the task once the essential information was provided. This is assuming the participants did some progress during the basic window and only needed the essential information to finalize key details.

5.5.1 Results

5.5.2 Discussion Weaknesses/Threats!

6 Related Work

6.1 UML Profiles.

Building on the powerful concepts and semantics of UML, and its wide adoption in modelling artefacts of object oriented software and systems, UML profiles enable the development of DSLs by extending (and constraining) elements from the UML metamodel [11]. More specifically, UML profiles make use of extension mechanisms (e.g., stereotypes, tagged definitions and constraints) through which engineers can specialise generic UML elements and define DSLs that conform to the concepts and nature of specific application domains [36]. Compared to creating a tailor-made DSL by defining its metamodel and developing supporting tools from scratch, the use of UML profiles introduces several benefits including lightweight language extension, dynamic model extension, model-based representation, preservation of metamodel state and employment of already available UML-based tools [29]. Driven by these benefits, several UML profiles have been standardised by the OMG including MARTE [15] and SysML [10]) which are now included in most widely used UML tools (e.g., Papyrus [30]).

The flexibility and open-ended boundaries of UML profiles facilitated the development of profiles in applications as diverse as performance analysis [43] and Quality-of-Service investigation [6] in component-based systems, as well as context modelling in mobile distributed systems [37], Web applications [32] and smart homecare services [40]. In safety-critical application domains such as railway, avionics and network infrastructures, developed UML profiles support the specification and examination of security patterns [4, 34], analysis of intrusion detection scenarios [20], and modelling and verification of safety-critical software [3, 45],

Other researchers have designed UML profiles for the specification [7,31] and visualisation of design patterns [8]. Also, [38] proposes a methodology

for formalising the semantics of UML profiles based on fUML [18], a subset of UML limited to composite structures, classes and activities with a precise execution semantics. For an analysis of qualitative characteristics of several UML profiles and a discussion of adopted practices for UML profiling definition, see [33]. Likewise, interested readers can find a comprehensive review on execution of UML and UML profiles in [5].

Irrespective of the way these UML profiles were developed, either following ad-hoc processes or based on guidelines for designing well-structured UML profiles [11, 36], they required substantial designer effort. Also, the learning curve for new designers interested in exploring whether UML profiles suit their business needs is steep. In contrast, Jorvik automates the process of generating UML profiles using a single annotated ECore metamodel and reduces significantly the developer's effort for specifying, designing and validating UML Papyrus profiles (cf. Section 5).

6.2 Automatic Generation of UML Profiles.

Relevant to Jorvik is research introducing methodologies for the automatic generation of UML profiles from an Ecore-based metamodel [27]. The work in [28] 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 [42]. The methodology introduced in [13, 14] facilitates the derivation of a UML profile using a simpler as input. The methodology requires the manual definition of an intermediate metamodel that captures the abstract syntax to be integrated into a UML profile, and automates the comparison of this intermediate metamodel against the UML metamodel to automatically identify a set of required UML extensions, as well as the transformation of the intermediate metamodel into a corresponding functioning UML profile. Similarly, [27] introduces an approach for the automatic derivation of a UML profile and a corresponding set of OCL expressions for stereotype attributes using annotated MOF-based metamodels. Another interesting research work is JUMP [2] that support the automatic generate profiles from annotated Java libraries [2]. Despite the potential of these approaches, they usually involve non-trivial human-driven tasks, e.g., a UML profile skeleton [28] or an intermediate metamodel [13,14], or have limited capabilities (e.g., support of UML profile derivation with generation of OCL constraints [27]). In contrast, Jorvik builds on top of standard ECore metamodels that form the building blocks of MDE [16]. Furthermore, Jorvik facilitates the development of a fully-fledged UML profile and a distributable Papyrus graphical editor including the generation of OCL constraints and the definition of optional polishing transformations (cf. Section 4.11).

6.3 From ECore to UML profiles and back.

Jorvik 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 [42] simplifies the specification of mappings between a profile and its corresponding Ecore metamodel using a dedicated bridging language. Through an automatic generation process that consumes these mappings, the technique produces UML profiles and suitable model transformations. Along the same path, the approach in [12] employs an integration metamodel to facilitate the interchange of modelling information between Ecore-based models and UML models. Compared to this research, Jorvik automatically generates UML profiles (like [42] and [12]), but requires only a single annotated Ecore metamodel and does not need any mediator languages [42] or integration metamodels [12]. Also, the transformation of models from UML profiles to Ecore is only a small part of our generic approach (Section 4.10) that generates not only a fully-fledged UML profile but also a distributable custom graphical editor as en Eclipse plugin.

7 Conclusions and Future Work

In this paper we presented an approach towards automatic generation of UML profiles and supporting distributable Papyrus editors from an annotated Ecore metamodel. Our approach automatically generates the appropriate UML profile and all the needed artefacts for a fully functional Papyrus editor for the profile. In addition, it allows users to override/complement the built-in transformations to further polish the generated editor.

In the current version, although users are able to create compartments using an already existing compartment relationships in UML (e.g., Package-Class, Class-Property, 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. In the future, we plan to better support compartments by overriding the way in which these are displayed in Papyrus. In addition, in the current version we only support the automatic generation of OCL constraints for connectors for the *Association* base element. In the future more connectors, as well as opposite references will be supported.

Finally, the annotations provided in the Ecore file, might contain errors (e.g., users might point to a base class that does not exist). Validation scripts should be run against the Ecore file to check for such errors and produce meaningful error messages to the user in the future.

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".

We would like to thank Mr. Sina Mandani, Ms. Beatriz Sanchez Pina and Dr. Xiaotian Dai for their contributions to the evaluation of this paper.

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]
- bold: The label should be in bold font [optional - false by default]
- fontHeight: The font size of the label [optional - Papyrus default value if not provided]

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]
- 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]
- lineStyle: The style of the line (possible values: solid, dashed, dotted, hidden, double). [optional]
- bold: The label should be in bold font [optional - false by default]
- fontHeight: The font size of the label [optional - Papyrus default value if not provided]

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. Bernardi, S., Flammini, F., Marrone, S., Mazzocca, N., Merseguer, J., Nardone, R., Vittorini, V.: Enabling the usage of UML in the verification of railway systems: The dam-rail approach. Reliability Engineering & System Safety 120, 112 – 126 (2013)
4. Bouaziz, R., Coulette, B.: Applying security patterns for component based applications using uml profile. In: 15th International Conference on Computational Science and Engineering. pp. 186–193 (2012)
5. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of uml models: a systematic review of research and practice. Software & Systems Modeling pp. 1–48 (2018)
6. Cortellessa, V., Pompei, A.: Towards a uml profile for qos: a contribution in the reliability domain. ACM SIGSOFT Software Engineering Notes 29(1), 197–206 (2004)
7. Debnath, N.C., Garis, A.G., Riesco, D., Montejano, G.: Defining patterns using uml profiles. In: IEEE International Conference on Computer Systems and Applications. pp. 1147–1150 (2006)
8. Dong, J., Yang, S., Zhang, K.: Visualizing design patterns in their applications and compositions. IEEE Transactions on Software Engineering 33(7), 433–453 (2007)
9. Erickson, J., Siau, K.: Theoretical and practical complexity of modeling methods. Communications of the ACM 50(8), 46–51 (2007)
10. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to SysML: the systems modeling language (2014)
11. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to UML profiles. UML and Model Engineering 2 (2004)
12. 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)
13. 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)
14. Giachetti, G., Valverde, F., Pastor, O.: Improving Automatic UML2 Profile Generation for MDA Industrial Development, pp. 113–122 (2008)
15. Object Management Group: Modeling And Analysis Of Real-Time Embedded Systems. ONLINE (2011), http://www.omg.org/spec/MARTE/1.1/
16. Object Management Group: Meta object facility (MOF) core specification. ONLINE (2014), http://www.omg.org/mof/
17. Object Management Group: Unified Modeling Language. `http://www.omg.org/spec/UML/` (June 2015)
18. Group, O.M.: Semantics of a foundational subset for executable uml models. In: Technical Report (2010)
19. Haren, V.: Archimate 2.0 specification (2012)

20. Hussein, M., Zulkernine, M.: Umlintr: a uml profile for specifying intrusions. In: 13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06). pp. 8–pp. IEEE (2006)
21. Iacob, M.E., Jonkers, H., Lankhorst, M.M., Proper, H.A.: ArchiMate 1.0 Specification. Zaltbommel: Van Haren Publishing (2009)
22. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: Atl: a qvt-like transformation language. In: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. pp. 719–720. ACM (2006)
23. Kolovos, D., Paige, R., Polack, F.: The Epsilon Transformation Language. Theory and Practice of Model Transformations pp. 46–60 (2008)
24. 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)
25. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Rigorous methods for software construction and analysis. chap. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages, pp. 204–218. Springer-Verlag, Berlin, Heidelberg (2009), http://dl.acm.org/citation.cfm?id=2172244.2172257
26. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon object language (EOL). In: Model Driven Architecture–Foundations and Applications. pp. 128–142. Springer (2006)
27. Kraas, A.: Automated tooling for the evolving SDL standard: From metamodels to UML profiles. In: SDL 2017: Model-Driven Engineering for Future Internet - 18th International SDL Forum. pp. 136–156 (2017)
28. Lagarde, F., Espinoza, H., Terrier, F., André, C., Gérard, S.: Leveraging Patterns on Domain Models to Improve UML Profile Definition, pp. 116–130 (2008)
29. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: From UML profiles to EMF profiles and beyond. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. pp. 52–67. Springer (2011)
30. 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)
31. Mak, J.K.H., Choy, C.S.T., Lun, D.P.K.: Precise modeling of design patterns in uml. In: 26th International Conference on Software Engineering (ICSE'04). pp. 252–261 (2004)
32. Moreno, N., Fraternali, P., Vallecillo, A.: Webml modelling in uml. IET Software 1(3), 67–80 (2007)
33. Pardillo, J.: A Systematic Review on the Definition of UML Profiles, pp. 407–422 (2010)
34. Rodríguez, R.J., Merseguer, J., Bernardi, S.: Modelling and analysing resilience as a security issue within uml. In: 2nd International Workshop on Software Engineering for Resilient Systems (SERENE'10). pp. 42–51 (2010)
35. 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)
36. 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)

37. Simons, C., Wirtz, G.: Modeling context in mobile distributed systems with the {UML}. Journal of Visual Languages & Computing 18(4), 420 – 439 (2007)
38. Tatibouët, J., Cuccuru, A., Gérard, S., Terrier, F.: Formalizing execution semantics of uml profiles with fuml models. In: International Conference on Model Driven Engineering Languages and Systems. pp. 133–148. Springer (2014)
39. 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)
40. Walderhaug, S., Stav, E., Mikalsen, M.: Experiences from Model-Driven Development of Homecare Services: UML Profiles and Domain Models, pp. 199–212 (2009)
41. 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)
42. Wimmer, M.: A semi-automatic approach for bridging dsmls with uml. International Journal of Web Information Systems 5(3), 372–404 (2009)
43. Xu, J., Woodside, M., Petriu, D.: Performance analysis of a software design using the uml profile for schedulability, performance, and time. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. pp. 291–307. Springer (2003)
44. Zolotas, A., Wei, R., Gerasimou, S., Rodriguez, H.H., Kolovos, D.S., Paige, R.F.: Towards automatic generation of uml profile graphical editors for papyrus. In: European Conference on Modelling Foundations and Applications. pp. 12–27. Springer (2018)
45. Zoughbi, G., Briand, L., Labiche, Y.: A uml profile for developing airworthiness-compliant (rtca do-178b), safety-critical software. In: International Conference on Model Driven Engineering Languages and Systems. pp. 574–588. Springer (2007)