The Journal of Systems & Software 175 (2021) 110912

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment☆

Lorenzo Addazi[1], Federico Ciccozzi[1,*]

School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden

ARTICLE INFO

Article history:
Received 8 May 2020
Received in revised form 20 October 2020
Accepted 19 January 2021
Available online 23 January 2021

Keywords:
Blended modelling
Multi-view modelling
UML profiles
MARTE
Xtext
Papyrus

ABSTRACT

Domain-specific modelling languages defined by extending or constraining the Unified Modelling Language (UML) through the profiling mechanism have historically relied on graphical notations to maximise human understanding and facilitate communication among stakeholders. Other notations, such as text-, form-, or table-based are, however, often preferred for specific modelling purposes, due to the nature of a specific domain or the available tooling, or for personal preference. Currently, the state of the art support for UML-based languages provides an almost completely detached, or even entirely mutually exclusive, use of graphical and textual modelling. This becomes inadequate when dealing with the development of modern systems carried out by heterogeneous stakeholders. Our intuition is that a modelling framework based on seamless blended multi-notations can disclose several benefits, among which: flexible separation of concerns, multi-view modelling based on multiple notations, convenient text-based editing operations (inside and outside the modelling environment), and eventually faster modelling activities.

In this paper we report on: (i) a proof-of-concept implementation of a framework for UML and profiles modelling using blended textual and graphical notations, and (ii) an experiment on the framework, which eventually shows that blended multi-notation modelling performs better than standard single-notation modelling.

1. Introduction

Expectations on software functionality and quality are increasing at a fast pace. Additionally, the interconnected nature of software-intensive systems makes software grow exponentially in complexity. The combination of high functional and extra-functional demands with ever-growing complexity leads to large increases in development time and costs.

To combat this threat, Model-Driven Engineering (MDE) has been adopted in industry as a powerful means to effectively tame complexity of software, systems and their development, as shown by empirical research (Hutchinson et al., 2011), by using domain-specific abstractions described in Domain Specific Modelling Languages (DSML) (Mussbacher et al., 2014). DSMLs allow domain experts, who may or may not be software experts, to develop complex functions in a more domain-focused and human-centric way than if using traditional programming languages. DSMLs formalise (for computer-based analysis and synthesis purposes) the *communication* language of engineers at the level of domain-specific concepts such as an engine and wheels for a car. These concepts may not exist in another domain. Moreover, DSMLs support more efficient integration of software with designs and implementations of other disciplines. In this paper, we focus on DSMLs based on the Unified Modelling Language (UML).

UML is the de-facto standard in industry (Hutchinson et al., 2011) and an ISO/IEC (19505-1:2012) standard. It is general-purpose, but it provides powerful profiling mechanisms to constrain and extend the language to achieve UML-based DSMLs (hereafter also called 'UML profiles'). Domain-specific modelling demands high level of customisation of MDE tools, typically involving combinations and extensions of DSMLs as well as customisations of the modelling tools for their respective development domains and contexts. In addition, tools are expected to provide multiple modelling means, e.g. textual and graphical, to satisfy the requirements set by development phases, different stakeholder roles, and application domains.

Nevertheless, domain-specific modelling tools traditionally focus on one specific editing notation (such as text, diagrams, tables or forms). This limits human communication, especially across

☆ Editor: [DOO-HWAN BAE].

* Corresponding author.
E-mail addresses: lorenzo.addazi@mdh.se (L. Addazi), federico.ciccozzi@mdh.se (F. Ciccozzi).

[1] The authors have equally contributed to the work presented in this manuscript.

https://doi.org/10.1016/j.jss.2021.110912
0164-1212/

stakeholders with varying roles and expertise. Moreover, engineers may have different notation preferences; not supporting multiple notations negatively affects throughput of engineers. Besides the limits to communication, choosing one particular kind of notation has the drawback of limiting the pool of available tools to develop and manipulate models that may be needed. For example, choosing a graphical representation limits the usability of text manipulation tools such as text-based diff/merge, which is essential for team collaboration. When tools provide support for both graphical and textual modelling, it is mostly done in a mutual exclusive manner. Most off-the-shelf UML modelling tools, such as IBM Rational Software Architect (IBM, 2020a) or Sparx Systems Enterprise Architect (SparxSystems, 2020b), focus on graphical editing features and do not allow seamless graphical–textual editing. This mutual exclusion suffices the needs of developing small scale applications with only very few stakeholder types.

For systems with heterogeneous components and entailing different domain-specific aspects and different types of stakeholders, mutual exclusion is too restrictive and void many of the MDE benefits. Therefore, modelling tools need to enable different stakeholders to work on overlapping parts of the models using different modelling notations (e.g., graphical and textual).

Paper contribution. In this paper we describe our work towards a full-fledged framework able to provide seamless blended graphical–textual modelling for UML profiles. Differently from current practices, our framework is based on a lightweight form of blended modelling, where both graphical and textual editors operate on a common underlying model resource, rather than on separate persisting resources, thus heavily reducing the need for explicit synchronisation between the two. To maximise the accessibility of our solutions, we leverage open-source platforms and technologies only. We implemented a proof-of-concept framework, as well as designed and ran an experiment to assess potential benefits of blended multi-notation modelling as opposed to standard single-notation modelling.

Note that the area of so called action languages, such as for instance the UML actions (as in Charfi et al. (2009)) or the VIDE[2] action language, also for UML, has focused on how to integrate textual notations for description of algorithmic behaviours (i.e. defined by a limited and fixed subportion of the original metamodel or a new ad-hoc one) in graphical (structural) models. In our work, we focus on a broader and more complex problem, namely the provision of a fully blended modelling environment for any portion (partial or full) of a UML-profile, being it structural or behavioural (or both).

Paper outline. The remainder of the paper is organised as follows. Section 2 outlines the motivation behind the research work on blended modelling reported in this paper, while Section 3 provides a snapshot of the states of the art and practice related to blended modelling and an introduction of core concepts. In Section 4 we outline our approach, the intended benefits, and the differences with current practices. Details on the actual implementation of the framework and exemplifications on a UML profile are provided in Section 5. The experiment's set-up, execution, results and threats to validity are described in Section 6 with discussing explicitly results in relation to research hypotheses and experimental questions. In Section 7 we provide a retrospective on the benefits and limitations of our approach in relation to state of the art. We conclude the paper with Section 8.

2. Blended modelling and potential benefits

We have previously defined the notion of *blended modelling* (Ciccozzi et al., 2019) as:

> *the activity of interacting seamlessly with a single model (i.e., abstract syntax) through multiple notations (i.e., concrete syntaxes), allowing a certain degree of temporary inconsistencies.*

At first sight, the notion of blended modelling may seem similar or overlapping with multi-view modelling (Cicchetti et al., 2019) that is based on the paradigm of viewpoint/view/model as formalised in the ISO/IEC 42010 standard.[3]

Multi-view modelling is commonly based on viewpoints (i.e. "conventions for the construction, interpretation and use of architecture views to frame specific system concerns" Emery and Hilliard, 2009) that are materialised through views that are composed of one or more models. In blended modelling, the focus is *not* on identifying viewpoints and related views, but rather on providing multiple blended editing and visualising notations to interact with a set of concepts. In short, blended modelling could be seen as orthogonal to multi-view modelling. While multi-view modelling aims at defining viewpoints/views, blended modelling aims at providing a powerful multi-notation characterisation that may be used to define viewpoints/views. Multi-view modelling approaches focus on the creation of viewpoints/views and mechanisms for consistency management across them (Boucké et al., 2008; Cicchetti et al., 2019). Blended modelling focuses on the specific problems related to the provision of multiple concrete syntaxes for a set of abstract syntactical concepts, independently of whether the base modelling approach is multi-view or not.

The intuition is that establishing a seamless blended modelling environment, which allows stakeholders to freely choose and switch between graphical and textual notations, can greatly contribute to increase productivity as well as decrease costs and time to market.

Such an environment is expected to support at least graphical and textual modelling notations in parallel as well as properly manage synchronisation to ensure consistency among the two. The possibility to visualise and edit the same information through a set of diverse perspectives always in sync has the potential to greatly boost communication between stakeholders, who can freely select their preferred notation or switch from one to the other at any time. Besides obvious notation-specific benefits, such as for instance the possibility to edit textual models in any textual editor outside the modelling environment, a blended framework would disclose the following overall benefits.

2.1. Flexible separation of concerns and better communication

Providing graphical and textual modelling editors for different aspects and sub-parts (even overlapping) of a DSML enables the definition of concern-specific views characterised by either graphical or textual modelling (or both). These views can interact with each other and are tailored to the needs of their intended stakeholders. Due to the multi-domain nature of modern software systems (e.g., cyber–physical systems, Internet-of-Things), this represents a necessary feature to allow different domain experts to describe specific parts of a system using their own domain-specific vocabulary and notation, in a so called *multi-view modelling* (Cicchetti et al., 2019) fashion. The same information can then be rendered and visualised through other notations in other perspectives to maximise understanding and boost communication between experts from different domains as well as other stakeholders in the development process.

[2] https://cordis.europa.eu/project/id/033606.

[3] https://www.iso.org/standard/50508.html.

There are many other aspects related to separation of concerns that could characterise a blended modelling framework, such as layered accessibility to shared information with multiple levels of read/write access rights, enforcement of specific notations depending on stakeholder roles, and customisability of perspectives, to mention a few. In this paper we do not focus on these features, but rather work on the infrastructural support for providing a blended modelling framework, that is to say the ground upon which these features can be yielded.

2.2. Faster modelling tasks

We expect that the seamless combination of graphical and textual modelling has the potential to reduce modelling effort in terms of time thanks to the following two factors.

(1) Any stakeholder can choose the notation that better fits her needs, personal preference, or the purpose of her current modelling task, at *any time*. For instance, while structural model details can be faster to describe by using diagrammatic notations, complex algorithmic model behaviours are usually easier and faster to describe using textual notations (e.g., Java-like action languages).

(2) Text-based editing operations on graphical models,[4] such as copy&paste and regex search&replace, syntax highlighting, code completion, quick fixes, cross referencing, recovery of corrupted artefacts, text-based differencing and merging for versioning and configuration, are just few of the features offered by modern textual editors. These would correspond to very complex operations if performed through graphical editors; thereby, most of them are currently not available for diagrams. Seamless blended modelling would enable the use of these features on graphically-described models through their textual editing view. These would dramatically simplify complex model changes; an example could be restructuring of a hierarchical state-machine by moving the insides of a hierarchical state. This is a demanding re-modelling task in terms of time and effort if done at graphical level, but it becomes a matter of a few clicks (copy&paste) if done at textual level.

In this paper we provide a blended modelling framework and experiment with it to assess whether its use can potentially speed-up modelling activities, if compared to standard single-notation modelling.

3. Related work and core concepts

Several open-source tools and approaches have been proposed to intermix textual and graphical concrete syntaxes. Here we present them highlighting their strengths and weaknesses in relation to our research goals.

3.1. Text-based modelling with generation of graphical visualisations

Umple (Umple team, 2020c) merges the concepts of programming and modelling by adding modelling abstractions directly into programming languages and provides features for actively performing model edits on both textual and graphical concrete syntaxes. Nevertheless, it does not provide blended modelling support for custom UML profiles.

A plethora of other open-source tools such as FXDiagram (Koehnlein, 2020d), Eclipse Sprotty (Eclipse Foundation, 2020e), LightUML (Hakala, 2020f), TextUML (TextUML team, 2020g), MetaUML (Gheorghies, 2020h), PlantUML (PlantUML team, 2020i) focuses on textual concrete syntax for actively editing the modelling artefacts, while providing a graphical notation for visualisation purposes only.

FXDiagram is based on JavaFX 2 and provides on-the-fly graphical visualisation of changes in the textual concrete syntax including change propagation; the focus is on EMF models. Both notations are predefined and not customisable, the graphical notation is read-only and there is no support for UML profiling mechanisms.

Eclipse Sprotty, as FxDiagram, focuses on the visualisation of textual models, but it provides a certain degree of editing of the models too. Nevertheless, concrete syntaxes are predefined and not customisable and there is no support for UML profiling mechanisms

LightUML focuses more on reverse engineering by generating a class diagram representation of existing Java classes and packages. It is the least advanced of the analysed solutions and displays all limitations listed at the end of the section.

TextUML allows modellers to leverage a textual notation for defining UML models and providing textual comparison, live graphical visualisation of the model in terms of class diagrams, syntax highlighting and instant validation. In this tool, the graphical notation is read-only and cannot be customised. Also, the subset of UML supported is fixed and not easily extensible. No support for UML profiling mechanisms is provided.

MetaUML is a MetaPost library for creating UML diagrams through a textual concrete syntax and it supports a few read-only diagrams. Similarly, PlantUML allows the modelling of UML diagrams by using a textual notation; graphical visualisations are read-only. In both cases, the support UML concepts and diagrams is limited and not extensible and there is no support for UML profiling mechanisms.

3.2. Synchronised textual and graphical modelling

No technology provides combined means for synchronised editing in both textual and graphical syntaxes and customisation of the concrete syntaxes for UML profiles. JetBrains MPS (Jetbrains, 2020j) is a meta-modelling environment for the development of DSML tools which support synchronised editing in multiple (customisable) concrete syntaxes for non-UML DSMLs, but lacks out-of-the-box support for UML profiles and has limited automated support for integration with domain-specific environments. Furthermore, the "textual" view of a model in MPS is not actual text but a form-based representation with a fixed format. This implies, not only that these form-based editors restrict the user, but more critically, standard text-based tools such as regex search/replace, or diff/merge cannot be used on the model.

Synchronised editing in multiple (customisable) concrete syntaxes for non-UML DSMLs can also be realised with Qt (The Qt Company, 2020k) by using a model-view architecture where the 'model' reflects the DSML concepts and any 'view' is actually an editor for some concrete syntax. However, Qt has no support for DSML-based automation in any way, which means that a DSML engineer needs to basically program most of it manually (although well supported by IDE tools).

3.3. Mixed textual and graphical modelling

Several research efforts have been directed to mixing textual and graphical modelling. A textual editor for the Action Language for Foundational UML (Alf) has been developed based on Xtext (Lazăr, 2011).

In Andrés et al. (2007), the authors provide an approach for defining combined textual and graphical DSMLs based on the

[4] Please note that by *graphical/textual model* we intend a model rendered using a graphical/textual notation.

AToM3 tool. Starting from a metamodel definition, different diagram types can be assigned to different parts of the metamodel. A graphical concrete syntax is assigned by default, while a textual one can be given by providing triple graph grammar rules to map it to the specific metamodel portion. The aim of this approach is similar to ours, but it targets specific DSMLs defined through AToM3 and is not applicable to UML profiles.

Charfi et al. (2009) explore the possibilities to define a single concrete syntax supporting both graphical and textual notations. Their work is very specific to the modelling of UML actions and has a much narrower scope than our work. In addition, the defined textual notation is exclusively defined for part of the concepts of UML actions and cannot be customised. Support for UML profiling is not envisioned in the solution either.

In Scheidgen (2008), the authors provide the needed steps for embedding generated EMF-based textual model editors into graphical editors defined in terms of GMF. That approach provides pop-up boxes to textually edit elements of graphical models rather than allowing seamless editing of the entire model using a chosen syntax. The focus of that paper is on the integration of editors based on EMF, while ours is to provide seamless textual and graphical modelling for UML profiles. Moreover, the change propagation mechanisms proposed by the authors are on-demand triggered by modeller's commit, while we focus on on-the-fly change propagation across the modelling views.

Related to the switching between graphical and textual syntaxes, two approaches are proposed to ease transformations of models containing both graphical and textual elements. The first is Grammarware (Wimmer and Kramler, 2005), by which a mixed model is exported as text. The second is Modelware (Wimmer and Kramler, 2005), by which a model containing graphical and textual content is transformed into a fully graphical model. Transformation from mixed models to either text or graphics is on demand rather than on-the-fly and the approach does not allow concurrent editing.

Mixed textual and graphical modelling can also be realised with Qt, where the approach is to use a graphical environment with embedded textual editors. However, DSML engineers need to realise most of this manually. Mixed notations and the possibility to switch between them are supported in JetBrains MPS for non-UML DSMLs and rely on the principle of projectional editing.

3.4. Projectional editing

Projectional editing is another research area which investigates the use of various concrete syntaxes for editing models by displaying the different concrete syntaxes as projections. JetBrains MPS and MelanEE (Atkinson and Gerbig, 2013) apply this approach. With projectional editing, the user edits the model through a syntax-specific view or editor, which itself updates the underlying abstract syntax model and these changes are automatically reflected in other views or editors for alternative concrete syntaxes. The main advantage is that the model can be projected in various concrete syntaxes depending on what the user prefers. However, it adds a considerable overhead for the DSML developer as the user actions (i.e., keyboard, trackpad and mouse events) have to be translated into change actions on the abstract syntax tree. For parser-based textual DSMLs (e.g., when using Xtext), a text editor in combination with a lexer/parser combination can be used.

3.5. Summary

To summarise, current solutions for blended textual and graphical UML modelling present at least one of the following limitations:

- L1: one of the notations is read-only, intended for visualisation means;
- L2: at least one of the two notations is enforced for a specific, self-contained portion of UML (or a profile) only;
- L3: concrete syntaxes are predefined and not customisable;
- L4: synchronisation among different concrete syntaxes is not automated and on-the-fly but rather manual or on-demand;
- L5: no out-of-the-box support for custom UML profiles.

In Table 1, we summarise the limitations of each of the described tools and approaches.

We have provided a first attempt to blended modelling in Addazi et al. (2017). In this paper, we revisited and extended every constituent of that, from the framework's implementation to the experiments. The framework was re-implemented to provide support for multiple UML stereotype applications, a core feature when dealing with profiles. A new experiment, with more than double the participants, was run to assess the potential benefits of blended modelling in general and analyse the use of user-preferred and task-optimal notations in particular.

4. Providing blended modelling in Papyrus

Our overall goal is to provide a full-fledged framework enabling blended modelling for UML and profiles. To maximise accessibility to our solutions around which a community of researchers and practitioners can be built, we chose to only leverage de-facto standard open-source tools, i.e. Eclipse modelling Framework (Eclipse Foundation, 2020l) (EMF) as platform, Papyrus (Eclipse Foundation, 2020m) for UML (graphical) modelling, and Xtext (Eclipse Foundation, 2020n) for textual modelling. Note that what we propose is a seamless solution leveraging two different technologies – Xtext and Papyrus – exploiting the common EMF infrastructure. We did not create a textual editor in Papyrus, but rather leverage all the advantages provided by Xtext and we modified its resource management routines to not break Papyrus models and vice versa. Summarise, we use Xtext as textual-focused technology and Papyrus as graphical-focused one, combining and leveraging their strengths. In Fig. 1, we depict the differences between existing Eclipse-based solutions for blended UML modelling and our framework.

The current state of the practice (see left-hand side of Fig. 1) relies on approaches that achieve UML-based blended modelling by keeping graphical and textual modelling almost fully detached. Graphical and textual modelling are performed on two separate models, which are both separately persistent in two physical resources (Maro et al., 2015). Given a UML profile, a corresponding Ecore-based DSML representing the profile is automatically generated or manually provided. EMF provides automation for this task, but the resulting Ecore model needs often manual tuning in order to be made useable. Starting from the Ecore-based DSML, Xtext provides features for automatic generation for a textual language (in terms of a grammar) and related editors.

Graphical modelling is performed using UML editors and the model persists as UML model resource. On the other hand, textual modelling is performed using generated Xtext editors and the textual representation persists in a separate text file. Moreover, Xtext works internally with an Ecore model resource. More specifically, the Ecore model represents the AST obtained from parsing the textual model at a given time and is used by validators, generators, and other features. The parser always generates a new Ecore model, rather than update an existing one. This means that the only source of reliable persistent information is the textual file itself.

Table 1
Aggregated elapsed modelling times (in seconds).

Tool/approach	L1	L2	L3	L4	L5
Andres et al. (Andrés et al., 2007)			✓	✓	✓
Charfi et al. (Charfi et al., 2009)		✓	✓		✓
FXDiagram (Koehnlein, 2020d)	✓		✓		✓
Grammar/modelware (Wimmer and Kramler, 2005)			✓	✓	✓
Lazar et al. (Lazăr, 2011)	✓	✓	✓		✓
LightUML (Hakala, 2020f)	✓	✓	✓	✓	✓
JetBrains MPS (Jetbrains, 2020j)					✓
Maro et al. (Maro et al., 2015)		✓	✓	✓	
MelanEE (Atkinson and Gerbig, 2013)					✓
MetaUML (Gheorghies, 2020h)	✓	✓	✓		✓
PlantUML (PlantUML team, 2020i)	✓	✓	✓		✓
Scheidgen et al. (Scheidgen, 2008)				✓	✓
Sprotty (Eclipse Foundation, 2020e)			✓		✓
Qt (The Qt Company, 2020k)			✓	✓	✓
TextUML (TextUML team, 2020g)	✓	✓	✓		✓
Umple (Umple team, 2020c)		✓			✓

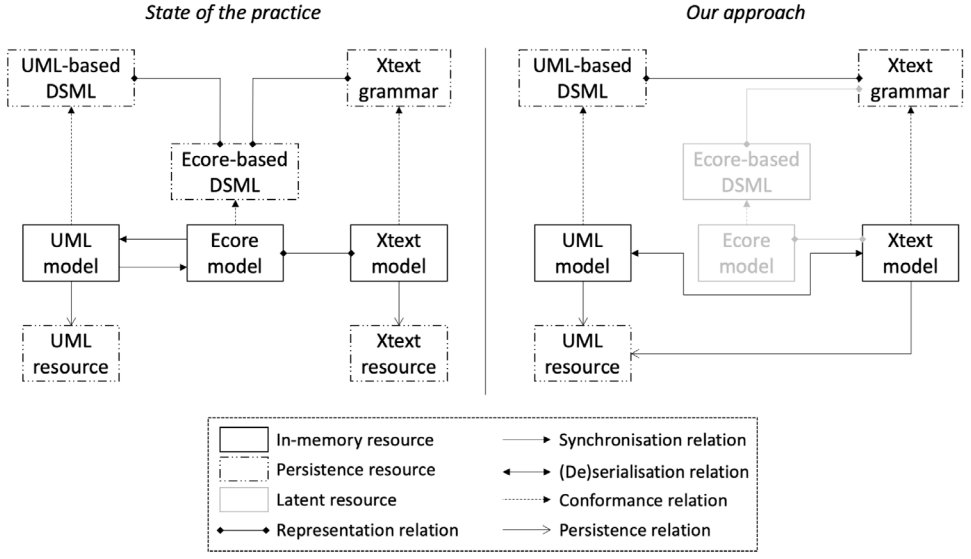

Fig. 1. Our approach compared to the state of the practice.

To synchronise graphical and textual models, semi-automated mechanisms in the form of synchronisation model transformations are used. These model transformations are, in some approaches, also generated, thanks to higher-order model transformations (HOTs) (Maro et al., 2015). Although this provides a certain degree of flexibility when it comes to the evolution of the involved UML-based DSML and automatic co-evolution of the synchronisation mechanisms, HOTs would stop working as soon as the generated Xtext grammar is manually edited. This practice is very often needed in order to make the grammar (and related editors) fit the developer's needs. But why would grammar customisations be needed?

As a concrete example of the need to customise a DSML grammar, consider the UML-RT language (Selic et al., 1994). UML-RT has two core concepts: *capsules* and *protocols*. Capsules are active classes and have a well-defined interface consisting of *ports* typed by protocols. Capsules may have an internal structure consisting of *parts* that hold capsule instances linked by connectors bound to the corresponding capsule ports. All interactions between capsule instances takes place by message-passing through connected ports.

UML-RT is implemented as a UML profile. If we start from the UML-RT profile, we obtain an Xtext grammar that contains rules like the following:

```
1   Capsule returns Capsule:
2       'Capsule'
3       '{'
4           'base_Class' base_Class=[uml::Class|
5               EString]
6       '}';
7
8   Class returns uml::Class:
9       Class_Impl | Activity | Stereotype |
10      ProtocolStateMachine | StateMachine_Impl
11      | FunctionBehavior | OpaqueBehavior_Impl
12      | Device | Node_Impl | ExecutionEnviron-
13      ment | Interaction | AssociationClass
14      | Component;
15
16  Class_Impl returns uml::Class:
17      'Class'
18      '{'
19          ('name' name=String0)?
20          ('visibility' visibility=Visibility-
21          Kind)?
22          'isLeaf' isLeaf=Boolean
23  ...
24          ('useCase' '(' useCase+=[uml::Use-
25          Case|EString]
26              ( "," useCase+=[uml::UseCase|
27              EString])* ')' )?
28  ...
29          ('ownedAttribute' '{' ownedAttri-
30          bute+=Property
31              ( "," ownedAttribute+=Property)*
32              '}' )?
33          ('ownedConnector' '{' ownedConnect-
34          or+=Connector
35              ( "," ownedConnector+=Connector)*
```

```
36            '}' )?
37    ...
38       '}';
```

This clearly entails a great amount of information related to UML, but not relevant to UML-RT. In fact, the rule for `Class_Impl` includes clauses for each and every feature of the UML Class metaclass, many of which we removed for the sake of space. Of these clauses, many, such as `useCase`, are irrelevant to the DSML, and only a few, such as `ownedAttribute` and `ownedConnector`, are relevant, but they do not reflect the concepts of UML-RT, and even the concrete syntax may not be desirable. For UML-RT, we would like to obtain a grammar with rules that reflect the DSML's concepts directly and hides away any additional UML structure that may be used to represent the concept. For example, instead of having a single clause `ownedAttribute`, we would like to have clauses for ports and parts, in a rule like this:

```
1   Capsule returns Capsule:
2      'capsule' name=EString
3      '{'
4         (ports+=RTPort)*
5         (parts+=CapsulePart)*
6         (connectors+=Connector)*
7         StructuredTypeCommonCoreFragment
8         BehaviourFragment
9      '}';
```

Another reason for customising grammars is to enable swift, condensed and convenient application of stereotypes, especially in case of multiple applications. When generating grammars from UML profiles, Xtext generates separate explicit metaclasses for each stereotype applicable to a UML base element. Let us consider the MARTE profile (Selic and Gérard, 2013) and a snippet of the grammar generated by Xtext as follows:

```
1   HwComputingResource returns marte::HwCompu-
2   tingResource:
3      'hwComputingResource'
4      '{'
5         ....
6         'base_Classifier' base_Classifier=
7         [uml::Classifier]
8         ....
9      '}';
10
11  HwTimingResource returns marte::HwTiming-
12  Resource:
13     'hwTimingResource'
14     '{'
15        ....
16        'base_Classifier' base_Classifier=
17        [uml::Classifier]
18        ....
19     '}';
```

Through these two rules, the Xtext grammar represents two stereotypes, `HwComputingResource` and `HwTimingResource`, applicable to the base element `Classifier`. Clearly, this mimics a stereotype application, but in reality it is quite far from it. In fact, the grammar allows to create a component A and afterwards two separate elements of type `HwComputingResource` B and `HwTimingResource` C, both referring to A:

```
1   component
2      {
3         name = A;
4         ...
5      };
6
7   hwComputingResource
8      {
9         name = B;
10        base_element = A;
11        ...
12     };
13
```

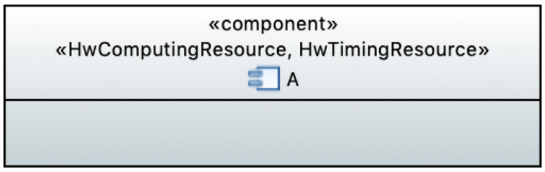

Fig. 2. Stereotype application in Papyrus.

```
14  hwTimingResource
15     {
16        name = C;
17        base_element = A;
18        ...
19     };
```

This does not properly reflect the stereotype application mechanism in UML (shown in Fig. 2). The same, using a UML graphical editor (e.g., Papyrus), would be defined as a single element, component A, stereotyped with both `HwComputingResource` and `HwTimingResource`:

We want to reproduce this modelling pattern also in the related grammar as:

```
1   hwComputingResource hwTimingResource component
2      {
3         name = A;
4         ...
5      };
```

The reasons for reproducing this pattern are the following:

- **Conformity** to UML modelling: as described above, applying a stereotype to a base element is realised by extending the base element itself rather than creating a brand new element, typed as the stereotype, and referring to the base element.
- **Conciseness**: apart from being in line with UML profiling and application of stereotypes, this solution is textually much more concise, since base element and applied stereotypes are described by one element only.
- **Modifiability**: when the base element or the applied stereotypes need revision and changes, the fact that they can all be found in one single place helps identifying the elements and modifying them, without jeopardising consistency nor breaking model conformance.
- **Automation**: when manipulating models (e.g., through model transformations), their navigation can be expensive in terms of execution time, especially when dealing with profiles (Ciccozzi et al., 2013). Clearly, having all information about one element (A) and its stereotypes in one single place, rather than spread across the model, simplifies model navigation, thus potentially leading to more performant model transformations.

Although in our grammar we enforced the aforementioned all-in-one stereotypes application pattern, the approach itself does not hinder the user from customising a grammar to work with separate stereotype applications.

Overall, Xtext is designed for being used with EMF-based modelling languages. The UML implementation in Eclipse is EMF-based, thus Xtext can be used to define textual concrete syntaxes for UML. However, Xtext is neither designed to work with UML resources nor with UML-based DSMLs directly. This raises the need for explicit complex synchronisation between the two, both at abstract and concrete syntax level. This boils down to two facts:

1. The Ecore-based DSML is the pivot abstract syntax. If either the UML-based DSML or the grammar is changed, the

representation relationships from them to the Ecore-based DSML are broken and so are the synchronisation relations among the related conforming models (concrete syntax). In summary, those changes would break the synchronisation across notations.
2. UML and Xtext models are persistent in two different resources. The textual resource can be edited with, virtually, any editor, while the UML resource is editable with UML editors. The modelling is not blended, but rather detached and complex model transformations keep the two notations in sync.

Our approach is inherently different (see right-hand of Fig. 1). In fact, we make Xtext work with UML profiles by exploiting a single underlying abstract syntax (UML-based DSML), two concrete syntaxes (graphical given by UML and textual given by Xtext), one single persistent resource (UML resource), and thereby reducing the need for ad-hoc heavyweight synchronisation mechanisms. Synchronisation is achieved extending the default content management operations performed in Xtext editors. The parsing process does not use the content of the persisted UML resource, but rather the result of a model-to-text transformation applied on it. Inversely, the serialisation process does not persist the plain-textual editor content. The textual model is merged with the persisted UML model using a model-to-model transformation, which propagates the changes and manages the application/removal of stereotypes.

Advancing the state of the art and practice. Our solution provides the following improvements to the current state of the practice:

- **Grammar customisability.** Given a specific UML profile, Xtext grammars are semi-automatically derived to provide a textual language for the profile (or part of it). Xtext grammars can be customised and refactored to fit the stakeholder's needs. This does not jeopardise the (de-)serialisation mechanisms as long as it does not break the conformance of models to the UML profile specification (i.e. metamodel).
- **Multiple stereotypes application.** Given a specific UML profile, Xtext provides an out-of-the-box feature for generating grammar and editor for a textual language related to the profile. This generation is not customisable nor parametric. More importantly, there is no feature to automatically derive cross-profile grammars. A cross-profile grammar would entail the possibility to apply stereotypes from different profiles to the same base UML model. This is a very common feature in UML modelling and in this work we demonstrate how to achieve it using Xtext grammars.
- **Cross-profile modelling.** Virtually, any UML profile can be leveraged without the provision of ad-hoc complex synchronisation transformations. In practice, for complex profiles as well as in case of multiple applications of stereotypes to the same base UML elements, (de-)serialisation might need additional input from the blended DSML developer so to better reflect the purposes of the textual language(s) (see stereotypes application transformation described in Section 5).
- **On-the-fly changes propagation.** Model changes done in one view (e.g., UML graphical) are seamlessly reflected and visible on-the-fly in the other view (e.g., Xtext textual). This is possible thanks to the single persistent resource shared among the views. Such a propagation is incremental, thus not producing tangible delays in the rendering of the changed model across notations.
- **Cross-notation multi-view modelling.** Multiple Xtext grammars/languages representing different sub-sets (even partially overlapping) of the UML profile (or several profiles) exposed to stakeholders in ad-hoc views/editors can seamlessly work on the same UML resource, along with UML editors. This, along with the possibility to "import" profiles in a joint Xtext grammar for multiple stereotypes application, provides a full-fledged blended modelling framework for UML and profiles. An overall precondition for the framework to properly function is that Xtext grammars always enforce model conformance to the entailed profiles.

Other benefits stem from the aforementioned ones. An example is the fact that code generators can reuse a single, shared abstract syntax for both graphical and textual representations of a model, without relying on additional transformations which result in added maintenance costs. Another is that different stakeholders can view and edit model parts of their collaborators in their preferred syntax (or in a syntax that is optimised for them). In this way, potential inconsistencies can be identified very early already during the modelling process and communication among different stakeholders is greatly improved.

In the next section we describe our blended modelling solution from a technical perspective, providing concrete exemplifications of the aforementioned benefits.

5. Technical solution

Our blended graphical–textual modelling approach combines Xtext and Papyrus for UML (a demo of the running framework can be found at https://bit.ly/blended_demo). As mentioned in Section 3, existing blended modelling approaches using these technologies rely on detached sets of abstract and concrete syntaxes, as well as persistence resources. This results in separated graphical and textual modelling support, where partial blended modelling is achieved through explicit synchronisation between separate resources representing different concrete syntaxes. However, relying on different abstract syntaxes makes the synchronisation process non-trivial as it requires complex exogenous DSML-specific model transformations.

Given a UML-based DSML, our solution supports blended modelling using multiple concrete syntaxes with a single abstract syntax and persistence resource, as depicted in Fig. 1. Unfortunately, Xtext does not provide built-in UML support and two major challenges hinder the feasibility of our approach, i.e. resource persistence and profiling support. Given a grammar specification, Xtext automatically generates an ANTLR parser and an Ecore metamodel. At runtime, the parser produces AST instances conforming to this metamodel. The instances are subsequently used to provide features such as validation or code generation, but are not stored in a persistent resource. Models are rather treated as plain-textual resources, which makes the framework fundamentally incompatible with editors using XMI serialisation, e.g. Papyrus for UML. Furthermore, the Ecore-based nature of Xtext implies lack of support for UML-specific features, such as profiling.

The following sections describe in detail how these challenges have been tackled. First, we illustrate how Xtext grammar rule and validation patterns can be combined to mimic UML profiling tasks at runtime, e.g. stereotypes application and editing (the blended modelling environment in Papyrus is depicted in Fig. 5, in Appendix). Then, we describe how the Xtext resource management workflow can be extended to parse and serialise models conforming to UML-based DSMLs. To demonstrate our approach, we define a textual modelling language, `XMarte`, supporting a small portion of the MARTE profile. In Tables 2 and 3 we list metaclasses, stereotypes and features from (UML and) MARTE included in XMarte.

Table 2
XMarte - Metaclasses.

Metaclass	Features	Stereotypes
Model	name packagedElement	–
Component	name packagedElement	HwProcessor HwCache Allocated

Table 3
XMarte - Stereotypes.

Stereotype	Features
HwProcessor	nbCores caches
HwCache	level
Allocated	kind

5.1. UML profiling in Xtext

The integration of UML profiling in Xtext languages requires appropriate mechanisms supporting stereotypes application and value editing of their features. In particular, multiple stereotypes should be applicable on the same UML base element, and their features should be editable along with those of the base element.

5.1.1. Stereotypes application

Given a stereotyped UML element, the set of editable features consists of metaclass and stereotype properties. An intuitive approach to represent the stereotype application on a given metaclass instance would consist in defining specific grammar rule alternatives originating from the metaclass rule. Indeed, the Ecore metamodel inference process would generate a metaclass extending the UML metaclass and containing the stereotype properties. For example, the rule representing elements of type `Component` stereotyped as `HwProcessor` could be defined as follows.

```
1  Component returns uml::Component:
2    HwProcessor | ComponentImpl
3
4  HwProcessor returns HwProcessor:
5    'processor' name=ID ';'
```

Exploiting dedicated rule alternatives for each stereotype individually presents major drawbacks. First, adopting such a strategy to represent the application of a single stereotype on multiple base elements could lead to ambiguities, as illustrated in the grammar below. There, multiple parsing paths connect `HwProcessor` rule instances from the `packagedElement` property in the `Model` rule.

```
1  Model returns uml::Model:
2    'model' name=ID
3      packagedElement+=Class*
4
5  Class returns uml::Class:
6    HwProcessor | Component | ClassImpl
7
8  Component returns uml::Component:
9    HwProcessor | ComponentImpl
10
11 HwProcessor returns HwProcessor:
12   'processor' name=ID ';'
```

Moreover, the inferred stereotype metaclass would only extend leaf target metaclasses, i.e. `Component` and not `Class`. Therefore, representing elements of type `Class` stereotyped as `HwProcessor` is not possible. Finally, no support is provided for multiple stereotype applications on a single element. Although grammar ambiguities could be addressed by defining dedicated stereotype rules for each target metaclass, e.g. `HwProcessorClass` and `HwProcessorComponent`, this issue would remain.

In our solution, we addressed the problem by defining a single generic rule for each metaclass, and possible stereotype applications as boolean properties, as follows.

```
1  Model returns uml::Model:
2    'model' name=ID
3      packagedElement+=Class*
4
5  Class returns uml::Class:
6    XClass | Component
7
8  XClass returns XClass:
9    (
10     isHwProcessor?='processor'?
11     & isAllocated?='allocated'?
12   ) 'class' name=ID ';'
13
14 Component returns uml::Component:
15   XComponent
16
17 XComponent returns XComponent:
18   (
19     isHwProcessor?='processor'?
20     & isAllocated?='allocated'?
21   ) 'component' name=ID ';'
```

The grammar above provides an example of stereotypes application using the `XMarte` language. The initial rule allows to define a `Model` containing `Component` and `Class` instances as values of the `packagedElement` containment reference. The `XComponent` and `XClass` rules illustrate our stereotype application approach, see lines 8–12 and 17–21. In Xtext, the `?=` operator defines boolean properties with value depending on the presence of a given token, e.g. `processor` or `allocated`. In particular, the value is *true* if present, *false* otherwise. In our context, the value indicates whether or not the corresponding stereotype is applied. Finally, the `&` operator indicates *unordered groups*, i.e. sets of properties whose order is irrelevant. An example of model conforming to the above grammar, showing unordered combinations of tokens, is illustrated below.

```
1  model m1 {
2     processor component c1;
3     allocated class c2;
4     processor allocated class c3;
5     allocated processor component c4;
6  }
```

In this, the `m1` model contains four stereotyped elements, two components and two classes. Lines 4–5 provide an example of multiple stereotypes applied on the same element.

5.1.2. Stereotype properties

An intuitive approach to support the all-in-one-place editing of properties of the UML base element and applied stereotypes could consist in aggregating both sets into a single grammar rule, as follows.

```
1  XComponent returns XComponent:
2    (
3      isHwProcessor?='processor'? &
4      isAllocated?='allocated'? &
5      isHwCache?='cache'?
6    )
7    'component' name=ID '{'
8      'nbCores' '=' nbCores=INT?
9      'kind' '=' kind=Kind?
10     'level' '=' level=INT?
11     'caches' = '{'
12        caches+=XComponent*
13     '}'
14     'packagedElements' '=' '{'
15        packagedElement+=Component*
16     '}'
17   '}'
18 ;
```

However, this solution presents two fundamental issues. In order to integrate editing of properties of stereotypes and UML base element, application of a given stereotype and editing its properties are located in two different parts of the rule. This solution supports access to the values of stereotype properties, but does not provide control over their modifiability. For example, the `nbCores` property should only be included (thereby modifiable) if the `HwProcessor` stereotype is applied, i.e. `isHwProcessor` is *true*. Furthermore, treating stereotype and UML metaclass containment features in the same way leads to illegal models, see lines 11–16. There, `XComponent` instances in `caches` cause problems as their container remains unset. The instances should rather be inserted in the `packagedElement` containment feature and their insertion in `caches` managed whenever persisting the model.

Similarly to stereotype applications, our approach provides control over the modifiability of a given stereotype property by integrating boolean rule properties and Xtext validation rules. In particular, the solution above is modified as follows.

```
1  XComponent returns XComponent:
2    (
3      isHwProcessor?='processor'? &
4      isAllocated?='allocated'? &
5      isHwCache?='cache'?
6    )
7    'component' name=ID '{'
8      (
9        (hasNbCores?='nbCores''='nbCores=INT)?
10       & (hasKind?='kind''='kind=Kind)?
11       & (hasLevel?='level''='level=INT)?
12     )
13     packagedElement += Component*
14   '}'
15 ;
```

For each stereotype property, a simple validation check is defined to invalidate the model if the corresponding stereotype is not applied. The following Xtend snippet represents a validation check regulating `nbCores` and `HwProcessor` on `Component` instances.

```
1  @Check
2  def void checkHwProcessorCores
3       (XComponent xComponent) {
4    if (xComponent.hasNbCores &&
5      !xComponent.isHwProcessor) error(...)
6  }
```

5.2. Resource management extension

Given a grammar specification, Xtext generates a dedicated textual editor. The information flow among this and the modified model resource is orchestrated through a document provider component. All languages share a default implementation loading and serialising models as plain textual resources. Inevitably, introducing Xtext-based languages directly editing UML resources requires adaptations of the document provider to avoid that Xtext and UML editors corrupt the models whenever they are edited.

In order to address this issue, an extended document provider implementation introducing a transformation step whenever loading and persisting models is proposed. On the one hand, the editor is populated with the result of a model-to-text transformation applied on the persisted UML resource and conforming to the Xtext grammar. Inversely, the editor content is parsed using the generated Xtext language-specific parser and merged with the persisted UML resource using a model-to-model transformation. Both transformations only take into consideration metaclasses, stereotypes and properties covered by the Xtext-based textual language representing the specific UML profile.

The following Xtend template illustrates the model-to-text transformation rule serialising `Component` instances in XMarte. In this, stereotype application checks and property accesses are encapsulated into separate extension methods for the sake of brevity.

```
1  def transform(Component component) '''
2    «IF component.isAllocated»
3      allocated
4    «ENDIF»
5    «IF component.isHwProcessor»
6      processor
7    «ENDIF»
8    «IF component.isHwCache»
9      cache
10   «ENDIF»
11     component «component.name» {
12   «IF component.hasKind»
13     kind = «component.kind»
14   «ENDIF»
15   «IF component.hasNbCores»
16    cores = «component.nbCores»
17   «ENDIF»
18   «IF component.hasLevel»
19     level = «component.level»
20   «ENDIF»
21   «FOR element : component.packagedElement»
22     «element.transform»
23   «ENDFOR»
24 }
25 '''
```

Code in lines 2–10 checks whether the `Allocated`, `HwProcessor` or `HwCache` stereotypes are applied on the element. If yes, the corresponding token is added. Code in lines 12–20 handles stereotype properties. Finally, the execution continues recursively transforming the elements contained in the `packagedElement` containment feature.

The model-to-text transformation replaces the serializer generated by Xtext that is not able to handle stereotype serialisation nor to process UML elements not included in the Xtext language but present in the resource. The model-to-model synchronisation rule addressing pairs of `XComponent` and `Component` instances is illustrated below.

```
1  def merge(XComponent xComponent, Component component) {
2    // Component.name
3    component.name = xComponent.name
4    // Allocated
5    if (xComponent.isAllocated) {
6      component.applyStereotype(Allocated)
7      // Allocated.kind
8      if (xComponent.hasKind) {
9        component.kind = xComponent.kind
10     }
11   }
12   // HwProcessor
13   if (xComponent.isHwProcessor) {
14     component.applyStereotype(HwProcessor)
15     // HwProcessor.nbCores
16     if (xComponent.hasNbCores) {
17       component.nbCores = xComponent.nbCores
18     }
19   }
20   // HwCache
21   if (xComponent.isHwCache) {
22     component.applyStereotype(HwCache)
23     // HwCache.level
24     if (xComponent.hasLevel) {
25       component.level = xComponent.level
26     }
27   }
28
29   // Component.packagedElement - deleted
30   deletedComponents(xComponent, component)
31     .forEach[destroy]
32   // Component.packagedElement - inserted
33   insertedComponents(xComponent, component)
34     .forEach[create]
35   // Component.packagedElement - updated
36   updatedComponents(xComponent, component)
37     .forEach[merge]
38
39   // HwProcessor.caches
40   if (xComponent.isHwProcessor) {
41     component.caches = getComponents(component)
42       .filter[isHwCache]
43   }
44 }
```

Code in lines 4–27 propagates `XComponent` stereotype applications and properties on the `Component` instance, if set. As previously mentioned, extension methods are used to encapsulate operations involving stereotypes. Code in lines 29–37 manages the `packagedElement` containment reference with specific focus on `Component` instances. There, deleted components represent those persisted elements not having a counterpart in the Xtext model. Inversely, inserted components are only contained in the Xtext model. Finally, code in lines 39–43 handles the `HwProcessor.caches` stereotype reference. In our approach, indeed, stereotype references are not explicitly managed in the grammar. The reference is simply processed inserting the components stereotyped as `HwCache` from `packagedElement`.

The model-to-model transformation is needed to handle the merging of the textual model with the persistent UML resource, and more specifically to preserve UML elements in the resource that are not covered by the textual language.

6. Experiment

We set up and ran an experiment following in general the guidelines for experimentation in software engineering by Wohlin et al. (2012) and in particular the practical guide to experiments of software engineering tooling with human participants by Ko et al. (2015).

6.1. Design and execution

The variables identified for our experiment were:

- Independent variable: editing notations available for modelling purposes. Possible values: single-notation (S), blended multi-notation (B).
- Dependent variable: modelling effort in terms of time (MT). To test the effects on it, we separate it into modelling time with single notation (MT_S) and modelling time with blended notations (MT_B)

The independent variable was controlled to test the effects of its possible values on the dependent variable.

Our null and alternative hypotheses were:

Null hypothesis (H_0)**.** modelling time using seamless blended modelling features (MT_B) is equal or greater than modelling time using standard single-notation modelling features (MT_S): $MT_B \geq MT_S$.

Alternative hypothesis (H_1)**.** modelling time using seamless blended modelling features (MT_B) is lower than modelling time using standard single-notation modelling features (MT_S): $MT_B < MT_S$.

The experimental questions (EQs) that we wanted to answer while testing $\mathbf{H_0}$ were:

EQ1 – *Given blended modelling support, does usage of user-preferred modelling notations decrease modelling time compared to standard single-notation?*

EQ2 – *Given blended modelling support, does usage of task-optimal modelling notations decrease modelling time compared to standard single-notation?*

EQ3 – *Given blended modelling support, which among usage of task-optimal and usage of user-preferred modelling notations is more efficient in terms of modelling time?*

Before addressing EQs to test $\mathbf{H_0}$, we carried out an informal review of state of the art and practice of blended modelling in order to identify possible existing solutions to use for our purposes. As explained in detail in Section 3, there was no solution with fully-fledged blended modelling features. So, we first designed and implemented a proof-of-concept solution for blended modelling for UML profiles, focusing on textual and graphical concrete syntaxes and multiple stereotype application support. This solution, outlined in Section 4 and detailed in Section 5, allowed us to test $\mathbf{H_0}$ and answering to **EQ1-2-3**.

As suggested by Ko et al. we designed and ran our experiment through the following key activities:

Recruitment. We recruited potential participants by sending personal emails to entice people. We sent out the invitation to 50 subjects and gathered 18 willing to participate.

Selection. Out of the 18 recruited subjects, 14 were selected after applying inclusion criteria, which were the following:

- $\geq$3 years of experience with software design and development; based on the experience trichotomy proposed by Falessi et al. (2018), we did not include subjects with shorter experience level (0–2 years).
- $\geq$2 previous projects with UML-based software design in Eclipse/Papyrus.

Consent. We described the experiment in detail (including the final purpose and intent to report it as scientific peer-reviewed publication) to the subjects, whom consented by accepting to participate. All selected participants gave their verbal consent.

Procedure. The subjects were to carry out all tasks right after training. The experimenter prepared the modelling environment and switched between editors to initialise tasks. He also managed time keeping by measuring actual elapsed time per task. Questions were allowed in the training sessions and before starting with a task. Once a participant had completed all the tasks, we asked whether (s)he had a preferred notation; we used that information to aggregate data for answering to EQs.

Assignment to tasks. To distribute variation in participants' behaviour across conditions evenly, we chose to make all subjects carry out all tasks.

Training. The notations that we exploited were the UML standard graphical notation and two custom textual languages defined by us with Xtext, with related grammar and editor: *XMarte*, representing a sub-set of the `HwLogical` package of the MARTE profile for UML, and *SmText*, representing a minimal sub-set of UML state machines (supporting only states and transitions). All subjects started the experiment with a 3-hour training time to study the Xtext languages for *XMarte* and *SmText* as well as the tasks to be performed, including questions to the experimenter. More specifically, 1.5 h was dedicated to an introduction of by the interpreter of the technologies, the languages to be used and description of the experiment and modelling tasks. After that, the subjects had 1h for experimenting and trying out the modelling environment and the languages. To conclude the training session, 0.5 h was dedicated to an open session for Q/A (questions to the experimenter could be asked during the entire training session, too). During the training sessions, it was quite clear that different people reacted differently (i.e., varying learning curves) to the two notations, depending on their previous skillset and familiarity to them.

Tasks. We defined the following four modelling tasks:

- C1: create a UML platform package with two processors as follows:
  1. create a `Class` called 'Platform'
  2. create a `Component` called 'AProcessor' with applied stereotypes `Allocated` and `HwProcessor`
  3. set `kind` property of `Allocated` stereotype to 'executionPlatform'
  4. set `nbCores` property of `HwProcessor` to '4'
  5. add pre-existing `HwCache` element 'ACache' to `caches` property of `HwProcessor` of 'AProcessor'
  6. create a `Component` called 'BProcessor' with applied stereotypes `Allocated` and `HwProcessor` to 'BProcessor'
  7. set `kind` property of `Allocated` stereotype to 'executionPlatform'
  8. set `nbCores` property of `HwProcessor` to '1'
  9. add pre-existing `HwCache` 'BCache' to `caches` property of `HwProcessor` of 'BProcessor'
- M1:
  1. create a `Class` called 'CCache' with applied stereotypes `Allocated` and `HwCache`
  2. add 'CCache' to `caches` property of `HwProcessor` of 'AProcessor'

  `caches` property of `HwProcessor` stereotype of `Component` 'AProcessor'
- C2: populate a UML state-machine diagram (see Fig. 4) as follows:
  1. create an `Initial` called 'state_0'
  2. create a `State` called 'state_1'
  3. create a `State` called 'state_2'
  4. create a `State` called 'state_3'
  5. create a `Join` called 'state_4'
  6. create a `FinalState` called 'state_5'
  7. create a `Transition` from 'state_0' to 'state_1'
  8. create a `Transition` from 'state_1' to 'state_2'
  9. create a `Transition` from 'state_1' to 'state_3'
  10. create a `Transition` from 'state_2 to 'state_4'
  11. create a `Transition` from 'state_3' to 'state_4'
  12. create a `Transition` from 'state_4' to 'state_5'
- M2: rename all states from 'state_x' to 's_x'

The graphical models resulting from task C1 and C2 are depicted in Figs. 3 and 4 respectively, while the textual ones are listed in Listing 1 and Listing 2.

```
package Platform{
  allocated processor component AProcessor {
    kind = executionPlatform
    cores = 4
    allocated cache component ACache {
      kind = executionPlatform
      level = 1
    }
    allocated cache component CCache {
      kind = executionPlatform
      level = 1
    }
  }
  allocated processor component BProcessor {
    kind = executionPlatform
    cores = 1
    allocated cache component BCache {
      kind = executionPlatform
      level = 2
    }
  }
}
```

Listing 1: Textual model resulting from the C1 modelling task

```
statemachine SM{
   states{
     initialState state_0
     state state_1
     state state_2
     state state_3
     joinState state_4
     finalState state_5
   }
   transitions{
     state_0 to state_1
     state_1 to state_2
     state_1 to state_3
     state_2 to state_4
     state_3 to state_4
     state_4 to state_5
   }
}
```

Listing 2: Textual model resulting from the C2 modelling task

Each participant was asked to perform all tasks described above sequentially (in the order above). The experimenter prepared the modelling environment and the specific (graphical or textual) editor to be used. In graphical mode, tasks were performed in the Papyrus graphical editor (including diagram editor, model explorer and properties view): C1 and M1 started in an editor initialised with an empty class diagram, while C2 and M2 in an editor initialised with an empty state machine diagram. Note that the graphical layout (i.e., the placement of graphical elements) was not considered in the experiment and participants were free to place elements as they liked in the graphical editing space. In textual mode, C1 and M1 were performed in a Xtext textual editor generated for the *XMarte* language, while C2 and M2 in a Xtext textual editor generated for the *SmText* language. Each participant carried all tasks twice, once per notation (graphical and textual).

Outcome measurement. We were interested in task completion time (i.e., modelling effort in terms of time). For each modelling task and used notation, elapsed time for modelling was measured for each participant and the arithmetic mean across subjects was calculated.

Debriefing. We debriefed the participants by asking for general comments and feelings while carrying out the tasks.

6.2. Results

The modelling tasks were performed individually by 14 subjects, with varying experience levels (in years: min 3, max 9, mean 4) in software (UML-based) design and development. Three user types were considered:

- *G*, preferring graphical notation;
- *T*, preferring textual notation;
- *N*, no preference.

The distribution of types was: G = 6, N = 3, T = 5. Table 4 shows the results of the experiment, where we provide the arithmetic mean (μ) of the individual sets of values overall, per notation, per user-preferred notation, and per task-optimal notation. We

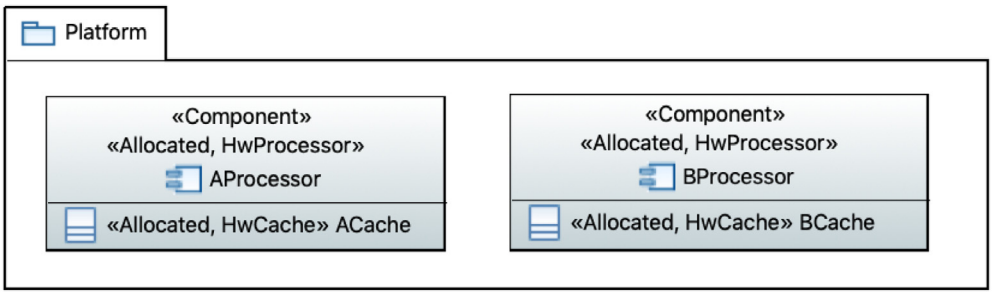

Fig. 3. Resulting class diagram from C1 modelling task.

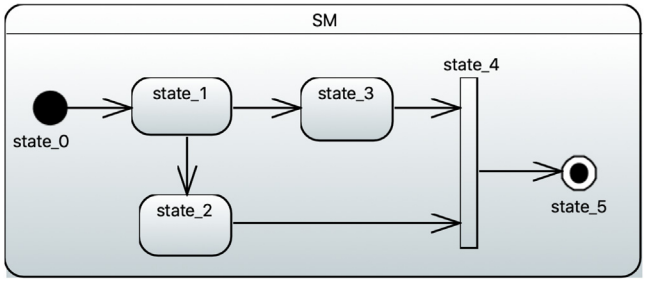

Fig. 4. Resulting state-machine diagram from C2 modelling task.

Table 4
Aggregated elapsed modelling times (in seconds).

	Settings		μ modelling time					
	User type	Notation	C1	M1	C2	M2	Σ	σ
1	G	**Graphical**	130.2	39	**63.5**	**32.8**	**265.5**	18.9
2		Textual	**103.9**	**21.4**	164.1	35.2	324.6	25.2
3	N	Graphical	150.9	43.2	**83.5**	42.1	319.7	23.1
4		Textual	**93.2**	**22.6**	165.8	**28.1**	**309.7**	24.9
5	T	Graphical	171.1	81.5	**101.2**	61.2	415	35.7
6		**Textual**	**71.6**	**18.7**	134.5	**20.3**	**245.1**	12.9
7	All	Graphical	150.7	54.6	**82.7**	45.4	333.4	
8		Textual	**89.6**	**20.9**	154.8	**27.9**	**293.2**	
9	G-T	User-pref.	100.9	24.6	84	27.7	237.2	
10	All	***Task-opt.***	***89.6***	***18.1***	***82.7***	***24.1***	***214.15***	

provide the standard deviation on the individual total modelling times per user type and notation too.

We measured the time it took for each participant to perform the tasks with each notation (only active modelling time) and calculated the arithmetic mean for each group and task by notation. Rows 1–2, 3–4 and 5–6 show the mean times (μ) for each user type (G, N, T respectively) to complete each of the four tasks by notation. The user-preferred notation is highlighted in bold (i.e., 'Graphical' for G users, 'Textual' for T users, while N users did not have any preference).

Looking at the total modelling times (column Σ), the use of blended notations (rows 9–10) leads to lower modelling times than by using a single notation (rows 1–8). This made us refute $\mathbf{H_0}$ and accept $\mathbf{H_1}$.

Looking at the performance of the individual user types, we can notice that, overall, G (row 1) and T (row 6) performed better using their preferred notation (Σ column). In some specific cases though, the free choice of notation does not pay off (i.e., C1 and M1) and a task-optimal notation (i.e., textual for C1 and M1) would be preferable. In fact, we noticed that different notations are more suitable for different modelling tasks and that, besides previous experience with one or another, in general enforcing the use of a task-optimal notation decreases modelling time. In our experiment, the usage of task-optimal notations led to the fastest modelling times (row 10) overall, independently of the task at hand.

Overall (rows 7–8), textual editing resulted faster when creating stereotyped elements and setting their properties (C1). This is due to the possibility to customise Xtext grammars to only require a minimum amount of information to be entered by the modeller (while the underlying base UML elements are created by our stereotypes application transformation). The same goes for the modification of an existing model by inserting a new model element (M1). One of the issues related to graphical editing is the need to interact with multiple windows (e.g., properties view for profile-related operations) leading to an overly high amount of mouse clicks needed to navigate across views and edit model elements. The problem of mouse clicks affects the diagram editing too, but it is mitigated in some cases by the intuitiveness of visual diagrams.

The creation of state-machines resulted to be faster with the graphical notation (C2). This is mainly due to a swifter creation of transitions between states using the graphical editor (more effective also than using copy&paste&modify in textual editing) As expected, the textual notation resulted to be faster when renaming model elements (M2). This was thanks to the possibility to copy&paste in a swift manner. We did not use regex search&replace to avoid editor-related bias. In fact, regex search&replace would not work for, e.g., applied stereotype names using graphical editors (those changes would require a delete/add pair of actions instead), while it would work on text.

Let us summarise the experiment's results by answering our EQs:

- **EQ1** – *Given blended modelling support, does usage of user-preferred modelling notations decrease modelling time compared to standard single-notation?* Yes, the usage of user-preferred notations can decrease modelling time (Σ in row 9 as opposed to Σ in rows 7–8).
- **EQ2** – *Given blended modelling support, does usage of task-optimal modelling notations decrease modelling time standard single-notation?* Yes, the usage of task-optimal notations can decrease modelling time (Σ in row 10 as opposed to Σ in rows 7–8).
- **EQ3** – *Given blended modelling support, which among usage of task-optimal and usage of user-preferred modelling notations is more efficient in terms of modelling time?* The usage of task-optimal notations is more efficient in terms of modelling time (Σ in row 10 as opposed to Σ in rows 9).

While this experiment was not meant to provide a definitive quantification of benefits of blended modelling, it was very useful to give a glimpse on them when users have different editing preferences/skills and face various modelling tasks, and we can conclude that blended capabilities, no matter whether

the inclination is towards user-preferred or enforced task-optimal notations, bring improvements in the modelling activities and decreases modelling time overall (rows 9–10). Despite the results related to EQ3, we tend to believe that a balanced combination of user-preferred and task-optimal notations could represent the best modelling solution. The goodness of such a combination depends on two factors: modelling tasks to be performed and stakeholder's preference/skills. For this reason, it is hard to identify a generic optimal combination.

Note that the modelling tasks were run sequentially and individually, thus not requiring merge/diff support. In a fully collaborative scenario, such a support would be vital, therefore it is paramount to equip multi-notation modelling with powerful collaborative features.

6.3. Threats to validity

In this subsection we argument on the potential validity threats of our experiment and how we eventually mitigated them.

6.3.1. Internal validity

Internal validity refers to extraneous variables and inaccurate settings that may have had a negative impact on the design of the experiment (Brewer and Crano, 2000). In this study, subjects were selected in a homogeneous population of computer scientists with focus on software engineering and modelling experience. The results show that possibly small differences in the background of the subjects did not play any significant role. Furthermore, there was no repeated testing on the same subjects and no change in the modelling environment. Since each subject was only subject to one complete experiment treatment, no differential attrition in terms of subjects withdrawing between experiment rounds occurred. Although we tried to shape the modelling tasks so as not to favour any subject type, it might always occur that their actual construction may favour one over another. Nevertheless, given the experiment results we believe that this possibility was not to affect the drawn conclusions.

6.3.2. External validity

External validity refers to the generalisability of causal findings with respect to the desired population and settings (Brewer and Crano, 2000). In our experiment, researchers with software design and development experience between three and nine years were employed as experiment subjects. Could the results be generalised to even more experienced researchers? The fact that, in our population, we did not notice extreme advantages for more experienced researchers make us believe that this generalisation can be made, although it is hard to quantify it. Moreover, the subjects were a mix of academic researchers (10) with experience in industrial projects and industrial researchers (4). Can the results be generalised to industrial practitioners with more specific (narrow) expertise and stronger notation-specific preferences? The results of the experiment make us believe in this generalisation, although an extended experiment would be needed to quantify the generalisability of the results.

6.3.3. Construct validity

Construct validity refers to the extent to which an identified causal relationship can be generalised from the particular methods and operations of a specific study to the theoretical constructs and processes they were meant to represent (Brewer and Crano, 2000). Regarding the modelling scenarios, although they were limited in terms of complexity, they represented a fair variety of common modelling situations to test the usefulness of different modelling notations. Regarding the time measurements, the validity of the elapsed times to complete the individual scenarios can always be questioned since possibly affected by several variables out of control. We minimised this threat by measuring active monotonous elapsed modelling times rather than total times for completing the scenarios.

6.3.4. Ecological validity

Ecological validity concerns the level to which experimental conditions approximate the real world (Brewer and Crano, 2000). In our experiment, we used the actual modelling environment we wanted to assess and real software developers/designers to maximise ecological validity. One threat is represented by the fact that the modelling scenarios where not taken directly from a real modelling project, but rather chosen ad-hoc to be easily understandable by non experts of specific domains, to exploit the most common aspects of UML and profiles, and to be simple enough for the execution times to mostly reflect the blended modelling features rather than the modelling of complex concepts. We are planning more extensive studies in industrial settings, which would increase ecological validity even more.

6.3.5. Conclusion validity

This refers to the relationship between experiment measurements and obtained findings (Wohlin et al., 2012). We mitigated potential threats by systematically applying and documenting synthesis of measurements. We only considered active elapsed modelling times (i.e., time when subjects were actively carrying out a modelling task) and only calculated average means across sub-groups of two subjects in each subjects group. Note that this experiment was limited in terms of population size and complexity of modelling tasks, since our goal was to get a proof-of-concept of the potential benefits of blended solutions, given the current prototypical framework. Further experiments with more complex modelling scenarios and a larger population will be run once the framework reaches a release level. Then, we will also test the possibility to use multiple notations for different sub-tasks, where we believe great advantages of using blended modelling would arise. Anyhow, considering the overall results of the experiment (lower modelling time with blended notations) in relation to the limited complexity of modelling scenarios, we expect the benefits of blended solutions to be even sharper when dealing with more complex modelling scenarios and carried out by domain experts.

7. Discussion

To experiment on whether a blended modelling environment could provide beneficial improvements to modelling activities and decrease modelling time, we implemented a prototype based on Papyrus and Xtext in the Eclipse environment. There are alternative approaches to the development of a blended modelling environment for UML profiles, but any such tool must address some common issues such as the synchronisation of multiple representations by editors as well as their persistence, or the handling of cross-references between model elements.

For the purpose of synchronisation, some tools may choose to keep separate representations and use explicit transformations between them to keep them up-to-date. An advantage of this approach is that representations could be persisted in the format preferred by the users or tools. The disadvantage is that such transformations must be explicitly written and maintained, when the language evolves. This entails the problem of keeping the transformations consistent with the language and with each other. Furthermore, keeping separate representations would be prone to error in a collaborative environment, as multiple copies

of the same model could be modified separately in inconsistent ways.

A different approach is that of projectional editors which, as previously discussed, keep only one underlying representation of the model's abstract syntax and different editors and viewers are responsible for the conversion between abstract and concrete syntax. The obvious advantage here is that it avoids the problems described in the previous paragraph, as there is no need for transformations between representations. A disadvantage is that it may not provide the same flexibility of storing multiple representations.

For the purpose of supporting UML-based profiles specifically in a blended environment, there are additional challenges, in particular, the support for multiple stereotype applications. Our solution leverages Xtext's own mechanisms which infer an appropriate Ecore meta-model upon which the generated environment operates. To achieve the seamless synchronisation we override Xtext's document provider behaviour, which maps domain elements to documents, and is used by (textual) editors to update the textual representation, the abstract syntax elements, notify listeners of changes, etc. By overriding Xtext document provider, we are able to maintain only one common resource.

Our approach can be considered as semi-projectional, differing from existing projectional tools, such as MPS, in several ways: (1) it addresses support for DSMLs defined as UML profiles; (2) the textual representation is truly textual, as opposed to a form-based representation, and thus enables the use of text-based tools (e.g. regex search); (3) it relies on existing mature frameworks for graphical modelling with UML (Papyrus) and text-based IDEs (Xtext).

In Section 4, we listed a set of five improvements to current practices brought by our blended modelling framework, which we discuss in the followings.

7.1. Grammar customisability

Given a specific UML profile, Xtext grammars could be semi-automatically generated to define a textual concrete syntax for the profile (or part of it). Xtext provides an out-of-the-box features for generating grammars from UML profiles. The generation process is not customisable nor parametric. That is to say, generated grammars are in most cases unusable in practice without refactoring and manual tuning. That was the case for the MARTE profile. In our solution, we manually created and customised an Xtext grammar from a sub-set of MARTE. Doing so, we were able to provide a customised and convenient solution to blended modelling for UML and MARTE.

Nevertheless, a blended modelling environment shall provide a specific feature for parametric and semi-automatic generation of Xtext grammars from profiles, so as to be customisable and refactorable to fit the stakeholder's needs. This would not jeopardise the (de-)serialisation mechanisms as long as it does not break the conformance of models to the UML profiles specification (i.e., metamodel). We are currently working on such a feature, which is expected to heavily simplify the job of a DSML developer.

7.2. Multiple stereotypes application in textual format

The possibility to apply multiple stereotypes (coming from the same and/or from different profiles) is crucial to provide a full-fledged modelling environment for UML and profiles. Currently, since Xtext does not provide out-of-the-box features for generating grammars entailing concepts embodied in different profiles (or packages within a profile), this is not trivial to achieve. In this work, we showed how we provided our framework with such a feature combining grammar rules representing stereotyped metaclass instances and runtime validation checks. Stereotype applications correspond to boolean rule properties, literally indicating whether or not a given stereotype is applied. A similar approach supports stereotype properties together with dedicated validation checks regulating their availability, i.e. prevent editing the properties of non-applied stereotypes. Each stereotype is managed independently, hence an indefinite number of stereotype applications can be handled on a single element. Furthermore, users have the possibility to easily introduce additional constraints concerning one or more stereotypes in the form of validation checks, e.g. `HwProcessor` instances should contain at least one packaged element stereotyped as `HwCache`.

When a stereotype is applicable to multiple UML base elements, there would be duplicates cvx in the grammar (one rule per base element). Initially, we tried to avoid this by using Xtext's *fragment* rules. However, since fragments were not compatible with unordered sets in Xtext and we did not want to impose a specific order for stereotype tokens and stereotype properties, we decided to get rid of fragments and opt for better usability of the textual languages instead rather than a more condensed grammar.

7.3. Cross-profile modelling

One of the main characteristics of our solution is that it does not entail complex *profile-specific* explicit synchronisation transformations between textual and graphical notations. This makes most of the framework cross-profile. The only transformations needed for propagating stereotype applications across the notations are mainly based on the grammar, hence generalisable. The mechanism itself is cross-profile and profile-specific instances such as XMarte can be generated from its metamodel definition in a semi-automated manner, with the help of the blended DSML developer for more complex cases.

7.4. On-the-fly changes propagation

Model changes done in one view are seamlessly reflected and visible in the other views (graphical, textual and tree-based views in Fig. 5). On-the-fly propagation is achieved thanks to a single persistent resource shared among the views. Although the propagation does not produce tangible delays in the rendering of the changed model across notations, the stakeholder may want to disable it for specific reasons (e.g., sketching purposes with non-conforming models). This feature is currently not available in the framework, but we are working on it.

7.5. Cross-notation multi-view modelling

We showed how an Xtext-based textual language (XMarte), with related grammar and editor, representing only a sub-set of the `HwLogical` package of MARTE can seamlessly work on a UML resource containing other UML and MARTE concepts (e.g., UML elements in `SW_Functions` package and MARTE `«Allocate»` in Fig. 5). For instance, XMarte would be suitable for a platform modeller, who might not need or want to view functional details. This is possible thanks to our enhanced Xtext resource management, which, instead of overwriting the in-memory model with plain text, propagates changes directly to the UML resource, the same used for editing and rendering UML models in the graphical and tree-based views by Papyrus.

As further enhancements of the multi-view nature of our framework, we plan to provide features for layered accessibility to shared information with multiple read/write access rights levels, enforcement of specific notations depending on the stakeholder roles, wizard-based customisability of perspectives, and inclusion of additional notations (besides graphical and textual, such as tabular, form, etc.).

8. Conclusion

In this paper we described our work towards an open-source framework for blended graphical–textual UML modelling based on Eclipse, Papyrus and Xtext. The framework aims at advancing the state of the practice of blended UML modelling by providing support for: textual grammar customisability, flexible application of multiple stereotypes on UML base elements using textual languages, cross-profile modelling based on blended notations, on-the-fly changes propagation across notations, and cross-notation multi-view modelling. We ran two experiments to assess the potential impact of blended solutions on modelling effort, in terms of time.

Different notations are more suitable for different modelling tasks and, besides previous experience of the modeller with one or the other, enforcing the use of a task-optimal notation can decrease modelling time. Overall, stakeholder's free choice of notation does decrease modelling time and subjects used to a specific notation perform overall better using their preferred notation. Nevertheless, for specific modelling tasks, enforcing a task-optimal notation has a better impact on modelling time. To summarise, we can conclude that a balanced combination of freely chosen and fixed task-dependent notations may represent the optimal solution. The goodness of such a combination depends on two factors: modelling tasks to be performed and stakeholder's preferences. For this reason, it is hard to identify a generic optimal combination. In any case, we could observe that blended capabilities bring improvements in the modelling activities and decreases modelling time.

9. Future work, dissemination and communication

One limitation of our approach is that the process of developing the blended environment for UML-based DSMLs is not fully automated, as the designer is expected to adapt the Xtext grammar according to her taste and write the endogenous model-to-model transformation described in the technical solution. This transformation may be achieved in a (semi-)automated fashion, but we leave this for future work, as it was not needed to address our EQs.

Our current solution does not persist a textual file representing the model since synchronisation is achieved in-memory. Nevertheless, we are aware of the fact that the user could want to work on a plain-textual file for, e.g. storing in a version control system. We plan to enhance the approach to give the users the possibility to persist the model in its textual format, and transform it to and from an in-memory model to exploit the blended modelling environment. When the model is in memory, all synchronisations would occur on it, but when saving it to a persistent resource, the user would get the choice to transform it back to text or as a UML resource.

In order to provide a more definitive quantification of the potential improvements brought by blended modelling, we plan to run additional experiments with practitioners from different domains and at different phases of the development lifecycle (e.g., requirements modelling, software design, information modelling, etc.). We have built an international consortium across 4 countries and just started a project in the ITEA3 cluster programme on blended graphical–textual modelling called BUMBLE.[5] In that context, we plan to build upon the work reported in this paper for the framework to reach a full-fledged level. In the same context, we will run more extensive controlled experiments and industrial case-studies too.

The solutions originated from this work and enhanced in the BUMBLE project will be released as part of the Eclipse ecosystem and with EPL license. We aim at establishing a long term Eclipse project providing and supporting those solutions. Since the base technologies are meant to be released in the open-source community, an important element of the dissemination plan consists in leveraging the different opportunities provided in the Eclipse community, including Eclipse conferences (e.g., EclipseCon Europe) and marketing. We will also collaborate with the Eclipse Working Groups, Papyrus and Capella Industry Consortia to reach out to industrial MDE tool users.

We plan to disseminate results via research forums (conferences, workshops), corporate presentations, participation to industrial events like expos, on-line community forums for Eclipse, social media, fact sheets and wikis.

The intended end users for these technologies are practitioners, researchers, teachers and students as follows:

- Companies using modelling tools: seamless blended modelling has the potential to lower development time. Communication between stakeholders will improve and the communication overhead due to mismatching views and different familiarity level to specific notations is expected to decrease both at provider level (company) and at end-user level (company's customers). The learning time for a new user is reduced thanks to multiple notations. The possibility to alternate graphical and textual is expected to provide a better mutual understanding between collaborating users and across different stakeholders already from the learning phase.
- Research: blended modelling will provide a flexible multi-notation infrastructure boosting research in multiple domains. The possibility to switch between notations allows to focus on the semantics of the information to be modelled rather than on their representation. Graphical notations provide a means for reasoning and sketching, while textual notations can be used for maximising the efficiency of model manipulations and transformations. Collaborative features of the framework will boost research carried out in a collaborative fashion too.
- Education/training: blended modelling techniques and tools can be employed for teaching and training, and are expected to dramatically push down the learning curve for modelling activities, by making students/trainees less reliant on one single specific notation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research is funded by VINNOVA, Sweden through the BUMBLE project (18006) and the Knowledge Foundation through the HERO project (20180039).

Appendix. Blended modelling environment in Papyrus

See Fig. 5.

[5] https://itea3.org/project/bumble.html.

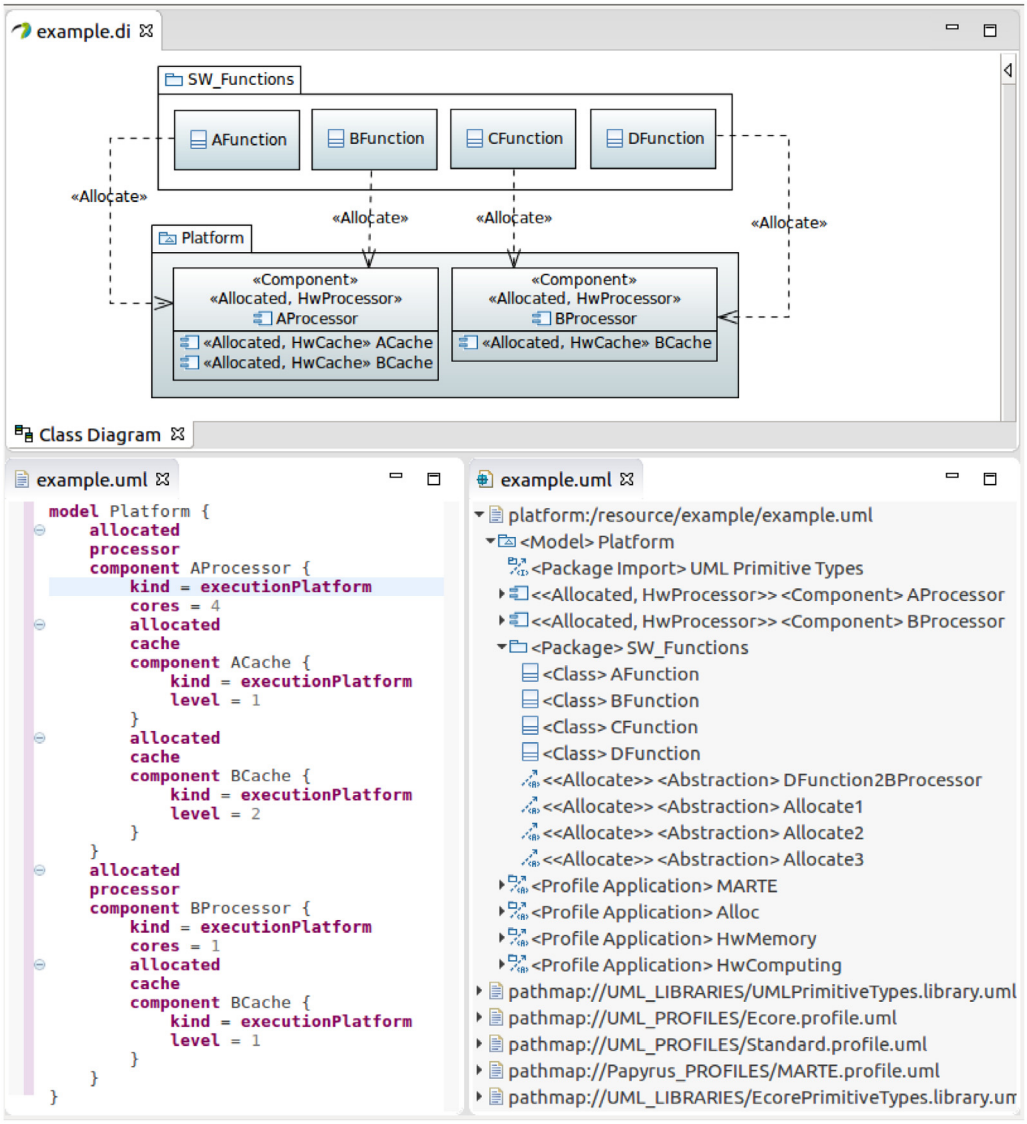

Fig. 5. XMarte textual editor (bottom-left side), Papyrus tree-based editor (bottom-right side), and Papyrus graphical editor (upper side) in Eclipse.

References

Addazi, L., Ciccozzi, F., Langer, P., Posse, E., 2017. Towards seamless hybrid graphical–textual modelling for UML and profiles. In: Procs of ECMFA. Springer, pp. 20–33.

Andrés, F.P., De Lara, J., Guerra, E., 2007. Domain specific languages with graphical and textual views. In: Procs of AGTIVE. Springer, pp. 82–97.

Atkinson, C., Gerbig, R., 2013. Harmonizing textual and graphical visualizations of domain specific models. In: Procs of GMLD, pp. 32–41.

Boucké, N., et al., 2008. Characterizing relations between architectural views. Software Architecture. Springer Berlin Heidelberg, pp. 66–81.

Brewer, M.B., Crano, W.D., 2000. Research design and issues of validity. In: Handbook of research methods in social and personality psychology. pp. 3–16.

Charfi, A., Schmidt, A., Spriestersbach, A., 2009. A hybrid graphical and textual notation and editor for UML actions. In: Procs of ECMFA. Springer, pp. 237–252.

Cicchetti, A., Ciccozzi, F., Pierantonio, A., 2019. Multi-view approaches for software and system modelling: a systematic literature review. Softw. Syst. Model..

Ciccozzi, F., Cicchetti, A., Sjödin, M., 2013. Round-trip support for extra-functional property management in model-driven engineering of embedded systems. Inf. Softw. Technol. 55 (6), 1085–1100.

Ciccozzi, F., Tichy, M., Vangheluwe, H., Weyns, D., 2019. Blended modelling – What, why and how. In: MPM4CPS Workshop. http://www.es.mdh.se/publications/5642-.

Eclipse Foundation, 2020e. Eclipse sprotty. https://projects.eclipse.org/proposals/eclipse-sprotty, Latest access: 2020-05-05.

Eclipse Foundation, 2020l. Eclipse modeling framework. https://www.eclipse.org/modeling/emf/, Latest access: 2020-05-05.

Eclipse Foundation, 2020m. Papyrus. https://eclipse.org/papyrus/, Latest access: 2020-05-05.

Eclipse Foundation, 2020n. Xtext. http://www.eclipse.org/Xtext/, Latest access: 2020-05-05.

Emery, D., Hilliard, R., 2009. Every architecture description needs a framework: Expressing architecture frameworks using ISO/IEC 42010. In: Procs of ECSA.

Falessi, D., Juristo, N., Wohlin, C., Turhan, B., Münch, J., Jedlitschka, A., Oivo, M., 2018. Empirical software engineering experts on the use of students and professionals in experiments. Empir. Softw. Eng. 23 (1), 452–489.

Gheorghies, O., 2020h. MetaUML. https://github.com/ogheorghies/MetaUML, Latest access: 2020-05-05.

Hakala, A., 2020f. LightUML. http://lightuml.sourceforge.net/, Latest access: 2020-05-05.

Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S., 2011. Empirical assessment of MDE in industry. In: Procs of ICSE. IEEE, pp. 471–480.

IBM, 2020a. IBM Rational software architect. http://www-03.ibm.com/software/products/en/ratsadesigner/, Latest access: 2020-05-05.

Jetbrains, 2020j. Jetbrains MPS. https://www.jetbrains.com/mps/, Latest access: 2020-05-05.

Ko, A.J., Latoza, T.D., Burnett, M.M., 2015. A practical guide to controlled experiments of software engineering tools with human participants. Empir. Softw. Eng. 20 (1), 110–141.

Koehnlein, J., 2020d. FXDiagram. http://jankoehnlein.github.io/FXDiagram/, Latest access: 2020-05-05.
Lazăr, C.-L., 2011. Integrating ALF editor with eclipse UML editors. Stud. Univ. Babes-Bolyai Inform. 56 (3).
Maro, S., Steghöfer, J.-P., Anjorin, A., Tichy, M., Gelin, L., 2015. On integrating graphical and textual editors for a UML profile based domain specific language: An industrial experience. In: Procs of SLE, pp. 1–12.
Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-M., Cheng, B.H., Collet, P., Combemale, B., France, R.B., Heldal, R., Hill, J., et al., 2014. The relevance of model-driven engineering thirty years from now. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 183–200.
PlantUML team, 2020i. PlantUML. http://plantuml.com/, Latest access: 2020-05-05.
Scheidgen, M., 2008. Textual modelling embedded into graphical modelling. In: Procs of ECMFA. Springer, pp. 153–168.
Selic, B., Gérard, S., 2013. Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. Elsevier.
Selic, B., Gullekson, G., Ward, P.T., 1994. Real-Time Object Oriented Modeling. Wiley & Sons.
SparxSystems, 2020b. SparxSystems enterprise architect. http://www.sparxsystems.eu/enterprisearchitect/, Latest access: 2020-05-05.
TextUML team, 2020g. TextUML. http://abstratt.github.io/textuml/, Latest access: 2020-05-05.
The Qt Company, 2020k. Qt. https://www.qt.io/, Latest access: 2020-05-05.
Umple team, 2020c. Umple. http://cruise.eecs.uottawa.ca/umple/, Latest access: 2020-05-05.
Wimmer, M., Kramler, G., 2005. Bridging grammarware and modelware. In: Procs of MoDELS. Springer, pp. 159–168.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.

Lorenzo Addazi is a Ph.D. student in parallel modelling and programming languages for heterogeneous embedded systems at IDT, Mälardalen University (Sweden). His research interests also include model-driven software engineering, model versioning and comparison, hybrid modelling, domain-specific languages, software language engineering and programming models. Contact him at lorenzo.addazi@mdh.se, or visit http://www.es.mdh.se/staff/3276-Lorenzo_Addazi.

Federico Ciccozzi is an Associate Professor at Mälardalen University (Sweden). His research specialises in: definition of DSMLs, model transformations, system properties preservation, multi-paradigm modelling, model versioning, combination of MDE and CBSE for complex systems, blended modelling, language and compiler engineering.

Federico has organised over 40 conferences, tracks, sessions, workshops and journal special issues. He has been program committee member of over 40 scientific events in the last year. He is associate editor of IET Software, guest editor of SoSyM and JISA. He has (co-)authored over 100 peer-reviewed publications. More info at: http://www.es.mdh.se/staff/266-Federico_Ciccozzi.