Automatically bridging UML profiles to MOF metamodels

Ivano Malavolta
Gran Sasso Science Institute
L'Aquila (Italy)
ivano.malavolta@gssi.infn.it

Henry Muccini
University of L'Aquila
Department of Information Engineering, Computer Science and Mathematics
henry.muccini@univaq.it

Marco Sebastiani
University of L'Aquila
Department of Information Engineering, Computer Science and Mathematics
marco.sebastiani@univaq.it

***Abstract*—In Model-Driven Engineering, UML profiles and MOF-based Domain Specific Modeling Languages (DSMLs) are the most used approaches for describing domain specific applications. The choice of the right approach depends on several aspects, such as tool support, expressivity, complexity of models, company policies. In general, profiled UML models are very much used since they are intuitive for designers and model editors already exist, however they are intrinsically complex for model manipulation (e.g., transformation, analysis); conversely, domain specific models are more concise and easy to be manipulated, but they require an initial effort in terms of designers training and model editors development.**

In this paper we propose an approach that allows getting the best of the two worlds: on one side designers can use UML profiles familiar to them, on the other side DSML models (automatically generated from profiled UML models) enable a better model manipulation. Our approach is based on an automatic bridge between UML profiles and MOF metamodels (which are the main artifacts of MOF-based DSMLs). The bridge is transparent to the user since it autonomously operates both on UML profiles and all the involved models. The bridge is realized through model transformation techniques in the Eclipse platform. In this paper we show its application on a case study based on SysML.

I. Introduction

Domain specific applications require notations enabling to capture specificities of the domain at the right level of abstraction. Domain Specific Languages (DSLs [1]) have been introduced for this purpose, facilitating the specification of domain information. In Model Driven Engineering (MDE), UML profiling [2] and Domain Specific Modeling [3] are the most used techniques for defining a DSL.

The *UML profiling* technique consists in extending the UML modeling language with concepts coming from the domain of interest [2]. A UML profile is the definition of such an extension; the application of a profile to a UML model allows the designer to tailor it with domain-specific information, by specializing existing UML meta classes. Among the most popular UML profiles, we can cite SysML [4] for modeling systems engineering applications, and MARTE [5] for real-time and embedded applications.

The *Domain Specific Modeling* technique consists in defining and using languages dedicated to a specific domain. Those languages are called Domain Specific Modeling Languages (DSMLs) and their concepts are usually formalized via MOF metamodels [6]. Each language can have its textual or graphical representation, and tool support can be provided either by generic (metamodeling tools) or ad-hoc modeling environments [3]. Examples of DSML are AADL to model embedded real-time systems[1], and BPMN for business processes[2].

Currently, in many MDE projects both UML profiles and DSMLs are extensively used and each technique has its own strengths and weaknesses. For example, in a recent study about MDE approaches for modeling Wireless Sensor Networks [7] it emerged that 75% of those approaches are based on DSMLs, whereas the remaining 25% are based on generic modeling languages, mainly UML profiles and Simulink. The choice of the right approach depends on several aspects, such as tool support, language expressiveness, models complexity, and company policies [8]. What can be noticed today is that UML profiles became extremely popular, and used as an extensive tool for describing domain specific applications. However, while of more immediate use for practitioners (since a profiled model is essentially an annotated UML model that can be created using existing UML tools), developing a tool to manipulate profiled UML models is a complex activity; this is a consequence of the inherent complexity of the involved models [8][9]. Such a complexity is mainly due to (i) the intricacy of the UML language, where many strictly related concepts are scattered across its metamodel (the so called "metamuddle"[9]) , and (ii) because the application of a profile imposes additional constraints in the way models are manipulated [10]. It is important to note that the complexity of manipulating profiled UML models is not strictly related to the system of interest, but relies on the profiling technique itself.

Goal of this work is to support those kind of projects in which the system is modeled using UML profiles, and there is a strong need of automatic model manipulation (e.g., to perform some kind of analysis [11]). This paper proposes an **automatic bridge** between UML profiles and MOF metamodels; such a bridge alleviates the accidental difficulty in manipulating profiled UML models, without forcing designers to abandon UML-based notations. By using the bridge, on one side designers describe the system using UML profiles, and on the other side tools operate on *automatically generated* MOF-based models. The bridge is totally transparent to designers since it operates at both metamodeling and modeling levels of

[1]SAE AADL website: http://www.aadl.info
[2]BPMN information website: http://www.bpmn.org/

abstraction. At the metamodeling level, the bridge generates a MOF metamodel MM_x representing the concepts of the UML profile. At the modeling level, it transforms each UML model into a model conforming to MM_x, and vice versa. Since it is recurrent that designers use only a small subset of UML diagrams [9], it is fundamental to generate a small and concise MOF metamodel containing only such elements, while discarding any other information; for this purpose, a **slicing mechanism** is presented. The proposed bridge is implemented as an Eclipse plugin and in this paper we apply it on a case study based on OMG Systems Modeling Language (SysML). It is important to clarify that while this work might not provide some novel theoretical results, it provides a concrete solution to a problem that exists in practice.

The remainder of this paper is organized as follows. In Section II we discuss the main motivations for our work. We describe our automatic bridge in Section III and the mechanism for slicing the obtained MOF metamodel in Section IV. Then, Section V gives some implementation details, and Section VI describes the application of the bridge on a case study. Related work is discussed in Section VII. Finally, in Section VIII we discuss future work directions and draw the conclusions.

II. Motivation

Many state-of-the-art model transformation and analysis engines (like ATL[3]) need to be tailored or extended to support the transformation of profiled UML models. These problems are caused by a set of constraints which make the manipulation of profiled UML models difficult for both model transformations users and transformation engines developers. For example, specific mechanisms to apply (and un-apply) either UML profiles or stereotypes must be implemented in the model transformation engine itself (or at least as an ad-hoc extension for it), or the model transformation language must expose specific constructs for accessing tagged values associated to a UML model element. By automating the transition from profiled UML models to MOF-based models and vice versa, the above mentioned constraints are avoided, thus allowing both users and developers of transformation engines to assume to work on MOF-based models only. Also, the proposed bridge (along with its slicing algorithm) helps transformation and analysis tools developers to reason on smaller and more concise MOF metamodels.

Moreover, tools working on meta-concepts (i.e., metaclasses, attributes, etc.) that can be represented either as metamodels or UML profiles are emerging. For example, in **D**U**AL**Ly, an automated framework for architectural languages interoperability [12], a set of higher-order transformations (HOTs) is executed on an initial set of meta-concepts; at any time a transformation needs to access a meta-concept, it has to analyse whether a meta-concept is represented as a MOF element (e.g., an attribute) or as a UML profile element (e.g., a tagged value), making the computation more complex.The proposed bridge enables designers to simplify their tools: by using the proposed bridge as a pre-processing tool, UML profiles can be transformed into MOF metamodels, and tools may simply elaborate on MOF metamodels.

[3]ATL project website: http://www.eclipse.org/atl

III. The Automatic Bridge

Figure 1 provides a high-level view on how the proposed bridge works. The starting point of the whole bridging mechanism is a UML profile and models conforming to it (see Figure 1). The profile and its models can be developed using standard UML modeling tools. Then, all the other modeling artifacts are automatically generated:

1) the MOF MM metamodel containing all the concepts corresponding to the elements of the UML profile,
2) a set of model-to-model transformations enabling the transformation of profiled UML models into models conforming to the MOF metamodel, and vice versa.

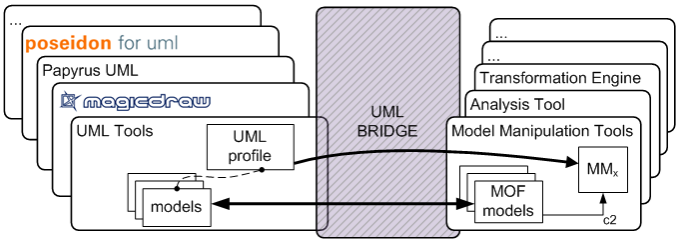


Fig. 1: High-level view of the proposed bridge

Modeling artifacts involved in the bridging procedure lie at different levels of abstraction (i.e., metamodeling and modeling, in the MDE modeling stack). In order to keep the bridging procedure well defined and manageable, we decided to keep this distinction in our bridge by decomposing it into two main phases:

- **phase 1**: done at the metamodeling level, it consists in the generation of MM from the UML profile.
- **phase 2**: it works at the modeling level, and consists in the generation of model-to-model transformations between UML and MM.

It is important to note that phases 1 and 2 are executed only once for each UML profile p_x, then the generated transformations are re-used every time a UML model profiled with p_x has to be bridged. Moreover, the proposed bridge is *completely automatic*, i.e., when a UML profile is defined, the above mentioned bridging phases do not require any additional effort to the user. As a matter of fact, designers work on the UML side, while tool developers work on standard MOF metamodels, thus enabling a clear separation of concerns. Next sections describe in details the two phases of the bridge.

A. Phase 1: The bridge at the metamodeling level

At the metamodeling level, the bridge takes an initial UML profile and generates a corresponding MOF metamodel. As shown in Figure 2, a model-to-model transformation called *UMLprofile2MOF* is used for this purpose. This transformation generates a MOF metamodel (MM_x in figure) starting from (i) the definition of the UML profile and (ii) the UML metamodel. The latter input is needed since the transformation has to access UML metaclasses referenced by the various stereotypes of the profile.

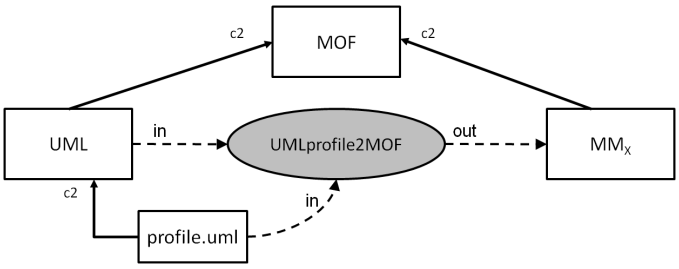


Fig. 2: The bridge at the metamodeling level

The *UMLprofile2MOF* transformation contains a transformation rule for each element that can appear in the definition of a UML profile:

- **Stereotype2Class**. Each UML stereotype is mapped into a MOF metaclass (e.g., *XComponent* in Figure 3). Tagged values of the source stereotype are separately managed by the *Property2Feature* rule (described below). The rule also checks whether the source UML stereotype specializes some other elements, and recreates the generalization hierarchies in the target MOF metamodel, accordingly. A special reasoning is applied to transform the relationship between a stereotype and the UML metaclasses it extends. According to the UML superstructure [2], *"the MOF construct equivalent to an extension is an aggregation from the extended metaclass to the extension stereotype"*. So, this rule transforms each UML extension into a MOF containment reference with cardinality *0..1*.
- **Class2Class**, each UML class is mapped to a MOF metaclass. Properties of the source UML class are managed by the *Property2Feature* rule and, similarly to *Stereotype2Class*, contains a mechanism for managing its generalization hierarchy.
- **Profile2Package**, each UML profile is mapped to a MOF package (e.g.,the *XProfile* in figure). Specific bindings populate the newly generated package with the correct metaclasses (generated either by the *Class2Class* or the *Stereotype2Class* rules) in order to recreate the same hierarchy of containments.
- **Package2Package**, each UML package is mapped to a MOF package. It is populated in a similar way as in the *Profile2package* rule.
- **Property2Feature**, each UML property (like the *executionTime* in figure) is mapped to a MOF attribute or reference depending on their type; more specifically, if the type of the source property is either a data type, an enumeration or a primitive type, then a MOF attribute is generated, otherwise the rule generates a MOF reference.
- **DataType2Datatype**, each UML data type is mapped to a MOF data type.
- **Enumeration2Enumeration**, each UML enumeration (*type* in figure) is mapped into a MOF enumeration containing the same literals.

UMLprofile2MOF also contains a set of auxiliary rules and bindings (e.g., the one setting the name of each MOF metaelement with the name of the corresponding source UML element, etc.), not described in this paper for sake of simplicity. Furthermore, *UMLprofile2MOF* creates an ad-hoc package in MM_x where it copies all the UML metaclasses in there. In this way, the generated MOF metamodel is self-contained, and does not depend on the UML metamodel itself.

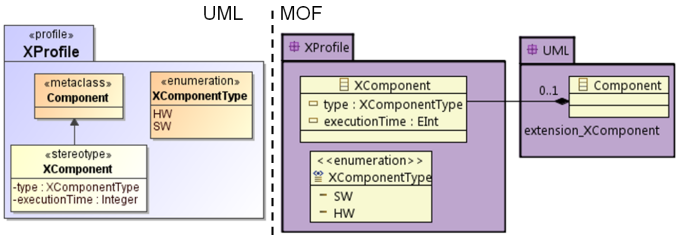


Fig. 3: Example of generated MOF metamodel

B. Phase 2: The bridge at the modeling level

At the modeling level, the proposed bridge automatically generates model-to-model transformations between UML profiles and MOF metamodels. More specifically, Figure 4 shows the two model transformations generated by our bridge:

- $UML2MM_x$ returns a model conforming to MM_x, starting from a UML model conforming to the $profile.uml$ profile;
- MM_x2UML performs the opposite task: it takes as input a model conforming to MM_x and produces UML models profiled according to $profile.uml$.

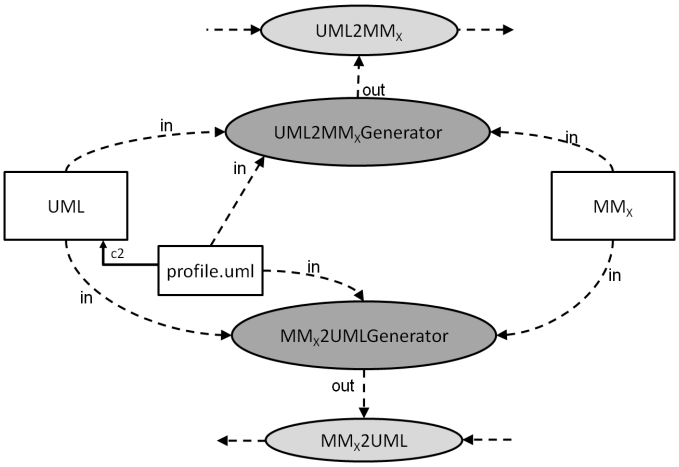


Fig. 4: The bridge at the modeling level

$UML2MM_x$ has the following high-level logical structure: (i) a set of rules transform each standard UML element into an instance of its corresponding metaclass in MM_x; (ii) other rules transform each UML stereotype into an instance of the corresponding metaclass in MM_x; and (iii) for each rule, ad-hoc imperative code is generated in order to automatically manage the application of stereotypes. MM_x2UML works in

the other way round: (i) firstly it applies the UML profile to the target UML model; (ii) then, it transforms each instance of metaclasses in MM_x into an instance of the corresponding UML element; (iii) stereotypes are applied to previously generated elements. Specific rules manage the order in which profiles and stereotypes are applied, and how tagged values are accessed in the models. Figure 5 shows an example of models produced by the bridge.

Such transformations are automatically generated by means of the execution of two Higher-Order Transformations (i.e., transformations taking other transformations as input or producing other transformations as output): $UML2MM_xGenerator$ and $MM_x2UMLGenerator$. It is important to note that (i) even if higher-order transformations are quite complex (they are composed of 285 lines of ATL code), they are independent form the input UML profile, thus we developed them only once and they are valid for each future application of the proposed approach, and (i) higher-order transformations make the bridge generic (and so, totally automatic), i.e., it does not depend either on source UML profiles or on the generated MOF metamodels.

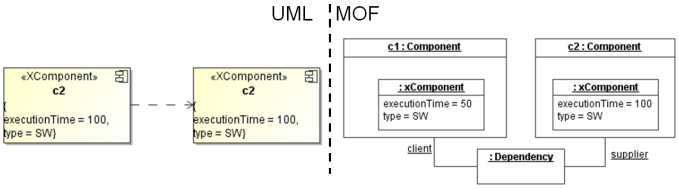


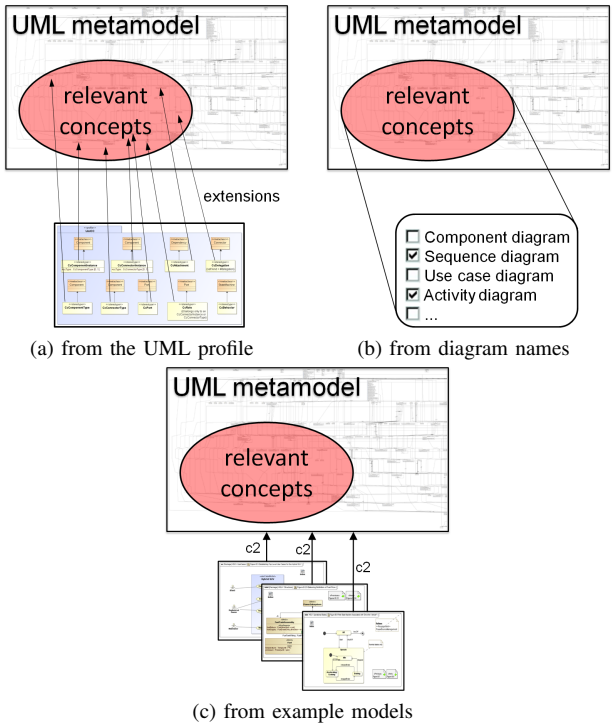
Fig. 5: Example of bridged models

It is important to note that we generate two model transformations, rather than a single bidirectional one, because we chose ATL as technological enabler of our bridge (ATL is a unidirectional model transformation language). We chose ATL because it is a well-accepted and known model transformation language and because of the stability of its underlying model transformation engine. Nevertheless, as future work we will investigate on the generation of a single bidirectional model transformation (instead of two unidirectional ones), e.g. using the JTL model transformation language [13].

IV. Slicing the obtained metamodel

The default behavior of the bridge is to consider the whole UML metamodel and to keep its concepts also in the target MOF metamodel. So, while ensuring loss-less transformation (and this is our primary goal while designing the bridge), our approach moves the complexity of the UML metamodel (currently containing 246 classes and 583 properties) into the target MOF metamodel. In order to avoid such a complexity, we couple the bridge with a **slicing mechanism** that reduces the generated metaclasses to a *subset of relevant UML concepts*.

The subset of relevant UML concepts can be considered as the set of metaclasses that are expected to be instantiated in the context of a specific project. A typical example is when designers assume that they will only use, e.g., use case diagrams. In this scenario, relevant UML concepts are only those pertaining to use case diagrams (e.g., actor, use case, association) and there is no need to keep modeling concepts for other UML diagrams. The practical value of the slicing mechanism resides in the production of a very much simpler MOF metamodel as output of our approach. In this section we describe the slicing algorithm, then we describe how relevant UML concepts can be represented as annotation models, and then we discuss how the slicing mechanism can be executed on both metamodels and transformations.



(a) from the UML profile (b) from diagram names

(c) from example models

Fig. 6: Mechanisms for defining the set of UML relevant concepts

Slicing algorithm. Our slicing mechanism extends a generic slicing algorithm for MOF metamodels presented in [14]. Intuitively, it considers the set of relevant metaclasses and then navigates both their references and generalizations in order to return a self-contained subset of the initial metamodel. More formally, let MM be a metamodel and let SC be a subset of relevant elements in MM; $slice$ is defined as follows:

$$slice(SC) = SC \cup \bigcup_{c \in SC} slice(neighbour(c))$$

where $neighbour(c)$ is the set of all super metaclasses of c, of all metaclasses referred (both with association and aggregation) by c, and of all types of attributes in c. It is important to note that even though $slice$ is defined as a set of metaclasses, since each metaclass contains also references to other metaclasses, the final result is a subset of the metamodel MM with both metaclasses and their meta-relations. In this work we modified the original slicing algorithm in two ways: (i) we implemented it as a reusable library for model-to-model transformations (previously it was embedded into the tool itself), and (ii) we introduced specific helpers to manage UML-specific issues (e.g., every UML model must contain a root element called `Model`, so we added a specific helper that ensures that the slicing algorithm does not leave out this element from the target MOF metamodel).

Annotation models. The set of relevant metaclasses is represented as an *annotation model* [4] linked to the target MOF metamodel. The annotation model contains a link to each relevant metaclass coming from UML: the slicing mechanism uses those links to identify the initial set of metaclasses that will not be sliced out by the slicing procedure. Even though the effort for creating the annotation model may be put only once for each specific project, manually creating such an annotation model can be both a heavy and error-prone task, as the designer has to manually annotate each UML metaclass that is considered as relevant. As a solution to this potential issue, the current version of the bridge provides three mechanisms to (semi) automatically obtain the annotation model that will drive the slicing procedure:

1) the annotation model is automatically generated from a UML profile (Figure 6a); intuitively, it creates a link in the annotation model for each UML metaclass extended by at least a stereotype in the profile.
2) a GUI allows designers to specify only the types of UML diagram they will use (e.g., activity, sequence diagrams), and the corresponding annotation model is generated (Figure 6b); more specifically, designers provide the names of the UML diagrams, and this mechanism deduces the set of metaclasses belonging to those diagrams, then a link for each of those metaclasses is created in the annotation model.
3) designers provide an example of a UML model, and then the annotation model is automatically generated from it (Figure 6c). Basically, this mechanism checks which metaclasses are instantiated in the example model and creates a link in the annotation model for each of them.

These mechanisms are implemented as model transformations. Each mechanism has a different level of automation and requires different input artifacts. Of course, if full control over the generated metamodel is needed, designers can manually define the annotation model by means of a dedicated graphical editor. Furthermore, those mechanisms can be executed incrementally; for example a metamodel could be sliced by means of the first mechanism, and then the resulting metamodel could be sliced multiple times via the third mechanism with different example models: the resulting metamodel will be more concise and accurate.

Slicing metamodels. As can be seen in Figure 7a, the slicing algorithm is used at the M2 level by a model-to-model transformation called *MMslicer*. It takes as input a MOF metamodel MM_x and an annotation model am_x, and generates a new metamodel MM_{sliced}. The resulting metamodel is self-contained (i.e., it does not contain any reference to external metamodels) and contains a subset of MM_x according to the metaclasses referenced in am_x.

Slicing transformations. Once the target MOF metamodel has been sliced, the previously generated model transformations of the bridge may refer to missing metaclasses in the metamodel. For example, if all the concepts related to state machines have been sliced out, the generated $UML2MM_x$ transformation still contains transformation rules for transforming states, transitions, etc.; so, it is not aligned with the newly sliced metamodel and its execution will be erroneous. Our approach provides a mechanism for adapting also the $UML2MM_x$ and MM_x2UML transformations to the newly sliced metamodel. We are aware that in literature there are generic approaches managing the coupled evolution of metamodels and model transformations [16]; however, since in our approach elements can be only deleted from metamodels (neither added nor updated), and since we need a fully automatic mechanism, we developed our minimalistic solution to automatically adapt model transformations to sliced metamodels. Our solution takes inspiration from the EMFMigrate project[5] and is based on an higher-order transformation: $Tslicer$.

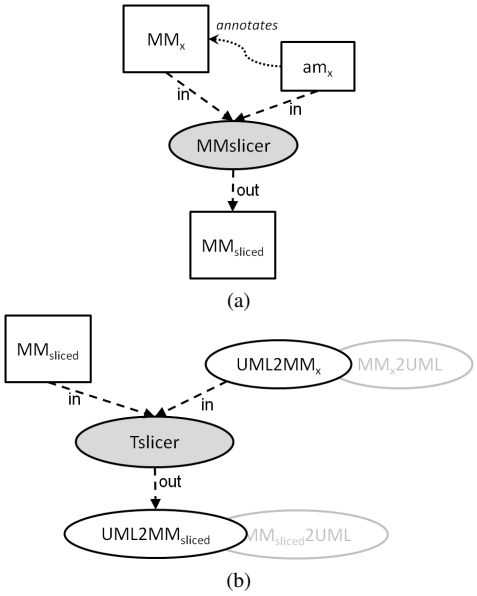


Fig. 7: Slicing the MOF metamodel (a) and the generated transformations (b)

Figure 7b gives an idea of how the *Tslicer* transformation works. It takes as input (i) the sliced metamodel (MM_{sliced} in figure) and (ii) the model transformation to be adapted ($UML2MM_x$ in figure). $Tslicer$ traverses all the instructions of the input transformation (e.g., transformation rules, assignments) and deletes those instructions referring to meta-elements not contained into the MM_{sliced} metamodel. $Tslicer$ can also be applied on the transformation in the other direction (i.e., MM_x2UML); this ensures that using the slicing mechanism does not affect the bridge in terms of automation and bidirectionality. In conclusion, it is important to note that the whole slicing mechanism (both on metamodels and transformations) acts as a post-processing activity of the artifacts generated by the bridge described in Section III; this means that, according to project-specific needs, the proposed bridge and slicing mechanism can be used independently.

V. Implementation of the bridge

We implemented the proposed bridge as a plugin of the Eclipse platform. In our approach, models and metamodels

[4] An annotation model is a model containing auxiliary information about another model (the annotated model)[15]

[5] EMFMigrate project website: http://www.emfmigrate.org

are represented by means of the EMF framework[6]. It includes an implementation of the MOF meta-metamodel called Ecore, and runtime support for models including, among all, persistence support by serializing models in the OMG XML Metadata Interchange (XMI). UML profiles are defined using UML2, the implementation of the UML metamodel for the Eclipse platform[7]. This provides the bridge full compliance with OMG standards (specifically UML 2.0 and MDA) and interoperability with other UML modeling tools Thanks to this choice, designers can graphically design UML profiles with any UML modeling tool, and directly import them into the Eclipse environment. The same rationale holds for UML models.

Both model transformations and annotation models are based on the Atlas Model Management Architecture (AMMA) [17]. More specifically, the model transformations constituting the bridge are specified using the Atlas Transformation Language (ATL) , a hybrid model transformation language with both declarative and imperative constructs (Listing 1 shows an excerpt of ATL code). The use of ATL allows designers to execute generated transformations (e.g., $UML2MM_{[sliced]}$ in Figure 7) on their models: via ANT scripts, manually as a standard ATL transformation, or directly from Java code (this is particularly useful if the proposed bridge must be embedded in already existing tools). The technology used for representing and manipulating annotation models is the Atlas Model Weaver (AMW) . It allows the graphical definition of correspondences among models and links among model elements. Such links are captured by weaving models conforming to an extensible weaving metamodel [15]. The AMMA platform has been selected since it best fits the requirements required to implement the bridge: (i) it provides a flexible model transformation engine, (ii) it supports the concept of transformation model, thus enabling the development of higher-order transformations, and (iii) it is integrated into Eclipse and its modeling facilities. The current version of the bridge (along with its source code) is available http://www.di.univaq.it/malavolta/umlbridge/.

VI. Case Study

In this section we show the application of the proposed UML bridge to a non-trivial case study based on the SysML profile. SysML is a general-purpose modeling language for systems engineering applications [4], it has been proposed by the OMG group and supports the specification of hardware, software, processes, and facilities of a system. The objective of this case study is to present how each aspect of the proposed approach works in practice on SysML models. As shown in Figure 8, we organized the case study as a seven-steps process:

1) transformation of the SysML profile into a MOF metamodel called MM_{sysml};
2) automatic generation of the model transformation that creates MOF-based models from SysML-profiled models;
3) creation of an annotation model am_{sysml} to slice MM_{sysml};
4) execution of *MMslicer* and *Tslicer* according to the am_{sysml};
5) definition of a SysML-profiled UML model representing a Hybrid gas/electric powered Sport Utility Vehicle (HSUV) ($HSUV.uml$ in figure);
6) transformation of $HSUV.uml$ into its MOF-based counterpart ($HSUV.xmi$);
7) development of a simple manipulation tool that works on $HSUV.xmi$.

Steps 1-4 are executed only once, then the generated ATL transformation can be re-used every time a SysML model has to be bridged.

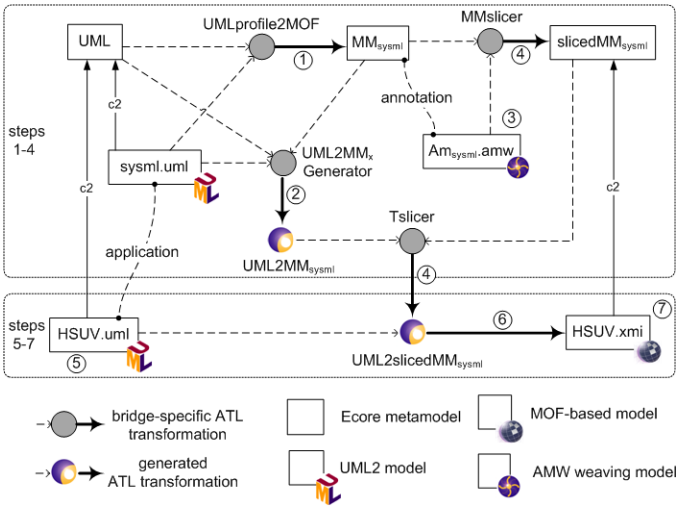


Fig. 8: Overview of the HSUV case study

Step 1. We firstly consider the SysML profile definition and then we transform it into the corresponding MOF metamodel by means of the $UMLprofile2MOF$ transformation (it is described in Section III-A). The resulting MM_{sysml} metamodel is composed of 322 metaclasses (246 from the UML metamodel), 122 attributes and 665 references (either associations or aggregations).

Step 2. In this step we execute the HOT called $UML2MM_xGenerator$ in order to obtain the model transformation that takes as input SysML-profiled models and returns MOF-based models. The generated transformation ($UML2MM_{sysml}$ in figure) is composed of 292 transformation rules, 3625 feature bindings, 10 helpers; the total size of the transformation is 5680 lines of ATL code.

Step 3. In order to slice the obtained MM_{sysml} metamodel and the $UML2MM_{sysml}$ transformation, we (automatically) generate an annotation model by means of the second generation mechanism (see Section IV) by assuming that only UML class diagrams and component diagrams are used. The generated annotation model (am_{sysml} in figure) contains 37 links to UML metaclasses like Class, Dependency, Interface and so on. It is important to note that am_{sysml} contains links to only concrete metaclasses in the lower part of the generalization hierarchy, the complete generalization hierarchy is automatically computed by the slicing algorithm presented in Section IV.

[6]EMF project website: http://www.eclipse.org/modeling/emf/
[7]UML2 project Web site: http://www.eclipse.org/uml2/.

Step 4. The newly created annotation model drives the execution of *MMslicer* and *Tslicer*. They adapt MM_{sysml} and $UML2MM_{sysml}$ by leaving out those concepts that do not belong either to class or component diagrams. The newly adapted metamodel ($slicedMM_{sysml}$) contains 171 metaclasses, 80 attributes and 460 references. The size of the adapted transformation ($UML2slicedMM_{sysml}$) is 2690 lines of ATL code; it contains 139 transformation rules, 1721 feature bindings and 10 helpers.

Step 5. In order to impartially test the bridge, in this case study we consider the example model provided in the SysML official specification. This model represents a HSUV by focussing on its requirements, performance, structure, and behavior. Figure 9a shows a fragment of SysML diagram in which low-level requirements (e.g., *RegenerativeBraking*) are derived from system requirements (e.g., *Braking*).

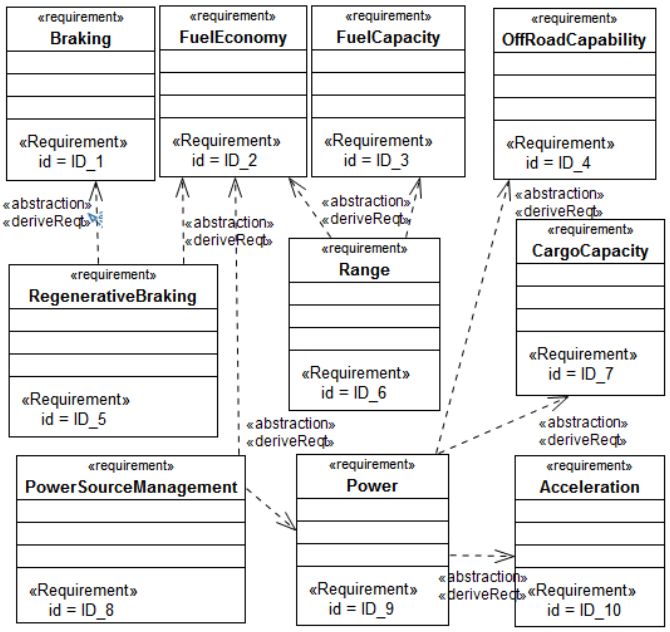
Step 6. At this point we can execute the $UML2slicedMM_{sysml}$ transformation to obtain a MOF-based representation of the HSUV model. Figure 9b shows a fragment of the obtained model; it contains the same elements of the source UML model, but (i) each stereotype application (e.g., *Requirement* was applied to the *Brake* UML class) has been transformed into an instance of the metaclass corresponding to the applied stereotype, and (ii) tagged values (e.g., *text* or *id* of *Requirement*) have been translated to standard MOF properties. These two differences will facilitate the manipulation of the HSUV model because stereotypes and tagged values can be accessed as standard MOF elements.

Step 7. In this step we provide a simple manipulation tool that operates on bridged MOF models. Technical details and accuracy are not in the focus of this step of the case study, its main goal is to show as clearly as possible how manipulation tools may benefit from our UML bridge. The manipulation tool is implemented as an ATL transformation checking if a SysML model follows a simple naming convention (see Listing 1). It takes as input a SysML model and checks if the identifier of each `Requirement` starts with "ID_" (line 3); the transformation generates a `Problem` for each requirement that does not follow the convention (lines 10 to 14).

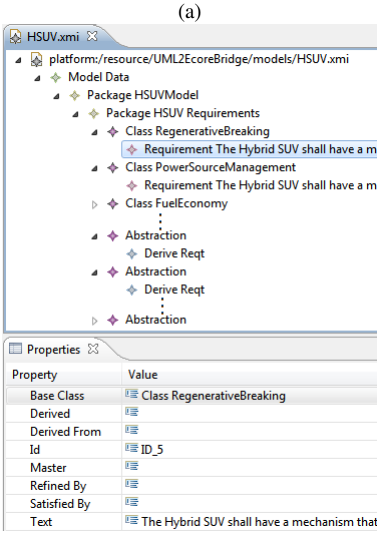
```
1  rule checkIDsConvention {
2   from
3    s : SYSML!Requirement(not s.id.startsWith('ID_'))
4  --  s : UML2!CLass(
5  --    s.isStereotypeApplied(thisModule.requirementStereotype
       ) and
6  --    not s.getValue(s.getAppliedStereotypes()->select(e |
7  --    e = thisModule.requirementStreotype
8  --     ).first(), 'id').toString().startsWith('ID_')
9  --  )
10  to
11   t : Problem!Problem (
12    severity <- #warning,
13    description <- 'the id of ' + s.base_Class.name + ' must
          start with ID_')
14 }
```

Listing 1: ATL transformation manipulating SysML models

The proposed bridge facilitates the development of this transformation in different ways, above all: (i) the target domain of the transformation (i.e., $slicedMM_{sysml}$) is smaller, making the transformation more testable and maintainable, and (ii) accessing stereotypes applications and tagged values does not require to use UML-specific constructs (lines from 4 to 9 in Listing 1 show how the condition in line 3 could have been if the input UML model was not bridged with with our approach). In this case study we showed how the proposed



(a)



(b)

Fig. 9: HSUV system: a UML model (a) and its corresponding MOF-based model (b)

bridge works and how manipulation tools developers benefit from its application. The workflow of the HSUV case study presented in this section can be fully reproduced since all metamodels, models, and transformations described above are available at the UML bridge web page.

VII. Related Work

Many approaches to bridge UML profiles and MOF metamodels have been proposed in research, each of them with its own features and usage scenarios.

Abouzahra et al. [18] proposed an integration process starting from a UML profile, a MOF metamodel and a weaving model linking their concepts. The tool takes as input the three artifacts and generates two ATL transformations that enable the bridge to be executed at the modeling level. Similarly to our work, the ATL transformations are generated by means of higher-order transformations. Further, also their approach works both at the metamodeling and modeling level, and it is based on the AMMA platform. In spite of these commonalities, the two approaches are conceptually different. While in our approach a UML profile is the only input required, the approach of Abouzahra et al. requires designers to define the MOF metamodel and the weaving model linking it to the initial UML profile too.

In [19], a specular approach is proposed: the authors assume to start from the DSML metamodel, and then to automatically generate its corresponding UML profile; model transformations for transforming DSML models to UML models and vice versa are also generated. Similarly to Abouzahra's work, this approach is based on the manual definition of a weaving model between the profile and the DSML metamodel. Moreover, the two approaches proposed in [18] and [19] have a common limitation: since the weaving model is defined by the user, elements for which there is no mapping in the weaving model are lost while round-tripping from UML to the DSML metamodel. Basically, our approach does not suffer from this limitation since the MOF metamodel is automatically generated and the mappings are defined at the M3 level. The only case in which our approach may experience the loss of information problem is when the slicing mechanism is used with wrong assumptions made by designers.

In the Eclipse platform a transformation from UML profiles to MOF metamodels is provided by EMF. A Java class called `Profile2EPackageConverter` implements such a mechanism and can be executed via a standard Java method call. This class converts UML profiles to representative Ecore packages. The transformation logic is similar to the one we proposed at the metamodeling level (in Section III-A), but it does not provide any mechanism to execute the bridge at the modeling level.

VIII. Conclusions

In this paper we presented an automatic bridge between UML profiles and MOF metamodels. The problem we want to alleviate is that, even if profiled UML models are very used since they are intuitive for designers and model editors already exist, they are intrinsically complex for model manipulation. So, the main goal of the proposed bridge is to support those kind of projects in which the system is modeled using UML profiles, and there is a strong need of automatic model manipulation. The bridge is fully automatic and loss-less with respect to the information provided in the models. The bridge is coupled with a slicing mechanism that allows designers to obtain smaller and more concise target MOF metamodels. In order to test its capabilities, we applied the proposed bridge to a non-trivial and widely used profile: SysML.

As future work, we are studying how to customize the generation of the target MOF metamodel in different ways: by deciding a priori the number of levels in its generalizations hierarchy, by specifying whether or not some kind of attributes should be part of it, or more in general, to specify a priori some kind of characteristics that the target MOF metamodel will have. We are also planning to evolve the slicing mechanism by providing a richer annotation model; for example, designers can define a black list of meta-elements that must not be part of the sliced metamodel. Finally, we are planning to embed the bridge in stable releases of other research and industrial tools; this gives us two main benefits: (i) on one side those research tools will benefit from the features of the UML bridge, (ii) on the other side it also gives us the possibility to continuously evaluate the UML bridge itself from the practitioner's perspective.

References

[1] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[2] Object Management Group, "OMG/Unified Modelling Language(UML) V2.3." 2010.

[3] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, pp. 26–36, June 2000. [Online]. Available: http://doi.acm.org/10.1145/352029.352035

[4] OMG, "SysML specification version 1.2 (formal/2010-06-01)." 2010.

[5] OMG, "MARTE specification version 1.0 (formal/2009-11-02)." 2009.

[6] OMG, "Meta Object Facility (MOF) 2.0 core specification." 2006.

[7] I. Malavolta and H. Muccini, "A Study on MDE Approaches for Engineering Wireless Sensor Networks," in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. IEEE, 2014, pp. 149–157.

[8] I. Weisemoller and A. Schurr, "A Comparison of Standard Compliant Ways to Define Domain Specific Languages," in *Models in Software Engineering*, ser. Lecture Notes in Computer Science, H. Giese, Ed. Springer Berlin / Heidelberg, 2008, vol. 5002, pp. 47–58.

[9] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-Driven Development Using UML 2.0: Promises and Pitfalls," *Computer*, vol. 39, pp. 59–66, 2006.

[10] L. Fuentes-Fernandez and A. Vallecillo-Moreno, "An introduction to UML profiles," in *UML and Model Engineering*, vol. 5, 2004.

[11] S. Bernardi, J. Merseguer, V. Cortellessa, and L. Berardinelli, "UML Profiles for Non-functional Properties at Work: Analyzing Reliability, Availability and Performance," in *NFPISML*, 2009.

[12] D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, "Model-driven techniques to enhance architectural languages interoperability," in *Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 26–42.

[13] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "JTL: a bidirectional and change propagating transformation language," in *Software Language Engineering*. Springer, 2011, pp. 183–202.

[14] D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, "Developing next generation ADLs through MDE techniques," in *ICSE 2010*. IEEE Computer Society, 2010.

[15] M. D. D. Fabro, "Metadata management using model weaving and model transformation. PhD thesis. University of Nantes," September 2007.

[16] L. Rose, A. Etien, D. Mendez, D. Kolovos, R. Paige, and F. Polack, "Comparing model-metamodel and transformation-metamodel coevolution," in *in International Workshop on Models and Evolution (ME)*, 2010.

[17] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez, "Modeling in the large and modeling in the small," in *LNCS, Vol. 3599*, 2005.

[18] A. Abouzahra, J. Bézivin, M. Didonet, D. Fabro, and F. Jouault, "A Practical Approach to Bridging Domain Specific Languages with UML profiles," in *in OOPSLA*, 2005.

[19] M. Wimmer, "A semi-automatic approach for bridging dsmls with uml," *IJWIS*, vol. 5, no. 3, 2009.