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

Automatic Generation of UML Profile Graphical Editors for Papyrus

Ran Wei[1,2], Athanasios Zolotas[2], Horacio Hoyos Rodriguez[2], Simos Gerasimou[2], Dimitrios S. Kolovos[2], Richard F. Paige[2,3]

[1] School of Artificial Intelligence, Dalian University of Technology, Dalian, China
e-mail: ranwei@dlut.edu.cn
[2] 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
[3] Department of Computer Science, McMaster University, Hamilton, Canada
e-mail: paigeri@mcmaster.ca

10 May, 2020

Abstract UML profiles offer an intuitive 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 and their supporting editors in Papyrus typically requires the developers to hand craft and maintain a number of interconnected models through a loosely guided, labour-intensive and error-prone process. We demonstrate how metamodel annotations and model transformation techniques can help manage the complexity of Papyrus in the creation of UML profiles and their supporting 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 editor 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* demonstrating the substantial productivity and maintainability benefits that *Jorvik* delivers.

Key words Model Driven Engineering, UML Profiling, Papyrus

1 Introduction

The Unified Modelling Language (UML) [19] 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* [13]. UML profiling enables the users to derive Domain-Specific Languages (DSL) from UML's set of general language concepts. The flexibility and open-ended boundaries of UML profiles facilitated the development of profiles in applications such as performance analysis [48], Quality-of-Service investigation [8] in component-based systems, as well as context modelling in mobile distributed systems [41], Web applications [36] and smart homecare services [44].

An important advantage of UML profiling for designing DSLs is that it allows the reuse of existing UML tools whilst exploiting readily, widely available UML expertise. The basic premise of UML profiling is that all domain specific concepts are derived as extensions or refinements of existing UML concepts (referred to as UML *meta-element*s). The extension mechanism is realised using UML *Stereotype*s, which extend standard UML *meta-elements* to describe user defined DSLs. Consequently, a profile-based UML 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 with less effort.

Although Domain-Specific Modelling Languages and tools that support them, like Sirius [43] or Eugenia [28], are becoming more popular, UML is still widely used in Model-Based Software Engineering (MBSE) [11]. As a result, alternative ways to define Domain-Specific Languages using dedicated metamodels and textual/graphical editors are available to the users [4, 37].

Papyrus [34] 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 of development, Papyrus is close to becoming 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 [12] and MARTE [17] offer implementations for Papyrus. As highlighted in the recent systematic survey on execution of UML models and UML profiles [7], Papyrus is the most widely used tool for developing UML profiles.

Despite the ability of Papyrus to support the development of non-trivial UML profiles, the learning curve and development effort required for developing these profiles is substantial. As reported in [47], the manual definition of new UML profiles is typically a tedious, time-consuming and error-prone process.

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

We evaluate the efficiency of *Jorvik* for the automatic generation of Archimate, a non-trivial enterprise modelling language, and its corresponding Papyrus editor against an equivalent manually created UML profile and its Papyrus editor. We then apply our approach on several other DSMLs of varying size and complexity [46], to demonstrate its generality and wide applicability. Furthermore, we evaluate the productivity improvement of *Jorvik* in a user study where developers are asked to create two UML profiles and their Papyrus graphical editors, both manually and with *Jorvik*. We report our findings, and the efficiency of *Jorvik* based on the results.

This paper extends the prototype approach for automated generation of UML profile graphical editors for Papyrus from our conference paper [49] in the following ways:

1. *Refactoring of Jorvik to adapt to the new underlying structure of Papyrus 3.0+.* Since Papyrus 3.0, a standard of creating Papyrus editors have been introduced (i.e. using the new Architecture design). This involves changes to Papyrus' underlying metamodels. In turn, the process of creating editors have changed significantly. We therefore re-implement a majority of our work to adapt the changes.
2. *Enhanced experimental evaluations by conducting user experiments.* Significant time has been spent on studying the efficiency of our approach, compared to the manual approach.
3. *Additional validations to check user defined DSLs.* Additional validation script provides useful feedback to the users to assist them in creating correct DSLs (with proper annotations).
4. *Support of more styling properties for diagrams created with our approach.* We add support for font styling and we add support for line styling.

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 while 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 evaluations we con-

ducted. In Section 6, we discuss related work and 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 its supporting graphical editing facilities in Papyrus. This process typically involves the manual creation of a number of inter-related models and configuration files. We motivate the need of automatic generation of these artefacts by highlighting the labour-intensive and error prone activities involved in creating a UML profile and its supporting Papyrus editor.

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, their properties and relationships among the elements.

Papyrus offers, among other choices, the mechanism of creating UML profiles using a *Profile Diagram* editor. Users can use 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*). The 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 *Stereotype*s 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. To do this, 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 *Stereotype*s 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*. As illustrated in Section 4.3, this 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 arguably labour-intensive and repetitive. In addition, 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. For the first option, 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 a new diagram is created. In addition, it cannot be shared in case the editor 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, hence its use is discouraged. The third option is to create a Papyrus editor associated to the UML profile, which includes the manual creation of a number of inter-related models and artefacts, including a palette configuration model, an ElementTypesConfiguration model, which is used to associate *Stereotype*s in the UML profile with its UML meta-element and the concrete syntax of the meta-element, etc. Although this option provides a whole solution to create a UML profile editor, in the paper we illustrate that it 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.

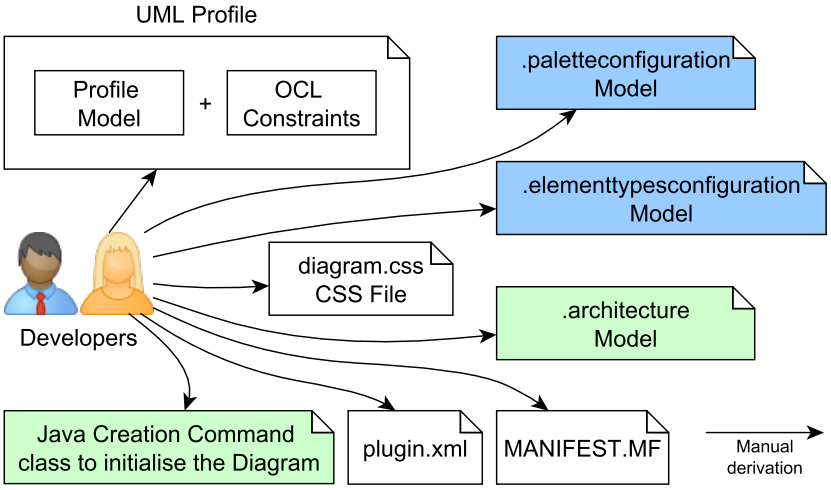

Fig. 1: Models/files developers need to write manually to develop a fully functional distributable Papyrus profile editor for Papyrus 3.0+

To create a distributable (i.e. an Eclipse plugin) 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. Figure 1 shows all the artefacts needed to be created for having a distributable Papyrus UML profile editor for Papyrus 3.0+[1]. These artefacts include:

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

Through our experiment (see Section 5), we found out that it is difficult, if not impossible, to create these models without any working examples,

[1] Metamodels for Element Types Configuration, Palette Configuration have been changed since our previous work (pre-Papyrus 3.0, 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).

taking also into account that the documentation of Papyrus provides limited useful insight in this matter.

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 and repetitive tasks (e.g., model creation) should be done. This is backed by our studies in the evaluation, which is discussed in Section 5.

As mentioned in [25], in different phases of the system engineering process, different metamodels created using the same modelling language could be used based on the current phase's needs. Papyrus opted for the option that allows the definition of UML profiles based on the UML specifications defined by the OMG standard. As a result, it may be the case that problems, when using tools that implement UML, arise due to the use of an inappropriate version of the metamodel by the tool vendors. Examples of solutions in tackling such problems that are a result of potential design flaws in OMG specifications are presented in [3] and [26]. However, our personal experience, which is verified by the evaluation results, shows that the problems with the creation of UML profiles and graphical editors in Papyrus, arise from the labour-intensive and repetitive tasks related mostly in the definition of all the other artefacts (which are a result of a design decision by the tool vendor) rather than the UML profile itself. We argue that this *labour-intensive*, *repetitive* and *error-prone* tasks 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 the generation of a distributable Papyrus editor that supports the UML profile.

3 Proposed Approach

In this section we present *Jorvik*, an automatic generation tool for Papyrus UML profiles and Papyrus editors that support modelling with them. Through *Jorvik*, developers can define the abstract syntax and the concrete (graphical) syntax of their DSLs (i.e., 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 models/artefacts needed for a distributable Papyrus editor. This is done through 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.

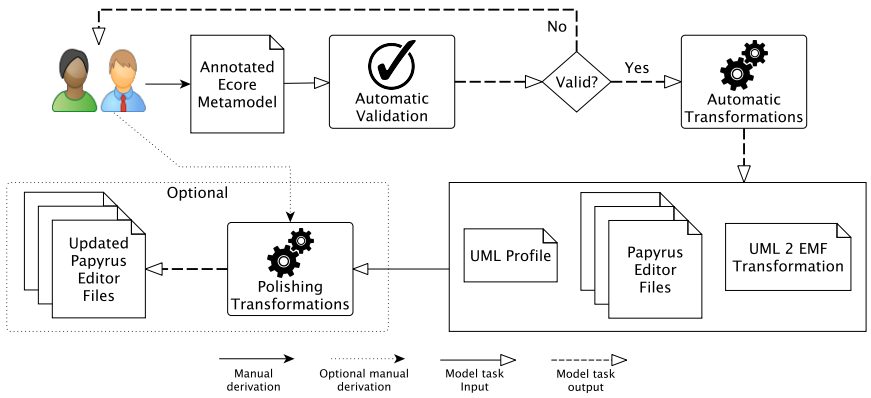

Fig. 2: An overview of the proposed approach

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[2]. 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 EClasses are automatically transformed into *Stereotype*s. *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 annotations on EClasses/EReferences (i.e., @Node or @Edge, respectivelly). A detailed list of all valid annotation properties is provided in Appendix A.

[2] Due to the vast amount of edge types in UML, we currently support only the following types: Association, Dependency, Generalization, Realization, Usage and InformationFlow

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) [29] (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 metamodel is 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) [27] and the Epsilon Generation Language (EGL) [39]. In principle, any other model validation, M2M and M2T language could be used (e.g., ATL [24] and Acceleo[3]). 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 (in cases where the users change their UML profiles, they wish to propagate the changes back to their EMF metamodels). The benefit of adopting this process is that model management programs already developed to run against models conforming to the EMF metamodel can be re-used.

Jorvik offers the option of *polishing transformations*, where developers are able to write their own (optional) transformations to fine-tune the generated artefacts produced by the built-in transformations. In the following section, the proposed approach is explained via a running example.

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[4], a textual notation for Ecore, which is shown in Listing 1 (ignore the annotations at 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. Due to the relevance of our work with Eugenia [28], a tool that

[3] `https://www.eclipse.org/acceleo/`

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

```
1  @namespace(uri="sdpl",prefix="sdpl")
2  @Diagram(name="SDPL", icon="ic/sdpl.png")
3  package Process;
4
5  @Node(base="Class", shape="sh/step.svg", icon="icons/step.png")
6  class Step {
7    attr String stepId;
8    ref Step[1] next;
9  }
10 @Node(base="Class", shape="sh/tool.svg", icon="icons/tool.png", bold="
       true")
11 class Tool {
12   attr String name;
13   attr int version;
14 }
15 @Node(base="Class", shape="sh/per.svg", icon="icons/per.png")
16 class Person {
17   attr String name;
18   attr int age;
19   @Edge(base="Association", icon="icons/line.png", fontHeight="15")
20   ref Tool[*] familiarWith;
21 }
22 @Edge(base="Association", icon="icons/line.png", source="src", target="
       tar")
23 class Role {
24   attr String name;
25   ref Step[1] src;
26   ref Person[1] tar;
27 }
```

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.

uses the same principles for the generation of GMF editors and inspired our work, and to be as consistent as possible, the syntax of our annotations match those in Eugenia, where possible.

- **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 DSL elements (i.e., 'Steps", 'Roles" and 'Tools") should be *Stereotype*s in the UML profile that will be represented as nodes in the diagram. The *base* parameter is used to define which UML meta-element the *Stereotype* should extend (i.e., Class in this example[5]). The shape of

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

the node in the diagram 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).

- **Lines 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* annotation. 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. Note in here we use keywords *source* and *target* to denote the origin of the Association (*ownedEnd*) and the destiny of the Association (*memberEnd*). We do this to avoid confusions to EMF users (as EReferences may naturally map to Association). However, there are other types of edges that we support, i.e. Dependency, Generalization, Usage, Realization and InformationFlow. These types are UML::DirectedRelationship and therefore also have *source* and *target*. It is to be noed that the *@Edge* annotation is applicable to EReferences with any multiplicity. In line 19, we set the font height to 15 for the labels of the 'familiarWith" edges.
- **NB:** In 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 are then executed on the Ecore file and the produced SDPL Papyrus editor (with supporting palette and custom shapes for the SDPL profile) is presented in Figure 3.

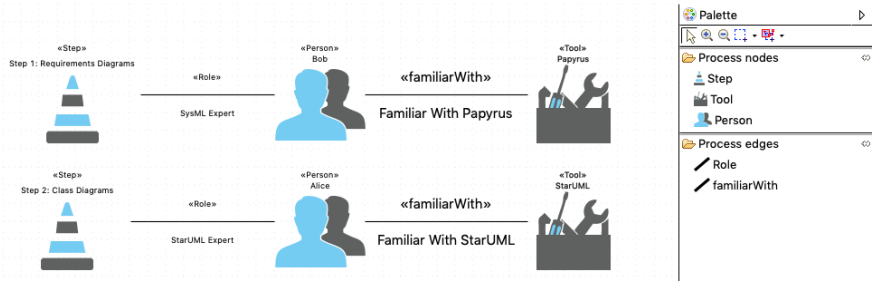

Fig. 3: The SDPL editor for Papyrus where two steps in the software development process are defined and responsible persons are attached to them along with the tools they are specialized on.

3.2 Polishing Transformations

The generated editor is fully functional but it can be further customised to fit the 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, users 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 executed, which will set the visibility of the *Stereotype*s to false automatically.

```
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.

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

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

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

The EGL template in Listing 2 generates a CSS rule in lines 1-3 that sets the visibility property of the *Stereotype*s' 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 *Stereotype*s, 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 shown in Listing 4.

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

Listing 4: 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 workflow of *Jorvik*. Each step in the workflow is identified by a number (#1 – #9 in Figure 4) for easier reference. Before generating all the artefacts, 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 Plug-in are also generated while icons and shapes are placed next to the annotated metamodel (#8 in Figure 4). As the transformations consist of about 1000 lines of code we will describe them omitting low level technical details[6].

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 is

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

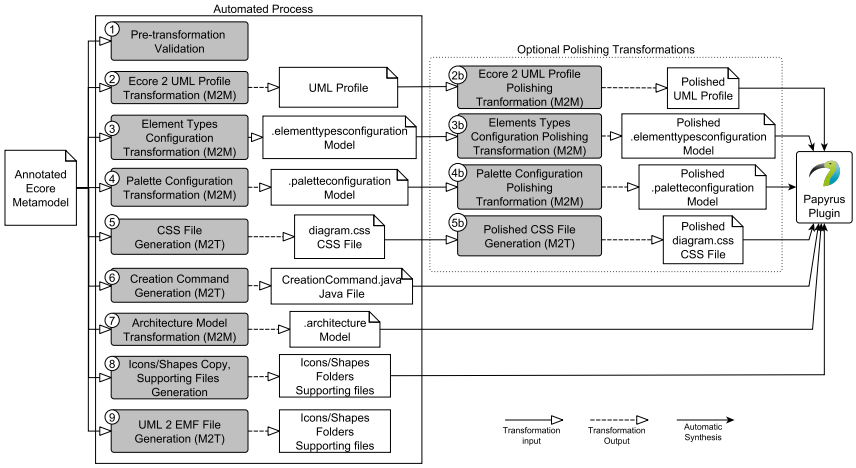

Fig. 4: An overview of the transformation workflow

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 5 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) [7].

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

Listing 5: 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 checked.

[7] Interested readers can find the Ecore metamodel that includes some of the elements used in this EVL example and other listings of this paper in `http://www.kermeta.org/docs/org.kermeta.ecore.documentation/build/html.chunked/Ecore-MDK/ch02.html`

#	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
11	Edge annotated elements have different names	There are no elements annotated as @Edge that have the same name

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

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 that are annotated as @Edge in the Ecore metamodel. No attributes are added to the *Stereotype* as EReferences do not support attributes in Ecore.

When all *Stereotype*s 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*, where *ref* denotes a non-containment reference and *val* denotes a containment reference) 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* 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:

– **End Types**: one of the elements a 'familiarWith" association connects to, has the 'Person" steretotype applied to it while the other has the 'Tool" *Stereotype*s applied to it;

[8] The attributes of the *Stereotype*s are omitted for simplicity.

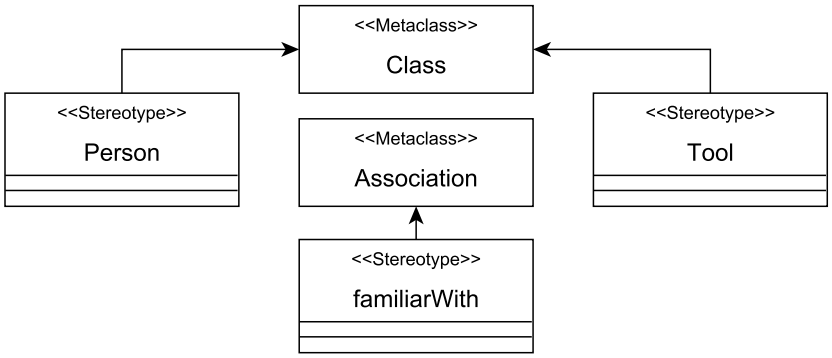

Fig. 5: Example UML profile for SDPL showing Person, Tool and the familiarWith association

– **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 6 shows the OCL code for the *End Types* constraint[9]. 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 either a 'Person" *Stereotype* or a 'Tool" *Stereotype*. In this way, if a 'familiarWith" association connects two types that are not 'Person" or a '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 6: The End Types constraint in OCL

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

[9] Please refer to Section 11.8.1 in the UML2.5.0 specification for properties of UML::Association `https://www.omg.org/spec/UML`

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

Listing 7: The Navigability constraint in OCL

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)[10].

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 element of type *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

[10] For detailed explanation for the ElementTypesConfiguration framework of Papyrus, please refer to Papyrus Guide/Toolsmith Guide/ElementTypeConfiguration Framework in Papyrus official documentation

model[11]. In addition to *SpecializationTypeConfiguration*s, for each *Stereotype* element 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 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, each *ToolConfiguration*, needs 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* element and determine the UML syntax it specialises.

4.6 CSS File Generation (#5)

As stated above, the look and feel of diagram elements in Papyrus can be customised using CSS. In this transformation we generate our default 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, shapes, 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

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

CSS. Finally, we generate the CSS rules for each edge, e.g., if a *lineStyle* parameter is set, then the *lineStyle* 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 UML diagram (with the user-defined UML profile).

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 written in the Epsilon Object Language (EOL) [30]. The program needs 1) the annotated Ecore metamodel for the DSL, 2) the Element Types Configuration model and 3) the Palette Configuration model thus it should be executed after the transformations that generate the latter 2 artefacts (i.e., Steps #3 and #4). 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 *ArchitectureDescriptionLanguage* points to the Element Types Configuration and the Creation Command class. The *PapyrusDiagram*s point to the Palette Configuration model and the CSS file.

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 its supporting editor in either a new Eclipse plug-in project or in the same Eclipse 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 '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 transformation generated 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.

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[12].

4.10 UML to EMF Transformation Generation (#9)

```
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 8: 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.

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

[12] The diagram layout information cannot be generated and is not related to this work

source Ecore metamodel given as input to the approach. The reason behind this is to allow the users to propagate any changes they make in their transformed UML profiles back to their EMF models. One rule is generated for each of the *Stereotype*s 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 8. 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.

ETL provides the ::= operator for rule resolution. 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 8), 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 (#2b - #5b, whereas #2b - #4b are model-to-model transformations and #5b is a model-to-text transformation) 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.

4.12 Adding support for nested relations

In EMF, a reference between two classes can be flagged as a containment relation, which is consistent with the *containment* definition of UML associations (with exception of the deletion cascade mechanism). These types of relations, when presented visually, can benefit from showing the contained elements shapes inside the shape of the container, as opposed to the

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

Table 2: Polishing Transformations File Names

line/arrow presentation. For example, if a `package` is represented with an empty rectangle, classes contained in the package would appear inside this rectangle.

Ideally, a custom profile editor should allow this containment relations to be represented as visual containment too. Papyrus does allow providing custom shapes for elements, so we explored the feasibility of supporting visual containments too. However, the current structure of the Papyrus visual editor does not allow this functionality. Currently, the *shape* concept at the editor level provides separate graphical sections for the custom shape and the contained elements. These two sections are presented visually one after the other. Hence, with the current Papyrus implementation it is not possible to have nested elements inside a custom shape.

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 [21,23]. We use the Adocus Archimate for Papyrus[13] (an open-source tool that includes a profile for Archimate and the appropriate editors for Papyrus) for reference. Archimate is an open and independent enterprise architecture modelling language to support the description, analysis and visualisation of architecture within and across business domains in an unambiguous way. 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 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 [46]. 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 ask participants to build Papyrus editors for two UML Profiles. We first ask the participants to create the profiles and editors manually, and then ask them

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

to create the same profiles and editors using *Jorvik*. We measure the time, report problems encountered during the experiment for both approaches 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 diagram 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 all five Archimate diagrams. 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. Finally, 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 transformations. We do not go into details of the polishing transformations for this specific example[14].

In our previous work [49], we compared our approach with Archimate for Papyrus. However, as we mentioned in Section 2, Papyrus changed its underlying metamodels and the mechanism for creating UML specific editors. To ensure that our results are still valid for Papyrus 3.0+, we re-generated all the Archimate editors using *Jorvik*. We add the lines of code needed for *Jorvik* to our findings in the previous work.

Table 3 summarises the efficiency of *Jorvik*, both for *Jorvik* pre-Papyrus 3.0 version and for *Jorvik* post-Papyrus 3.0 version[15]. To make a fair comparison, we count both the lines of code and the number of model elements in each artefact that constitutes to a working editor. Hence, the numbers in Table 3 are shown in the format of *Lines of Code/Number of Model Elements*. For artefacts which are not models (e.g., the CSS file) we only provide

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

[15] Cells in gray are artefacts not needed for implementation. E.g. Creation Command and Architecture Model are concepts in Papyrus version 3.0+, and therefore are not applicable to *Jorvik* pre-Papyrus 3.0 version and Archimate for Papyrus

the lines of code metric as well for artefacts created by polishing transformations in *Jorvik*, as these were generated by the polishing transformation scripts.

For *Jorvik* pre-Papyrus 3.0 (columns under *Jorvik (pre-Papyrus 3.0)*), we need to manually create five annotated Ecore metamodels, which involves writing 436 lines of code in Emfatic, which is equivalent to 668 model elements. For polishing transformations, we need to write 11 lines of code in the transformation script for Element Types Configuration, 50 lines for Palette Configuration, 195 lines for CSS and 10 lines for Types Configuration. For *Jorvik* post-Papyrus 3.0 (columns under *Jorvik (post-Papyrus 3.0)*), we need the same Ecore metamodels (i.e., five), thus the numbers do not change. For polishing transformations, we need to write 50 lines for the Palette Configuration and 195 lines for CSS.

It can be observed from the numbers, for *Jorvik*, we create 63.5% less objects and we write 63.7% code in CSS in order to produce editors that matches the original Archimate for Papyrus editors. In addition to the working editors (which offer the same functionalities and features as the original Archimat for Papyrus tool), *Jorvik* also produces the OCL constraints for the profiles, as well as the ETL transformations which allows the interoperability from UML profiles to annotated EMF metamodels.

5.1.1 Threats to Validity There are 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 creation wizards. For those to be created, the developers needs to extend the 'plugin.xml' file. In addition, the line decoration shapes of *Stereotype*s 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 for a fair comparison.

	Jorvik (pre-Papyrus 3.0) LoC/Number of Model Elements			**Jorvik (post-Papyrus 3.0)** LoC/Number of Model Elements			**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	**436/668**	**266**	**702/668**	**436/668**	**245**	**681/668**	**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 test *Jorvik* with nine more Ecore metamodels from different domains. The names of the metamodels (including Archimate) and their size (in terms of types) are provided 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.

As illustrated in Table 4, the metamodels vary in size, from small profiles (with 5 *Stereotypes*) to large profiles (with up to 57 *Stereotypes*). The approach is able to produce the profiles and the editors for *all* metamodels, demonstrating that it can be used to generate the desired artefacts for a wide spectrum of domains. The time needed for the generation varies from miliseconds up to a few seconds. In the future, we plan to assess further the scalability of our approach using larger metamodels.

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)

5.3 User Experiment

We have argued that *Jorvik* provides significant gains in productivity when building custom UML Profile editors for Papyrus. We design a user experiment to substantiate our claim and quantify the productivity improvement. As discussed in Section 4, there are eight major steps to be taken in order to create a UML profile as well as its supporting editor. In this experiment, we compare the time needed to develop an editor using Papyrus infrastructure (hereby referred to as the *Papyrus approach*) with the time needed to develop the same editor using *Jorvik*. For the Papyrus approach we design eight tasks, each with its own deadline (see Table 5) for the participants to complete towards manually creating a UML profile and a working UML editor for the profile. For *Jorvik*, we design one task for the participants to complete to automatically generate a UML profile and a working UML

editor for that profile. We ask two participants to take part in the experiment and work on two profiles we choose. We record the time taken for the participants to complete the experiment using both approaches and we compare the times.

5.3.1 Papyrus Approach Experiment Set-Up For the purpose of this experiment, we have chosen a participant with relatively more experience in modelling (hereby referred to as Participant 1), and a candidate with less experience in modelling (hereby referred to as Participant 2). Both participants have an Eclipse IDE installed on their computers, with Eclipse Epsilon 1.6 Interim version[16] and Eclipse Papyrus 4.0.0[17] installed. We ask the participants to perform the tasks involved in the Papyrus approach firstly on one profile (the Website profile[18]) and then repeat the experiment for a second profile (the Fault Tree profile[19]). The SVG shapes and icons for both cases are provided to the participants. Before the experiment is conducted, a pre-experiment questionnaire is handed to the participants, to assess their expertise in UML, UML profiles and Papyrus[20]. In addition, a 20-30 minutes introduction to UML profiles and Papyrus is given to them while an example of a custom UML profile Papyrus editor is being presented to them.

For each of the eight steps, there is a set deadline; the participants are asked to try to complete the step within the deadline. The tasks and the deadlines are derived from our own experience in developing an UML profile and its distributable Papyrus editor. Initially we spent 3 months on creating an example editor, due to the lack of documentation and the lack of tool support when referencing model elements among the models required for the editor. After we found out how to create an editor, we recorded the amount of time required for us to perform the eight steps to derive the deadlines. We then normalise the deadlines through a pilot study with a volunteer from our research group (we also make adjustments to our experiment set-up in the pilot study based on what we learnt from it).

In each step, the participants are asked to complete a minimal task first (e.g., for UML profile, create a *Stereotype* that is displayed as a node and a *Stereotype* that is displayed as an edge)[21]. They are asked to continue with the rest if there is still time left. If the participants miss the deadline but they are working towards the correct solution, they are asked to give an estimate of how long they believe it would take them to finish the whole step.

[16] `https://www.eclipse.org/epsilon/`
[17] `https://www.eclipse.org/papyrus/download.html`
[18] `https://github.com/wrwei/Jorvik/tree/master/org.papyrus.website`
[19] `https://github.com/wrwei/Jorvik/tree/master/org.papyrus.faulttree`
[20] The questions can be viewed in `https://github.com/wrwei/Jorvik/wiki/Pre-Experiment-Self-Assessment-Questionnaire`
[21] Detailed descriptions of the tasks can be found at `https://github.com/wrwei/Jorvik/tree/master/User_Experiment`

Table 5: Tasks and times (in minutes) for the Papyrus approach.

Task	Total (m)	Default (m)	Essential (m)
1. UML Profile	60	40	20
2. Element Types Configuration Model	60	40	20
3. Palette Configuration Model	30	20	10
4. Cascading Style Sheet	30	20	10
5. Creation Command	30	20	10
6. Architecture Model	40	30	10
7. Plug-in Configuration	20	12	8
8. OCL Constraints	60	40	20

At the beginning of each step, we provide a piece of *Default* knowledge[22], which covers ground knowledge for the step to be completed. Participants are also allowed to search for any information over the Internet which may assist them in their tasks at any point during the experiment. At a certain point for each step, we assess if the participants are able to complete the step within the time frame, and we provide a piece of *Essential* knowledge[23], which contains key information (which is not accessible on the Internet) for the participants to complete the step.

Table 5 lists an overview of the tasks. It also includes the times (in minutes) for the total time given (i.e., Total (m)) and the deadlines (in minutes) for the task with the default (i.e., Default (m)) and the essential (i.e., Essential (m)) knowledge. The task descriptions are as follows:

1. UML Profile - An image of a UML profile is provided to the participants, they are required to create the profile within 60 minutes.
2. Element Types Configuration Model - Participants are asked to create an Element Types Configuration model for the editor for the profile they create in task 1.
3. Palette Configuration Model - Participants are asked to create a Palette Configuration model for the editor/profile.
4. Custom Style - Participants are asked to create a CSS file to customise the styles of the editor.
5. Creation Command - Participants are asked to create the creation command Java class to initialise the Papyrus diagram .
6. Architecture Model - Participants are asked to create an Architecture model for the editor/profile.
7. Plug-in Configuration - Participants are asked to configure their plug-ins in order to make use of all the models/artefacts to form a working editor
8. OCL constraints (optional) - In this optional task, participants are asked to create OCL constraints mentioned in Section 4.3 for all connector

[22] See an example at `https://github.com/wrwei/Jorvik/blob/master/User_Experiment/Step1_Default.pdf`

[23] See an example at `https://github.com/wrwei/Jorvik/blob/master/User_Experiment/Step1_Essential.pdf`

*Stereotype*s, within 60 minutes. We do not expect this task to be taken by participants, as it typically requires experienced MDE practitioners 2 weeks to complete the constraint templates described in Section 4.3. For the record, no participants agreed to take this optional task.

For each step, when participants express that they have completed the step, we stop the timer and assess the solutions. If the solutions are not correct, we tell the participants that they need more work and resume the timer. Since each step depends on the previous being completed, at the beginning of each step, the participants are given a solution that contains complete and correct assets for all the previous tasks.

After participants have completed the manual process, a post-experiment questionnaire is handed to them, in which they evaluate the difficulties of each task and if enough time was provided for each task[24].

5.3.2 Jorvik Experiment Set-up The experiment then proceeds to the use of *Jorvik*, where the participants need to complete one task: **Annotated Ecore Metamodel and Generation** - Participants are provided with the same images of the profiles (one each time), they are asked to create an annotated Ecore metamodel for the profile and generate the UML profile and its supporting Papyrus editor, within 50 minutes. The *Default* information is provided at the beginning. 30 minutes in the task, we assess the participants' status and provide the *Essential* information.

5.3.3 Results Table 6 shows the times obtained from the user experiment. Participant #1 is a PhD student who has a high level of expertise in modelling, and has used Papyrus before. Participant #2 is a post-doctoral researcher who has an intermediate level of expertise in modelling, and has used Papyrus on a limited number of occasions. Both of the participants have no experience in creating distributable editors for UML profiles using Papyrus.

In the table, the *Task* column specifies the name of the steps. The *Time Given* column specifies the time we give the participants for each step having the *Default* knowledge only and in parenthesis the time we gave to them having the *Essential* knowledge information. The *Time Taken* column records the time taken for the participants to complete the Website profile (*Web*) and the Fault Tree profile (*FTA*). Times with an *asterisk (*)* denote that the participant asked and was given the essential information[25]. The *Correctness* column records if the participants are able to provide correct,

[24] The questions can be viewed in `https://github.com/wrwei/Jorvik/wiki/Post-Experiment-Self-Assessment-Questionnaire`

[25] In some tasks participants knew how to complete the task with no more information given to them, but they hit the deafult knowledge deadline. In such cases we were assessing the solution and if it was indeed towards the correct direction they were allowed to use the time remaining to complete the task without giving the essential information.

partial correct (e.g. participants miss some style properties in the CSS file) and incorrect solutions. The participants are distinguished using the *Participant* column. We record any comments/remarks made by the participants in the *Remarks* column (se discuss the remarks made by the participants later in this section).

	Task	Time Given Default (Essential)	Time Taken Web	Time Taken FTA	Correctness Web	Correctness FTA	Participant	Remarks
Papyrus	1. UML Profile	40m (20m)	25m 30s	15m	●	●	#1	–
			40m	24m	◐	●	#2	–
	2. Element Types Configuration	40m (20m)	56*	38m	●	●	#1	①
			60m*	58m*	○	○	#2	②
	3. Palette Configuration	20m (10m)	26m30s*	25m	●	●	#1	③
			30m*	30m*	○	◐	#2	④
	4. Custom Style	20m (10m)	15m30s	16m	◐	●	#1	–
			30m*	19m	◐	◐	#2	–
	5. Creation Command	20m (10m)	25m*	16m	○	●	#1	⑤
			30m*	25m	○	●	#2	–
	6. Architecture Model	30m (10m)	40m*	25m	○	●	#1	⑥
			40m*	40m*	○	◐	#2	⑦
	7. Plug-in Configuration	12m (8m)	20m*	5m	○	●	#1	⑧
			19m	10m	●	●	#2	–
	Total	182m (88m)	208m30s	140m			#1	
			249m	206m			#2	
Jorvik	EMF + Annotations	30m (20m)	32m 30s*	14m 51s	●	●	#1	–
			36m 55s*	37m 27s*	●	●	#2	–

Legend

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

Table 6: Times (in minutes) obtained from the Papyrus approach and the Jorvik approach experiments.

Below is an example of how the table should be read (the summary of the experiment for Participant 1 for the Papyrus approach for the Website profile):

1. Participant 1 was able to finish the UML profile creation in 25 minutes without the *Essential* knowledge.
2. Participant 1 finished the Element Types Configuration in 56 minutes with the help of the *Essential* knowledge.
3. Participant 1 finished the tasks in Step 3 in 26 minutes with the help of the *Essential* knowledge.
4. Participant 1 finished a partial solution for the CSS in 15 minutes.
5. Participant 1 could not figure out how to create a creation command, therefore *Essential* knowledge is provided, and she finished the step in 25 minutes in total.
6. Participant 1 could not figure out how to create an Architecture model, even with the *Essential* knowledge provided and missed the deadline.
7. Participant 1 was not able to configure the editor Plug-in to successfully run the editor, even with the *Essential* knowledge.

We also note some remarks made by the participants during the experiment. Below is a list of the description of the remarks in the table, which should be read together with the experiment results:

- Remark ①: In the Website experiment, in Step 2, Participant 1 claims that she found a solution online[26] that made the task significantly easier. She also claims that without the solution there is no way he/she could have finished the task, even with the *Essential* knowledge.
- Remark: ② In both the Website and the Fault Tree experiment, in Step 2, Participant 2 claimed that he could never complete the step, without the essential information. He also claims that the Element Types Configuration is rather confusing. In the Fault Tree experiment, he claims he would need 40+ minutes to complete the whole model.
- Remark ③: In the Website experiment, in Step 3, Participant 1 finishes the minimal task with the help of the essential information. She claims that she would need 20 more minutes to finish the model.
- Remark ④: In the Website experiment, in Step 3, Participant 2 misses the deadline even with the essential information. He claims that he would need 30 more minutes to finish the step.
- Remark ⑤: In the Website experiment (and presumably in the Fault Tree experiment), in step 5, participant 1 claims that she copied the actual solution for the essential information provided to her.
- Remark ⑥: In the Website experiment, in step 6, participant 1 misses the deadline even with the essential information. She claims that the tool support for the Architecture model by Papyrus is not well implemented (it does not support the reference to model elements in other models).

[26] Which is the forum thread where the authors obtained the correct way of creating Element Types Configurations: `https://www.eclipse.org/forums/index.php/t/1096471/`.

- Remark ⑦: Participant 2 in both rounds of this experiment claims that he finishes the step before the deadline (both with the help of the essential information), but he could not get the solutions right. This is typically due to the fact that there are somewhat confusing model elements in the Architecture metamodel by Papyrus.
- Remark ⑧: In the Website experiment, in step 7, Participant 1 cannot configure the plug-in to a working order, she claims that she would need more than 20 minutes to inspect other models to find out what went wrong.

For the results obtained using *Jorvik*, participant 1 is able to generate the correct Papyrus editor for the Website profile in 32 minutes. She is also able to create the correct editor for the Fault Tree profile in 15 minutes. Participant 2 needs 37 minutes and 38 minutes for the creation of a correct Papyrus editor for the Website and the Fault Tree profiles, respectively, using *Jorvik*.

5.3.4 Analysis We begin our analysis with useful insights from the responses to the pre and post-experiment questionnaires. From the pre-experiment questionnaires[27], we found out that both participants had intermediate knowledge of UML but have not created a UML profile in the past using any tool (including Papyrus).

By analysing the responses to the post-experiment questionnaires[28] for the Papyrus approach, both participants feel that the time assigned to the tasks, most of the times, is not enough for the first round of the experiment (the Website experiment). However, they feel the time was enough for the second round of the experiment (the Fault Tree experiment). This is because participants are able to refer to their Website solution in the second round. Both participants mention it is difficult to find the documentation needed to finish the steps (*NB*: this applies only to responses after the first experiment as they mostly referred to the editor produced in that one when executing the second).

For all the steps in the Papyrus approach, except the one in which participants have to create the UML profile, they declare that they have low to moderate confidence that they will be able to complete the task before receiving the essential information. This can be explained from the lack of experience in developing UML profile editors with Papyrus before. For the same steps (i.e., 2-7), participant 2 highlights that he feels completely lost before receiving the *Essential* information. However, both of them declare to have moderate to high confidence after receiving the *Essential* information, except for step 2, where both are still confused by the concept of Element Types Configuration. This confirms our findings as this task is the one that participants perform the worst (see Table 6).

[27] Available at `https://github.com/wrwei/Jorvik/blob/master/User_Experiment/Pre-Experiment_Self_Assessment_Responses.pdf`

[28] Available at `https://github.com/wrwei/Jorvik/blob/master/User_Experiment/Post_Experiment_Self_Assessment_Responses.pdf`

To summarise, from the responses received, step 2 (Element Types Configuration), step 3 (Palette Configuration), Step 5 (Creation Command) and step 6 (Architecture Model) are identified as typical obstacles for the participants in completing the whole experiment.

Regarding the questionnaires from the *Jorvik* experiment, participant 1 describes herself as knowledgable of EMF, has created EMF metamodels in the past, and has also used annotations. Participant 2 had limited EMF experience in the past and has created a limited number of EMF metamodels. Finally, both mention that the time given is enough and that they feel more confident after receiving the essential information.

By analysing the results provided in Table 6, comparing the Papyrus approach with *Jorvik*, we can conclude that using *Jorvik*, the users are able to increase the productivity by at least 10 times (especially when participants claim that they would need additional time to finish the complete solution for some steps). This is also due to the fact that both participants chose not to complete the optional Step 8 (the OCL constraints), which may take significant amount of time, even for experienced OCL programmers.

We are also able to draw the conclusion that it is rather difficult to derive the models/artefacts needed for a working UML profile-specific editor. This is based on the experiment results that both participants get the majority of their models/artefacts wrong for the Website profile, which is the first profile and editor they work on. Although we provide the *Essential* information, parts of which, to the best of our knowledge, are not available in Papyrus documentation, participants still cannot get the models/artefacts right because of the inter-related nature of the models. For example, both participants find it difficult to comprehend the purpose of the Element Types Configuration, they actually find that it is the most challenging part of the experiment. The candidates also find it difficult to link creation tools in the Palette Configuration model to elements in the Element Types Configuration. They also find it hard to understand the rationale behind referencing to Element Types Configuration. Finally, both participants claim that it is rather difficult to create the Architecture model, as there are concepts defined in it which purposes are not orthogonal to their experience. In addition, due to the inter-connected nature among the models/artefacts, participants find it difficult to debug their solutions, as there are many places where things may go wrong. In contrast, *Jorvik* provides feedback based on the validation rules applied to the annotated Ecore metamodel, which helped participants in debugging. In practice, during the *Jorvik* experiment, we notice that participants make use of the feedback provided by *Jorvik* to debug their annotated Ecore metamodels.

When participants work on the Fault Tree profile, they are able to refer to their solutions to the Website profile. Therefore we observe that the correctness of the models/artefacts for the Fault Tree profile improves significantly comparing to the Website profile. This matches our experience with using the Papyrus infrastructure for the development of distributable editors for UML profiles, where we have to reverse engineer other editors avail-

able online to try to understand how to proceed. Although they may have adapted their Website solutions to their Fault Tree solution, the recorded time shows that *Jorvik* still performs significantly better than the manual process, especially taking the OCL constraints into consideration.

It is worth mentioning that Participant 1 is an expert user of Ecore, where Participant 2 only used Ecore from the training provided prior to the experiment. We observe the advantage of being familiar with Ecore based on Participant 1's time taken for the experiments (especially after she got familiar with the annotation rules for *Jorvik*). However, for both levels of expertise in Ecore, the experiment suggests that the time taken is still significantly better using *Jorvik* compared to the Papyrus approach.

5.3.5 Threats to Validity For the experiment we invite participants that do not specialise in creating UML profiles and their supporting editors for Papyrus. The time taken if our participants were Papyrus experts might be lower. However, as we have spent significant time working with Papyrus and *Jorvik*, we have run the experiment ourselves for both the profiles and we observe the same time benefits (about 10 times faster using *Jorvik*).

In the Papyrus experiment, we take a waterfall approach. We derive the eight steps in the Papyrus experiment based on our own experience, each step depends on previous steps. For example, step 2 (Element Types Configuration Model) depends on the whole solution of step 1 (the UML profile), step 3 (Palette Configuration Model) depends on step 2. There is one exception that step 5 (Creation Command) depends on step 7 (Plug-in Configuration) as the Creation Command Java class relies on the definition of URI mappings, required in the plug-in. However, this does not affect the experiment results, as when participants work on step 7, we notify them that they need to also alter their solution in step 5.

The Jorvik experiment runs in both cases after the Papyrus one. Participants may be familiar with the domain described in the metamodel after finishing the Papyrus experiment and this could reduce the time for understanding the domain in the Jorvik experiment. However, the knowledge of the domain described in the metamodel is mostly useful when constructing the UML profile. This was the task that the participants perform better in the Papyrus experiment, thus we do not believe that this has any (significant) impact in the results presented.

Finally, participants require the whole solution of the editor in order to test the correctness of the models/artefacts produced in each step. Both participants find that it is difficult to test the models they develop, because it requires the whole solution in order to test a single model. We do not consider this a flaw in our experiment as it is a replicate of our own experience. In addition, to mitigate this issue, we performed a manual review of the models/artefacts when it is requested to inspect their correctness.

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 [13]. More specifically, UML profiles make use of extension mechanisms (e.g., *Stereotype*s, 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 [40]. 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 [33]. Driven by these benefits, several UML profiles have been standardised by the OMG including MARTE [17] and SysML [12]) which are now included in most widely used UML tools (e.g., Papyrus [34]).

In safety-critical application domains such as railway, avionics and network infrastructures, developed UML profiles support the specification and examination of security patterns [6, 38], analysis of intrusion detection scenarios [22], and modelling and verification of safety-critical software [5, 45, 50].

Other researchers have designed UML profiles for the specification [9,35] and visualisation of design patterns [10]. Also, [42] proposes a methodology for formalising the semantics of UML profiles based on fUML [20], 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 [37]. Likewise, interested readers can find a comprehensive review on execution of UML and UML profiles in [7].

Irrespective of the way these UML profiles were developed, either following ad-hoc processes or based on guidelines for designing well-structured UML profiles [13, 40], they required substantial designer effort. Also, the learning curve for new designers interested in exploring whether UML profiles suit their business needs is steep [15]. 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 [31]. The work

in [32] 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 [47]. The methodology introduced in [15, 16] facilitates the derivation of a UML profile using a simpler DSL as input. The methodology requires the manual definition of an intermediate metamodel that captures the abstract syntax to be integrated into a UML profile. The intermediate metamodel is then compared against the UML metamodel to identify a set of required UML extensions, as well as the transformation of the intermediate metamodel into a corresponding functioning UML profile. Similarly, [31] 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 relevant research work is JUMP [4] that supports the automatic generation of profiles from annotated Java libraries [4]. Despite the potential of these approaches, they usually involve non-trivial human-driven tasks, e.g., a UML profile skeleton [32] or an intermediate metamodel [15, 16], or have limited capabilities (e.g., support of UML profile derivation with generation of OCL constraints [31]). In contrast, Jorvik builds on top of standard Ecore metamodels that form the building blocks of MDE [18]. 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 [2], 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 [47] 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 [14] 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 [47] and [14]), but requires only a single annotated Ecore metamodel and does not need any mediator languages [47] or integration metamodels [14]. 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 an Eclipse plugin.

6.4 Automatic generation of modeling facilities

There are a number of works that uses MDE techniques to generate modeling facilities for models. The most widely used tool for graphical modelling is the Graphical Modeling Framework [1] (GMF) provides editor generation facilities for EMF models. GMF provides the foundation for a number of tools. However, as stated in [28], creating graphical editors in GMF typically involves non-trivial, repetitive and error-prone tasks. Hence, a number of tools emerged, such as Sirius [43], Eugenia [28], and Papyrus. Whilst other tools focus on creating editing facilities for EMF models, Papyrus provides an open source solution for UML modelling and UML Profiling.

It is to be noted that editors are only one particular aspect of an MDE environment supporting UML profiles. There are a number of tools which focus on transformations, validations, code generations and comparison tools for UML models. For example, a work which addresses the generation of basic change operations over UML profile models has been presented in [3, 26].

7 Conclusions and Future Work

In this paper we presented an approach for automatic generation of UML profiles and their supporting distributable Papyrus editors from annotated Ecore metamodels. 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.

We evaluated the efficiency of Jorvik in terms of the amount of effort required through replicating what Archimate for Papyrus does. We evaluated the completeness of Jorvik in terms of its wide applicability through generating a number of Papyrus editors for a variety of EMF metamodels. We evaluated the boost in productivity of Jorvik in terms of the amount of time required through a user experiment. We conclude that Jorvik is a complete solution, which improves the efficiency and boosts the productivity for developers in creating UML profiles and their supporting editors in Papyrus.

In the current version we only support the automatic generation of OCL constraints for connectors for the *Association* base element. In the future work, we will try to support other connector types, such as Dependency and Composition. Currently, our approach is a one way transformation from annotated Ecore metamodels to UML profiles and their editors. In the future work, we plan to support the generation of editors based on a UML profile. In addition, since there was a major change in infrastructure of Papyrus version 3.0.0, we plan to build a migration tool, which uses the generation of editors based on UML profiles and support the migration of editors built for pre-Papyrus 3.0.0 to Papyrus 3.0+.

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 outgoing node for the edge. [required]
- target (*for EClasses only*): The name of the EReference of the EClass that denotes the type of the incoming 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. The graphical modeling project (gmp). ONLINE, `http://www.eclipse.org/modeling/gmp/`
2. 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)
3. Alanen, M., Porres, I.: Subset and union properties in modeling languages. Tech. rep., Technical Report 731, TUCS (2005)
4. Bergmayr, A., Grossniklaus, M., Wimmer, M., Kappel, G.: JUMP—From Java Annotations to UML Profiles, pp. 552–568 (2014)
5. 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)
6. 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)
7. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of uml models: a systematic review of research and practice. Software & Systems Modeling pp. 1–48 (2018)
8. 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)
9. 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)
10. Dong, J., Yang, S., Zhang, K.: Visualizing design patterns in their applications and compositions. IEEE Transactions on Software Engineering 33(7), 433–453 (2007)
11. Erickson, J., Siau, K.: Theoretical and practical complexity of modeling methods. Communications of the ACM 50(8), 46–51 (2007)
12. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to SysML: the systems modeling language (2014)
13. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to UML profiles. UML and Model Engineering 2 (2004)
14. 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)
15. 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)
16. Giachetti, G., Valverde, F., Pastor, O.: Improving Automatic UML2 Profile Generation for MDA Industrial Development, pp. 113–122 (2008)
17. Object Management Group: Modeling And Analysis Of Real-Time Embedded Systems. ONLINE (2011), http://www.omg.org/spec/MARTE/1.1/
18. Object Management Group: Meta object facility (MOF) core specification. ONLINE (2014), http://www.omg.org/mof/
19. Object Management Group: Unified Modeling Language. `http://www.omg.org/spec/UML/` (June 2015)

20. Group, O.M.: Semantics of a foundational subset for executable uml models. In: Technical Report (2010)
21. Haren, V.: Archimate 2.0 specification (2012)
22. 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)
23. Iacob, M.E., Jonkers, H., Lankhorst, M.M., Proper, H.A.: ArchiMate 1.0 Specification. Zaltbommel: Van Haren Publishing (2009)
24. 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)
25. Kehrer, T., Kelter, U.: Versioning of ordered model element sets. Softwaretechnik-Trends 34(2) (2014)
26. Kelter, U., Schmidt, M.: Comparing state machines. In: Proceedings of the 2008 international workshop on Comparison and versioning of software models. pp. 1–6. ACM (2008)
27. Kolovos, D., Paige, R., Polack, F.: The Epsilon Transformation Language. Theory and Practice of Model Transformations pp. 46–60 (2008)
28. 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)
29. 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
30. 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)
31. 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)
32. Lagarde, F., Espinoza, H., Terrier, F., André, C., Gérard, S.: Leveraging Patterns on Domain Models to Improve UML Profile Definition, pp. 116–130 (2008)
33. 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)
34. 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)
35. 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)
36. Moreno, N., Fraternali, P., Vallecillo, A.: Webml modelling in uml. IET Software 1(3), 67–80 (2007)
37. Pardillo, J.: A Systematic Review on the Definition of UML Profiles, pp. 407–422 (2010)

38. 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)
39. 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)
40. 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)
41. Simons, C., Wirtz, G.: Modeling context in mobile distributed systems with the {UML}. Journal of Visual Languages & Computing 18(4), 420 – 439 (2007)
42. 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)
43. 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)
44. Walderhaug, S., Stav, E., Mikalsen, M.: Experiences from Model-Driven Development of Homecare Services: UML Profiles and Domain Models, pp. 199–212 (2009)
45. Wei, R., Kelly, T.P., Dai, X., Zhao, S., Hawkins, R.: Model based system assurance using the structured assurance case metamodel. Journal of Systems and Software 154, 211–233 (2019)
46. 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)
47. Wimmer, M.: A semi-automatic approach for bridging dsmls with uml. International Journal of Web Information Systems 5(3), 372–404 (2009)
48. 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)
49. 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)
50. 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)