Blended Graphical and Textual Modelling of UML-RT state-machines: an industrial experience

Malvina Latifaj[1][0000−0002−2754−9568], Federico Ciccozzi[1][0000−0002−0401−1036], Muhammad Waseem Anwar[1][0000−0002−1193−5683], and Mattias Mohlin[2]

[1] Mälardalen University, Västerås, Sweden
{name.surname}@mdh.se
[2] HCL Technologies, Malmö, Sweden
mattias.mohlin@hcl.com

Abstract. The ever increasing complexity of modern software systems requires engineers to constantly raise the level of abstraction at which they operate to suppress the excessive complex details of real systems and develop efficient architectures. Model Driven Engineering has emerged as a paradigm that enables not only abstraction but also automation. UML, an industry de-facto standard for modelling software systems, has established itself as a diagram-based modelling language. However, focusing on only one specific notation limits human communication and the pool of available engineering tools. The results of our prior experiments support this claim and promote the seamless use of multiple notations to develop and manipulate models. In this paper we detail our efforts on the provision of a fully blended (i.e., graphical and textual) modelling environment for UML-RT state-machines in an industrial context. We report on the definition of a textual syntax and advanced textual editing for UML-RT state-machines as well as the provision of synchronization mechanisms between graphical and textual editors.

Keywords: UML-RT · HCL RTist · Xtext · QVTo · model transformation · model synchronization · blended modelling

1 Introduction

The complexity of software systems has been growing at an unbelievable pace for decades now. Relying merely on human efforts to develop high-quality software is presently regarded as a futile attempt. It can be argued that in the face of this complexity, without an efficient and effective architecture, software is inscrutable [8]. To tackle the architectural complexity of software development, Model Driven Engineering (MDE) has emerged as a software engineering paradigm that focuses on raising the level of abstraction when architecting software systems [2,19]. This is done by promoting modelling languages and models, which are closer to human understanding, instead of code, closer to machines, as core architectural and engineering artefacts. This approach imposes limits

to the problem-domain, facilitates the identification of relevant abstractions, and avoids superfluousness. Together with that, MDE pledges automation by exploiting modelling technologies that among others, enable the generation of fully fledged code from architectural models.

Domain-specific abstractions facilitating the architectural description of software systems are defined using formal specifications expressed in Domain Specific Modeling Languages (DSMLs), which capture the core aspects of a domain, thus promoting productivity, efficiency and comprehensibility of domain-specific problems.

UML is the most used architecture description language in industry [13], the de-facto modelling standard in industry [9], 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, called UML profiles; in this paper, we focus on the UML real-time profile (UML-RT) [18], as this is the profile implemented in the commercial tool HCL RTist[3] of our industrial partner. We also leverage an open-source implementation of it provided in the Eclipse Papyrus-RT[4] tool.

1.1 Problem, motivation, and the RTist case

Domain-specific modelling tools, like RTist, traditionally focus on one specific editing notation (such as text, diagrams, tables or forms). This limits human communication, especially across stakeholders with varying roles and expertise. Moreover, architects and engineers may have different notation preferences; not supporting multiple notations negatively affects their throughput. Besides the limits on 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[5] or Sparx Systems Enterprise Architect[6], 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. RTist is not an exception. It provides support for modelling UML-RT architectures and applications based on graphical *composite structure diagrams*, to model structure, and *state-machine diagrams*, to model behavior. In addition, the implementation of UML-RT in RTist provides support for leveraging C/C++ action code for the description of fine-grained, algorithmic, behaviors within graphical state-machines. That is needed to enable the definition of full-fledged

[3] https://www.hcltechsw.com/rtist
[4] https://www.eclipse.org/papyrus-rt/.
[5] http://www-03.ibm.com/software/products/en/ratsadesigner/
[6] https://sparxsystems.com/

UML-RT models from which executable code can be automatically generated. While providing means to model graphical entities and "program" algorithmic behaviours textually, the two are disjoint, since the modelling of UML-RT is graphical only and the textual C/C++ is injected in graphical models as a "foreign" entity and with almost no overlapping with graphical model elements. The aim is instead to achieve a modelling tool that is able to make different stakeholders to work on overlapping parts of the models using different modelling notations (e.g., graphical and textual) in an automated manner.

1.2 Paper contribution

In this paper we describe our proposed solution for providing a fully blended graphical-textual modelling environment for UML-RT state-machines in an industrial setting. Our experiments in a previous study with blended graphical-textual modelling showed that the seamless use of different notations can significantly boost the architecting of software using UML profiles [1]. The results of those experiments together with the exposed wish of RTist customers of being able to design software via multiple notations led us to initiate this work towards an automated support for blended modelling of UML-RT in RTist. In a prior work [11], we describe the effort of designing, implementing and integrating a textual notation for UML-RT state machines in RTist. In this paper, we extend that work, and address the problem formulated in the previous section by providing the following additional research contributions.
C1. Definition of a textual editor for UML-RT state-machines with advanced formatting features including systematic support for hidden regions which group hidden tokens (e.g., comments, whitespaces) between two semantic tokens.
C2. Provision of synchronization mechanism between textual and graphical notations to achieve a seamless blended modelling environment and validation of the solution.

1.3 Paper outline

The remainder of the paper is organized as follows. In Section 2 we describe the concept of blended modelling and in Section 3 we detail the design of our proposed solution. The implementation details of the solution are presented in Section 4, whereas the validation is discussed in Section 5. The related works are detailed in Section 6 and the paper is concluded in Section 7 with a brief summary and an overview of the current and upcoming enhancements to the overall blended modelling approach.

2 Blended Modelling: what and why

We have previously defined the notion of *blended modelling* [4] 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.

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.

Flexible separation of concerns and better communication. Providing graphical and textual modelling editors for different aspects and sub-parts (even overlapping) of a DSML like UML-RT enables the definition of concern-specific architectural 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* [3] 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.

Faster modelling activities. We have experimented with blended modelling of UML profiles [1] and the seamless combination of graphical and textual modelling has shown a decreased modelling effort in terms of time thanks to the following two factors:

1. Any stakeholder can choose the notation that better fits his/her needs, personal preference, or the purpose of the 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[7], such as copy&paste and regex search&replace, syntax highlighting, code completion, quick fixes, cross referencing, recovery of corrupted artefacts, text-based diff and merge

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

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.

3 Design Solution

In this section, we detail the solution design, illustrated in Fig. 1, for the provision of a blended modelling environment for UML-RT state-machines. Note that in order to maximise accessibility to our solution, we describe the solution for an open-source tool, Eclipse Papyrus-RT, which is orthogonal to the one in RTist (which also is Eclipse EMF-based).

The starting point is the already existing Ecore-based DSML formalizing the UML-RT profile (i.e., MM_G), which is utilized to instantiate graphical models (i.e., M_G) in both Papyrus-RT and RTist. Using this DSML as blueprint, we define a textual language (i.e., MM_T) in Xtext[8] that will be used to instantiate textual models (i.e., M_T) of UML-RT state-machines. Moreover, using Xtext's formatting APIs, we also customize the textual editor to preserve essential textual information, such as lines, formatting and hidden regions like comments. This provides our first contribution **C1**. Subsequently, we design and implement the synchronization mechanisms between the two notations by model-to-model (M2M) transformations [16]. These transformations are defined on the basis of implicit mappings between metaelements of the source and target metamodels and implemented in terms of the operational version of the Query/View/Transformation language (QVTo[9]) in Eclipse. QVTo supports only unidirectional transformations, thus, to achieve bidirectionality, we defined two unidirectional transformations; MM_T2MM_G, where the source metamodel is MM_T and target metamodel is MM_G, and MM_G2MM_T, where the source metamodel is MM_G and the target metamodel is MM_T. Both model transformations are horizontal as the source and target model reside in the same abstraction level, and exogenous as the models are expressed in different modelling languages. This makes for our second contribution **C2**. Further details on the definition of the textual syntax and synchronization mechanisms can be found in Section 3.1 and 3.2. The implementation details are included in Section 4, and the validation of the solution is detailed in Section 5.

[8] https://www.eclipse.org/Xtext/
[9] https://wiki.eclipse.org/QVTo

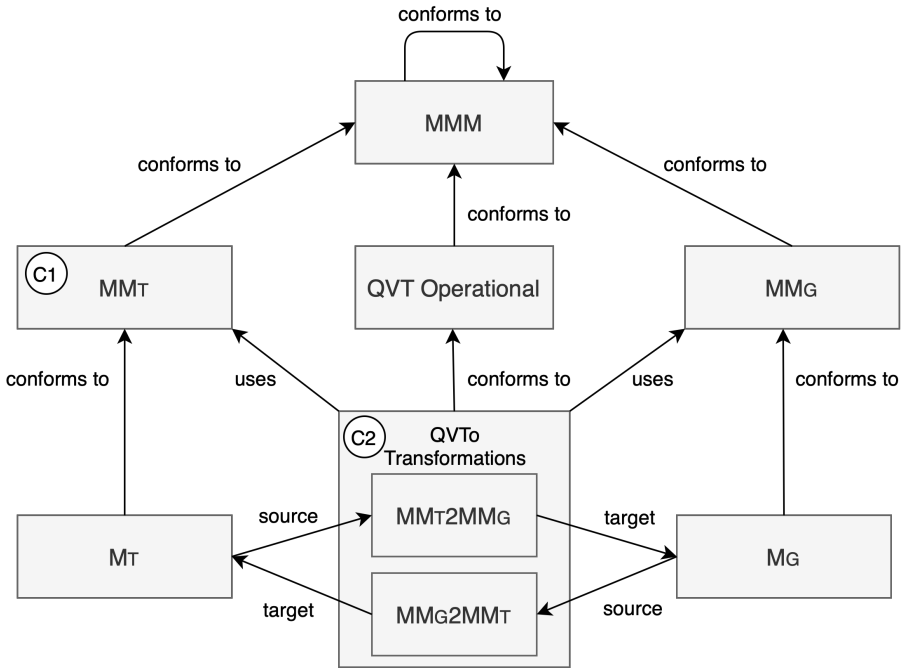


Fig. 1: Synchronization solution design

3.1 Textual notation for UML-RT state-machines

Textual language workbench. To complement the existing graphical editor in RTist with a textual notation and editor, a suitable language workbench needs to be carefully selected. HCL RTist and Papyrus-RT are Eclipse-based environments that leverage the Eclipse Modeling Framework (EMF)[10] as backbone. Thereby, by choosing an EMF-based language workbench, we could leverage EMF as a common data layer. For this reason, we chose Xtext, a framework for the development of textual DSMLs, based on EBNF grammars. The textual editor supports an outline view, syntax highlighting, error checking, quick-fix proposals, and many other features provided by Xtext. Furthermore, Xtext provides code completion for keywords and cross-references by increasing the usability of the language and decreasing the learning curve.

Textual notation definition. Our goal was to introduce a textual notation (and related editor) to the already existing UML-RT profile supported by RTist. A possible alternative was to use the underlying metamodel consumed by the RTist's graphical editor as an input for Xtext to automatically generate a textual editor. However, although easy to implement, this solution generates erroneous

[10] https://www.eclipse.org/modeling/emf/

and unintuitive grammar, too far from the expectations of RTist's architects and customers. Manually editing this generated grammar would have been a tedious and potentially error-prone process. Therefore, we decided to design a textual notation in terms of an Xtext grammar, from scratch. Starting from a wish-list of RTist's customers and architects, and using the UML-RT metamodel portion describing state-machines as blueprint, we manually defined our UML-RT textual notation for state-machines in Xtext. The steps needed for the definition of the grammar were the following.

① *Identify reserved keywords:* When defining a DSML, it is crucial to identify the reserved keywords used to typify the core concepts of the language. The importance of these keywords lies in improved readability, higher language familiarity, and efficient parsing as they serve as directives for specific concepts. The chosen keywords for the textual syntax for UML-RT state-machines are the following: *capsule, statemachine, state, initial, junction, choice, entry, exit, entrypoint, exitpoint, history, transition, when, on* and *inherits*. A more detailed description of the concepts represented by each keyword can be found in the official documentation [11] of UML-RT by HCL.

② *Elements' ordering strategy:* Even though it is not mandatory for our language to have a fixed order of elements, this approach enhances readability and navigation of the textual syntax, as well as increased predictability on where the elements created by using the graphical notation will be placed in the textual syntax. Our grammar is based on the vertical distance approach where elements that affect each other's understandability and are closely related [14], are grouped together and have a low vertical distance. Furthermore, being that this grammar prohibits cross-references before element declaration, we take the aforementioned statement into consideration and make sure that elements that need to be cross-referenced will be declared before the cross-reference occurs.

③ *A spoonful of syntactic sugar:* The majority of programming languages, including C++, which is used as action code for behavioral state-machines, makes use of statement terminators in the form of semi-colons. Being that one of the main goals when introducing this textual syntax is for developers to use it jointly with the C++ action code, we introduced consistent use of semi-colons for indicating statement termination to make the grammar more conforming to C++ and to increase readability. For the same readability reasons and developers' preferences, we also introduce colons after transition names. Furthermore, to make the grammar more compact, we allow the declaration of multiple objects of the same type in one single line of code. Due to the combination of the textual syntax with action code, we need to handle C++ code blocks so we can "isolate" them and make them distinguishable from the rest of the grammar. For this reason, we include back-ticks in order to enclose code snippets and to make the lexer aware of where the code block begins and ends.

The overall goal during this process was to keep a fixed concrete syntax while

[11] https://rtist.hcldoc.com/help/topic/com.ibm.xtools.rsarte.webdoc/pdf/RTist%20Concepts.pdf

simultaneously enhancing the abstract syntax, even though frequently we had to trade-off between ease of expression in the concrete syntax and extra complexity in the abstract syntax.

Enforcing UML-RT's modularity. Scoping in Xtext is concerned with the visibility of elements; therefore, the scope provider computation returns the target candidates that are visible in the current context and by a given reference. In order to enforce the UML-RT's modularity, it is necessary to specify a custom scope provider. The default behavior of Xtext allows establishing a cross-reference to all the elements of a particular type that are located inside the same Eclipse resource (i.e., project). By customizing the scope provider, we restrict this behaviour, and only allow cross-references for elements declared in the same model file. The rationale behind this decision lies in the fact that multiple model files containing different capsules can be located inside the same resource, and a particular capsule should not be able to cross-reference the elements of other capsules. However, a key concept in which UML-RT relies on to reuse and extend parts of existing state-machines is the inheritance mechanism. When capsule A inherits capsule B, the state-machine of capsule A implicitly inherits the state-machine of capsule B. Therefore, to support inheritance, we need to customize the scope provider so that it allows cross-references for elements not only from the capsule itself, but also from the inherited capsule, in case there is one. Another default behavior of Xtext consists in allowing cross-references for all elements of a particular type declared in the same model file, regardless of their level of nesting. This contradicts an important UML-RT concept; compound transitions. Since transitions in UML-RT state-machines can not cross state boundaries, the concept of compound transitions is applied, consisting of multiple segments that are connected by means of pseudo-states. However, with the default behaviour of Xtext, a transition can cross state boundaries. Therefore, the scope provider is customized to restrict that and provide the desired behavior in conformance with UML-RT concepts by allowing 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.

Advanced textual editing features. The aforementioned steps provide a solid platform for developing a sophisticated editor for the specification of textual state-machines. Besides the editing features provided out-of-the-box by Xtext for textual languages, we incorporated formatting features like text indentation and syntax highlighting in the textual editor to simplify the specification of these textual models. Furthermore, the support to associate both single and multiline comments within textual specifications is provided too.

3.2 Synchronization transformations

Model transformation language. For model transformations, we chose QVTo, which is an implementation of the Operational Mapping Language defined by

Object Management Group's (OMG's) Meta-Object Facility (MOF) 2.0 Query/ View/Transformation (QVT[12]). 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 declarative and imperative QVT and it is very well-suited for both exogenous and endogenous transformations, also in-place.

Transformation structure. The transformations are executed in Eclipse QVTo, the only actively maintained QVTo implementation, adhering to its default structure composed of the modeltype declarations, the transformation declaration, main function and mapping operations. In the following, we detail the QVTo structure.

- *Modeltype declaration*: The modeltype declaration in QVTo serves as a reference to the metamodels that will be used for the transformation. When declaring the modeltype, it is obligatory to define the name and reference. The latter can be specified either by using the package `nsURI` or file location `URI`. In our use case, we reference the metamodels via the `nsURI` which is resolved via the global package registry in the running platform instance. Optionally, the modeltype definition can include the conformance kind (i.e., strict or effective) and a set of constraint expressions (i.e., OCL expressions). In our case we do not define the conformance kind, thus by default it assumes an effective conformance. The rationale behind this decision is to allow the transformations to be applied to similar metamodels. Moreover, we do not define any constraint expressions as we have no additional restrictions over the set of the involved models. As an example, the modeltype definition that references MM_T, is detailed in the following.

  modeltype *MM_T* ***uses*** *MM_T_Package_Name (MM_T_Package_nsURI)*

- *Transformation declaration*: The transformation declaration defines the name of the unidirectional transformation and specifies the involved metamodels. Additionally it details the direction kind of the transformation via the following values; `in`, `out`, and `inout`. As an example, the MM_T2MM_G transformation declaration has the following structure that details the name of the transformation, the involved metamodels, and the direction of the transformation.

  transformation *MM_T2MM_G (**in** source:MM_T, **out** target:MM_G)*;

- *Main function*: The main function is also referred to as the entry point of the transformation as it initiates the execution of the transformation by executing the operations defined in the body of the function. As an example, for the MM_T2MM_G transformation, the defined operation selects the root

[12] https://www.omg.org/spec/QVT/1.3/About-QVT/

metalelements (i.e., metaelements at the highest level) of MM_T, and filters out the `StateMachine` metaelement. Additionally, it invokes the `SM2SM()` "top-level" mapping rule that maps the `StateMachine` metaelement of MM_T to the `StateMachine` metaelement of MM_G.

```
main() {
src.rootObjects()[MMT::StateMachine] -> map SM2SM();
}
```

– *Mappings:* The transformations in QVTo are executed by means of mapping operations. Each mapping operation consists of a signature, an optional guard (i.e., *when* clause), a mapping body, and an optional post condition (i.e., *where* clause). The signature of the mapping operation minimally includes the following elements:
Mapping Type: A mapping operation can either be an abstract mapping or a concrete mapping (non-abstract). An abstract mapping operation is distinguished by the `abstract` keyword which indicates that the mapping can not be invoked in isolation. Such mapping operations are common when the target metaelement is abstract and are usually inherited by other mappings with concrete target types.
Metaelements: QVTo does not strictly require the fully qualified name of the metaelements that are to be mapped (i.e, metamodelName::metaelementName), but in the presence of source and target metamodels that contain similar concepts, the fully qualified name is used to resolve possible ambiguities.
Mapping Name: Serves to identify the mapping and it is always unique.
As an example of a mapping signature, in the following we detail an abstract mapping between the concrete `State` metaelement of MM_T and the abstract `State` metaelement of MM_G, where we use the fully qualified name to separate them from one another.

```
abstract mapping MMT::State::State2State() : MMG::State { ... }
```

Moreover, a mapping declaration can include mapping guards described with OCL expressions and distinguished by the `when` keyword. If the guard evaluates to true, it restricts the execution of the mapping operation only to a subset of elements; alternatively, the mapping operation is not invoked.
Finally, the body of the mapping operation is populated by assigning EReferences and EAttributes of the source metaelement to corresponding EReferences and EAttributes of the target metaelement. For EReferences, a type-dependent mapping operation is invoked by using the `map` keyword. When invoking the mapping on a single element, the element is followed by a dot which precedes the `map keyword`. Alternatively, when invoking the mapping on a collection of elements (i.e., Set, Bag, Sequence, or OrderedSet), the latter is followed by an arrow which precedes the `map keyword`. Moreover, the `self` and `result` variables, refer to the source and target metaelements, respectively. As an example, in the following we detail a regular mapping

between `State` and `CompositeState` that is extended with a mapping condition.

```
mapping MMT::State::State2CMPState() : MMG::CompositeState
when {not(self.states -> isEmpty() ...)}
{
result.choicePoints := self.choice.map Choice2Choice();
result.name := self.name; }
```

To conclude the definition of the synchronization transformations, we detail two additional QVTo concepts that we used for this purpose: `inherits` and `disjuncts`. Inheritance enables the reuse of other mapping operations with the condition that the signature of the mapping which is inheriting must conform to the signature of the mapping that is being inherited. In short, the source and target metaelements of the inheriting mapping operation must either be the same or subtypes of the source and target metaelements of the inherited mapping, respectively. This feature allows for a more compact code and increased readability as the operations are defined once in the inherited mapping and reused in each inheriting mapping. In the following, we provide an example of these mapping operations. Mapping operation `State2SimpleState` inherits mapping operation `State2State`. The signatures are conformant as the source metaclasses are the same, while the target metaclass of the inheriting mapping (i.e., `State2SimpleState`), is a subtype of the output metaclass of the inherited mapping (i.e., `State2State`). By inheriting this mapping, in the `State2SimpleState` mapping operation, we do not need to rewrite what is already defined in the `State2State` mapping operation, as it is automatically invoked when the `State2SimpleState` mapping operation is executed.

```
mapping MMT::State::State2SimpleState() : MMG::SimpleState
inherits MMT::State::State2State { ... }
```

Moreover, mapping operations can be defined as disjunctions of multiple other mappings, which are then extended with distinct guards. When invoking such operation, the guards of the mapping operation alternatives specified after the `disjuncts` keyword are sequentially checked. When the first guard evaluates to `true`, the corresponding mapping operation is invoked; alternatively if they all evaluate to `false` it returns `null`. The body of such mapping operations is always empty, because that part of the code is unreachable. This concept is primarily applied to operations transforming abstract types that are extended by multiple subtypes. In this case, the alternative mapping specified after the `disjunct` keyword, consists of subtypes of the original mapping.

mapping *MM$_T$::State::StateDisjunct() : MM$_G$::State*
disjuncts *MM$_T$::State::State2SimpleState,*
MM$_T$::State::State2CompositeState { }

4 Implementation

In this section we present the implementation details of the proposed solution and show examples both of the textual syntax and model instances after applying the model transformations for synchronization.

4.1 Textual language and editor for UML-RT

Based on the aforementioned approach (see Section 3.1), in this section we detail the implementation specifics of the textual language and editor in Eclipse Xtext. We focus particularly on the customization of the scope provider in Xtext to enable inheritance and compound transitions in our textual UML-RT, as well as on the customization of the Xtext's formatter for advanced textual editing and formatting features.

Customization of the scope provider. Listing 1.1 provides a snippet of the customized scope provider in Xtext for supporting the concept of inheritance. The conditional `if` statement in Line 1 checks whether the current capsule inherits another capsule. If this condition evaluates to true, the `EObject` is down casted to a `Capsule` object in Line 3. Depending on the instance type of the context's container, the desired elements of the inherited capsule are added to the list of eligible candidates that the scope provider will return as detailed in Lines 4-6.

```
1 if (rootCapsule.getSuperclass() != null) {
2   parentinheritance = rootCapsule.getSuperclass();
3   Capsule inheritedCapsule = (Capsule) parentinheritance;
4   if (context.eContainer() instanceof StateMachine) {
5   transitionfrom.addAll(inheritedCapsule.getStatemachines().getStates());
6   ...
7   }
8 }
```

Listing 1.1: Inherited capsule scope provider

Listing 1.2 provides instead a snippet of the customized scope provider in Xtext for supporting the concept of compound transitions. The `eContainer()` method in Line 2 is used to return the containing object of the context object. The list of objects `T_F` in Line 3, is initialized to be used for storing all the eligible candidates that can be cross-referenced. The block of code to be executed if the specified condition of the `if` statement in Line 4 evaluates to true, down casts the `currentParent` EObject into a `StateMachine` object and uses the `addAll()` method to add all elements of a specific type that are contained in the state-machine as the context element, to the list.

```
else if (reference == HclScopingPackage.Literals.TRANSITION__FROM) {
  EObject currentParent = context.eContainer();
  List<EObject> T_F = new ArrayList<>();
  if (currentParent instanceof StateMachine) {
    StateMachine s = (StateMachine) currentParent;
    T_F.addAll(s.getStates());
    ....
    for (State states : rootCapsule.getStatemachines().getStates()) {
          transitionfrom.addAll(states.getEntrypoint());
          transitionfrom.addAll(states.getExitpoint());
    }
  }
return Scopes.scopeFor(T_F, N_C, IScope.NULLSCOPE);
```

Listing 1.2: Compound transitions scope provider

Customization of the textual editor Parsing and serialization are two major concepts in Xtext associated with the textual model and Abstract Syntax Tree (AST), respectively. The instance of a grammar in the editor, technically referred to as XtextResource, is represented through a textual model. The equivalent AST is generated from the textual model through the parser. On the other hand, the serializer converts the AST into the equivalent textual model. The conversions between textual model to AST and vice versa are very frequent and, therefore, Xtext supports the exploitation of built-in APIs to customize certain functionalities that may be required before or after the conversions from one to the other. We exploited the built-in APIs for customizing our textual editor.

The synchronizations targeted in our solution between the textual and graphical models lead to frequent changes in the AST related to the textual model, like deletion or addition of textual elements. In this case, the line numbers and other hidden region elements like comments need to be updated in the textual editor. Furthermore, the formatting of the text needs to be preserved in the textual editor after synchronizations since it brings along semantic information in most cases. To achieve this, we utilized Xtext's formatting infrastructure. In particular, we extended the `AbstractFormatter2` class to implement a customized state-machine formatter, composed of two core functions (i.e., *Lines* and *Hidden Regions*). The *Lines* function updates the sequence of lines in the textual editor according to the synchronized changes to the AST. *Hidden Regions* instead preserves the place of hidden regions that group all hidden tokens (e.g., whitespace, newlines, tabs and comments) between two semantic tokens upon changes to the AST.

4.2 Synchronization

Based on the aforementioned approach (see Section 3.2), in this section, we detail the implementation specifics of the synchronization model transformations in Eclipse QVTo. Synchronization mechanisms are provided in terms of two unidirectional M2M transformations; MM_T2MM_G and MM_G2MM_T. The majority of metaelements between the two metamodels require a one-to-one mapping, thus the mapping rules are rather straightforward. In the following, for each

model transformation, we discuss the mapping operations that highlight a few interesting and less simple cases.

Textual to graphical synchonization – MM_T2MM_G

– The `StateMachine` metaelement behaves as a root element both in MM_T and MM_G. Nevertheless, there is a notable difference between the two. In MM_T, `StateMachine` has multiple children and its containment of elements `State` has a zero-to-many (0..*) cardinality. Instead, in MM_G, `StateMachine` has a one-to-one (1..1) cardinality to `CompositeState`. In short, whilst in MM_T `StateMachine` can contain many `State`s as immediate children, in MM_G the `StateMachine` can only contain one `CompositeState` as its immediate child, and in turn `CompositeState` would contain the other elements. Consequently, when transforming a `StateMachine` in MM_T to a `StateMachine` in MM_G, two possible narratives need to be taken into account. First, if we consider a model instance of MM_T (i.e., M_T) and the `StateMachine` of this model instance contains only one `State` and no other immediate children, `State` is transformed to a `CompositeState` in M_G (i.e., model instance of MM_G) as detailed in Lines 4-5 in Listing 1.3. Otherwise, if the `StateMachine` in M_T contains more than one immediate state, when transforming to a `StateMachine` in M_G, a `CompositeState` object is created (Lines 9-10) and the immediate children of the `StateMachine` in M_T are assigned as immediate children (Line 11) of the `CompositeState` in M_G.

```
1  mapping text::StateMachine::SM2SM() : graph::StateMachine{
2  result.name:=self.name;
3
4  if (self.states -> size() = 1 and self.initialtransition -> isEmpty() and
       self.transition -> isEmpty() and self.junction -> isEmpty() and self
      .choice -> isEmpty()) {
5    result.top := self.states -> first().map State2CMPState();
6  }
7
8  else {
9  var cs := object graph::CompositeState{};
10 top :=cs;
11 cs.substates := self.states.map toState();
```

Listing 1.3: `StateMachine` to `StateMachine`

– With respect to states, MM_T considers only the `State` metaclass, while MM_G makes a distinction between `SimpleState` and `CompositeState`, which extend the `State` metaclass. Thus, when transforming a `State`, the mapping operations in Lines 1-9 in Listing 1.4 need to be extended with mapping guards that determine if the `State` metaclass will be transformed to a `SimpleState` or `CompositeState`. Moreover, an additional mapping operation is defined in Lines 13-14, which is a disjunction of the aforementioned mapping operations and is invoked in Line 10. Upon its execution, the guards of `State2SimpleState` and `State2CMPState` are checked in a sequential order. For a `State` to be transformed to a `SimpleState`, the `State` should have no children. To evaluate that we use the OCL expression `isEmpty()`, which

evaluates whether the collection is empty or not, in Line 3. Alternatively, a `CompositeState` has children, thus in Line 8 we use the OCL expression `notEmpty()`.

```
1  mapping text::State::State2SimpleState() : graph:: SimpleState
2  inherits text::State::State2State
3  when {self.states -> isEmpty() and self.entrypoint -> isEmpty() and self.
       exitpoint -> isEmpty()  and self.junction -> isEmpty() and self.
       choice -> isEmpty() }
4  {  }
5
6  mapping text::State::State2CMPState() : graph:: CompositeState
7  inherits text::State::State2State
8  when {self.states -> notEmpty() or self.entrypoint -> notEmpty() or self.
       exitpoint -> notEmpty()  or self.junction -> notEmpty() or self.
       choice -> notEmpty()}
9  {
10 substates := self.states.map State2StateDisjunct();
11 ...
12 }
13 mapping text::State::State2StateDisjunct() : graph::State
14 disjuncts text::State::State2SimpleState, text::State::State2CMPState
15 {}
```

Listing 1.4: `State` to `SimpleState` and `CompositeState`

Graphical to textual synchronization – MM$_G$2MM$_T$

– `SimpleState` and `CompositeState` metaelements in MM$_G$ both have to be transformed to `State` in MM$_T$. Hence, two corresponding mapping operations are defined in Line 1 and Line 4 in Listing 1.5. In this particular situation, a disjunctive mapping operation (i.e., `State2StateDisjunct()`) is introduced in Line 10. Contrary to the first two mapping operations where a mapping body is defined, this operation specifies a list of mapping operations (i.e., `SimpleState2State` and `CMPState2State`) which are evaluated when the mapping operation is invoked. The invocation of the operation occurs in Line 7 when trying to map `substates` to `states` as the EType for `substates` is the abstract metaclass `State` which is extended by `SimpleState` and `CompositeState`.

```
1  mapping graph::SimpleState::SimpleState2State() : text::State
2  inherits graph::State::State2State {}
3
4  mapping graph::CompositeState::CMPState2State() : text::State
5  inherits graph::State::State2State
6  {
7      result.states := self.substates -> map State2StateDisjunct();
8      ...
9  }
10 mapping graph::State::State2StateDisjunct() : text::State
11 disjuncts graph::CompositeState::CMPState2State,
12           graph::SimpleState::SimpleState2State{}
```

Listing 1.5: `SimpleState` and `CompositeState` to `State`

– `Transition` metaclass in MM$_G$ can be transformed to either `HistoryTransition`, `InitialTransition`, `InternalTransition` or `Transition` in MM$_T$, all extending the `Transitions` metaclass, depending on the source and/or target

vertices it connects. First, the `T2Ts()` mapping operation is defined, which details the creation of the `TransitionBody` and the assignment operations for its children. This is then inherited by all other mapping operations that in their signature include `Transition` as source and a subtype of `Transitions` as target. Inheritance eliminates the need to rewrite the operations that are already defined in `T2Ts()` such as the creation of the `TransitionBody` and the corresponding assignments. In addition, all mapping operations that map `Transition` to subtypes of `Transitions` include a mapping guard (i.e., `when` clause) that specifies the type of vertices that the `Transition` must connect, to be transformed to a specific subtype of `Transitions`. The mapping body of these mapping operations details the assignments of the `sourceVertex` and `targetVertex` properties of the source metaelement, to `from` and `to` properties of the target metaelement, respectively as shown in Lines 18-19 in Listing 1.6. The mapping operation that is invoked on them is a disjunction of other mapping operations in which the source and target metaelements are subtypes of the source and target metaelements of the original mapping (i.e., `Vertex2VertexDisjunct`). This is because the types of the source and target metaelements of the invoked mapping (i.e., `Vertex2VertexDisjuncts`) must conform to the types of properties `from` and `to`.

```
1  mapping graph::Vertex::Vertex2VertexDisjunct() : text::Vertex
2  disjuncts graph::CompositeState::CMPState2State, graph::SimpleState::
       SimpleState2State, graph::EntryPoint::EntryPoint2EntryPoint, graph::
       ExitPoint::ExitPoint2ExitPoint,graph::ChoicePoint::ChoicePoint2Choice
       , graph::JunctionPoint::JunctionPoint2Junction{}
3
4  mapping graph::Transition::T2Ts() : text::Transitions
5  {
6      var TransitionBodyObject := object text::TransitionBody{
       transitionguard := self.guard.map Guard2TransitionGuard();
7          ....
8      };
9      transitionbody := TransitionBodyObject;
10     result.name := self.name;
11 }
12 mapping graph::Transition::T2T() : text::Transition
13 inherits graph::Transition::T2Ts
14 when {not(self.sourceVertex.oclIsTypeOf(InitialPoint) or
15       self.targetVertex.oclIsTypeOf(DeepHistory) or
16       self.sourceVertex = null or self.targetVertex = null)}
17 {
18     result._from := self.sourceVertex.map Vertex2VertexDisjunct();
19     result.to := self.targetVertex.map Vertex2VertexDisjunct();
20 }
21
22 mapping graph::Transition::T2HT() : text::HistoryTransition
23 inherits graph::Transition::T2Ts
24 when {self.targetVertex.oclIsTypeOf(DeepHistory)} { ...}
25
26 mapping graph::Transition::T2INI_T() : text::InitialTransition
27 inherits graph::Transition::T2Ts
28 when {self.sourceVertex.oclIsTypeOf(InitialPoint)} {...}
29
30 mapping graph::Transition::T2INT_T() : text::InternalTransition
31 when {self.sourceVertex = null and self.targetVertex = null} {...}
```

Listing 1.6: `Transition` to `Transitions`

– `Trigger` metaclass in MM_G can be transformed to either `Trigger`, `PortEventTrigger`, or `MethodParameterTrigger` in MM_T depending on which of the mapping guards (i.e., `when` clause) evaluates to true. Hence, three corresponding mapping operations are defined. The `PortEventTrigger` consists of a `Port` and `Event` separated by a dot (e.g., port.event), thus in order for a `Trigger` in MM_G to be transformed to a `PortEventTrigger` in MM_T it should match the pattern defined in Line 7 in Listing 1.7. Moreover, the `Port` and `Event` metaclasses in MM_T have no correspondence in MM_G therefore they should be created as new elements. Lines 9-17 detail the creation of `Port` and `Event` metaclasses and the assignment of the name attribute for each of them. The same procedure is applied to transform `Trigger` in MM_G to `MethodParameterTrigger` in MM_T and to create the `Method` and `Parameter` metaelements. The `MethodParameterTrigger` consists of a `Method` followed by parentheses, which may or not contain a `Parameter` (e.g., method(parameter)). In this case, the mapping condition specifies that the name of the `Trigger` in MM_G should match the pattern specified in Line 19.

```
1  mapping graph::Trigger::Trigger2Trigger() : text::Trigger
2  when {not(self.name.matches(".*\\(.*") or self.name.matches(".*\\..*"))}
3  {
4      result.name := self.name;
5  }
6  mapping graph::Trigger::Trigger2PETrigger() : text::PortEventTrigger
7  when {self.name.matches(".*\\..*")}
8  {
9      var PortObject := object text::Port{
10         name := self.name.substringBefore(".");
11     };
12     port := PortObject;
13     var EventObject := object text::Event{
14         name := self.name.substringAfter(".");
15     };
16     event := EventObject;
17 }
18 mapping graph::Trigger::Trigger2MPTrigger() : text::
       MethodParameterTrigger
19 when {self.name.matches(".*\\(.*")}
20 { ... }
```

Listing 1.7: `Trigger` to `Trigger`, `PortEventTrigger`, and `MethodParameterTrigger`

5 Validation and discussion

The M2M transformations detailed in this paper make possible the synchronization between multiple notations. As such, the correctness of the unidirectional transformations and the consistency between them is crucial.

The validation is conducted for RTist and Papyrus-RT by applying the model transformations to instances (i.e., models) of MM_G and MM_T. The representation of the graphical instance M_G is detailed in Fig. 4, while the representation of the textual instance M_T is detailed in Fig. 5. Fig. 4a details the graphical editor of M_G, whereas Fig. 4b details the Exeed editor (an extended version of the built-in tree-based reflective editor provided by EMF) of M_G. Alternatively, Fig. 5a

details the Xtext editor of M_T, whereas Fig. 5b details the Exeed editor of M_T. We have included the Exeed editor for both instances, as it conveys additional structural information that is not glaring in the other editors. In the following we describe the execution of the validation process.

① The correctness of the model transformations is validated by carrying out testing of the MM_G2MM_T and MM_T2MM_G model transformations at the unit level [21]. To carry out this procedure, we have defined a functional decomposition diagram for each model transformation. Such diagram, can facilitate the identification of the test *subjects*, as the nodes of the diagram (i.e., mapping operations) will represent the *subjects*. Additional consideration has been given to guarantee that the test cases cover all the mapping operations of a given model transformation. The metaelements of the input metamodels will be considered as *test inputs* whereas the *expected output* will be represented by a regular expression. For a test case to *pass*, the *expected output* of a given mapping operation must match *the actual output*. The latter is the result that we get after the execution of the mapping operation. In the following we use the `CMPState2State` mapping operation defined in the MM_G2MM_T model transformation to exemplify our manual testing process. In Fig. 2 we detail only a portion of the decomposition diagram to highlight `CMPState2State`. Then we extract the input of the `CMPState2State` mapping operation, which is the `CompositeState`. In Fig. 3 we detailed an instance of MM_G, which is used to test the `CMPState2State` mapping operation.

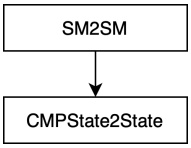


Fig. 2: Decomposition diagram portion

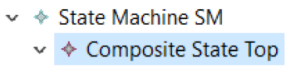
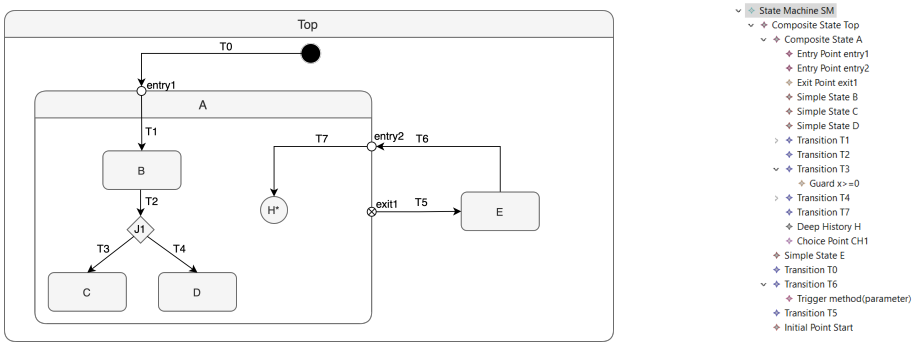


Fig. 3: Input

The *expected output* after the execution of the `CMPState2State` on the instance of MM_G detailed in Fig. 3 is as follows. The `CompositeState` **Top** included in the `StateMachine` **SM** in M_G must be transformed to a `CompositeState` **Top** included in the `StateMachine` **SM** in M_T. Lastly we execute the transformations and check whether the *actual output* is same as the *expected output*.

```
<hcl:StateMachine name="SM">
<states name="Top">
</hcl:StateMachine>
```

② The second step involves validating the consistency between the two model transformations. For achieving this, we apply the MM_G2MM_T model transformation to M_G detailed in Fig. 4. The output is M_T detailed in Fig. 5. We then apply the MM_T2MM_G model transformation to the output of the MM_G2MM_T model



(a) Graphical editor (b) Exeed Editor.

Fig. 4: Graphical model

transformation. The output of the MMT2MMG model transformation must be identical to the instance of MG detailed in Fig. 4 for the QVTo transformation to be considered consistent. The model transformations are revised until full consistency is achieved. In addition to the testing results, architects at HCL have assessed both the usability and usefulness of the textual notation, as well as the synchronization mechanisms between notations.

On another note, by inspecting the instances of MMG and MMT detailed in Fig. 4 and Fig. 5 respectively, we can highlight the most significant differences in terms of semantics and structure. For instance, `CompositeState A` and `SimpleState B` in MMG are transformed into `State A` and `State B` in MMT, respectively, where `State A` is the parent of `State B`. Although the structure is identical, there exist semantic differences between `SimpleState B` and `State B`, because `SimpleState B` cannot contain other elements, whilst `State B` can. For transitions, if we consider the same transformation, it is the opposite. For instance, `Transition T0` in MMG is transformed into `InitialTransition T0` in MMT and `Transition T7` in MMG is transformed into `HistoryTransition T7` in MMT. Alternatively, structural differences are prominent when creating new metaelements instead of transforming them (because of the lack of a corresponding source metaelement). This is the case of `TransitionBody`, `Port`, `Event`, `Method`, and `Parameter` metaelements in MMT. For instance, `Transition T1` in Fig. 4 has two children, `Trigger` and `ActionChain`, whilst `Transition T1` in Fig. 5 has only one child, `TransitionBody`, which contains the `PortEventTrigger` and `TransitionOperation`. Another interesting difference is that whilst in Fig. 4 `Transition T7` and `Deep History H` reside on the same level (i.e., they are siblings), in Fig. 5 `Deep History H` is a child of `Transition T7`. The reason for this is to allow the user to initialize `DeepHistory` when writing `Transition T7`, and not before initializing it, and then reference it in `Transition T7`.

```
statemachine SM {
    state Top {
        state A {
            entrypoint entry1 ;
            entrypoint entry2 ;
            exitpoint exit1 ;
            state B ;
            state C ;
            state D ;
            choice CH1 ;
            transition T1 : entry1 -> B on myPort . myEvent { "code" } ;
            transition T2 : B -> CH1 ; transition T3 : CH1 -> C when [ "x>=0" ] ;
            transition T4 : CH1 -> D when [ "x<0" ] ;
            historytransition T7 : entry2 -> history* ;
        } ;
        state E ;
        T0 : initial -> A.entry1 ;
        transition T6 : E -> A.entry2 on method ( parameter ) ;
        transition T5 : A.exit1 -> E ;
    } ;
} ;
```

(a) Xtext editor

(b) Exeed Editor

Fig. 5: Textual model

6 Related Work

The Action Language for Foundational UML (Alf) [17] is a textual language standardized by the Object Management Group (OMG) for representing UML models. Since its underlying semantics are indicated by a limited subset of UML, named Foundational UML (fUML), the Alf syntax is restricted within its bounds and does not support state-machines as they are not available in the fUML subset. tUML is a textual language for a limited subset of the standard UML metamodel targeted at real-time embedded systems that consists of class diagrams, composite structure diagrams, and state diagrams. The implementation of tUML has been carried out to have a very close proximity to the UML metamodel. Consideration has been given to propose tUML to OMG as an extension of Alf, being that the latter lacks support for state-machines [10]. There also exists a plethora of tools and modeling languages that support textual notations for UML models. Earl Grey [15] is a textual modeling language that supports the creation of UML class and state models. MetaUML [6] is a MetaPost library for creating UML diagrams using textual notations, and it supports class diagrams, package diagrams, component diagrams, use case diagrams, activity diagrams, and state diagrams. The textual notation is not only used to define the elements and their relationships but also their layout properties. PlantUML [13] is an open-source tool that supports the generation of both UML and non-UML diagrams from a textual language. Among the most important UML diagrams they support are sequence diagrams, class diagrams, activity diagrams, state diagrams, and more. Umple [12] is an open-source modeling tool that can be used to add UML abstractions to programming languages (i.e., Java, C++, PHP, and Ruby) and create UML class and state diagrams from a textual notation. The generated graphical view for class diagrams can be edited, while for state-machines,

[13] http://plantuml.com/guide

it is read-only. Textual, executable, and translatable UML (txtUML) [5] is an open-source project that supports the creation of models from a textual notation and generates the corresponding graphical visualization. TextUML Toolkit [14] is an open-source IDE that allows the creation of UML2 models from a textual notation. This toolkit is available on Cloudfier, as a plug-in for Eclipse IDE and as a standalone command-line tool.

There have been a handful of attempts at providing textual syntax for UML-RT and we have been involved in some of them. Calur [7] provides a textual syntax only for UML-RT's action language, not state-machines. Unlike our approach, both eTrice[15] and Papyrus-RT[16] provide a kind of all-or-nothing approach. They both provide syntax for both structure and behaviour, but the entire model is described as either textual or graphical, whereas in our approach the user can select only parts of the model to be represented textually. This allows the user to retain the ability to use existing RTist tooling for graphical modelling. Note also that our textual notation for UML-RT state-machines has been designed and implemented to maximise user experience of architects and engineers, as their throughput thanks to the possibility of blended modelling.

7 Outlook

In this work, we have reported on our work to provide a seamless blended graphical and textual modelling environment for UML-RT state-machines. Our proposed solution involves the provision of (i) a textual notation as complement to the existing graphical notation for UML-RT state-machines and (ii) ad hoc synchronization mechanisms between the metamodels underlying the two notations. The synchronization mechanisms have been designed as model-to-model transformations and implemented using the operational implementation of the QVT language in Eclipse. With regards to the limitations of this approach, we argue that the solution is language-agnostic (i.e., applicable to UML-RT state machines only). For any other language, the related editors and transformations have to be re-done from scratch. On a similar note, if the metamodels evolve, the model transformations would have to be manually updated.

For that reason, future work involves the definition of a mapping language that allows the definition of explicit mappings between arbitrary metamodels (MMs), and automatic generation of synchronization transformations via Higher Order Transformations (HOTs). The HOTs are transformations that take as input and/or generate as output other model transformations [20]. Given two MMs defined, a mapping model, conforming to the mapping language, would conceive the mapping rules for synchronizing models conforming to the two MMs. The mapping model together with the two MMs would be given in input to a set of HOTs that we are currently designing. The outputs of the HOTs are synchronization model transformations, as the ones defined manually in Operational QVT

[14] ttp://abstratt.github.io/textuml/
[15] https://www.eclipse.org/etrice/
[16] https://www.eclipse.org/papyrus-rt/

for the solution presented in this paper. The type of generated transformations (i.e., endogenous, exogenous, out-of-place, in-place) depends on the nature of the two MMs.

In fact, this architecture and the HOTs in it would entail multiple usage scenarios, as follows. In case the MMs are two entirely disjoint (but somehow connected/dependent) languages, the generated transformations provide synchronization across different languages (either same or different notations). In case the MMs represent two notations of the same language, the generated transformations provide synchronization across different notations of the language. In addition, in case the target MM represents an evolution of the source MM, the generated transformations provide co-evolution mechanisms for models conforming to MM.

This work is run in the context of an international consortium across 4 countries within the ITEA3 BUMBLE project[17]. In that context, we will run more extensive controlled experiments and industrial case-studies too. 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.

Acknowledgments

This work was supported by Vinnova through the ITEA3 BUMBLE project (rn. 18006). We would like to thank Ernesto Posse for his great support in technical discussions related to the UML-RT textual implementation in Papyrus-RT.

References

1. Addazi, L., Ciccozzi, F.: Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment. Journal of Systems and Software **175**, 110912 (2021)
2. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synthesis lectures on software engineering **3**(1), 1–207 (2017)
3. Cicchetti, A., Ciccozzi, F., Pierantonio, A.: Multi-view approaches for software and system modelling: a systematic literature review. Software & Systems Modeling (2019)
4. Ciccozzi, F., Tichy, M., Vangheluwe, H., Weyns, D.: Blended Modelling – What, Why and How. In: MPM4CPS workshop (September 2019), http://www.es.mdh.se/publications/5642-

[17] https://blended-modeling.github.io/

5. Dévai, G., Kovács, G.F., An, Á.: Textual, Executable, Translatable UML. In: OCL@ MoDELS. pp. 3–12. Citeseer (2014)
6. Gheorghies, O.: Metauml: Tutorial, reference and test suite (2005)
7. Hili, N., Posse, E., Dingel, J.: Calur: an action language for UML-RT. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018) (2018)
8. Hofmeister, C., Nord, R., Soni, D.: Applied software architecture. Addison-Wesley Professional (2000)
9. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Procs of ICSE. pp. 471–480. IEEE (2011)
10. Jouault, F., Delatour, J.: Towards Fixing Sketchy UML Models by Leveraging Textual Notations: Application to Real-Time Embedded Systems. In: OCL@ MoDELS. pp. 73–82 (2014)
11. Latifaj, M., Ciccozzi, F., Mohlin, M., Posse, E.: Towards automated support for blended modelling of uml-rt embedded software architectures. In: 15th European Conference on Software Architecture ECSA 2021, 13 Sep 2021, Virtual (originally Växjö), Sweden (2021)
12. Lethbridge, T.C., Abdelzad, V., Orabi, M.H., Orabi, A.H., Adesina, O.: Merging modeling and programming using Umple. In: International Symposium on Leveraging Applications of Formal Methods. pp. 187–197. Springer (2016)
13. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: A survey. IEEE Trans. on Soft. Eng. **39**(6), 869–891 (2012)
14. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2009)
15. Mazanec, M., Macek, O.: On General-purpose Textual Modeling Languages. In: Dateso. vol. 12, pp. 1–12. Citeseer (2012)
16. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic notes in theoretical computer science **152**, 125–142 (2006)
17. Object Management Group (OMG): Action Language for Foundational UML (Alf), Version 1.1. OMG Document Number formal/2017-07-04 (http://www.omg.org/spec/ALF/1.1) (2017)
18. Selic, B.: Real-time object-oriented modeling. IFAC Proceedings Volumes **29**(5), 1–6 (1996)
19. Selic, B.: The pragmatics of model-driven development. IEEE software **20**(5), 19–25 (2003)
20. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: European Conference on Model Driven Architecture-Foundations and Applications. pp. 18–33. Springer (2009)
21. Tiso, A., Reggio, G., Leotta, M.: Unit Testing of Model to Text Transformations. In: AMT 2014–Analysis of Model Transformations Workshop Proceedings. p. 14 (2014)