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

D4.1 Mapping rules for blended notations generation, bidirectional synchronization, and co-evolution

BUMBLE

Blended Modelling for Enhanced Software and Systems Engineering

Project Acronyms

<ACR>	<Acronyms>
BUMBLE	Blended Modelling 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
XLST	eXtensible Stylesheet Language Transformations
ETL	Epsilon Transformation Language
MML	Mapping modeling Language
ATL	Atlas Transformation Language
PSS	Portable test and Stimulus Standard
AMW	Atlas Model Weaver
HOT	Higher order transformation
MEO	Mapping Ecore-OWL
RDF	Resource Description Framework
DIML	Diagram Interchange Mapping Language
MOF	Meta-Object Facility

Table of contents

1. Introduction

In this deliverable we report on the activities carried out as part of tasks T4.1 and T4.2 in WP4, in particular for what concerns the definition, implementation, and validation via application to industrial use-cases of a *mapping language* in the context of the Eclipse Modeling Framework (EMF).

In this scope, a mapping language is a structured and formalised means for precisely describing mapping rules between two or more domain-specific modeling languages (DSMLs). In this context, a modeling language is intended to be defined in terms of a metamodel.

The definition of a mapping language is pivotal for multiple activities in BUMBLE. Explicit mapping rules enable us to link in a deterministic manner multiple DSMLs; in BUMBLE, the rules are a fundamental input to:

1. correctly generate editors from a DSML definition. In this case the DSML is mapped to one or more notation-specific DSMLs (see Section 3).
2. correctly synchronize models between two different DSMLs. In this case the DSMLs, which may represent two different notations (in terms of abstract and concrete syntaxes), are mapped to one another (see Section 4).
3. provide co-evolution mechanisms in terms of in-place model transformations for both generation (WP3) and synchronization purposes (WP4-5).

Once the mapping language is defined and implemented, its instances (mapping models) will be used as input to generation and synchronization transformations. More specifically, a mapping model represents the guiding principle driving the transformation to properly generate correct information from one model or to properly propagate changes across models. If defined at meta-metamodel level across multiple DSMLs, mapping models could even be used for driving co-evolution across DSMLs (and thereby notations).

The work and solutions described in this deliverable contribute to the project core requirements BC1, BC4, BC9 (as described in D2.2).

The remainder of the deliverable is structured as follows. In Section 2 we provide an overview of the state of the art in mapping modeling/description. In Section 3 we describe the actions and results in relation to explicit and implicit mapping of blended notations, with direct application to multiple industrial use-cases. In Section 4 we introduce a first version of our modeling language for flexible mapping and outline the next steps in Section 5.

2. State of the art on mapping modeling/description

Various mapping languages have been proposed in the literature to support different model management operations (e.g., model transformation, model migration, model integration). In the following, we present these contributions, and highlight their advantages and disadvantages with respect to our approach.

In [LHBJ06], authors propose a mapping metamodel based on the Eclipse Modeling Framework (EMF), that supports mapping specifications between two metamodels. Moreover, they contribute with the Atlas Model Weaver (AMW) tool, which simplifies mapping visualization, and enables the generation of transformation models conforming to Atlas Transformation Language (ATL) from the mapping model. In addition, AMW allows for the generation of a textual representation of the mapping model and validates

the conformity of the latter to its metamodel. However, the mapping metamodel only provides one-to-one, one-to-many, and many-to-one relationships, and restricts the specification of more complex mappings (e.g., target metamodel contains elements that do not have a correspondence with any of the elements of the source metamodel, thus, need to be created). Moreover, the AMW tool generates a read-only textual representation of the mapping model and is restricted to the generation of ATL model transformations. Ecore2Ecore1 is a plugin, distributed with EMF, that is originally implemented with the goal of supporting metamodel evolution and is widely used for such purpose. Nevertheless, being that it allows the definition of mappings between two metamodels, it can be used to define mapping models that could serve as input to higher-order transformations (HOTs) UI and generate language-specific model transformations. However, just like AMW, it does not provide a way for the user to specify more complex mappings and it does not restrict correspondences that are not valid bindings.

In [HBL+08], authors propose a textual mapping language called MEO (Mapping Ecore-OWL) that aims to enable the use of RDF resources as EMF objects and the serialization of EMF objects in RDF resources. The approach is based on EMF, and it defines correspondences between the domain model (conforming to Ecore), and the OWL ontology model. Moreover, it supports the generation of paired ATL transformations from HOTs, which automates the process of defining a bridge between EMF objects and RDF resources. However, the mapping metamodel is specific to OWL/RDF Resources.

In [KDSC14], authors propose a solution to enable the exchange of models between meta-modeling tools, thus, supporting interoperability, and avoiding vendor lock-in. This approach uses bridges at the metameta level to export metamodels from different environments into an intermediate one and uses binding components to create tree-structures of the metamodels. The main contribution of this approach is the graphical mapping language that is used to map between elements of the trees. Moreover, the mapping language is used as input to the code generator that outputs Epsilon Transformation Language (ETL) transformations. However, this mapping-based approach is focused on enabling the exchange of models between different meta-modeling tools, and as such, it specifies mapping correspondences between elements of meta metamodels, and generates model transformations for metamodels, while our approach aims to specify mapping correspondences between elements of metamodels and generate model transformations for models.

In [BBL08], authors propose Malan, a MApping LANguage that supports the definition of a schema mapping, between a source and target data schema. The mappings can be defined both textually and graphically (not simultaneously) using Papyrus. The graphical mapping is supported by the definition of a UML profile that contains a stereotype that defines the mapping concept for UML. In addition, these mappings are used as input to the Malan processor that generates a transformation. However, this approach manifests a few limitations. To begin with, the source and target schema should be expressed as UML class diagrams. Now even though UML is a widely used modeling language, and allows for the definition of the mapping concept using UML profiles, this restricts the use of Ecore metamodels. Moreover, the transformation program only generates XSLT stylesheets that convert XML documents into XML, HTML, or plain text documents.

In [ZKK07], authors propose a solution for the integration of heterogeneous modeling languages that incorporates both the definition of a mapping language and a rule definition language. Even though their objectives differ from ours, being that the mapping language is defined independently from the rule integration language, it can support other model manipulations (e.g., model transformations). However, all metamodels (i.e., source, target, mapping, and integration) conform to the ADONIS meta-metamodel, in order to avoid conflicts among metamodels. Unless we define mappings from the ADONIS meta-metamodel to Ecore meta-metamodel, we cannot use Ecore models and metamodels.

In [ALP06], authors propose DIML, a Diagram Interchange Mapping Language, that aims to define mappings between elements of MOF-based modeling languages (e.g., UML), and Diagram Interchange (DI) languages. DI is not restricted to UML, therefore, in a broad context, DIML can be used to create and transform visual diagrams for various MOF-based DSMLs. However, DIML is still a specific-purpose mapping language with a limited purpose of defining the concrete syntax of MOF-based modeling languages, and cannot be applied to more generic examples.

3. Explicit and implicit mapping of blended notations

In this section we describe the actions and results in relation to explicit and implicit mapping of blended notations, with direct application to multiple industrial use-cases (UC1, UC2, UC6, described in D2.1).

3.1. Explicit mappings

As part of a prototype for the generation and synchronization of blended editors in the EMF, we designed and implemented a mapping editor between input graphical and textual notations using Java and the WindowBuilder library. The inputs to the mapping editor are represented by:

- A DSML defined in terms of Ecore (in EMF)
- A library of symbols for mapping to the specific graphical notation
- A set of textual concepts, extracted from a given EBNF grammar, for mapping to the specific textual notation

Note that all these elements can be customized and replaced. More specifically, any DSML defined in Ecore can be given as input to the editor. The library of symbols can be customized by removing and adding symbols and the set of textual concepts can be any as long as it obeys to an EBNF grammar.

The mappings are saved in an ad-hoc XML format and it is used by another component of the prototype as input for the implementation of synchronization mechanisms between graphical and textual notations.

In terms of reusability and portability, the 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 XML files. Figure 1 displays the graphical and textual notations from the related XML files that are generated by the first component. Furthermore, it displays the repository of symbols to be associated with the graphical notation through the XML mapping file (containing addresses and IDs of symbols) and, therefore, other symbols specific to the domain can be added to the mapping editor with simplicity. Furthermore, 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 implement a corresponding EBNF grammar used for the synchronization process.

In Figure 1 we show the mapping between the Portable test and Stimulus Standard[1] (PSS) graphical and textual notations, as well as the association of graphical symbols to the PSS concepts. More specifically, the symbol with *Id =1* and *Name = Action* is associated with the graphical action concept.

[1]The Portable Test and Stimulus Standard (PSS) defines a specification to create a single representation of stimulus and test scenarios usable by a variety of users across many levels of integration under different configurations. This representation facilitates the generation of diverse implementations of a scenario that run on a variety of execution platforms, including, but not necessarily limited to, simulation, emulation, FPGA prototyping, and post-silicon. With this standard, users can specify a set of behaviors once and observe consistent behavior across multiple implementations.

The assigned symbol will be available in the blended modeling editor for the modeling of the graphical action. On the other side, the textual syntax for action is specified as: *action name {}*. Subsequently, this mapping between graphical and textual action can be added to the queue (*grid*). Similarly, the mapping between graphical and textual notations for other PSS concepts, like buffer, objects, etc. is performed and saved in an XML file as well.

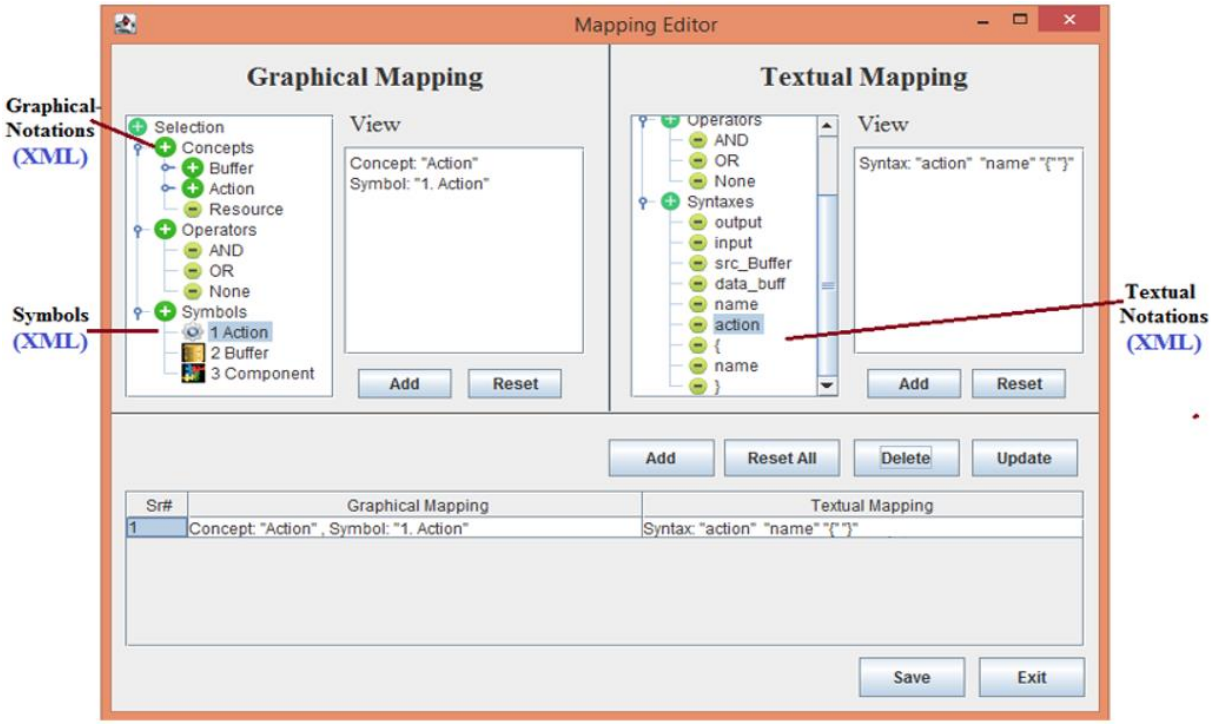

Figure 1 - Mapping editor

3.2. Implicit mappings

A different approach to the explicit mapping editor approach in Section 3.1 is represented by the implicit mapping rules that we encoded in a dedicated prototype for the synchronization of blended editors for UML-RT state-machines. In this case, since both notations for UML-RT are defined a-priori and not intended to be customized by the user, the mapping rules are embedded in the model transformations in charge of the synchronization between graphical and textual notations.

In Listing 1 we depict an excerpt of the transformation in charge of propagating changes from graphical to textual notation. Note that these transformations were implemented using ETL in EMF.

```
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 1 - Mappings implicitly defined in an ETL model transformation

As we can see in this specific rule, from an element of type *Trigger* in graphical UML-RT state-machine, the transformation generates a textual element of type *Trigger* and a set of other elements composing it. Although this is a transformation rule, it actually materializes a precise mapping rule between *Trigger* in the graphical notation and *Trigger* in the textual notation. While this solution may be preferable in the specific case where notations are not supposed to change or when the user is not intended to customize mappings, it is not flexible enough for our final purpose being a flexible mapping modeling solution that can be used for: generating blended editors, co-evolving them, and generate synchronization transformations. Nevertheless, both explicit and implicit mappings shown in this section laid the ground for the mapping modeling language described in the next section.

4. A flexible mapping modeling language

Given the experiences with mapping described in the previous section and in conjunction with the project requirements (core requirements BC1, BC4, BC9, as described in D2.2), we created a *mapping modeling language (MML)* defined as a metamodel, i.e. the most suitable form for our purposes. We investigated different possible technological choices, more specifically Xtext and JetBrains MPS for a textual mapping modeling language and Ecore for a tree-based mapping modeling language. Since the core usages of MML would be to (i) support the definition of explicit mapping rules between DSMLs in a user-friendly manner and (ii) provide a transformation-friendly input to generation of editors and synchronization mechanisms, we opted for an implementation in Ecore. The additional advantage is that a textual notation for it could be defined in Xtext exploiting the very same BUMBLE features.

In Figure 2 we depict the current version of the MML defined and implemented in Ecore. A *MappingModel* is a tuple *<name, input, output, Rules*>*, where *name* is a unique model name, *input* is the source *EPackage* (representing the root element of the source metamodel), *output* is the target *EPackage* (representing the root element of the target metamodel), and finally *Rules**, which is a possibly empty set of elements of type *MappingRule*.

A *MappingRule* is in turn a tuple *<name, action, input, output, ChildRules*>*, where *name* is a unique mapping rule name, *action* represents the type of mapping (i.e., transform, add, remove). *input* is a possibly empty set of elements of type *EObject* (representing the source model element(s) to be mapped from), *output* is a possibly empty set of elements of type *EObject* (representing the target model

element(s) to map to), and finally *ChildRules**, which is a possibly empty set of elements of type *MappingRule* representing sub-rules of the current mapping rule. Sub-rules are intended to be helper rules to the parent mapping rule.

Action is an enumeration with three mutually exclusive literals being:

- *transform*, used when 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;
- *add*, used when a non-empty set of output elements are added to the target model;
- *remove*, used when a non-empty set of input elements are removed from the target model. While this action may seem redundant, since it would not lead to any action in case the two models are conforming to different metamodels, it is a core feature in case the mapping rule is used for an in-place transformation, where the target model is an "updated" version of the source model (conforming to the same metamodel).

We have defined specific constraints in MML to enforce the correct type of *Action* to be selected depending on the cardinalities of *input* and *output* in *MappingRule*. More specifically:

- *transform*, if *input* > 0 && *output* > 0
- *add*, if *input* == 0 && *output* > 0
- *remove*, if *input* > 0 && *output* == 0

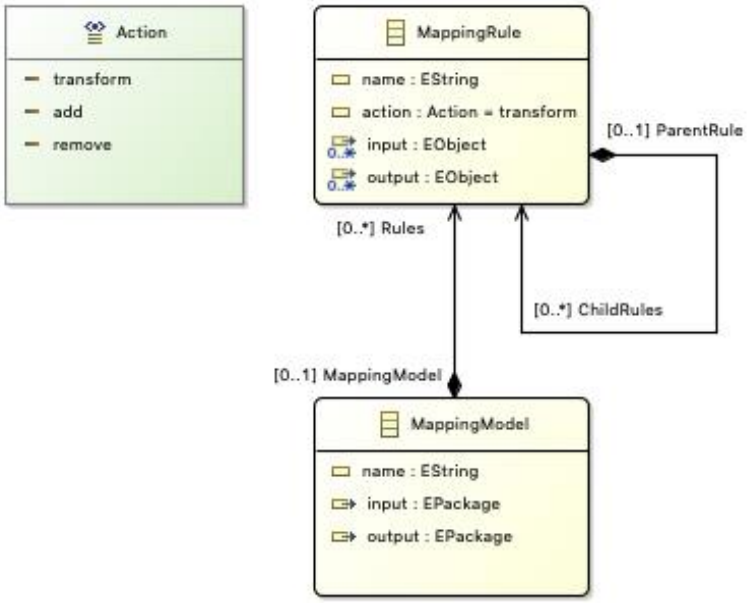

Figure 2 - Mapping metamodel in Ecore

5. Next steps

We are currently exploiting the MML to design the higher-order transformations in charge of generating synchronization transformations (see D4.2). MML will be employed in most automation aspects between notations in EMF. We will continue validating MML by applying it to more use cases as well as by formalizing mapping rules in MML between DSMLs of various nature.

References

[ALP06] Marcus Alanen, Torbjörn Lundkvist, and Ivan Porres. A mapping language from models to di diagrams. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, Model Driven Engineering Languages and Systems, pages 454–468, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[BBL08] Arnaud Blouin, Olivier Beaudoux, and Stephane Loiseau. Malan: A mapping language for the data manipulation. In Proceedings of the eighth ACM Symposium on Document Engineering, pages 66–75, 2008.

[HBL+08] Guillaume Hillairet, Frederic Bertrand, Jean Yves Lafaye, et al. Bridging emf applications and rdf data sources. In Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE, 2008.

[KDSC14] Heiko Kern, Vladimir Dimitrieski, Fred Stefan, and Milan Celikovic. Mapping-based exchange of models between meta-modeling tools. 10 2014.

[LHBJ06] Denivaldo Lopes, Slimane Hammoudi, Jean B´ezivin, and Fr´ed´eric Jouault. Mapping specification in mda: From theory to practice. In Interoperability of enterprise software and applications, pages 253–264. Springer, 2006.

[ZKK07] Srdjan Zivkovic, H Kuhn, and Dimitris Karagiannis. Facilitate modeling using method integration: An approach using mappings and integration rules. 2007.