ITEA Office
High Tech Campus 69 - 3
5656 AG Eindhoven
The Netherlands

T + 31 88 003 6136
E info@itea3.org
W www.itea3.org

ITEA 3 is a EUREKA strategic ICT cluster programme

BUMBLE Deliverable D4.2

Synchronisation of blended notations

Project Acronyms

<ACR>	<Acronyms>
BUMBLE	Blended modeling for Enhanced Software and Systems Engineering
DSML	Domain-Specific Modeling Language
UML	Unified Modeling Language
EMF	Eclipse Modeling Framework
UML-RT	UML for Real-time
EBNF	Extended Backus-Naur Form
XML	eXtensible Markup Language
Alf	Action language for foundational UML
ETL	Epsilon Transformation Language
MML	Mapping modeling Language
GMF	Graphical Modeling Framework
PSS	Portable test and Stimulus Standard
JAVACC	Java Compiler Compiler
HOT	Higher order transformation
DMA	Direct Memory Access
SM	State Machine
MOF	Meta-Object Facility
Papyrus-RT	Papyrus for Real-Time
QVT	Query/View/Transformation
QVTo	QVT operational

Versions

Release	Date	Reason of change	Status	Distribution
V0.1	20/09/2021	Defined structure	Draft	WP4 partners
V0.2	01/11/2021	Complete draft of the first version of the deliverable	Draft	BUMBLE consortium
V1.0	11/11/2021	Updated with comments on V0.2, to be submitted to ITEA portal	Final	Uploaded to ITEA portal
V1.1	08/01/2023	Draft of the updates for the second version of the deliverable	Draft	WP4 partners
V1.2	08/02/2023	Complete draft of the second version of the deliverable	Draft	BUMBLE consortium
V2.0	20/02/2023	Updated with comments on V1.2, to be submitted to ITEA portal	Final	Uploaded to ITEA portal

Executive Abstract

In this deliverable we describe the theory and process followed to achieve the model transformation mechanisms for synchronisation by means of higher-order transformations (HOTs). Starting from ad-hoc synchronisation solutions, we generalize them and provide the details on how the mapping rules were implemented into the HOTs as well as input and output formats for each of them. HOTs produce model transformations that can be used for multiple purposes, i.e. synchronisation, migration, reconciliation in case of metamodel evolutions. We also provide details on how the HOTs were validate on multiple use cases.

Table of contents

1. Introduction

In this deliverable, we describe the core activities and results of task T4.3 in Work Package 4 (WP4). More specifically, the theories defined in T4.1 and T4.2 in terms of mapping between DSMLs are exploited to drive higher-order transformations (HOTs), which are designed and implemented in T4.3.

HOTs are powerful transformations defined at the metamodeling level [7]. Given two DSMLs and a mapping model between them, HOTs are able to automatically generate bidirectional model transformations for synchronizing models conforming to the two DSMLs. Specific HOTs will realize the mapping rules defined in the previous tasks of the BUMBLE project (see D4.1).

The usage of HOTs entails at least three scenarios:

- synchronisation across different languages (either same or different notations)
- synchronisation across different notations of the same language
- Co-evolution mechanisms for models conforming to an evolving language

Except for specific synchronisation with the transformation tool DClare (see Section 3.4) that also applies to the MPS environment, the rest of this deliverable concerns synchronisation in the context of the Eclipse Modeling Framework (EMF), since the projectional nature of MPS requires a different kind of synchronisation mechanisms.

The work and solutions described in this deliverable contribute to the following BUMBLE Technology Bricks and requirements:

Technology bricks	Description of main contributions	Main requirements
Blended Model Access	We have provided both ad-hoc synchronisation mechanisms for specific languages and notations and provided a set of generic HOTs able to automatically generate synchronisation transformations between any pair of Ecore-based DSMLs	BC3, BC4, BC10, BC11, BC12, BC13, BT7, BT8, BT9, BT10, BT11, BT12, BT13, BT19, BT21, BT25
Platform Integration	We have integrated multiple Eclipse-based technologies as well as bridged Eclipse and MPS for blended (meta-)modelling	BT5, BT6, BT9, BT10, BT17, BT18
Meta-(model) co-evolution	HOTs are also used for co-evolution (in terms of migration and reconciliation) of blended models in case of evolutions of the metamodel	BC9, BT22, BT24

The remainder of this deliverable is structured as follows. In Section 2, we provide an overview of state of the art in synchronisation, including HOTs. In Section 3, we describe the actions and results in relation to the solutions achieved in this WP for synchronisation across notations as well as across different modeling artefacts. All solutions have successfully been implemented (or are being implemented) in industrial use cases in BUMBLE. In Section 4, we introduce the usage scenarios and

the intended solutions for HOTs, together with the selected technologies and the motivation behind those choices. We conclude the deliverable by outlining the next steps in Section 5.

2. State of the art on synchronisation and HOTs

Very little has been done in combined means for synchronised editing in both textual and graphical syntaxes and customisation of the concrete syntaxes for MOF-based languages (including Ecore and UML).

Synchronised editing in multiple (customisable) concrete syntaxes for non-MOF DSMLs can also be realised with Qt (https://www.qt.io/) 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 (e.g., autocompletion, validation), which means that a DSML engineer needs to program most of it manually (although well supported by IDE tools).

Several research efforts have been directed at mixing textual and graphical modeling. A textual editor for the Action Language for Foundational UML (Alf) has been developed based on Xtext [2].

In [1], the authors provide an approach for defining combined textual and graphical DSMLs based on the 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 its corresponding portion of the common metamodel.

Charfi et al. [3] explore the possibilities to define a single concrete syntax supporting both graphical and textual notations. synchronisation is provided only for a very small subset of UML while we focus on the generation of synchronisation mechanisms from any Ecore-based DSML.

In [12], the authors propose a Blended Metamodeling Framework (BMF) that enables the development of metamodels through both graphical and textual notations interchangeably. Particularly, a restricted natural language template was introduced for the specification of DSMLs textually. The explicit mappings were defined between the natural language template and Xcore elements to achieve runtime synchronizations among graphical (EMF) and textual (natural language) notations by utilizing model-to-text and text-to-model transformations. This facilitated the development of Domain-Specific Modeling Languages (DSMLs) exploiting multiple editing notations and language workbenches.

In [4], the authors provide the needed steps for embedding generated EMF-based textual model editors into graphical editors defined in terms of the Eclipse Graphical Modeling Framework (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 (i.e., blending). The focus of their solution is on the integration of editors based on EMF, while the aim in BUMBLE, and specifically T4.3, is to generate automatically synchronisation mechanisms across existing Ecore-based textual and graphical notations. Moreover, the change propagation mechanisms proposed by the authors are on-demand triggered by the modeller's commit, while in BUMBLE, we focus on on-the-fly change propagation across the modeling views also.

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 [5], by which a model is exported as text. The second is Modelware [5], 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 modeling can also be realised with Qt, where the approach is to use a graphical environment with embedded textual editors. However, DSML engineers would need to realise most of this manually. Mixed notations and the possibility to switch between them are supported in JetBrains MPS for non-MOF DSMLs and rely on the principle of projectional editing.

There exist approaches for synchronising graphical and textual models via semi-automated mechanisms in the form of synchronisation model transformations. These model transformations are, in some approaches, also generated, thanks to HOTs [6], which are specific to the DSML at hand. The aim of T4.3, and the work described in this deliverable, is to provide cross-DSML HOTs for any Ecore-based DSML.

3. Synchronisation solutions

In this section, we describe the solutions achieved in this WP for synchronisation across notations as well as across different modeling artefacts. Note that the transformations described in this section are not generated by HOTs, but they were manually written to gather a variegated set of blueprints for the HOTs. All solutions have successfully been implemented (or are being implemented) in industrial use cases in BUMBLE (UC1, UC2, UC6, described in D2.1).

3.1. Synchronisation for Portable test and Stimulus Standard (UC1)

In this section, we describe a generic solution to support seamless runtime synchronisation between graphical and textual syntaxes for multiple DSMLs. The high-level architecture of this solution is shown in Figure 1. The framework takes in input DSMLs in the form of: (i) metamodels defined in Ecor, (ii) textual domain-specific languages (DSLs), or (iii) grammar portions. From them, it generates graphical and textual notations automatically through the BUMBLE-M2T-metamodel and BUMBLE-Language-Analyzer components, respectively. The generated graphical and textual notations are saved in an XML format and serve as an input for mapping and synchronisation activities.

The BUMBLE-Mapping-Editor component is developed to define explicit mappings between graphical and textual syntaxes. Mappings are also saved in an XML format and serve as an input for the synchronisation mechanisms. For runtime synchronisation, mapping rules are materialized as an EBNF grammar through the BUMBLE-EBNF-Generator component.

In the following, we show the usage of the proposed framework [13] for blended modeling and, more importantly, synchronisation of graphical and textual notations for the Portable test and Stimulus Standard (PSS) use case.

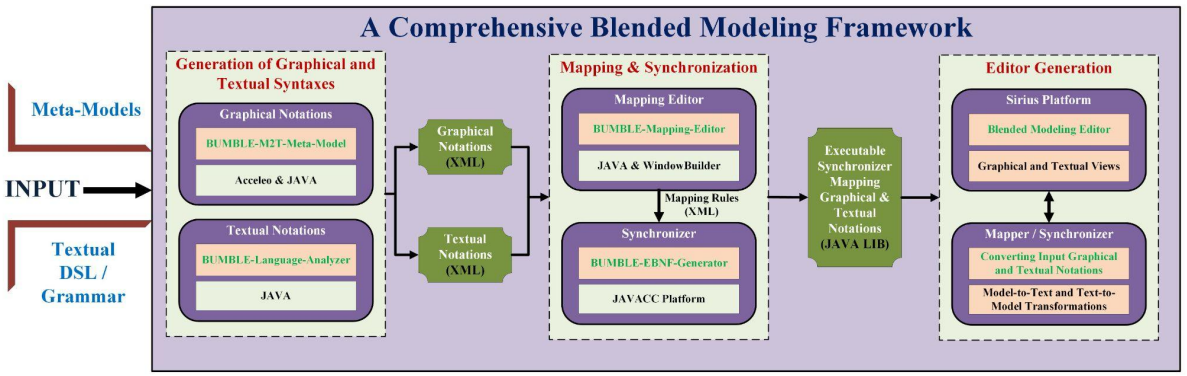

Figure 1 - The high-level architecture of proposed framework

Input artefacts

The proposed framework requires a textual DSL/grammar (plain text) and a metamodel (in Ecore) for the generation of textual and graphical notations, respectively. In our use case, PSS carries along a C++-like DSL for the specification of test intents. The samples of PSS DSL are available in the PSS specification[1] and can be given as input to our framework for the generation of textual notations for PSS. On the other hand, a canonical metamodel for PSS was not available, to the best of our knowledge. Therefore, we proposed such a PSS metamodel and implemented it using Ecore in EMF, as shown in Figure 2. The main PSS concepts like actions, objects, resources, etc., and their semantics are included in the metamodel. This metamodel is given as an input to our framework for the generation of graphical notations

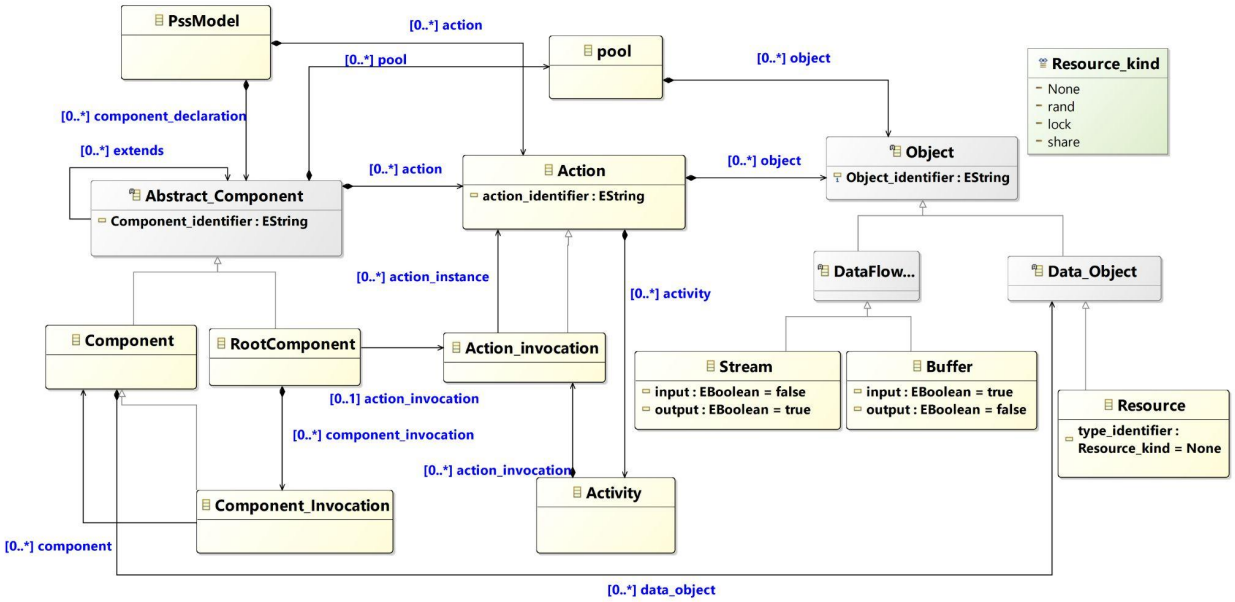

Figure 2 - PSS metamodel in Ecore

[1]https://www.accellera.org/downloads/standards/portable-stimulus#:~:text=The%20Portable%20Test%20and%20Stimulus,of%20integration%20under%20different%20 configurations

Generation of graphical and textual syntaxes

With the component “Generation of graphical and textual syntaxes” the user selects a DSL/grammar file as an input through the “Browse” button and starts the generation of the required tokens (textual notation) using the “Parse” Button. The generated grammar tokens can be saved in an XML file, which is further utilized in the mapping process. This module is capable of generating textual notations for different types of DSLs/grammars. For demonstration purposes, a grammar sample from the PSS DSL is considered, where an action is specified through input and output buffer types. The respective textual notation (tokens) are generated and, subsequently, saved in an XML file, as shown in Figure 3.

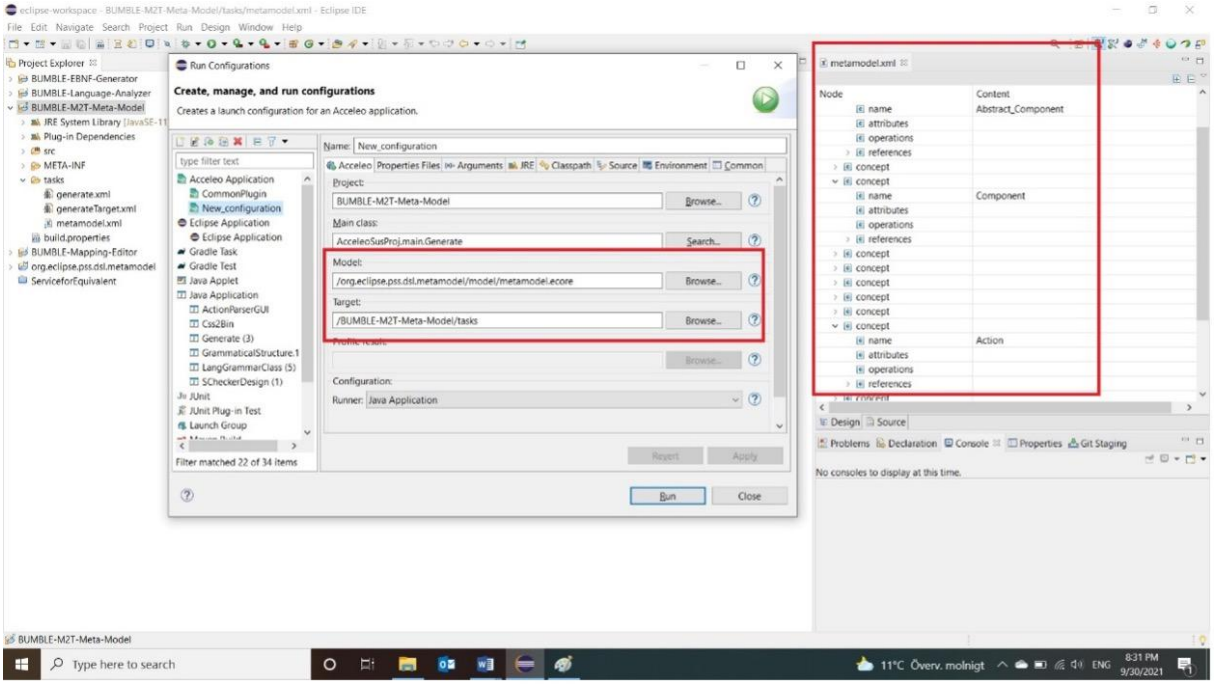

Figure 3 - BUMBLE-M2T-metamodel component for the generation of graphical syntax

For the generation of graphical notations, the BUMBLE-Language-Analyzer (Figure 4-a) is developed via model-to-text transformations in Acceleo and Java. Particularly, it takes a metamodel as an input and generates the corresponding graphical notation and semantic information in XML format. For demonstration, the PSS metamodel is given as an input, and the generated XML file is shown in Figure 4-b. This XML file is further utilized in the mapping process, as described in the remainder of this section.

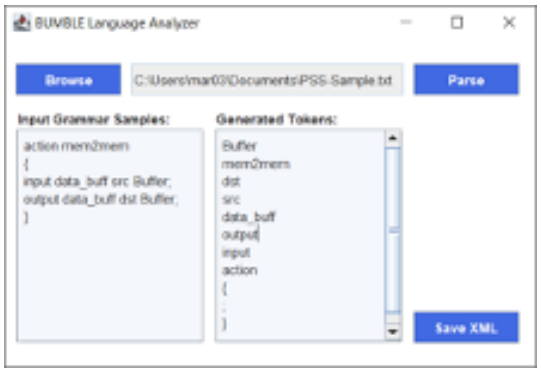

Figure 4-a - BUMBLE-Language-Analyzer

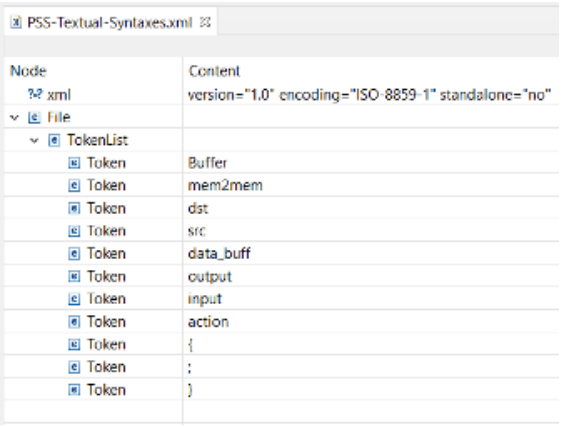

Figure 4-b - PSS textual syntaxes in XML

Mapping and synchronisation

The domain expert's feedback is crucial in the mapping process. For this purpose, a mapping editor is developed as shown in Figure 5. In terms of reusability and portability, BUMBLE-Mapping-Editor is flexible and can be used for any pair of graphical and textual notations. Importantly, all the interface components are generated dynamically through the XML files. Particularly, it displays the graphical and textual notations from the related XML files that are generated by BUMBLE-Language-Analyzer. Furthermore, it displays the repository of symbols to be associated with the graphical notation through mappings (containing addresses and IDs of symbols), and, therefore, other symbols specific to the domain can be added to the mapping editor with simplicity. In addition, it provides AND/OR operators to define complex mappings between graphical and textual elements (e.g., one graphical element may correspond to the combination of several textual elements and vice versa). This mapping file is utilized to generate the corresponding EBNF grammar for synchronisation purposes.

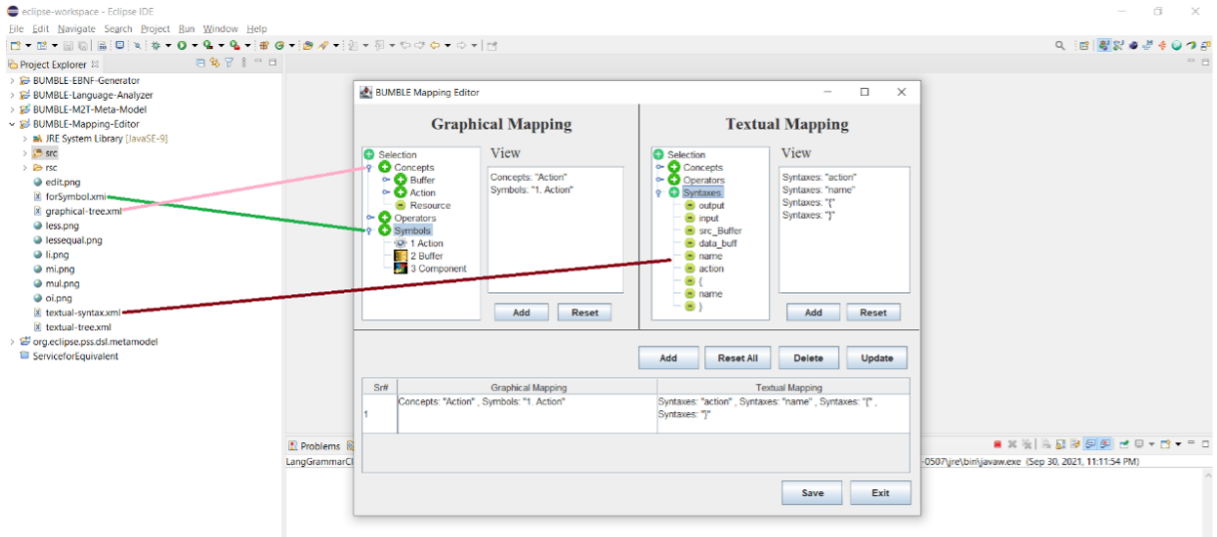

Figure 5 - BUMBLE Mapping Editor with PSS example

In Figure 6 we show the mapping between the PSS graphical and textual notations, as well as the association of graphical symbols to the PSS concepts, as furtherly described in deliverable D4.1.

The BUMBLE-EBNF-Generator component is responsible for generating an EBNF grammar by utilizing the mapping XML. This step is essential for seamless synchronisation and switching between graphical and textual notations. The JAVACC[2] platform is used to implement this component. Few EBNF mapping grammar rules for the PSS action and buffer concepts are given here for demonstration purposes:

Rule 1: <Action> : := <Graphical-Action> | <Textual-Action>
Rule 2: <Graphical-Action> : := <Name><Symbol> | <Name><Symbol><Relationship>(<Graphical-Data Buffer>)*
Rule 3: <Textual-Action> : := action <Name> { } | action <Name> { (<Textual-Data Buffer>)* }
Rule 4: <Data Buffer> : := <Graphical-Data Buffer> | <Textual-Data Buffer>
Rule 5: <Graphical-Data Buffer> : := <Type><Name><Symbol>
Rule 6: <Textual-Data Buffer> : := <Type> data_buff <Name>;
Rule 7: <Relationship> : := Containment <Symbol>
Rule 8: <Name> : := ([a-z][A-Z][0-9])*
Rule 9: <Symbol> : := ([a-z][/][\][A-Z][0-9])*
Rule 10: <Type> : := input | output

For better understanding, consider a simple PSS DSL example where an action having two buffers is defined as:

```
action mem2mem {
        input data_buff src Buffer;
        output data_buff dst Buffer;
}
```

Based on the aforementioned EBNF rules, a parsing tree of the given example is shown in Figure 6 and is used to achieve seamless synchronisation and switching between graphical and textual notations. Please note that terminal symbols are displayed in orange boxes.

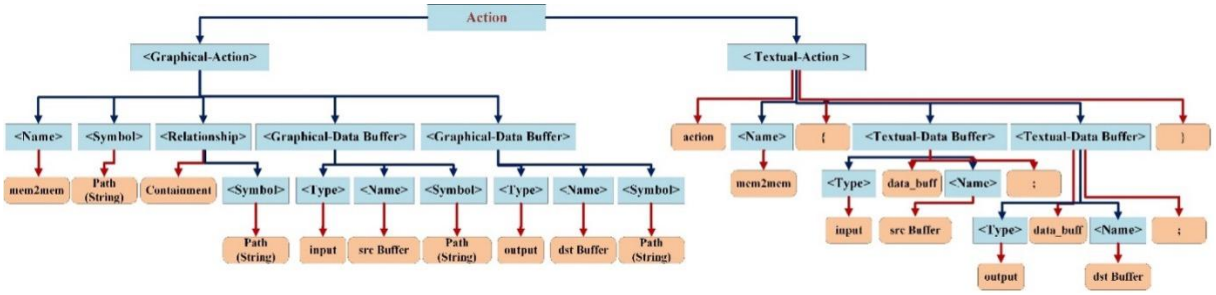

Figure 6 - Parsing tree of EBNF rules for PSS action concept

The implementation of EBNF is accomplished through the JAVACC platform, as shown in Figure 7-a. More specifically, all EBNF rules are implemented in a JAVACC template file (i.e., representing a grammar). It is pertinent to mention that the implementation of EBNF is done as a service so that it can be easily imported into other tools for editor generation like Eclipse Sirius. To verify the EBNF service in JAVACC, a test client application is also developed and shown in Figure 7-b. It provides an interface to input textual or graphical syntaxes and subsequently calls the EBNF service to receive the equivalent textual or graphical syntaxes accordingly, as shown in Figure 7-b.

[2] https://javacc.github.io/javacc/

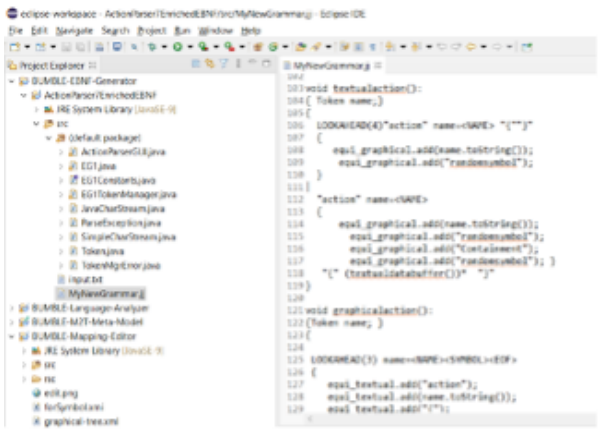

Figure 7-a - EBNF implementation in JAVACC

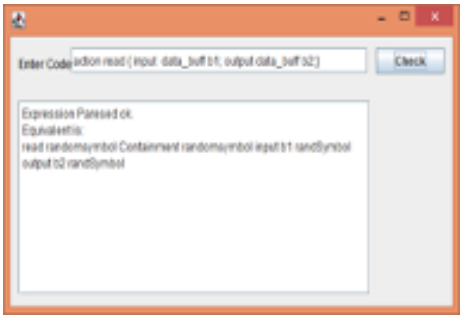

Figure 7-b - Test client for EBNF Service

Finally, Eclipse Sirius is used to generate the blended modeling editor. The architecture of the editor generation component is shown in Figure 8. In the first step, a graphical editor is generated in Sirius by utilizing the PSS metamodel. In the second step, a textual view is incorporated in the graphical editor for the specification of textual PSS. The blended modeling editor supports seamless synchronisation and switching between graphical and textual notations using Java services, model-to-text and text-to-model transformations, and the Synchronizer component. Particularly, Java services are used to communicate with the Synchronizer, while model-to-text and text-to-model transformations are intermediate components to perform suitable propagation of model changes between graphical and textual notations and display them in the blended modeling editor accordingly for seamless switching.

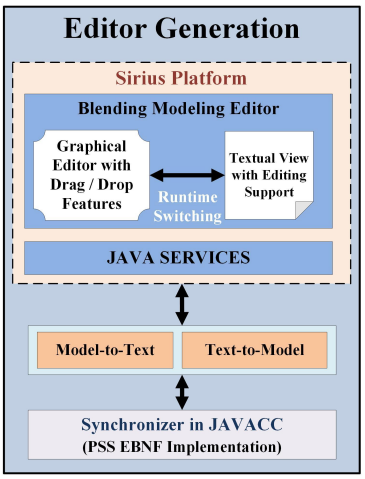

Figure 8 - Architecture of editor generation component

The generated PSS blended modeling editor is shown in Figure 9. Particularly, it offers drag/drop functionality for different PSS graphical elements (provided in a palette) with suitable symbols that were chosen during the mapping process. The “Update Views” button is provided to synchronize the two notations on-demand. The outcome of the synchronisation process (i.e., Successful, Done with

Errors, and Failed) is displayed too. Here, a simple PSS Direct Memory Access (DMA) example is considered where an action (mem2mem) and two buffers (input source and output destination) are modeled graphically, as shown in Figure 9. Subsequently, a seamless switching is performed via the “Update Views” button to generate the corresponding textual syntax. After successful synchronisation, a component (dma_c) is defined textually, and synchronisation is performed to represent the dma_c component graphically, as shown in Figure 9.

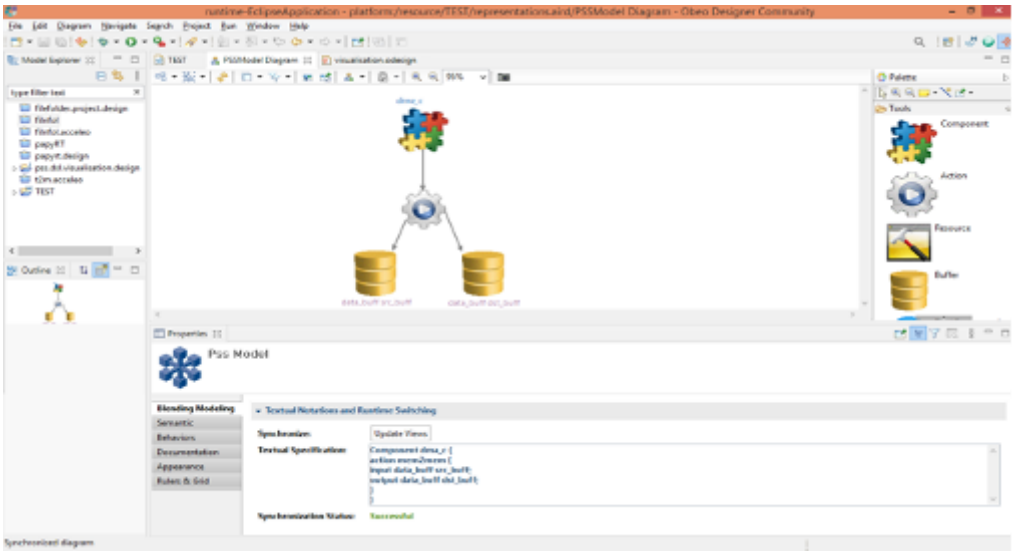

Figure 9 - PSS Blended Modeling Editor

3.2. Synchronisation for UML-RT State Machines (UC1, UC2)

UML-RT is a real-time profile for UML that aims to simplify the ever-increasing complex software architecture specification for real-time embedded systems, and it relies on state machines for behavior modeling. Most off-the-shelf UML modeling tools focus on graphical editing features and do not allow seamless graphical--textual editing. To overcome this challenge, we define a textual notation for UML-RT state machines (SMs) and synchronize it with the graphical notations available in Papyrus-RT (open source) and RTist (commercial tool from one of the BUMBLE’s partners, HCL). Note that in this section, we describe the solution for Papyrus-RT (UC1), which is orthogonal to the one in RTist (UC2) using Epsilon Transformation Language (ETL) In [9] we describe a similar approach using the Query/View/Transformation Operational (QVTo) language.

Figure 10 provides a high-level architecture that includes all the components required to synchronize between graphical and textual notations for UML-RT SMs and how they are connected together. In the following, we provide the description for each of the components.

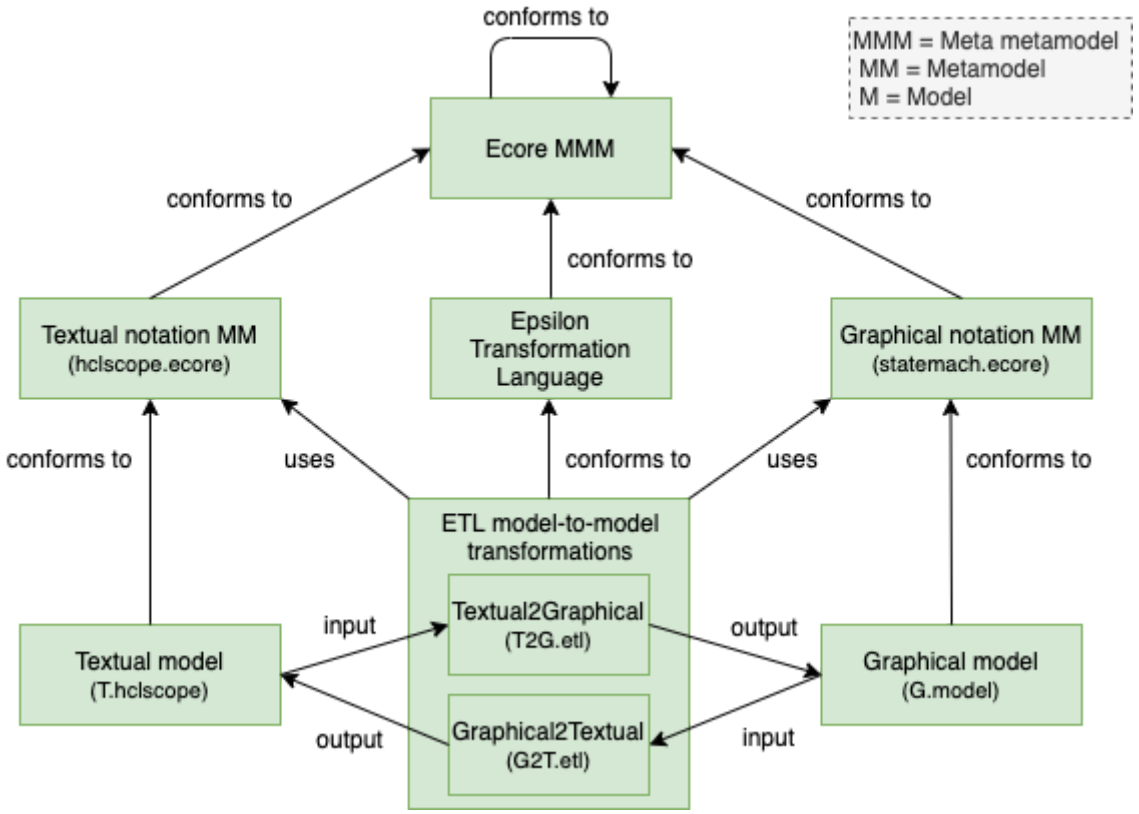

Figure 10: High-level architecture for the synchronisation between the graphical and textual notations for UML-RT state machines.

Ecore MMM: The Ecore meta-metamodel is a language that conforms to itself and is used to define Ecore metamodels.

Metamodels: Metamodels (defined in Ecore) conceive the concepts, relationships, and well-formedness rules that formalize how the meta-concepts can be legally combined for valid UML-RT SM models. The UML-RT SMs use case consists of two metamodels that are used to define the graphical and textual notations.

- **Graphical notation MM:** The metamodel used to describe the graphical model for UML-RT SMs is the underlying metamodel used in Papyrus-RT, and it is available as open-source (the metamodel in Ecore is depicted in Appendix 1).
- **Textual notation MM:** The metamodel used to describe the textual model for UML-RT SMs is defined manually using the Xtext language workbench (the metamodel in Ecore is depicted in Appendix 1). It takes into consideration i) the feedback from UML-RT's customers and architects in RTist and ii) the UML-RT metamodel portion describing state machines, as a blueprint. Moreover, the scope provider by Xtext is customized to only allow cross-references for elements declared in the same model file, support the inheritance mechanism, and restrict transitions to only cross-reference pseudo-states and states that are on the same level of nesting as the transition, or their immediate entry and exit points. Consequently, this customization contributes to the enforcement of the UML-RT's modularity. Further details on the textual notation can be found in [8].

Models: In order to instantiate the textual and graphical notation metamodels, we create two models (i.e., textual model and graphical model), respectively. These models will be used both as source and target models, depending on the direction of the transformation.

Epsilon Transformation Language (ETL): ETL is a hybrid model transformation language that provides both declarative execution schemes for simple transformations and imperative features for supporting complex transformations. ETL Transformations are organized in modules that contain a number of transformation rules that specify one source and target parameters.

ETL Transformations: In order to provide the synchronisation mechanisms for UML-RT SMs, we define two model-to-model unidirectional transformations. The transformations are exogenous and horizontal, as the models are expressed in different modeling languages and reside in the same abstraction level. The first step consists of identifying the mapping in order to determine which elements in the source model are actually mapped into a corresponding element in the target model. The majority of elements have a one-to-one mapping, as the modeling languages are conceptually similar. In such circumstances, the transformation rules are rather straightforward and contain one source and one target parameter (see Listing 1). On the occasions where one element in the source model is mapped to more than one element in the target model, the transformation rules contain one source and multiple target parameters (see Listing 2). Lastly, in the event of multiple elements in the source model mapping to one single element in the target model, the transformation rules need to be defined separately, as it is not possible to define transformation rules with multiple source parameters (e.g., transformation rules for transitions defined in the Textual2Graphical transformation).

Textual2Graphical: This transformation takes as input the textual model and produces as output the graphical model. Listing 1 is an example of the transformation rule written in ETL for transforming a Junction from the source metamodel (i.e., textual notation MM) into a JunctionPoint in the target metamodel (i.e., graphical notation MM).

```
rule Junction2Junction
   transform s: Source!Junction
   to t: Target!JunctionPoint
   {
     t.name=s.name;
   }
```

Listing 1 - Textual2Graphical ETL rule

Graphical2Textual: This transformation takes as input the graphical model and produces as output the textual model. Listing 2 shows an example of the transformation rule written in ETL for transforming a Trigger from the source metamodel (i.e., graphical notation MM) into multiple types of Trigger in the target metamodel (i.e., textual notation MM).

```
rule Trigger2Trigger
   transform s: Source!Trigger
   to t: Target!Trigger, mpt:Target!MethodParameterTrigger,m:Target!Method, pa:
Target!Parameter, pet: Target!PortEventTrigger,
p:Target!Port , e:Target!Event {
         if (s.name.matches(".*\\..*")){
           p.name = s.name.split("\\.").first();
           e.name = s.name.split("\\.").second();
```

```
            pet.port = p;
            pet.event = e;
          }
         else if (s.name.matches(".*\\(.*")){
            m.name = s.name.split("\\(").first();
            pa.name = s.name.split("\\(").second();
            pa.name = s.name.split("\\)").first();
            mpt.method = m;
            mpt.parameter = pa;
          }
          else {
          t.name = s.name;
          }
        }
```

Listing 2 - Graphical2Textual ETL rule

3.3. Synchronisation between system and software artefacts

In this section, we describe a solution for synchronisation between system and software engineers and thereby related (modeling) artefacts. Although the scenario originates from Saab's use case, the same settings are common in companies dealing with software-intensive systems.

Problem: a SysML system model is created to describe the decomposition of the system into function blocks that are then assigned to software or hardware components. Moreover, the intended functionality of those components is modelled via behavioural diagrams (e.g., state machines) and internal block diagrams. The software components are implemented in various ways, among which C++.

In the current state of practice, system engineers prepare handover presentations for the software engineers based on excerpts of the system model that need to be implemented in, e.g., C++. This is time-consuming and, thereby, usually a one-time activity. The result is that throughout the evolution of the system, the system model and the software implementation diverge.

Solution: To ensure that the model can be used as accurate documentation of the implementation and to ensure that the implementation correctly follows the plan as laid out in the system model, synchronisation between them is needed. Given the abstraction gap between the system model and code, this synchronisation cannot be completely automated (if there were enough detail in the model to do so, the code would be automatically generated instead of manually implemented). Hence, we propose to verify architectural guidelines and provide software engineers with insights into the model and, conversely, system engineers with more insight into the implementation. The automated evaluation of these guidelines provides faster feedback loops between the engineers.

We propose a bridge between the C++ IDE (JetBrains CLion) and the modeling tool (IBM Rational Rhapsody). The code and model notations will be blended in the following two ways: 1) when editing code in CLion, a software engineer will be able to see the related portion of the model she is editing. For example, when editing a class, she will see the state machine in which that class is a state. 2) When editing the model in Rhapsody, a system engineer will be able to see the related portion of the implementation she is editing. For example, when editing a state machine, the related code showing the class definitions corresponding to the states in that state machine will be shown. In both cases, the additional information is aimed to be shown inside the IDE by means of plug-ins for CLion and

Rhapsody, thus blending the two notations. Figure 11 gives an overview of the proposed bridge and its constituent components.

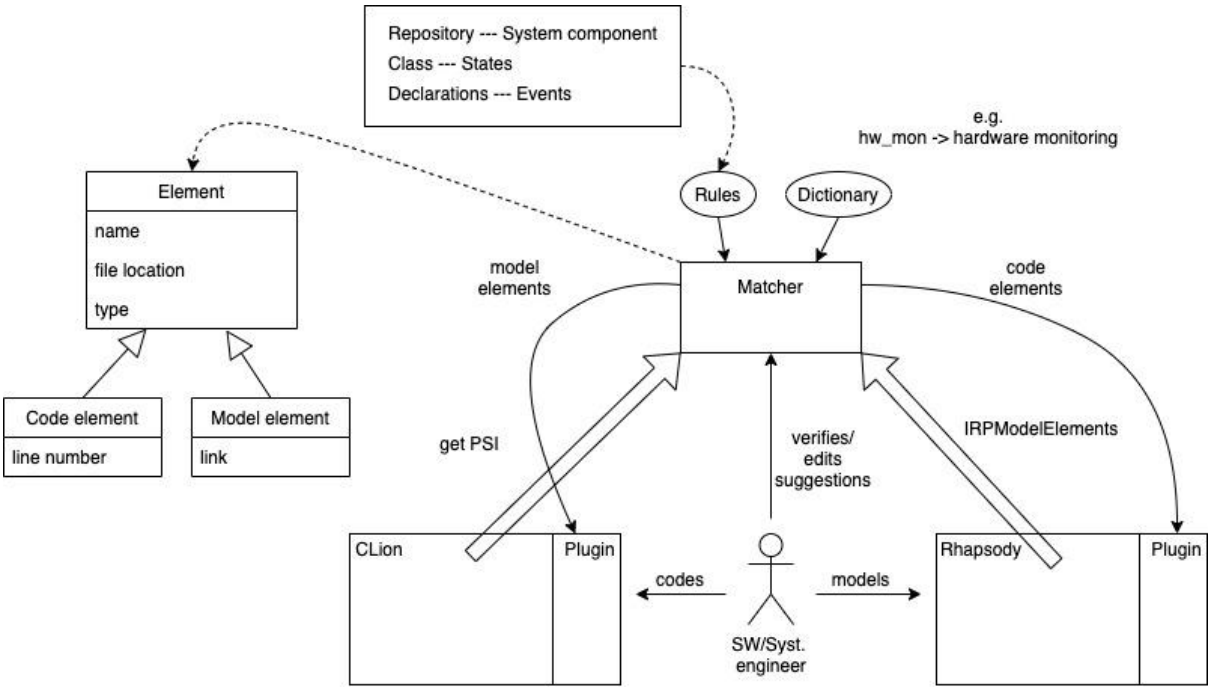

Figure 11 - System and software artefacts synchronisation

There, the two Matcher items match specific code and model elements, as defined by the rules, and store the mined elements in a generic common format as elements. In the current phase, we have identified three initial architectural guidelines that must be checked. Correspondingly, we match code repositories to system components in the model, classes in the code to states in the model, and specific declarations in the code to events (signals) in the model. The dictionary is used to assist the matcher in finding equivalent elements that are named differently in model and code (for example, because there can be no spaces in variable names in the code). Blending of the notations is shown in the figure as the plugins for the two development tools that will show the elements of the model/code.

As part of the study, we have performed a validation workshop (prior to starting implementation) with engineers at Saab. Initial feedback from the engineers indicates a positive view of the proposed solution. For example, a software engineer remarked that blending the notations would assist in improving the understanding of the model. “This will help me to learn to know the model and understand model better. Currently, I don’t look at the model.” Another remark: “As a software engineer, this will help me to see what has changed in the model and will give some help to see how complete the implementation is.”

Based on the outcome of the validation workshop, currently, a prototype implementation is being developed, focusing initially on creating the mapping rules (see D4.1) required for mapping implementation to model and vice versa. An important outcome of the validation workshop was that the mapping rules should be made in such a way that they do not become another maintenance burden in the development process (“As a system engineer, I would be unhappy with maintaining rules if it means to change a term in multiple places.”).

3.4. Synchronisation with DClare

DClare realises blended modelling by transforming the abstract syntaxes of different modelling languages. This is especially valuable in MPS, where the synchronisation of multiple concrete syntaxes for a given abstract syntax is already supported by MPS itself. However, in MPS the abstract syntax is very much driven by its concrete appearance in the editors. This implies that blending multiple concrete syntaxes in MPS requires the transformation of multiple abstract syntaxes. This is exactly what Dclare supports.

There is also a need for transformations between models of different Ecore-based models in EMF. That is why we plan to realise model transformations within EMF too. The engine is the same for EMF and MPS. There is already an integration with MPS, but a connector for EMF is still to be developed.

DclareForMPS is also going to support bi-directional transformations between declaratively defined GUI's and the abstract syntax (structures) in MPS. The declaratively defined GUIcan be of many flavors: Diagrammatic, tables, forms, trees, etc..

Figure 12 shows the basic architecture of DClare-based transformation specifications and executions. The small dark blue rectangle is the connector that connects MPS models to the DClare engine. The engine reacts to any change in the models and changes the opposite models according to the transformation specification. The generator of MPS is extended to also generate the transformation rules that are executed by the DClare engine.

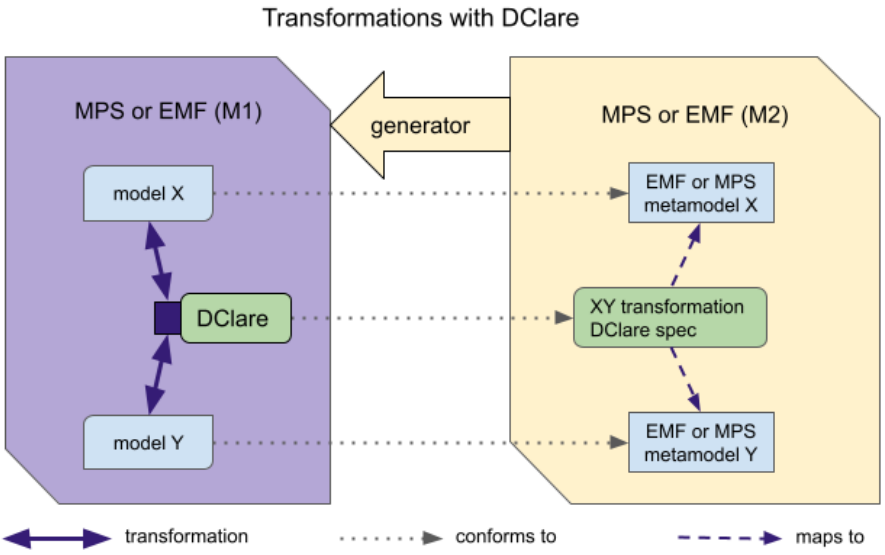

Figure 12 - Transformations with DClare

The specification of the transformation is based on attribute and rule specifications. Within MPS, we realised a so-called 'language aspect' for defining attributes and rules in MPS. The (meta)language for this is an extension of the base-language of MPS itself. This guarantees easy adaptation by the MPS community.

Figure 13 shows how transformations are defined in MPS using DClareForMPS rulesets.

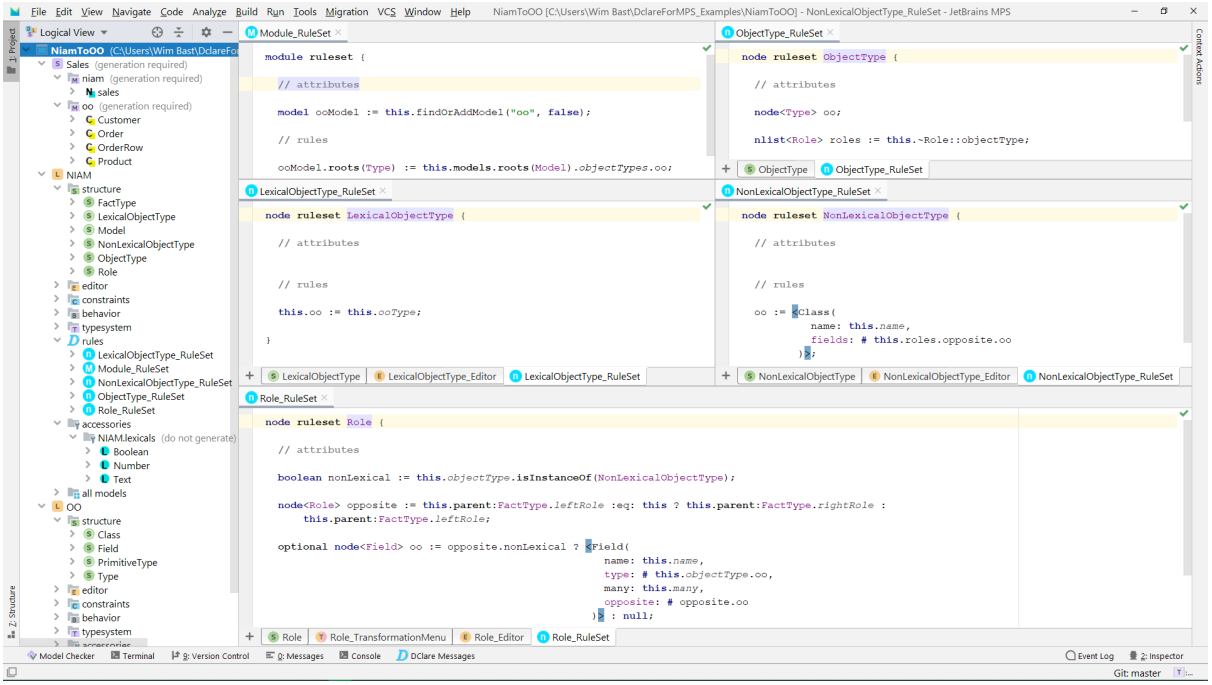

Figure 13 - DClareForMPS in MPS

4. HOTs for bidirectional synchronisation and co-evolution

The main objective of WP4 is to provide a set of HOTs to semi-automatically generate the synchronisation infrastructure, in terms of bidirectional model transformations, across concrete syntaxes. Being able to automatically generate synchronisation mechanisms through HOTs brings the following advantages: (i) the solution is generic and not tailored to a specific DSML; (ii) the solution is flexible and allows co-evolution in response to DSML evolution (since synchronisation mechanisms are not fixed but rather generated from the DSML metamodel, whenever the metamodel evolves, the mechanisms can be re-generated from the evolved DSML); (iii) the definition of HOTs is a one-time effort, which can then be transparently leveraged by developers to generate their synchronisation infrastructure for a given DSML.

On the one hand, the use cases refined in WP2, the architecture and methodology defined in WP3, and the visualization-specific DSMLs and mapping models defined in WP3-WP4 will be exploited to define the solutions in WP4 as well as to evaluate the resulting implementation. On the other hand, the capabilities of the solutions defined in WP4 will influence the overall architecture and methodology defined in WP3. We envision that results from WP4 (HOTs for the generation of synchronisation mechanisms from DSMLs) will also be exploited by WP5 for collaborative modeling.

In Figure 14, we provide an architecture for the generation and usage of model transformations generated via HOTs in BUMBLE. Given two DSMLs defined in terms of Ecore (in EMF), a mapping model, conforming to the mapping metamodel defined in D4.1, conceives the mapping rules for synchronizing models conforming to the two DSMLs. Interestingly, this architecture and the transformations in it entail multiple usage scenarios, as follows:

- In case the DSMLs are two entirely disjoint (but somehow connected/dependent) languages, the generated transformations provide **synchronisation across different languages** (either the same or different notations).
- In case the DSMLs represent two notations of the same language, the generated transformations provide **synchronisation across different notations** of the language.
- In addition, in case $DSML_B$ represents an evolution of $DSML_A$, the generated transformations provide **co-evolution** mechanisms for models conforming to $DSML_A$.

In Figure 14, we can see the mapping modeling language (MML) described in D4.1, which is used to formalize the mapping rules between $DSML_A$ and $DSML_B$ in two mapping models ($DSML_A$_2_$DSML_B$ and $DSML_B$_2_$DSML_A$). These models, together with the two DSMLs, are given as input to our set of HOTs defined in Xtend. The output of the HOTs is synchronisation model transformations defined in QVT Operational. These model transformations are then in charge of automated synchronisation between two instances of $DSML_A$ and $DSML_B$, that is to say, M_{DSML_A} and M_{DSML_B}. The type of transformation (i.e., endogenous, exogenous, out-of-place, in-place) depends on the nature of the two DSMLs, as explained in the scenarios above.

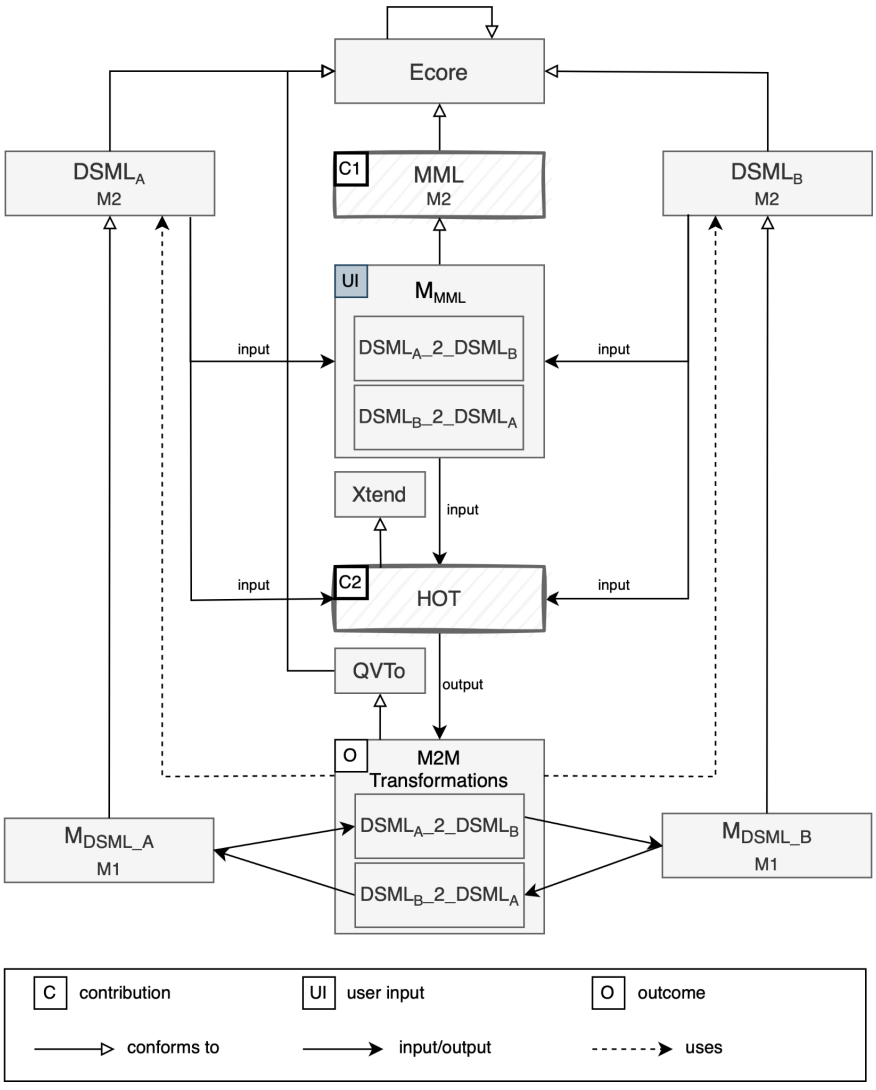

Figure 14 - Architecture for HOTs and bidirectional synchronisation

Xtend was chosen as the language for implementing the HOTs for multiple reasons. First of all, it is a flexible and expressive dialect of Java, and it compiles into readable Java 8-compatible source code. In addition, any existing Java library can be used seamlessly. The compiled output is readable and pretty printed, and usually runs at least as fast as the equivalent handwritten Java code. Xtend provides powerful macros, lambdas, operator overloading, and several other modern language features. Finally, Xtend is included in the Eclipse release train and follows all Eclipse releases.

For model transformations generated via our HOTs in Xtend, we chose QVT Operational (QVTo), which is an implementation of the Operational Mappings Language defined by MOF 2.0 Query/View/Transformation (QVT). The reasons behind this choice were, first of all, the fact that QVT is a MOF standard, and since our focus is on MOF languages, a transformation language also based on MOF is preferred. In addition, QVTo brings together benefits from both the declarative and imperative QVT, and it is very well-suited for both exogenous and endogenous transformations, also in-place.

In the following, we describe how the HOTs combine the input specified by the user in the mapping models with the information automatically extracted from the mapped DSMLs in order to generate the model transformations for synchronisation. Each paragraph corresponds to a metaconcept of MML defined in D4.1.

Mapping Model: `MappingModel` is the root element of the mapping language and represents the starting point for traversing a mapping model. The user assigns a name to it, which is then used to generate the name of the model transformation. If there is more than one `SourceMetamodel`, the user must select the `MainSourceMetamodel`, which represents the metamodel that is used as the entry point of the model transformation. Alternatively, in the case of only one `SourceMetamodel`, the latter is automatically selected as `MainSourceMetamodel`.

Metamodels: The information extracted from `SourceMetamodel` and `TargetMetamodel` is used to generate the `modeltype`, and `transformation signature` of the model transformations. The user loads the DSMLs and selects the EPackages to be mapped that will be used as the source and target of the model transformation. The HOTs automatically retrieve the name and nsURI of the EPackages to generate the `modeltype`, identify the direction of the model transformation, as specified in the mapping model, and generate parts of the transformation signature according to the following pattern:

```
in <<SourcePackageName>>Model : <<SourcePackageName>>,
out <<TargetPackageName>>Model : <<TargetPackageName>>
```

Mapping Rule: As described in D4.1, `MappingRules` contained in `MappingModel`, can be referred to as *immediate mapping rules*, while `MappingRules` contained in other `MappingRules` or `HelperStatements`, can be referred to as *child mapping rules.*
Immediate mapping rules are used to generate the mapping declaration, whereas child mapping rules are utilized to generate the body of the mapping operations. In the following, we detail the implementation of the features that apply to each category.

1. Immediate mapping rules

   Mapping operation name: The name of the mapping operation is automatically generated as

`<<sourceElementName>>2<<targetElementName>>`. This not only reduces the amount of manual effort from the user, but it also increases readability, as the naming follows a specific standard pattern and is rather intuitive. Moreover, to minimize the risk of errors when mapping elements with the same name, source, and target elements are printed using fully qualified names (i.e., `modelName::elementName`), thanks to our customized model editors.

Mapping operation type: The generated mapping operations can be abstract or non-abstract. Abstract mapping operations are used when the target of the mapping operation is abstract. This information is automatically extracted from the target DSML; hence, it does not require user input. Before printing a mapping rule, the HOTs determine whether the target element of the mapping rule is an abstract or non-abstract EClass. In the case of an abstract EClass, the mapping operation is printed as abstract according to the pattern:
`Abstract<<sourceElementName>>2<<targetElementName>>`.

Conditions: For immediate mapping operations, the source and target elements are EClasses, therefore, conditions that are manually defined by the user are automatically generated as when clauses that are evaluated to determine in which circumstances the mapping operation should be executed.

Inheritance: The concept of inheritance allows the reuse of mapping operations under the condition that the signature of the inherited mapping conforms to the one of the inheriting mappings. The source and target of any potential inherited mapping rule must be supertypes of, or identical to, the source or target of the inheriting mapping rule. The HOTs iterate through each of the immediate mapping rules in the mapping model and determine whether the mapping operation under analysis inherits from any of the iterated mapping rules. If it does, after the transformation signature and the `inherits` keyword, the names of the inherited mapping rules are printed (in the case of multiple inherited mapping rules, they are separated by a comma).

Disjunction: Invocation of a disjunct mapping operation results in an assessment of disjunct candidate mapping operations. To determine whether a mapping operation is disjunctive and, if so, to identify the disjunct candidates, the HOTs iterate through all the immediate mapping rules of the mapping model and identify those where the source and target are identical or subtypes of the source and target of the potentially disjunctive mapping rule. If these mapping rules exist, the analyzed mapping rule is considered disjunctive and is named according to the following pattern:
`<<sourceElementName>>2<<targetElementName>>Disjunct.`
After printing the signature of the mapping operation and the `disjuncts` keyword, the HOTs print the identified disjunct candidates. A mapping can be both abstract and disjunctive. The user needs to define the mapping rule only once in the mapping model, and the HOTs will generate two rules: one abstract and one disjunctive, since QVTo does not allow combining them into one.

2. Child mapping rules

As described in D4.1, for *child mapping rules,* there exist three different possible scenarios, depending on the values of the source and target attributes.
SC1: `source != null` **and** `target != null` - (a non-empty set of input elements in the source model are transformed into a non-empty set of output elements in the target model)
SC2: `source == null` **and** `target != null` - (a non-empty set of output elements are added to the target model)

SC3: `source != null` **and** `target == null` - (a non-empty set of input elements in the source model facilitates the navigation of model elements)

SC3 is used for complex and possibly ambiguous navigation cases, such as the one depicted in Figure 15, where on the left-side is illustrated an excerpt from the source metamodel and on the right side an excerpt from the target metamodel. Consider that the user defines an immediate mapping rule `Organization2Company` as detailed in Figure 16. The user then wants to map the value of the name attribute of `Person` in the source metamodel, to the `managerName` attribute of `Company` in the target metamodel, by defining the mapping rule `name2managerName`. The correct navigation path, in this case, would be `self.department.manager.name`, where `self = Organization`. To navigate to the `name` attribute, starting from `Organization`, there is, in fact, also another path; `self.department.secretary.name`. However, this navigation path is not considered correct, as it would map the name of a `Person` that is a secretary in `Department` to `managerName` in `Company`. Therefore, the information to navigate to `name` attribute via `manager` reference is to be decided by the user, as the HOTs cannot automatically determine which path to take. Therefore, the user needs to define an additional mapping rule `manager2null`, that will guide the HOTs in generating the expected model transformation. Being that `manager` is not an immediate reference of `Organization`, HOTs must automatically generate the navigation path from `Organization` to `manager`. For that reason, we implement a recursive Depth-First Search (DFS) algorithm, which starts at the root node (i.e., `Organization`) and explores as far as possible along each EClass, before backtracking (unless it finds the target).

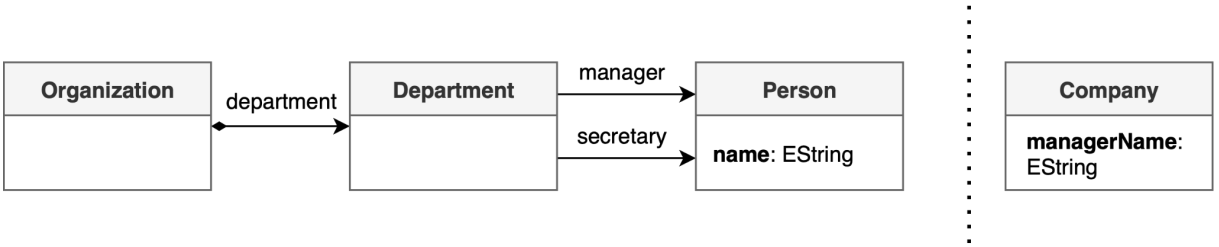

Figure 15: Ambiguous navigation

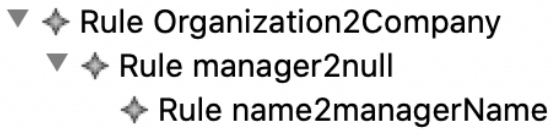

Figure 16: Ambiguous navigation mapping rules

Invoking rule: In QVTo, mapping operations are run with an explicit rule-invocation style, which initiates execution from an entry mapping operation generally found in the main function, and invokes the other mapping operations in a nested manner. The entry mapping operation that should be invoked is automatically determined.

OCL expressions: For sub-mapping operations, derived from child mapping rules, where the source and target elements are EReferences or EAttributes, `condition` attributes of the mapping rules are used to specify OCL expressions.

Navigation operators: The HOTs determine the navigation operator based on whether the source of the mapping rule is a single object or a collection of objects (i.e., by checking the `upperBound`). A single object is navigated using the *dot (.)* operator, whereas collections of objects are navigated using the *arrow (->)* operator.

HelperStatement: It is intended to facilitate the definition of complex mappings requiring the use of `for` loops, `while` loops, or `if/else` conditional statements. `HelperStatements` are contained in `MappingRules` (i.e., immediate mapping rules) similar to how they are defined in mapping operations in QVTo transformations. Moreover, they may contain other `HelperStatements` and `MappingRules` that are generated within the loop or statement defined by `HelperStatement`.

Further details on the solution can be found in [10,11].

5. Validation

Validation is performed by means of the use cases described in D4.1., and model-to-text tests classified by Tiso et al. (2013) as i) conformance tests, ii) semantic tests, and iii) textual tests. This validation process not only validates the correctness of the HOTs but also demonstrates the MML defined in D4.1. contains all the concepts required to specify deterministic mappings between two Ecore-based DSMLs, and mapping models can be effectively utilized to generate well-formed model transformations. For each mapping model defined in D4.1., we evaluated whether the generated QVTo transformations matched the expected QVTo transformations. In detail, we defined transformation test cases `<MappingModel_file, Exp_QVTo_file>`, where `MappingModel_file` represents the mapping model used to determine the links between elements of the source and target metamodels, and `Exp_QVTo_file` represents the expected QVTo transformations that we manually defined. For a test case to pass, the output of the HOTs (generated QVTo transformations) must match `Exp_QVTo_file`. In the following, we provide more details on the different types of model-to-text tests.

Conformance tests

They were used to verify whether the generated transformations were structured textual artefacts that conformed to the QVTo language. QVTo has specific rules that specify how statements can be written, and the set of these rules constitutes the syntax of the language. Failed conformance tests typically occur due to possible syntactical mistakes, such as missing or unbalanced parentheses, missing or unbalanced quotes, missing colons or semicolons, misspelled variables, and so on. When a QVTo file for which conformance tests have failed is opened, the syntax errors are flagged in the file, together with a message error. The most common message errors in these cases are: missing “x” to complete scope, “x” expected instead of “y”, “x” expected after “y”, unrecognized variable(x), and so on.

Semantic tests

They are used to verify whether the generated transformations adhere to the semantics of the QVTo language. Failed semantic tests are often due to missing mapping operations, incorrect hierarchical structure, incorrect type of mapping operations (abstract/non-abstract), missing/incorrect inheritance

and disjunct candidates, and so on. For instance, our HOTs are expected to automatically identify whether a mapping operation inherits another. Mistakes in the implementation could lead to the HOTs failing to identify inheriting mapping operations. This would not trigger any syntactical error in the file, and the generated transformations would be executed. However, the resulting models of the generated transformations would not be semantically correct. The lack of error messages makes these types of errors not easy to locate.

Textual tests

They are used to verify whether the textual elements of the generated model transformations have the required format. Errors discovered through these tests are not identified through conformance testing. Examples of textual testing involve checking whether the name of the transformation is the one inputted by the user or whether the names of the mapping operations are defined following the defined template. Furthermore, these tests verify whether the generated transformations adhere to the QVTo formatter (e.g., new lines, indents, white spaces).

6. Conclusion

Regarding the core activity of the WP, namely T4.3, after implementing the HOTs, we validated them by exploiting the use cases mentioned in this deliverable. Moreover, we exercised them for the three scenarios (i.e., sync across different notations, sync across languages, co-evolution) in order to fine-tune them.

References

[1] Andrés, F. P., De Lara, J., & Guerra, E. (2007, October). Domain specific languages with graphical and textual views. In International Symposium on Applications of Graph Transformations with Industrial Relevance (pp. 82-97). Springer, Berlin, Heidelberg.

[2] Lazăr, C. L. (2011). Integrating Alf editor into UML editors. Studia Universitatis Babes-Bolyai, Informatica, 56(3).

[3] Charfi, A., Schmidt, A., & Spriestersbach, A. (2009, June). A hybrid graphical and textual notation and editor for UML actions. In European Conference on Model Driven Architecture-Foundations and Applications (pp. 237-252). Springer, Berlin, Heidelberg.

[4] Scheidgen, M. (2008, June). Textual modeling embedded into graphical modeling. In European Conference on Model Driven Architecture-Foundations and Applications (pp. 153-168). Springer, Berlin, Heidelberg.

[5] Wimmer, M., & Kramler, G. (2005, October). Bridging grammarware and modelware. In International Conference on Model Driven Engineering Languages and Systems (pp. 159-168). Springer, Berlin, Heidelberg.

[6] Maro, S., Steghöfer, J. P., Anjorin, A., Tichy, M., & Gelin, L. (2015, October). On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience. In Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (pp. 1-12).

[7] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., & Bézivin, J. (2009, June). On the use of higher-order model transformations. In European Conference on Model Driven Architecture-Foundations and Applications (pp. 18-33). Springer, Berlin, Heidelberg.

[8] Latifaj, M., Ciccozzi, F., Mohlin, M., & Posse, E. (2021, September). Towards Automated Support for Blended Modelling of UML-RT Embedded Software Architectures. In ECSA (Companion).

[9] Latifaj, M., Ciccozzi, F., Anwar, M. W., & Mohlin, M. (2022, August). Blended Graphical and Textual Modelling of UML-RT State-Machines: An Industrial Experience. In Software Architecture: 15th European Conference, ECSA 2021 Tracks and Workshops; Växjö, Sweden, September 13–17, 2021, Revised Selected Papers (pp. 22-44). Cham: Springer International Publishing.

[10] Latifaj, M., Ciccozzi, F., & Mohlin, M. Higher-Order Transformations for the Generation of Synchronization Infrastructures in Blended Modeling. Frontiers in Computer Science, 4, 166, 2023.

[11] Latifaj, M. (2022, October). The path towards the automatic provision of blended modeling environments. In Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (pp. 213-216).

[12] M. W. Anwar and F. Ciccozzi, "Blended Metamodeling for Seamless Development of Domain-Specific Modeling Languages across Multiple Workbenches," 2022 IEEE International Systems Conference (SysCon), Montreal, QC, Canada, 2022, pp. 1-7.

[13] Anwar, M.W., Latifaj, M., Ciccozzi, F. (2022). Blended Modeling Applied to the Portable Test and Stimulus Standard. In: Latifi, S. (eds) ITNG 2022 19th International Conference on Information Technology-New Generations. Advances in Intelligent Systems and Computing, vol 1421. Springer, Cham.

Appendix 1 - Ecore- based metamodels for UML-RT graphical and textual notations

Ecore-based metamodel for UML-RT graphical notation

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="statemach"
nsURI="http://www.eclipse.org/papyrusrt/xtumlrt/statemach"
   nsPrefix="statemach">
  <eClassifiers xsi:type="ecore:EClass" name="StateMachine"
eSuperTypes="../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//Behaviour">
    <eStructuralFeatures xsi:type="ecore:EReference" name="top" lowerBound="1"
eType="#//CompositeState"
      containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Vertex" abstract="true"
eSuperTypes="../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//NamedElement">
    <eStructuralFeatures xsi:type="ecore:EReference" name="incommingTransitions" upperBound="-1"
      eType="#//Transition" eOpposite="#//Transition/targetVertex"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="outgoingTransitions" upperBound="-1"
      eType="#//Transition" eOpposite="#//Transition/sourceVertex"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Transition"
eSuperTypes="../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//RedefinableElement">
    <eStructuralFeatures xsi:type="ecore:EReference" name="targetVertex" lowerBound="1"
      eType="#//Vertex" eOpposite="#//Vertex/incommingTransitions"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="sourceVertex" lowerBound="1"
      eType="#//Vertex" eOpposite="#//Vertex/outgoingTransitions"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="triggers" upperBound="-1"
      eType="#//Trigger" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="guard" eType="#//Guard"
      containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="actionChain" eType="#//ActionChain"
      containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="State" abstract="true" eSuperTypes="#//Vertex
../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//RedefinableElement">
```

```
    <eStructuralFeatures xsi:type="ecore:EReference" name="entryAction" eType="ecore:EClass
../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//AbstractAction"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="exitAction" eType="ecore:EClass
../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//AbstractAction"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="entryPoints" upperBound="-1"
        eType="#//EntryPoint" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="exitPoints" upperBound="-1"
        eType="#//ExitPoint" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Pseudostate" abstract="true"
eSuperTypes="#//Vertex"/>
  <eClassifiers xsi:type="ecore:EClass" name="SimpleState" eSuperTypes="#//State"/>
  <eClassifiers xsi:type="ecore:EClass" name="CompositeState" eSuperTypes="#//State">
    <eStructuralFeatures xsi:type="ecore:EReference" name="substates" upperBound="-1"
        eType="#//State" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="transitions" upperBound="-1"
        eType="#//Transition" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="vertices" upperBound="-1"
        eType="#//Vertex" derived="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="initial" eType="#//InitialPoint"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="deepHistory" eType="#//DeepHistory"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="choicePoints" upperBound="-1"
        eType="#//ChoicePoint" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="junctionPoints" upperBound="-1"
        eType="#//JunctionPoint" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="terminatePoint"
eType="#//TerminatePoint"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Trigger"
eSuperTypes="../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//NamedEleme
nt"/>
  <eClassifiers xsi:type="ecore:EClass" name="Guard"
eSuperTypes="../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//NamedEleme
nt">
```

```
    <eStructuralFeatures xsi:type="ecore:EReference" name="body" eType="ecore:EClass
../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//AbstractAction"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ActionChain"
eSuperTypes="../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//NamedEleme
nt">
    <eStructuralFeatures xsi:type="ecore:EReference" name="actions" upperBound="-1"
        eType="ecore:EClass
../../org.eclipse.papyrusrt.xtumlrt.common.model/model/common.ecore#//AbstractAction"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="EntryPoint" eSuperTypes="#//Pseudostate"/>
  <eClassifiers xsi:type="ecore:EClass" name="ExitPoint" eSuperTypes="#//Pseudostate"/>
  <eClassifiers xsi:type="ecore:EClass" name="InitialPoint" eSuperTypes="#//Pseudostate"/>
  <eClassifiers xsi:type="ecore:EClass" name="DeepHistory" eSuperTypes="#//Pseudostate"/>
  <eClassifiers xsi:type="ecore:EClass" name="ChoicePoint" eSuperTypes="#//Pseudostate"/>
  <eClassifiers xsi:type="ecore:EClass" name="JunctionPoint" eSuperTypes="#//Pseudostate"/>
  <eClassifiers xsi:type="ecore:EClass" name="TerminatePoint" eSuperTypes="#//Pseudostate"/>
</ecore:EPackage>
```

Ecore-based metamodel for UML-RT textual notation

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="hclScope"
nsURI="http://www.xtext.org/example/hclscope/HclScope"
    nsPrefix="hclScope">
  <eClassifiers xsi:type="ecore:EClass" name="StateMachine">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="states" upperBound="-1"
        eType="#//State" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="initialtransition"
eType="#//InitialTransition"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="junction" upperBound="-1"
        eType="#//Junction" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="choice" upperBound="-1"
        eType="#//Choice" containment="true"/>
```

```
  <eStructuralFeatures xsi:type="ecore:EReference" name="transition" upperBound="-1"
    eType="#//Transition" containment="true"/>
 </eClassifiers>
 <eClassifiers xsi:type="ecore:EClass" name="State" eSuperTypes="#//Vertex">
  <eStructuralFeatures xsi:type="ecore:EReference" name="internaltransition" upperBound="-1"
    eType="#//InternalTransition" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="entryaction" upperBound="-1"
    eType="#//EntryAction" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="exitaction" upperBound="-1"
    eType="#//ExitAction" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="entrypoint" upperBound="-1"
    eType="#//EntryPoint" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="exitpoint" upperBound="-1"
    eType="#//ExitPoint" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="states" upperBound="-1"
    eType="#//State" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="initialtransition"
eType="#//InitialTransition"
    containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="junction" upperBound="-1"
    eType="#//Junction" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="choice" upperBound="-1"
    eType="#//Choice" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="transition" upperBound="-1"
    eType="#//Transition" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="historytransition" upperBound="-1"
    eType="#//HistoryTransition" containment="true"/>
 </eClassifiers>
 <eClassifiers xsi:type="ecore:EClass" name="EntryAction">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
 </eClassifiers>
 <eClassifiers xsi:type="ecore:EClass" name="ExitAction">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
 </eClassifiers>
 <eClassifiers xsi:type="ecore:EClass" name="InitialState">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

```
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Junction" eSuperTypes="#//Vertex"/>
  <eClassifiers xsi:type="ecore:EClass" name="Choice" eSuperTypes="#//Vertex"/>
  <eClassifiers xsi:type="ecore:EClass" name="EntryPoint" eSuperTypes="#//Vertex"/>
  <eClassifiers xsi:type="ecore:EClass" name="ExitPoint" eSuperTypes="#//Vertex"/>
  <eClassifiers xsi:type="ecore:EClass" name="DeepHistory">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="InitialTransition" eSuperTypes="#//Transitions">
    <eStructuralFeatures xsi:type="ecore:EReference" name="initialstate" eType="#//InitialState"
        containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="initialto" eType="#//Vertex"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Transition" eSuperTypes="#//Transitions">
    <eStructuralFeatures xsi:type="ecore:EReference" name="from" eType="#//Vertex"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="to" eType="#//Vertex"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="InternalTransition">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="transitionbody"
eType="#//TransitionBody"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="HistoryTransition" eSuperTypes="#//Transitions">
    <eStructuralFeatures xsi:type="ecore:EReference" name="from" eType="#//Vertex"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="to" eType="#//DeepHistory"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TransitionBody">
    <eStructuralFeatures xsi:type="ecore:EReference" name="methodparameter" upperBound="-1"
        eType="#//MethodParameterTrigger" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="portevent" upperBound="-1"
        eType="#//PortEventTrigger" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="trigger" upperBound="-1"
        eType="#//Trigger" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="transitionguard"
eType="#//TransitionGuard"
```

```
      containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="transitionoperation"
eType="#//TransitionOperation"
      containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TransitionGuard">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TransitionOperation">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Trigger">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Method">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Parameter">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="MethodParameterTrigger">
    <eStructuralFeatures xsi:type="ecore:EReference" name="method" eType="#//Method"
      containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="parameter" eType="#//Parameter"
      containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Port">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Event">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="PortEventTrigger">
```

```
    <eStructuralFeatures xsi:type="ecore:EReference" name="port" eType="#//Port"
containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="event" eType="#//Event"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Vertex">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Transitions">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="transitionbody"
eType="#//TransitionBody"
        containment="true"/>
  </eClassifiers>
</ecore:EPackage>
```