Software and Systems Modeling (2022) 21:51–79
https://doi.org/10.1007/s10270-021-00890-1

REGULAR PAPER

On the automation-supported derivation of domain-specific UML profiles considering static semantics

Alexander Kraas[1]

Received: 5 May 2019 / Revised: 27 March 2021 / Accepted: 6 May 2021 / Published online: 25 May 2021

Abstract

In the light of standardization, the *model-driven engineering (MDE)* is becoming increasingly important for the development of DSLs, in addition to traditional approaches based on grammar formalisms. Metamodels define the abstract syntax and static semantics of a DSL and can be created by using the language concepts of the *Meta Object Facility (MOF)* or by defining a UML profile.

Both metamodels and UML profiles are often provided for standardized DSLs, and the mappings of metamodels to UML profiles are usually specified informally in natural language, which also applies for the static semantics of metamodels and/or UML profiles, which has the disadvantage that ambiguities can occur, and that the static semantics must be manually translated into a machine-processable language.

To address these weaknesses, we propose a new automated approach for deriving a UML profile from the metamodel of a DSL. One novelty is that subsetting or redefining metaclass attributes are mapped to stereotype attributes whose values are computed at runtime via automatically created OCL expressions. The automatic transfer of the static semantics of a DSL to a UML profile is a further contribution of our approach. Our *DSL Metamodeling and Derivation Toolchain (DSL-MeDeTo)* implements all aspects of our proposed approach in Eclipse. This enabled us to successfully apply our approach to the two DSLs *Test Description Language (TDL)* and *Specification and Description Language (SDL)*.

1 Introduction

The use and development of *domain-specific languages (DSL)* is becoming increasingly important. In recent years, various approaches for developing DSLs have been proposed, which can be divided into two categories. The first category comprises approaches based on formalisms already used for general purpose languages, such as context-free grammars. The advantage is that existing language development tools such as parser generators can be employed for DSLs. The second category encompasses approaches that apply *model-driven engineering (MDE)* [51], where the most important activity [3,17,53] is the design and creation of a *metamodel*. Typically, a metamodel defines the abstract syntax and static semantics of the DSL to be implemented; it can also capture the semantics of a DSL [3,49]. Metamodels can be defined using the language concepts of the *Meta Object Facility (MOF)* [47], either the *Essential MOF (EMOF)* or the *Complete MOF (CMOF)*. The latter variant is based on EMOF but provides additional language concepts. A higher degree of abstraction and reuse of existing metamodels is achieved by employing the CMOF, which can be advantageous for the creation of more complex DSLs.

Apart from metamodelling, a DSL can also be implemented by customizing the *Unified Modeling Language (UML)* [40]. Either the UML metamodel can be modified or extended to meet the requirements of a DSL, or so-called *UML profiles* [11] can be used. Because UML profiles do not alter the UML metamodel, they are considered to be a lightweight extension to the UML [4,23]. When a UML profile is created, a set of UML stereotypes is introduced. A stereotype is a specific type of UML element that adds additional attributes, operations, and constraints to a metaclass. Different manual or generative approaches for implementing

Communicated by Sebastien Gerard.

This article is an extended version of our contribution [28] to the *15th European Conference on Modelling Foundations and Applications (ECMFA)* in 2017.

✉ Alexander Kraas
Alexander.Kraas@swt-bamberg.de

[1] Software Technologies Research Group, University of Bamberg, Bamberg, Germany

a DSL based on UML profiles can be found in the literature, e.g. in [15,30,50].

While working on a new version of the UML profile [20] for the *Specification and Description Language (SDL)* [21], we found several shortcomings [24,25] in the old edition [18], which resulted from the manual creation of the profile. The well-formedness rules of this profile are captured in natural language, not in a machine-processable language such as the *Object Constraint Language (OCL)* [45] that is used in the metamodelling domain. To create a toolchain [26] for the UML profile for SDL, we manually translated the well-formedness rules into OCL constraints. During this process, we noticed that these rules are often ambiguous. Likewise, the well-formedness rules of other standardised DSLs (e.g. [37,42,43]) are frequently provided only in natural language.

To remedy the shortcomings of hand-crafted UML profiles, we propose an approach for the fully automated derivation of UML profiles from CMOF-based metamodels. This is especially relevant for DSLs, for which a metamodel and a corresponding UML profile shall be created, which is often the case for standardized DSLs such as [6,43,44,46]. In contrast to a manual derivation of UML profiles from metamodels as proposed in [30,50,54], our approach performs this task fully automatically. In addition to generating model elements for a UML profile, our approach also enables the transfer of the OCL-defined static semantics of a metamodel to such a profile. While the work presented in [13,14] also supports this, the OCL constraints of a metamodel must be revised manually before they can be copied to a UML profile, whereas this is not required by our approach. Furthermore, we enable the mapping of subsetting or redefining metaclass attributes to OCL-defined stereotype attributes, which is a novel feature compared to [13,14,48]. Moreover, our approach supports the semi-automatic generation of CMOF-based metamodels from grammar production rules and of model transformations for model interoperability from DSL models to UML models with applied profile, and vice versa.

The remainder of this article is structured as follows. The next section provides an overview of our approach, while we detail the derivation of UML profiles from metamodels in Sec. 4. Then, we discuss our automatic transfer of the OCL-defined static semantics in Sec. 5. Based on a case study for the *Test Description Language (TDL)* [6], which is an international standard maintained by the *European Telecommunications Standards Institute (ETSI)*, we evaluate the results of our approach described in Sec. 6. Finally, we discuss the related work in Sec. 7, while Sec. 8 presents our summary and conclusions.

The concepts of our derivation approach have already been sketched as extended abstract in [28], and its applicability to automatically derive a UML profile for SDL has been demonstrated in another extended abstract [27]. In the present article, we give a fully comprehensive insight into our derivation of UML profiles and our automatic transfer of the OCL-defined static semantics. In particular, we detail our considerations that inspired this derivation approach. Furthermore, we analyse the applicability of our derivation approach based on a further case study involving TDL. To evaluate all aspects of our approach and to conduct case studies on SDL and TDL, we implemented the *DSL Metamodeling and Derivation Toolchain (DSL-MeDeTo)* [58] in Eclipse.

2 Overview

This section provides an overview of our overall approach to derive a metamodel, a UML profile, and model transformations for model interoperability. In addition, we give a short introduction to our DSL-MeDeTo toolchain that implements our approach.

2.1 Our overall approach

Our overall derivation approach shown in Fig. 1 consists of Steps (A)–(E), some of which are optional. The DSL-specific metamodel MM_{Domain} is the central artefact for almost all derivations. If production rules of a grammar-based DSL are available, the metamodel can be generated semi-automatically in Step (A); otherwise, it has to be created manually.

To obtain a metamodel that does not require too much effort for further refinement (e.g. adding additional metaclasses), we reuse 'Abstract Concepts' that are defined by an existing metamodel (MM_{AC}), as proposed in [10,49]. Apart from these works, the reuse of artefacts of existing languages is recommended in other works, e.g. in [3,22]). In contrast to the approach proposed in [10,49], we use particular annotations for a given DSL's production rules so that relationships between generated metaclasses and the 'Abstract Concepts' must not be created manually.

We define 'Abstract Concepts' as a set of generic language concepts that are commonly shared across several DSLs and not only applicable to a specific language. For example, language concepts such as generalization or redefinition can be regarded as 'Abstract Concepts'. Furthermore, we assume that each 'Abstract Concept' is represented by a particular metaclass contained in MM_{AC}.

If we derive MM_{Domain} from production rules in Step (A)[1], we have to review and, if necessary, refine it before it can be used as input for Steps (B)–(E). In particular, it must be

[1] Our approach to create metamodels from production rules is similar to that of existing tools (e.g. EMFText [57] or xText [59]) and is therefore not entirely novel. Further details can be found in [29].

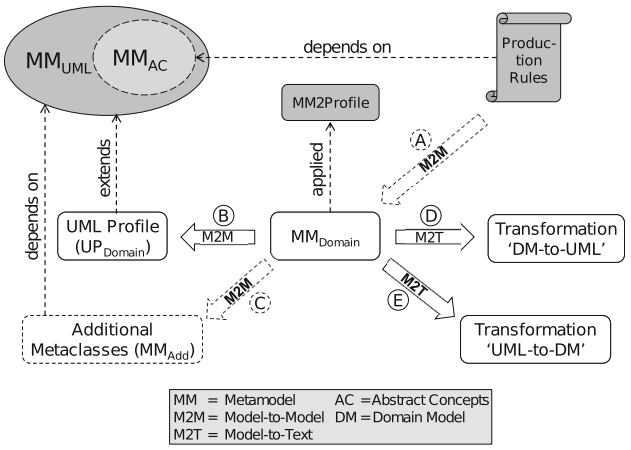


Fig. 1 Transformation steps and their derived artefacts

assessed whether all metaclasses derived for MM_{Domain} are required from a semantic point of view, and if not, the affected ones must be removed. Otherwise, we create MM_{Domain} from scratch. To do so, we copy the 'Abstract Concepts' from MM_{AC} to MM_{Domain}. Then, we capture the abstract semantics of the DSL of interest by creating appropriate metaclasses. Thereafter, we use inheritance relationships to associate these metaclasses with the 'Abstract Concepts' copied to MM_{Domain}. Finally, we capture the DSL's static semantics via OCL constraints and enrich MM_{Domain} with toolchain-specific meta-information. For this purpose, we apply the UML profile *'MM2Profile'* to MM_{Domain} as shown in Fig. 1. This UML profile contains five stereotypes and a set of constraints (see Sec. 4.2) that ensure a proper processing of MM_{Domain} via our toolchain.

After creating MM_{Domain}, we can automatically derive a DSL-specific UML profile UP_{Domain} in Step (B). If required, additional metaclasses (contained in MM_{Add}) that extend the MM_{UML} are derived in Step (C). The derivation of additional metaclasses may be an option if stereotypes cannot be employed due to their restrictions as defined by the UML [41]. For instance, such an approach is applied for the value and expression languages of the SDL-UML profile [20] and of the MARTE profile [42]. Because the input and output artefacts of Steps (B) and (C) are models, we realize both derivations by two dedicated *Model-to-Model (M2M)* transformations.

In Steps (D) and (E), we derive two M2M transformations that can be used to obtain model interoperability between DSL-specific models and UML models with an applied UML profile. For this purpose, we develop two *Model-to-Text (M2T)* transformations that generate the source code of the M2M transformations $T_{DM-to-UML}$ and $T_{UML-to-DM}$, resp.

Even though our automatic derivation approach eliminates the need for manual creation of UML profiles, the quality of these artefacts depends significantly on the experience of the language engineer who manually created the metamodels used as input. As with other hand-crafted software artefacts, metamodels can contain errors due to manual creation. Therefore, metamodels used as input for our derivation approach should be subject to quality assurance.

2.2 The 'Abstract Concepts'

The metamodel MM_{AC} holds a key role for our entire derivation approach, because metaclasses of the metamodel MM_{Domain} inherit from 'Abstract Concepts' defined by MM_{AC}. An important prerequisite for MM_{AC} is that it must match a subset of MM_{UML}. Otherwise, a straightforward mapping of MM_{Domain} to UP_{Domain}, as implemented by our approach, is impossible.

We consider a metamodel MM_{AC} to be matching with MM_{UML}, if the following constraints are fulfilled:

Constraint 1 *For each metaclass* `MC` *of* MM_{AC}*, a corresponding metaclass* `MC'` *with an equal name shall be present in* MM_{UML}*. In addition,* `MC` *shall have an equal or lesser number of attributes than* `MC'`*.*

This constraint is essential for the derivation of UML profiles according to our approach, because the UML metaclasses to be extended by *Stereotypes* are identified based on the correlation between 'Abstract Concepts' and UML metaclasses. To determine such a correlation, we employ the *name* property of the metaclasses. Even though an 'Abstract Concept' may have fewer attributes than the corresponding UML metaclass, we consider a correlation as given. For instance, such a situation may occur if some attributes of an 'Abstract Concept' are removed[2] because they are not required to define the syntax of a DSL.

Constraint 2 *For each attribute* `att` *of a metaclass* `MC`*, a corresponding attribute* `att'` *with an equal name shall be present in metaclass* `MC'`*. In addition,* `att` *and* `att'` *shall have the same properties, especially the same type and multiplicity.*

The condition imposed by Constraint 2 is important because OCL constraints in MM_{Domain} may capture attributes of 'Abstract Concepts'. When deriving a UML profile, such an attribute access must be translated into an access to a UML metaclass attribute. In addition, attributes of metaclasses in MM_{Domain} may redefine or subset 'Abstract Concept' attributes. During the UML profile derivation, we have to

[2] Removing metaclass attributes requires extreme care and should only be done for optional attributes (lower multiplicity == 0). Otherwise, it cannot be ensured that a generated UML profile is semantically compatible with the UML.

translate such attribute relationships into appropriate constructs that access UML metaclass attributes.

In addition to metaclasses, the 'Abstract Concepts' may contain *DataTypes*. Therefore, each of these *DataTypes* must have a corresponding type in MM_{UML}. We capture this condition by the following constraint:

Constraint 3 *A data type in MM_{AC} shall have a corresponding data type in MM_{UML}.*

Different approaches can be applied to obtain a metamodel MM_{AC}. Apart from creating such a metamodel from scratch, Clark et al. [3] argue that also a reuse of metaclasses of an existing metamodel, by copying or importing them, can be considered. Because MM_{AC} shall match with MM_{UML}, we consider a creation of MM_{AC} from scratch to be too error-prone and expensive. Another option is to use the MOF or the *UML Infrastructure Library* [39]. Because the metaclasses of these metamodels are primarily employed to define UML's 'Kernel' package, they may be reused to create an MM_{AC} that only supports structural language concepts (e.g. *Classifier*). Finally, also the reuse of parts of MM_{UML} may be considered if language concepts for behavioural specifications (e.g. *StateMachines*) are required.

Our approach does not support the import of metaclasses; otherwise, there would be a dependency between MM_{Domain} and the metamodel from which the metaclasses are imported. In addition, imported metaclasses cannot be modified. For example, it is impossible to remove non-required features from them. Therefore, we assume that MM_{AC} is created manually as a copy, or automatically using the *Package Merge* feature provided by the CMOF.

2.3 The DSL metamodelling and derivation toolchain

We have implemented all aspects of our approach in the novel *DSL Metamodelling and Derivation Toolchain (DSL-MeDeTo)* [58] using well-established standards and open source components. We choose the *Model Development Tools (MDT)*[3] edition of Eclipse to realize our toolchain, which consists of a set of particular plug-ins as shown in Fig. 2. To create a metamodel or to derive other artefacts such as UML profiles, the components of our toolchain have access to a common model repository. Because our derivation approach is designed for CMOF-based metamodels, we have to employ UML models instead of Ecore models [52]. The code generators of Eclipse-MDT can handle both formats. The most important plug-ins of our toolchain are:

Textual editor. The textual editor enables the specification of production rules for the DSL or computer language of interest. In addition, the production rules can be associated with different types of annotation, e.g. to define a relationship to the 'Abstract Concepts'. The parsed production rules are stored by the textual editor in an intermediate model-based format; thus, they can be transformed to a metamodel by employing an M2M transformation. As we have made good experiences with Spoofax for creating our SU-MoVal framework [26], we also employed this DSL workbench to create the textual editor for DSL-MeDeTo.

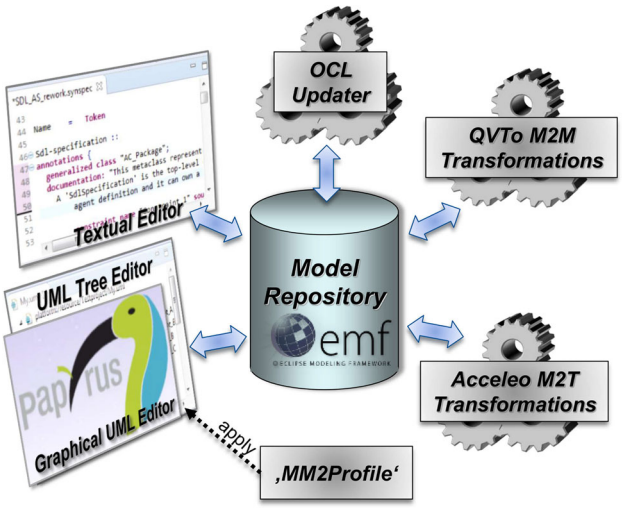


Fig. 2 The Eclipse-based toolchain and its components

UML profile 'MM2Profile'. The UML profile 'MM2Profile' is used to enrich a metamodel MM_{Domain} with additional information, which is processed by different components of DSL-MeDeTo. The application of this UML profile to a metamodel MM_{Domain} is a prerequisite for the derivation of a UML profile UP_{Domain} and of the additional metaclasses MM_{Add}. In addition, 'MM2Profile' defines a set of OCL constraints that must be met by MM_{Domain} so that its processing by our toolchain is sound. More details of 'MM2Profile' are discussed in Sec. 4.2.

OCL Updater. Before the metamodel MM_{Domain} with applied 'MM2Profile' can be used for the derivation of other artefacts, all its OCL-defined *Operations*, *Properties* and *Constraints* have to be adapted in such a way that they can be utilized for a derived UML profile. This adaptation is implemented by the 'OCL Updater' component, which consists of an OCL parser and a pretty printer. The update is based on the abstract syntax tree (AST) of a parsed OCL expression, as argued in Sec. 5.

M2M transformations. We employ the operational language of the *Query/View/Transformation* specification (QVT) [38] for implementing three M2M transformations that are used by DSL-MeDeTo to derive different kinds of artefacts. The first transformation implements Step (A) of our approach; its output is the CMOF-based metamodel MM_{Domain}. A sec-

[3] https://www.eclipse.org/modeling/mdt/

ond transformation derives the UML profile UP_{Domain} from MM_{Domain} (Step (B)). Optionally, the 'additional' metaclasses for MM_{Add} can be created with a third transformation (Step (C)).

M2T transformations. We use the *MOF M2T Language (MTL)* [36] for realizing two M2T transformations: the M2T transformation $T_{GenDM-to-UML}$ is employed to generate the M2M transformation $T_{DM-to-UML}$ in Step (D), while $T_{GenUML-to-DM}$ generates $T_{DM-to-UML}$ in Step (E). The source code of both M2M transformations is generated in terms of the operational language of QVT. We utilize the Acceleo[4] component of Eclipse to execute both M2T transformations.

Model editors. Apart from the discussed components, either Eclipse's UML tree editor or the UML modelling tool Papyrus[5] can be used for creating or modifying the metamodels that are processed by our toolchain (see Fig. 2). Hence, one of these tools also has to be utilized for applying the UML profile 'MM2Profile' to a metamodel.

3 Running example

The *Test Description Language (TDL)* [6] is a new DSL [32, 55] for the design and specification of test descriptions. Its development and standardization is driven by the *European Telecommunications Standards Institute (ETSI)*. Because a metamodel [6] and a corresponding UML profile are available, we employ TDL as an exemplary DSL for our case study in Sec. 6 in order to investigate the applicability of our derivation approach. Furthermore, TDL serves us as running example to illustrate the various steps in deriving a UML profile from a CMOF-based metamodel.

Background. Ulricht et al. [55] point out that TDL bridges the gap between high-level test requirement specifications and executable test cases. Thus, TDL test descriptions can serve as the basis to create executable test cases in any kind of target language, such as the *Testing and Test Control Notation-Version 3 (TTCN-3)* [8]. Because TDL is a new language, the literature concerning the application of TDL is rather limited. Marroquin et al. [34] report on a successful application of TDL in the telecommunications domain. The automatic derivation of TDL test descriptions from *Use Case Maps (UCM)* is discussed by Boulet et. al [2]. A TDL test description consists of the following parts:

Test configuration: A test configuration defines the tester components and the components of a system under test (SUT) involved in a particular test scenario. All components communicate among each other via defined gates and associated connections.

Test descriptions: A particular test scenario is described in terms of a test description that defines the set of interactions between the test and SUT components of its associated test configuration. Various types of behavioural elements are available for specifying the control flow of a test description, e.g. sequential or parallel behaviour.

Data definitions: Data types can be specified as simple or structured data types, and their instances are used in interactions or can be passed as arguments for the invocation of parameterized test descriptions.

Behavioural elements: The various behavioural element types can be used to define the control flow, send messages, start and stop timers, or set the test result.

A TDL test description example is given in Fig. 3. The shown diagram contains the definitions of data types and associated data instances. In addition, the 'DataResourceMapping' elements are used to specify a resource that contains an external representation for data types or their instances. Such an external representation is identified by a 'DataElementMapping'. For instance, the structured data type `MSG` is associated with the 'DataResourceMapping'.

Apart from data type definitions, the shown diagram also contains a *TestConfiguration* consisting of a tester and an SUT component. Because both components are instances of the *ComponentType* `DefaultCTwithVariable`, each of them owns a gate, which is of type `defaultGT`. The gates of the component instances are interconnected via a *Connection*.

Based on the type definitions and their instances, we can specify the behaviour of an TDL test description. The notation employed for TDL's test behaviour is similar to *Message Sequence Charts (MSC)* [19].

4 The derivation of UML profiles

As claimed in Introduction, we can automatically derive a UML profile based on a metamodel for a DSL in Step (B) of our approach. In this section, we first discuss the design decisions of our derivation approach. Then, we treat the prerequisites for the suitability of a metamodel as input for our UML profile derivation, and also consider the enrichment of

[4] http://www.eclipse.org/acceleo/

[5] https://www.eclipse.org/papyrus/

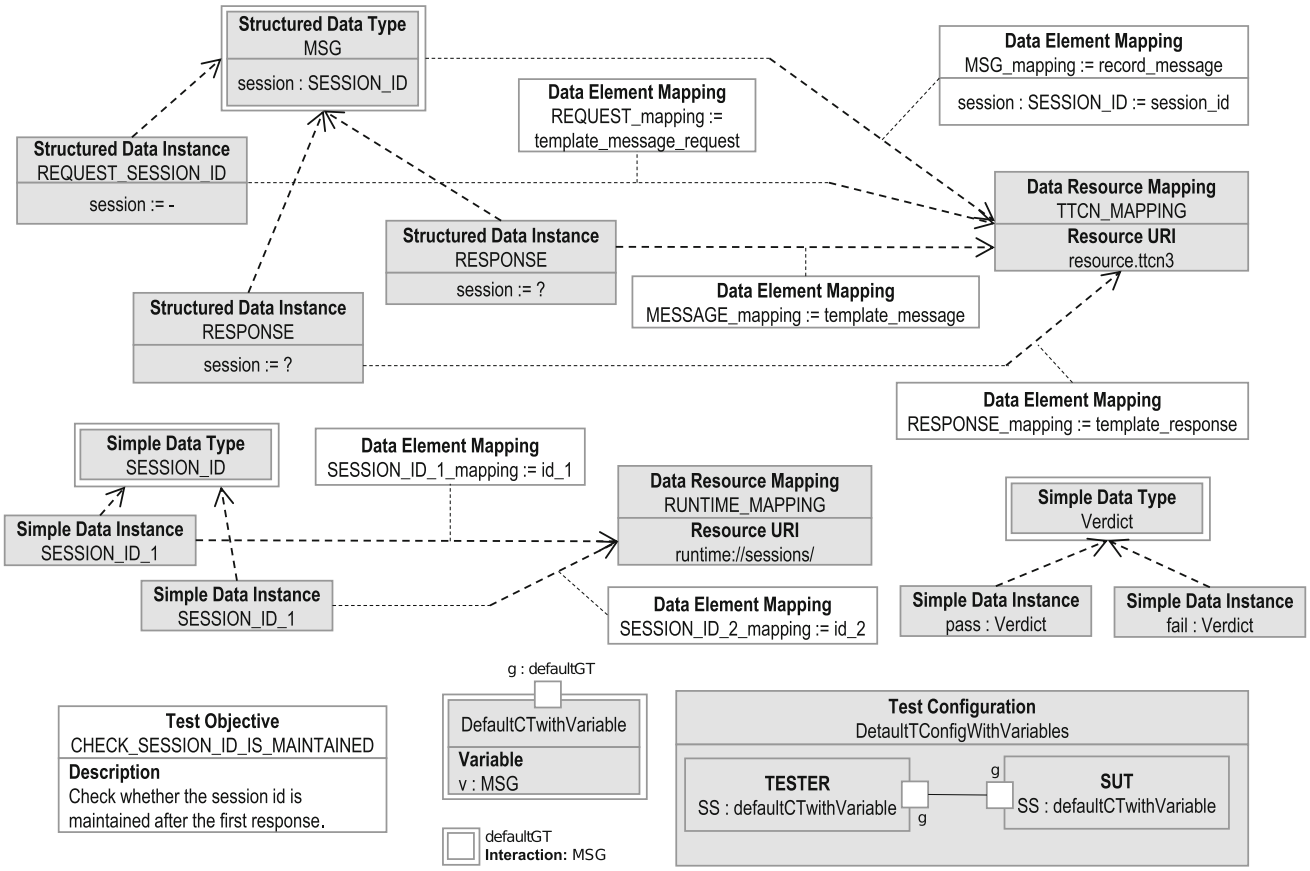


Fig. 3 Data and test configuration definition example ([7]—Annex A.1), specified in TDL's graphical notation

a metamodel with information that is required for the derivation. Thereafter, we discuss the details of our approach.

4.1 Design decisions for the profile derivation

As our objective is to derive a UML profile and optionally 'additional' metaclasses, the metaclasses contained in the metamodel MM_{Domain} have to be processed in different ways. In addition, we have to consider that MM_{Domain} also contains metaclasses that represent 'Abstract Concepts'. Hence, we presume that the metaclasses in MM_{Domain} can be divided into three different sets. The first set represents 'Abstract Concepts'; a metaclass of this set is denoted as MC_{AC}. The second set includes those metaclasses that shall be mapped to *Stereotypes* of the derived UML profile UP_{Domain}. We use the term MC_{St} to refer to a metaclass of this set. The third set contains metaclasses that map to 'additional' metaclasses in the metamodel MM_{Add}. A metaclass of this set is denoted by MC_{AMC}.

Design Decision 1 *A metamodel MM_{Domain} consists of a set of MC_{AC} metaclasses, a set of MC_{St} metaclasses, and an optional set of MC_{AMC} metaclasses.*

Each 'Abstract Concept' metaclass has a 'matching' UML metaclass in MM_{UML}, and their names are prefixed with `'AC_'` to avoid name clashes with those metaclasses that are generated based on production rules. Hence, we do not map MC_{AC} metaclasses contained in MM_{Domain}. In addition, we employ the name prefix to identify that a metaclass of MM_{Domain} is an MC_{AC} metaclass.

Design Decision 2 *An 'Abstract Concept' metaclass MC_{AC} has a name prefix `'AC_'` and is not mapped to any kind of element.*

Hence, all metaclasses without a name prefix are MC_{St} or MC_{AMC} metaclasses. To make a clear distinction between these two metaclass types, an additional qualifier is required. As the derivation of 'additional' metaclasses is optional, it is sufficient to use such a qualifier only for MC_{AMC} metaclasses. For this reason, we define metaclasses that have no special qualifier or name prefix as MC_{St} metaclasses. By default, we use this type of metaclasses of MM_{Domain} to derive *Stereotypes* for a UML profile, as shown in Fig. 4.

Design Decision 3 *A metaclass of MM_{Domain} without a qualifier is assumed to be an MC_{St} metaclass, whereas a metaclass with existing qualifier represents an MC_{AMC} metaclass.*

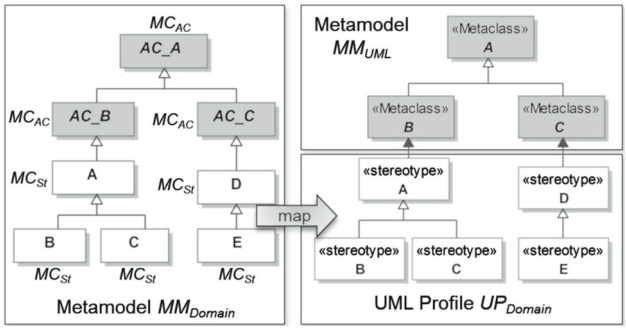


Fig. 4 Mapping of metaclasses to *Stereotypes*

> *"An element imported as a metaclassReference is not specialized or generalized in a Profile"* [41, Sec. 12.4.7.5].
>
> *"A Stereotype may only generalize or specialize another Stereotype"* [41, Sec. 12.4.9.6].

Hence, a *Stereotype* is not permitted to generalize a UML metaclass. Instead, an *Extension* association has to be used. Consequently, we introduce an *Extension* for each *Stereotype*, which is derived from an MC_{St} that inherits directly from an MC_{AC} in MM_{Domain}. For example, an *Extension* is introduced only for *Stereotypes* derived from metaclasses MC_{St} `A` and MC_{St} `D` in Fig. 4.

Design Decision 4 *An Extension shall be introduced for each Stereotype, which is derived from an MC_{St} metaclass that directly inherits from an MC_{AC} metaclass.*

In contrast to the previous case, a *Stereotype* can inherit from another *Stereotype* without restrictions. Therefore, we can introduce a *Generalization* between two *Stereotypes* that are derived from two MC_{St} metaclasses that are in a *Generalization* relationship (e.g. metaclasses MC_{St} `E` and MC_{St} `D` in Fig. 4).

Design Decision 5 *A Generalization is introduced for each Stereotype that is derived from an MC_{St} metaclass that inherits directly from another MC_{St} metaclass.*

One of our key objectives for the derivation of a UML profile is the preservation of the syntactic structure defined by a metamodel. This is a prerequisite for transferring the OCL-defined static semantics of a metamodel to a UML profile. Therefore, we map each *Property* that is an *ownedAttribute* of an MC_{St} to a corresponding *Property* of a *Stereotype*. However, we have to obey the following rule of the UML specification:

> *"The type of a composite aggregation Stereotype Property cannot be a Stereotype (since Stereotypes are owned by their Extensions) or a metaclass (since instances of metaclasses are owned by other instances of metaclasses)"* [41, Sec. 12.3.3.4].

Therefore, we map an MC_{St} attribute whose *type* property refers to a metaclass, to a stereotype attribute whose *aggregation* property has value 'none'. Thus, this stereotype attribute represents a reference to a metaclass instance.

Design Decision 6 *An MC_{St} attribute is mapped to a corresponding Stereotype attribute. If the type property of an MC_{St} attribute refers to a metaclass, the aggregation property of the mapped Stereotype attribute shall have value 'none'.*

If an *ownedAttribute* of an MC_{AC} is redefined or subsetted by an *ownedAttribute* of an MC_{St}, this kind of relationship cannot be preserved for a derived UML profile. This is caused by the two UML constraints above and the fact that redefinition and subsetting can only be used for *Classes* that are in a direct or indirect inheritance relationship.

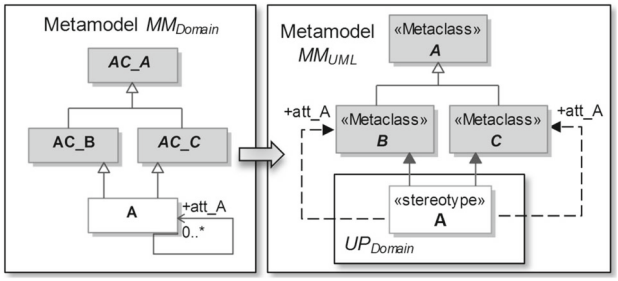
In general, such metaclass attributes may be mapped to stereotype attributes in two different ways. The first possibility is a one-to-one mapping to a stereotype attribute, where existing redefinition/subsetting relationships are removed. However, the drawback of this approach is that values for stereotype attributes have to be assigned manually. The second option is to map a metaclass attribute to a 'derived' stereotype attribute whose value is computed automatically via an OCL expression at runtime. Because of this advantage, we choose the latter option.

Design Decision 7 *An MC_{St} attribute that redefines or subsets an MC_{AC} attribute is mapped to a derived and read-only stereotype attribute, and an OCL expression is introduced as its defaultValue.*

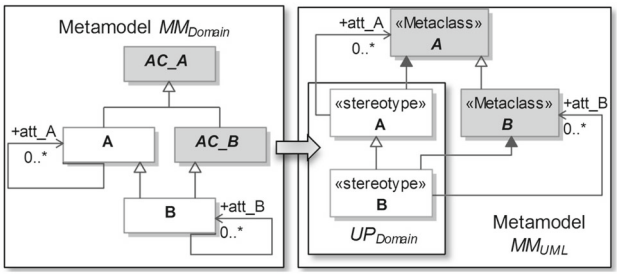
When specifying a UML model, a model element must be created before a stereotype can be applied to it. While the model element is always located at a certain position in the UML model, the instance of its applied stereotype is not directly contained in the model. However, the UML model and associated stereotype instances are contained in the same resource or container. This fact could become an issue when the *type* of an *ownedAttribute* of a *Stereotype* refers to another *Stereotype*. In this case, the designer of a UML model has to identify a valid *Stereotype* instance that shall be assigned as value for a *Stereotype* attribute, which is non-trivial because *Stereotype* instances have no unique identification feature.

Consequently, we do not use a *Stereotype* as the *type* of a stereotype attribute; instead, we refer to a UML metaclass that is extended by a *Stereotype*, or to an 'additional' metaclass contained in the MM_{Add} metamodel.

Design Decision 8 *The type property of an stereotype attribute either refers to a UML metaclass that is extended by a Stereotype, or to an 'additional' metaclass in MM_{Add}.*



(a) MC_{St} metaclass that inherits from two MC_{AC} metaclasses



(b) MC_{St} metaclass that inherits from an MC_{St} and an MC_{AC} metaclass

Fig. 5 Inheritance and the *type* of a stereotype attribute

The discussed recomputation of the *type* of a stereotype attribute also affects the modelling of inheritance relationships between MC_{St} and MC_{AC} metaclasses in MM_{Domain}. As long as an MC_{St} only inherits from a single MC_{AC}, there are no restrictions on the modelling of MM_{Domain}. But this is not the case for multiple inheritance.

Assume that an MC_{St} `A` that inherits from at least two MC_{AC} metaclasses is used as the *type* of a metaclass attribute `att_A`, and the MC_{St} is mapped to a *Stereotype* `A`, as shown in Fig. 5a. According to Design Decision 4, this stereotype would have two *Extension* associations with different UML metaclasses. Due to Design Decision 8, two possibilities would exist as *type* of the mapped attribute `att_A`.

To avoid this ambiguity, we must ensure that a derived *Stereotype* has only an *Extension* to a single UML metaclass. One possible solution would be that an MC_{St} of MM_{Domain} inherits from at most one MC_{AC}, so that only one *Extension* would be derived for a *Stereotype*. However, this approach would require a modification of MM_{Domain}, or its modelling would become too restrictive. Hence, we consider this approach to be inappropriate.

Another solution would be to introduce a specific kind of metadata that can be attached to an MC_{St}, so that the UML metaclass to be extended by a *Stereotype* can be defined explicitly. Because this approach does not require a modification of MM_{Domain}, we prefer this solution.

Design Decision 9 *A particular kind of meta-information (e.g. provided by an applied stereotype) shall be applicable to an MC_{St} so that the UML metaclass to be extended by a Stereotype can be defined explicitly.*

Table 1 *Properties* that redefine and/or subsets other properties of UML metaclasses (analysis based on the UML metamodel [40])

Number of 'Properties' subsetted	redefined	Number of occurrence in MM_{UML}
1	0	**437** (75,7%)
≥ 2	0	**79** (13,7%)
0	1	**53** (9,1%)
0	≥ 2	**3** (0,5%)
≥ 1	≥ 1	**6** (1,0%)

For the sake of completeness, it is mentioned that an MC_{St} can also inherit from an MC_{AC} and one or more MC_{St} metaclasses, as shown in Fig. 5b. In contrast to the previous scenario, no ambiguities for the determination of the *type* property of a stereotype attribute exist, because a derived *Stereotype* has only a single *Extension* to a UML metaclass. Due to Design Decision 8, *Generalizations* to other *Stereotypes* do not affect type determination.

A metaclass attribute can redefine and subset other attributes at the same time. In addition, a metaclass attribute cannot only redefine or subset a single attribute, but also several attributes. We have to consider this when introducing OCL expressions as substitute for redefining or subsetting attributes (see Design Decision 7).

Because subsetting or redefinition of several attributes usually occurs only in the case of multiple inheritance, we must pay particular attention to Design Decision 9, which requires that a derived stereotype extends only a single UML metaclass. Thus, only attributes of this UML metaclass can be invoked in generated OCL expressions for stereotype attributes. Hence, in the case of multiple inheritance, we must either be able to specify which UML metaclass attributes should be invoked in OCL expressions, or there must be a possibility to provide OCL expressions manually.

To decide which solution shall be implemented by our derivation approach, we have analysed the UML metamodel concerning the utilization of 'subsetting' and 'derivation'. The results are summarized in Table 1. Based on these, we suppose that the 'subsetting' of a single attribute (75.7%) is much more common than that of multiple attributes (13.7%). A similar situation exists for 'redefinition', whereas a combined use of 'redefinition' and 'subsetting' (1%) can be considered as a rarely used special case. Hence, we automatically introduce OCL expressions only for such stereotype attributes derived from MC_{St} attributes, which 'redefine' or 'subset' a single attribute. In all other cases, we prefer a manual specification of OCL expressions.

Design Decision 10 *An ownedAttribute of an MC_{St} can have an alternative OCL expression, which is used as defaultValue of a corresponding stereotype attribute.*

Due to Design Decision 8, we recompute the *type* property of a stereotype attribute so that it refers to a UML metaclass. A disadvantage is that syntactically invalid values can be assigned to such kind of attribute. For example, a value assigned to a stereotype attribute may have the correct UML type, but it may have applied an invalid stereotype. Therefore, we introduce OCL constraints to ensure that only those UML elements having applied a particular stereotype can be assigned to such stereotype attributes. However, we do not generate OCL constraints for 'derived' and 'read-only' stereotype attributes, because no values can be assigned to them manually (Design Decision 7).

Design Decision 11 *An OCL Constraint is created for each Stereotype attribute that is not defined as 'derived' and 'read-only' and that is mapped from an MC_{St} attribute with a type property that refers to an MC_{St}.*

Even though we do not introduce OCL constraints for 'derived' and 'read-only' stereotype attributes, this does not apply for those UML metaclass attributes that are employed as computational basis for the *defaultValue* of stereotype attributes (Design Decision 7). Because of the same reason as above, only UML elements having applied a particular stereotype shall be assignable to such UML metaclass attributes. Hence, we introduce appropriate OCL constraints. We identify the UML metaclass attributes that shall be constrained based on the redefinition and subsetting relationships of MC_{St} attributes.

Design Decision 12 *An OCL Constraint is created for each UML metaclass attribute that is the computation base for the value of a 'derived' and 'read-only' stereotype attribute.*

4.2 Enriching the source metamodel

Provided that MM_{Domain} is generated from production rules in Step (A) of our approach, we can use it directly as input for the UML profile derivation. By definition, we consider that such a metamodel contains a set of MC_{AC} metaclasses and a set of MC_{St} metaclasses (Design Decisions 2 and 3). In this case, we determine the UML metaclass to be extended by a derived *Stereotype* based on the inheritance relationships of the source MC_{St} in MM_{Domain}.

However, the direct use of MM_{Domain} to derive a UML profile is not always possible. For example, we cannot define that, instead of a 'matching' UML metaclass, one of its subtypes shall be extended by a *Stereotype*. Therefore, the stereotypes of our UML profile 'MM2Profile' presented below must be applied to metaclasses of MM_{Domain}.

As shown in Fig. 6, the UML profile 'MM2Profile' consists of five stereotypes that extend four different UML metaclasses. Except for stereotype `<<MM2Profile>>`

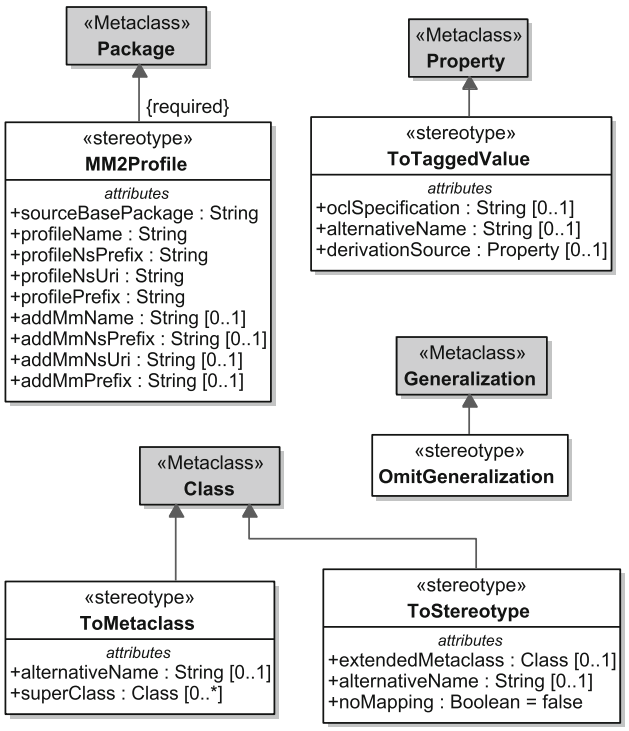


Fig. 6 Stereotypes of the UML profile 'MM2Profile'

that is automatically applied to a *Package* element, the application of all other stereotypes is optional. We thus enable the application of these stereotypes only in cases when additional information is required for the derivation.

The `<<MM2Profile>>` stereotype Because this stereotype has an *Extension* that is specified as required, it is automatically applied to the *Package* that represents MM_{Domain}. Most of the stereotype attributes define input parameters for the code generator of Eclipse and, therefore, are passed directly to the derived UML profile and to the metamodel MM_{Add}.

While attributes prefixed with `'profile'` are passed to the derived UML profile UP_{Domain}, all attributes with the `'add'` prefix are passed to the metamodel MM_{Add} that contains the 'additional' metaclasses.

The `<<ToStereotype>>` stereotype If the default creation of a *Stereotype* according to our approach is infeasible, the attributes of the `<<ToStereotype>>` can be used as follows:

extendedMetaclass	overrides the automatically determined UML metaclass to be extended by a *Stereotype*.
alternativeName	defines an alternative name for a derived *Stereotype*.
noMapping	determines whether a *Stereotype* is generated for the current MC_{St} metaclass.

The <<ToMetaclass>> stereotype According to Design Decision 3, an additional qualification for an MC_{AMC} metaclass in MM_{Domain} is required. Hence, we employ the stereotype `<<ToMetaclass>>` to define that a metaclass of MM_{Domain} represents an MC_{AMC} metaclass. The `<<ToMetaclass>>` stereotype provides the following attributes:

alternativeName defines an alternative name for an 'additional' metaclass contained in MM_{Add}.
superClass overrides the automatically determined UML metaclass from which an 'additional' metaclass inherits.

The <<ToTaggedValue>> stereotype We employ this stereotype to explicitly define a *name* or to specify an alternative OCL expression (used as *defaultValue*) for an attribute of a derived *Stereotype*. To support these, the `<<ToTaggedValue>>` stereotype provides the following attributes:

alternativeName defines an alternative name for an *ownedAttribute* of a derived *Stereotype*.
oclSpecification defines an OCL expression that is used as *defaultValue* for an *ownedAttribute* of a *Stereotype*.
derivationSource specifies the computation source for the `oclSpecification`.

The <<OmitGeneralization>> stereotype A metaclass can have more than one *Generalization* relation to other metaclasses. However, for the derivation of a UML profile, it may be required that a *Generalization* to a particular metaclass is not taken into account. This can be specified by applying the `<<OmitGeneralization>>` stereotype to a *Generalization* contained in MM_{Domain}.

For example, assume an MC_{St} `A` is given that inherits from an MC_{AC} at a high level of abstraction (e.g. `AC_Namespace`) and another MC_{AC} (e.g. `AC_DataType`). In this case, we want to prevent that the stereotype derived from `A` extends the UML metaclass `Namespace`. To achieve this, we apply the `<<OmitGeneralization>>` stereotype to the *Generalization* from `A` to `AC_Namespace`.

4.3 TDL Example: Input Metamodel

TDL's standardized metamodel MM_{std} is divided into several *Packages*, one of which is the `Foundation` package. The metaclasses contained in this package define generic language concepts, such as *NamedElement* and *PackageableElement*. These metaclasses are comparable to those contained in the UML 'Kernel' package. Thus, one can assume that these metaclasses of MM_{std} already correspond to the MC_{AC} metaclasses required by our approach. However, differences in syntax and semantics exist. For instance, metaclass attributes contained in TDL's `Foundation` package do not match those of UML metaclasses. Because identically named attributes of 'Abstract Concepts' and UML metaclasses are a prerequisite (see Sec. 2.2) of our derivation approach, we cannot use MM_{std} as input to derive a UML profile. Therefore, we have aligned MM_{std} to meet the requirements of our derivation approach. Below, we refer to the revised metamodel of TDL using the term MM_{TDL}.

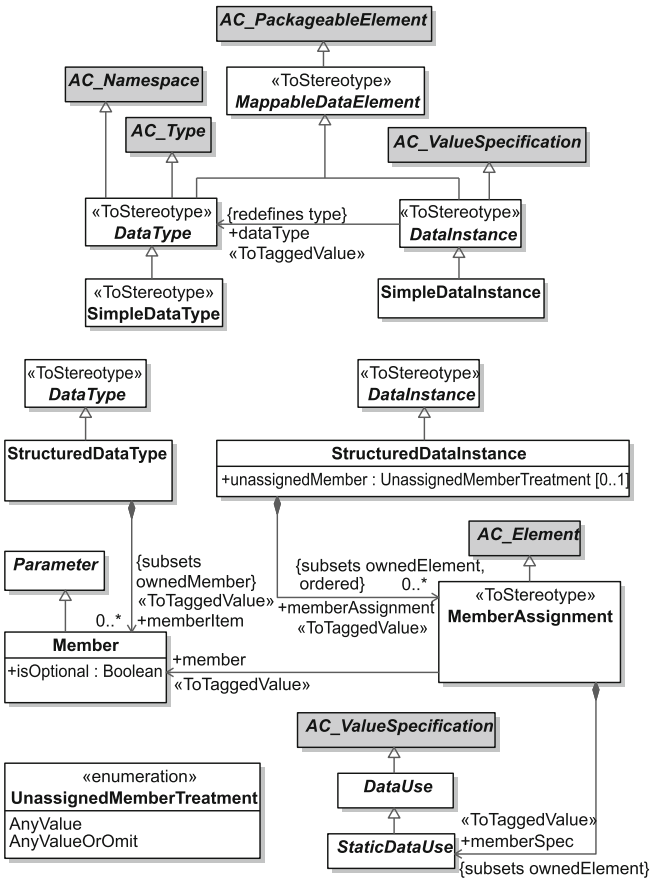
To obtain MM_{TDL}, we replace the metaclasses contained in TDL's `Foundation` package with equally named metaclasses of the UML `Kernal` package. Then, we put the copied metaclasses in inheritance relationships to TDL's metaclasses by introducing *Generalization* relationship. Furthermore, we redefine or subset the attributes inherited from the copied metaclasses where required. Finally, we apply our UML profile `MM2Profile` to enrich MM_{TDL} with the meta-information that we require to derive a UML profile.

An excerpt of our modified MM_{TDL} is shown in Figs. 7a and 7b, where the contained metaclasses define the syntax of TDL's data types and test configurations. Some of the metaclasses (e.g. `MappableDataElement`) and attributes contained in the figures have stereotypes of our UML profile `MM2Profile` applied. We assume that metaclasses whose names start without the prefix `'AC_'` represent MC_{St} metaclasses that are mapped to *Stereotypes*.

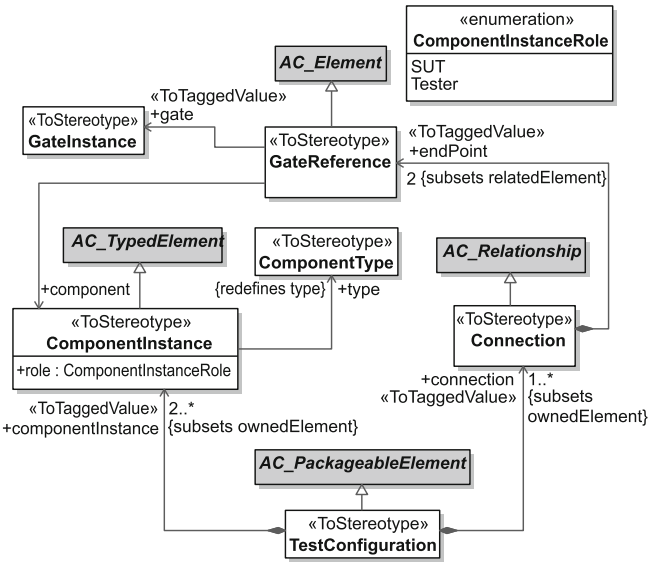
Without additional meta-information, a *Stereotype* derived from the `DataType` metaclass shown in Fig. 7a would extend the UML metaclass *Type* or *Namespace*. Consequently, this *Stereotype* could then be applied to all UML elements that are instances of subclasses of these UML metaclasses. This would not correspond to the syntax specified by the MM_{TDL} metamodel. Applying the `<<ToStereotype>>` stereotype on an MC_{St} metaclass remedies this issue, because its `extendedMetaclass` attribute can be employed to explicitly specify the UML metaclass to be extended by a *Stereotype*.

As the abstract metaclass `MappableDataElement` has neither constraints, nor attributes, or operations, we do not intend to derive a *Stereotype* for it. For this reason, we apply the `<<ToStereotype>>` stereotype to this metaclass and assign the value 'true' to its `noMapping` attribute.

We use the `<<ToTaggedValue>>` stereotype to prevent an MC_{St} attribute being mapped according to the default rules of our derivation approach. An MC_{St} attribute marked in this way is mapped to a stereotype attribute that is defined as 'read-only' and 'derived', and the `oclSpecification` of the applied `<<ToTaggedValue>>` stereotype is employed to define the *defaultValue*. For instance, the MC_{St} attributes `dataType` and `memberAssignment` in Fig. 7a have the `<<ToTaggedValue>>` stereotype applied.



(a) Metaclasses that defines TDL's data type concept



(b) Metaclasses that define TDL's test descriptions

Fig. 7 An excerpt of the metamodel MM_{TDL}

4.4 Our derivation approach

In the following, we discuss the details of our approach for deriving a UML profile UP_{Domain} from a metamodel MM_{Domain} that is enriched with the UML profile `MM2Profile`. To illustrate certain aspects of our approach, we use the MM_{TDL} metaclasses and their corresponding *Stereotypes* in the UML profile UP_{TDL} shown in Fig. 7. In addition to explaining our derivation approach by examples in this paper, we detail the various derivation steps using pseudocode in [29].

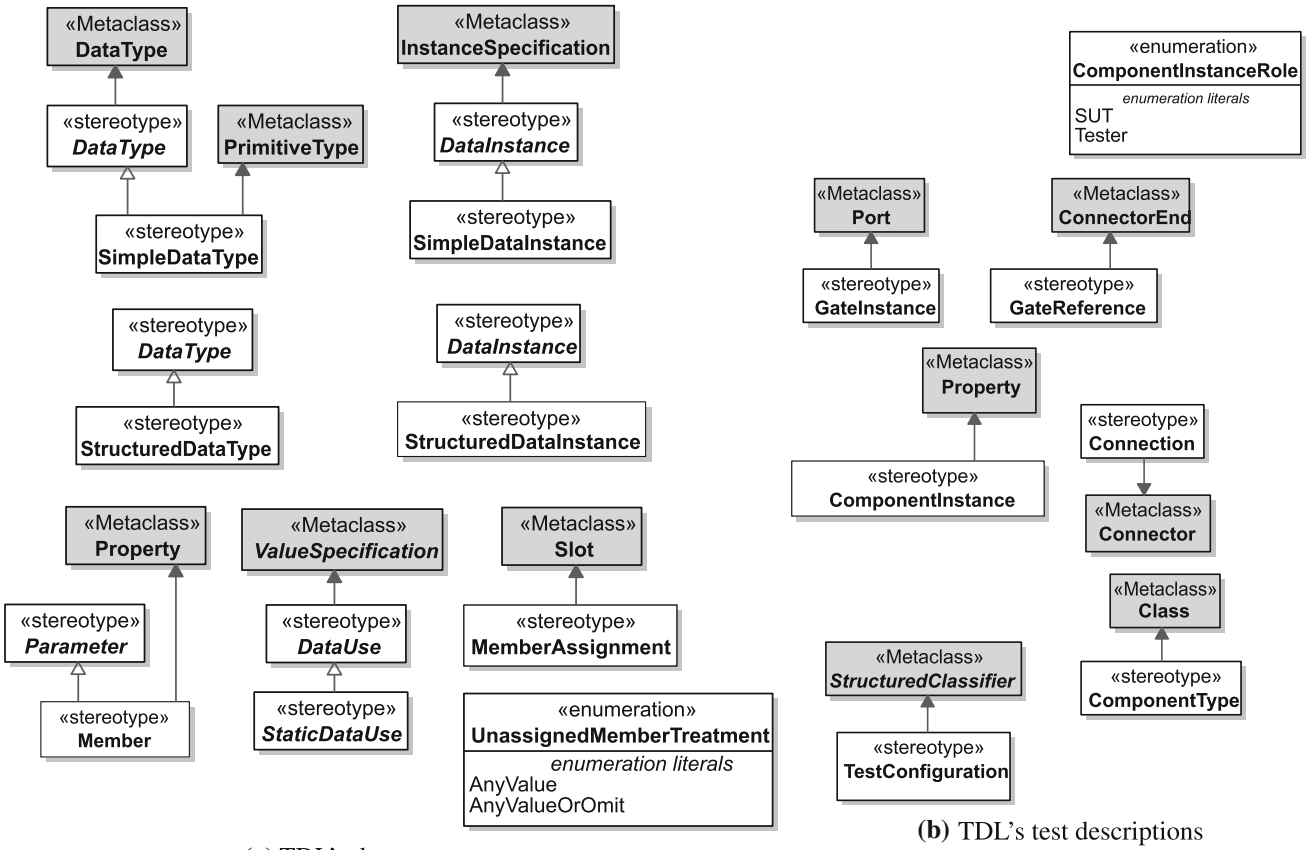
4.4.1 Mapping to 'Stereotypes'

The first step of our approach to derive a UML profile UP_{Domain} from a metamodel MM_{Domain} consists in the creation of *Stereotypes*. Therefore and as argued for Design Decision 3, we create a *Stereotype* and map various properties (e.g. *name* and *isAbstract*) of the source MC_{St} one-to-one to its corresponding properties, while a few properties must be processed in a particular manner. Further details concerning this topic are discussed below.

If an MC_{St} has applied the `<<ToStereotype>>` stereotype, the mapping to a *Stereotype* can be omitted due to the `noMapping` attribute. For example, when comparing Figs. 7a and 8a, we find that the latter does not contain a stereotype for the MC_{St} `MappableDataElement`. This is because we have applied the `<<ToStereotype>>` stereotype to this metaclass and assigned value 'true' to the `noMapping` attribute.

Relationships to super-types. After the stereotypes are created, we introduce inheritance relationships (i.e. a *Generalization*) between them and their super-types. However, according to Design Decision 5, a *Generalization* must be introduced only for those *Stereotypes* created from MC_{St} metaclasses that inherit from other MC_{St} metaclasses.

For the reason above, we identify the set of super-types for a *Stereotype* `ST` based on the *generalization* property of its source MC_{St} metaclass, where we regard only MC_{St} metaclasses that have not applied the `<<OmitGernalization>>` stereotype. Based on the resulting set of MC_{St} metaclasses, we identify the corresponding *Stereotypes* in UP_{Domain} and assign them to the *superClass* property of *Stereotype* `ST`. This also causes an



(a) TDL's data type concept **(b)** TDL's test descriptions

Fig. 8 Introduced *Extensions* and *Generalizations* in UP_{TDL}

implicit creation of *Generalization* relationships between *Stereotype* `ST` and the super-types.

An example is shown in Fig. 8a, where the `<<SimpleDataInstance>>` stereotype inherits from the `<<DataInstance>>` stereotype. This is because the two source metaclasses in Fig. 7a are MC_{St} metaclasses that are in an inheritance relationship.

Extension relationships to UML metaclasses. Apart from *Generalizations*, we create an *Extension* association between a mapped *Stereotype* `ST` and the UML metaclasses to extend, which can be determined in two ways. Implicitly, we can identify them based on generalization relationships between MC_{AC} and MC_{ST} metaclasses in MM_{Domain} (Design Decision 4). However, this can be overridden by explicitly specifying the UML metaclasses of interest (Design Decision 9) using the `<<ToStereotype>>` stereotype.

To obtain the set of UML metaclasses for a *Stereotype* `ST`, we employ its source metaclass MC_{St} `SM` in MM_{Domain}. First, we check whether UML metaclasses are explicitly defined via the `<<ToStereotype>>` stereotype, and if so, we use these UML metaclasses as extension target for stereotype `ST`. Otherwise, we determine the MC_{AC} metaclasses that are super-types of the MC_{St} `SM`. Then, we resolve the 'matching' UML metaclasses for that MC_{AC} metaclasses. Afterwards, we introduce an *Extension* between stereotype `ST` and each identified UML metaclass.

For example, the *Extensions* of the stereotypes `<<DataUse>>` and `<<Parameter>>` shown in Fig. 8 are create because the source MC_{St} metaclasses in Fig. 7 inherit from MC_{AC} metaclasses. In contrast, the *Extensions* of all remaining stereotypes in Fig. 8 are created based on an explicit specification using the `<<ToStereotype>>`, which is applied to the source MC_{St} metaclasses.

4.4.2 Mapping of *DataTypes*

Apart from *Stereotypes*, a UML profile can also introduce different kinds of data types, which can be *DataTypes*, *PrimitiveTypes*, or *Enumerations*. For the derivation of a UML profile, we determine those data types of MM_{Domain} that do not represent 'Abstract Concepts' and that are not mapped to data types of MM_{Add}. Then, we copy these data types one-to-one into the UML profile.

For instance, the *Enumerations* `UnassignedMemberTreatment` and `ComponentInstanceRole` in Fig. 7 do neither represent 'Abstract Concepts' nor data types that

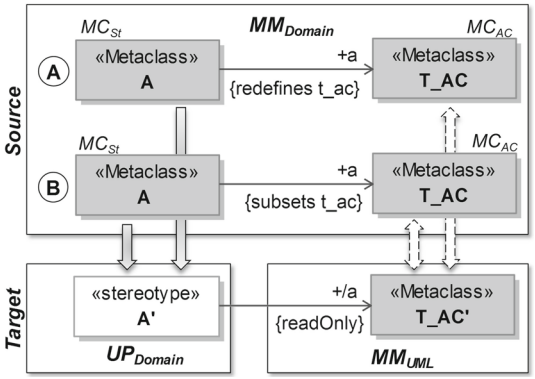


Fig. 9 Mapping to derived attributes

map to a metamodel MM_{Add}. Thus, they are copied one-to-one to UP_{TDL} as shown in Fig. 8.

4.4.3 Mapping to stereotype attributes

According to Design Decision 6, we map an MC_{St} metaclass attribute to a corresponding attribute of a *Stereotype*. However, we must also respect UML's extension mechanism for *Stereotypes*. Therefore, we cannot simply map every kind of metaclass attribute to a corresponding stereotype attribute. The most important rule in this context is that subsetting and redefinition are only applicable to stereotype attributes as long as no attribute of a UML metaclass is involved.

Subsetting and redefining metaclass attributes. Due to the reason above and according to Design Decision 7, we map a redefining or subsetting metaclass attribute to a 'derived' and 'read-only' stereotype attribute, and we introduce an OCL expression so that the attribute's value is automatically computed at runtime. To explain this mapping in more detail, we employ the example given in Fig. 9, where we distinguish two cases.

In Case (A) of our example, an MC_{St} `A` with an attribute `a` and an MC_{AC} `T_AC` are given. Furthermore, attribute `a` redefines attribute `t_ac`, and its *type* property also refers to `T_AC`. We obtain a 'read-only' and 'derived' attribute `a` of stereotype `A'` as mapping result, and the *type* property of `a` now refers to the 'matching' UML metaclass `T_AC'`. In addition, an OCL expression is introduced to define the *defaultValue* of `a`. Case (B) differs from (A) only in the detail that attribute `a` of MC_{St} `A` is subsetting attribute `t_ac`, instead of redefining it.

In both cases, we also create an OCL expression and assign it to the *defaultValue* property of a mapped stereotype attribute, as described in Sec. 4.5.

For example, see the `type` attribute of the MC_{St} metaclass `ComponentInstance` in Fig. 7b. As it redefines the `type` attribute of the MC_{AC} `AC_TypedElement`, it is mapped to a 'derived' and 'read-only' attribute of the stereotype `<<ComponentInstance>>` in Fig. 10b.

In addition to the scenario above, we also create 'derived' and 'read-only' stereotype attributes for all MC_{St} attributes that have an 'alternative' OCL specification (Design Decision 10). This can be defined via the `oclSpecification` attribute of the `<<ToTaggedValue>>` stereotype applied to an MC_{St} attribute in MM_{Domain}. During the mapping, we first create the stereotype attribute, and then we assign the given OCL specification to the attribute's *defaultValue* property.

For instance, see the various MC_{St} attributes with applied `<<ToTaggedValue>>` stereotype in Fig. 7b. According to Design Decision 10, these attributes are mapped to 'derived' and 'read-only' stereotype attributes shown in Fig. 10b.

Other metaclass attributes. All remaining kinds of MC_{St} attributes are mapped one-to-one to corresponding stereotype attributes, including those MC_{St} attributes that are redefining/subsetting other MC_{St} attributes, or that are specified to be a superset. Because only stereotype attributes are involved, no UML restrictions apply and we can preserve such relationships. To do so, we usually map all properties of an MC_{St} attribute to corresponding properties of the stereotype attribute to be created.

However, an exception are the properties *type* and *aggregation*. The mapping of the former property depends on whether the *type* property of the MC_{St} attribute in question refers to a metaclass. If this is the case, the *aggregation* property of a created stereotype attribute must be set to the value 'none', as we argued for Design Decision 6. Because the mapping rules for the *type* property are more complex, we discuss this separately in the next section.

4.4.4 Recomputing the type property

As we argued for Design Decision 8, a stereotype attribute shall not have a stereotype as its *type*. Therefore, we recompute the *type* property of a created stereotype attribute during the mapping; but for this, we must consider that a metamodel MM_{Domain} can contain three types of metaclasses (Design Decision 1) that are mapped in different ways. In the following, we analyse the different cases.

Let `A` be an MC_{St} in MM_{Domain} and `A'` its corresponding *Stereotype* in UP_{Domain}. In addition, assume that MC_{St} `A` has an attribute `att` that maps to a corresponding attribute `att'` of `A'`, and the *type* property of `att` can refer to an MC_{AC} (Case A), an MC_{St} (Case B), or an MC_{AMC} (Case C) metaclass, as shown in Fig. 11. We determine the *type* property of `att'` as follows:

Case A: By Design Decision 2, we assume that a 'matching' UML metaclass exists for each MC_{AC} metaclass

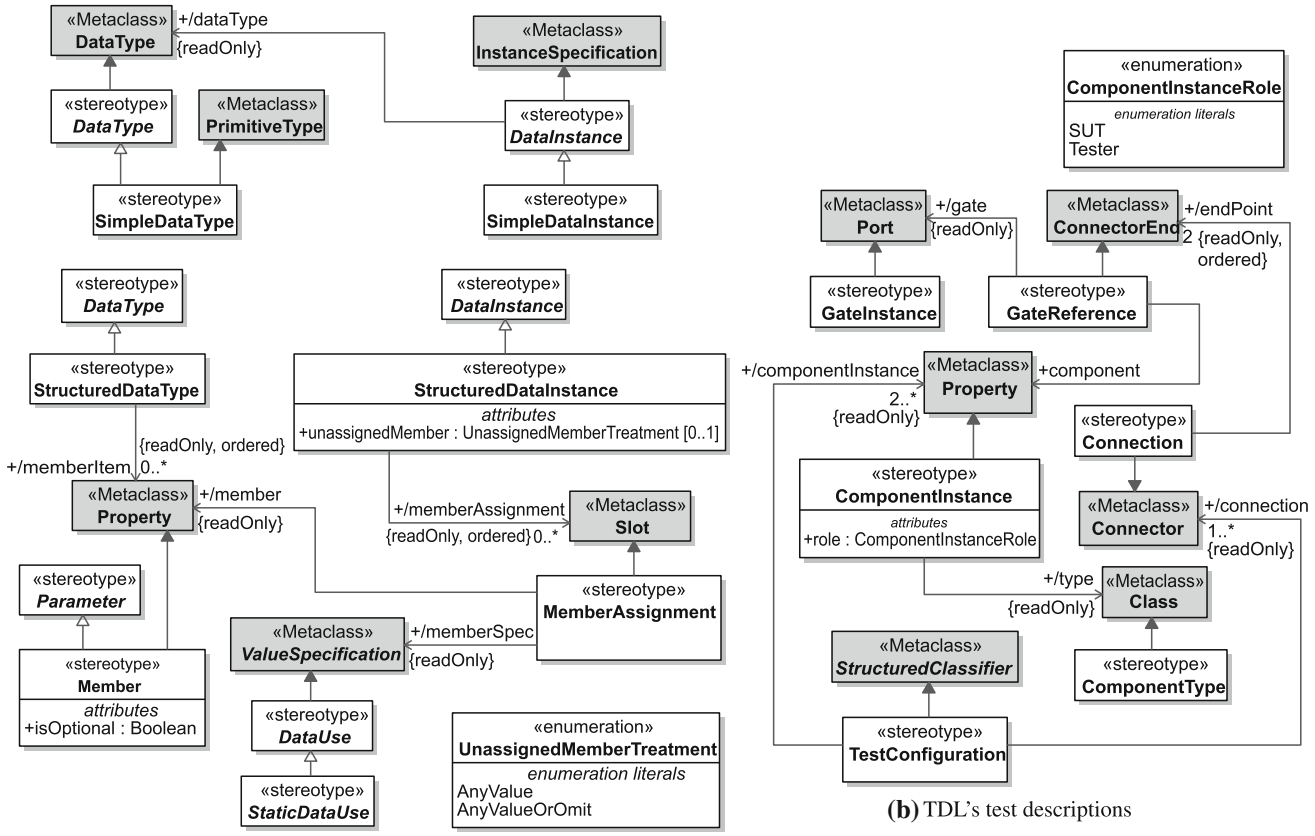


Fig. 10 Introduced *Properties* and *Associations* in UP_{TDL}

in MM_{Domain}. Hence, when we recompute a *type* property that refers to an MC_{AC}, we determine the 'matching' UML metaclass and use it as the new *type*.

For example, consider attribute `att` of MC_{St} `A` in Fig. 11a as input, and assume that the *type* property of `att` refers to an MC_{AC} `T_AC` in MM_{Domain}. Then, the recomputed *type* of `att'` refers to metaclass `T_AC'` in MM_{UML}.

Case B: If the *type* of an attribute refers to an MC_{St}, the *type* of the mapped attribute must refer to a UML metaclasses, as we argued for Design Decision 8. Thus, when we recompute a *type* property that refers to an MC_{St} metaclass, we determine a UML metaclass rather than the *Stereotype* that extends it.

For instance, consider attribute `att` of MC_{St} `A` in Fig. 11b as input, and assume the *type* property of `att` refers to MC_{St} `T_ST` in MM_{Domain}. Then, the recomputed *type* of `att'` refers `T_AC'` in MM_{UML}.

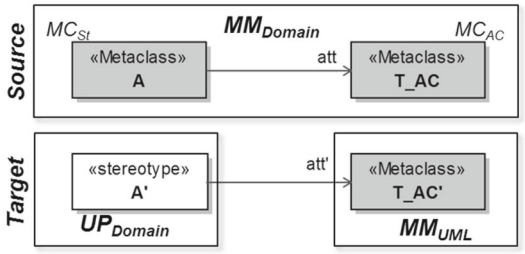
Case C: An MC_{AMC} in MM_{Domain} maps to an 'additional' metaclass in MM_{Add} (Design Decision 1). If such an MC_{AMC} is referenced as *type* of a metaclass attribute in MM_{Domain}, then only the corresponding metaclass in MM_{Add} has to be determined and used as *type* of the mapped attribute.

For example, let attribute `att` of MC_{St} `A` in Fig. 11c be the input for the recomputation, and assume that the *type* of `att` refers to the MC_{AMC} `T_AMC` in MM_{Domain}. Then, the recomputed *type* of `att'` refers to `T_AMC'` of MM_{Add}.

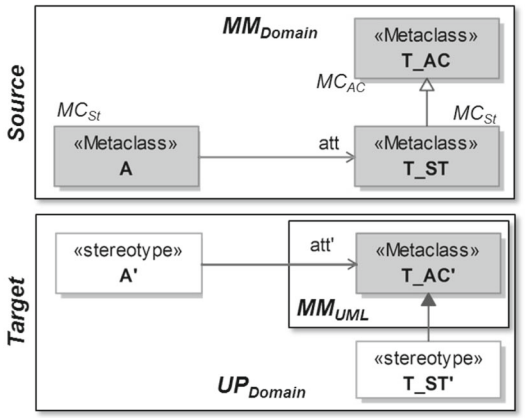
All metaclass attributes that are implicitly defined via association ends in Fig. 10 are an example of Case B because their *type* properties refer to UML metaclasses extended by *Stereotypes*.

In addition to metaclasses, a metamodel MM_{Domain} may contain data types, which we copy one-to-one to the UML profile UP_{Domain} or the metamodel MM_{Add}. Because such data types can also be used as attribute type in MM_{Domain}, we must recompute the *type* of a mapped attribute in this case, too. But unlike metaclasses, we only determine whether a data type is contained in UP_{Domain}, MM_{Add} or MM_{UML}; then, we use it as the new *type* of a mapped attribute.

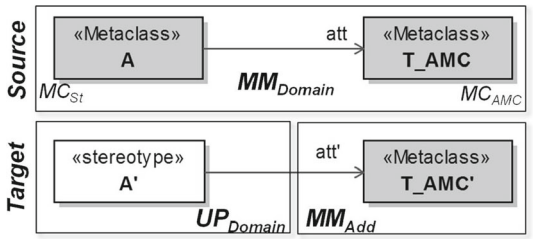
The `unassignedMember` attribute of the `<<StructureDataInstance>>` stereotype and the



(a) An MC_{AC} metaclass is used as attribute type



(b) An MC_{St} metaclass is used as attribute type



(c) An MC_{AMC} metaclass is used as attribute type

Fig. 11 Variants for recomputing the attribute type

`role` attribute of the `<<ComponentInstance>>` stereotype in Fig. 10 are examples for the described recomputation of an attribute type that refers to a data type.

4.4.5 Mapping to stereotype operations

In addition to attributes, a metaclass MC_{St} can also have *Operations* specified, which we map to corresponding items of a created *Stereotype* in UP_{Domain}. To do so, we map most of the properties of an MC_{St} operation one-to-one, excluding the *redefinedOperation* property and the operation's parameters.

Similar to attributes and because of the restrictions specified for UML profiles, an operation of a *Stereotype* is not permitted to redefine an operation of a UML metaclass. In contrast, no restrictions exist for a redefinition between stereotype operations. For these reasons, when mapping an

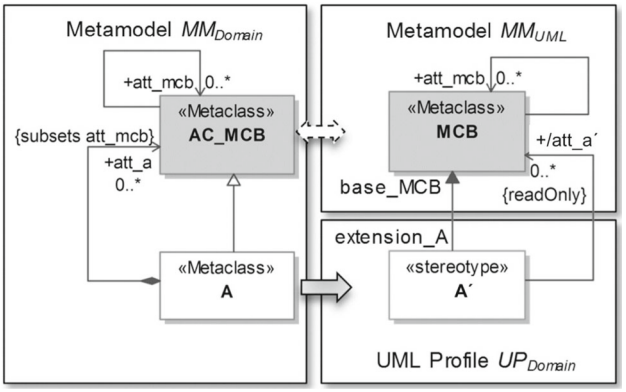


Fig. 12 Subsetting MC_{St} attribute and the corresponding 'derived' and 'read-only' stereotype attribute

MC_{St} operation to a corresponding stereotype operation, we only process those items of the *redefinedOperation* property, which refer to other MC_{St} operations. Consequently, a created stereotype operation in UP_{Domain} only redefines operations of other stereotypes, but not of UML metaclasses.

As discussed earlier, the *type* property must be recomputed while creating a stereotype attribute. Because a *Parameter* of an *Operation* also has a *type*, we recompute this property in the same way.

4.5 OCL expressions for stereotype attributes

By Design Decision 7, we map a redefining or subsetting MC_{St} attribute to a 'derived' and 'read-only' stereotype attribute. Consequently, the value of this attribute is computed at runtime based on its *defaultValue* property. To enable this computation, we introduce an OCL expression for each of these attributes. However, as argued for Design Decision 10, the automatic creation of OCL expressions is restricted to the subsetting or redefinition of a single attribute only. In all other cases, an OCL expression must be provided using the `<<ToTaggedValue>>` stereotype.

Assume we have given an MC_{St} `A` that inherits from an MC_{AC} `AC_MCB`, and `A` has an attribute `att_a` that is subsetting an attribute `att_mcb` of `AC_MCB`, as shown in Fig. 12. According to our mapping rules, `A` is mapped to a *Stereotype* `A'`, and `att_a` maps to a 'derived' and 'read-only' stereotype attribute `att_a'`. Because `AC_MCB` has a 'matching' UML counterpart `MCB`, `AC_MCB` is not mapped. In addition, we introduce an *Extension* between *Stereotype* `A'` and UML metaclass `MCB`, which implies the creation of the attributes `extension_A` and `base_MCB`. However, the subsetting relationship of `att_a` is not preserved for `att_a'`; instead, we create an OCL expression so that the attribute's value is computed at runtime.

In the created OCL expression, we first navigate from the instance of `A'` to the instance of `MCB` by employing attribute

base_MCB. Then, we access attribute att_MCB to collect all items required to compute the value of attribute att_a'. We select these items either based on their applied stereotypes or according to their type. The latter option is applied if the *type* property of att_a' refers to an 'additional' metaclass. Finally, we typecast the set of all determined values so that they match the type and cardinality of att_a'.

A generated OCL expression as a substitution for redefining or subsetting MC_{St} attributes always consists of the following parts:

- Navigation to UML metaclass MCB that is extended by a *Stereotype* ST;
- Navigation to attribute att_src of MCB, so this attribute serves as the source for the value computation;
- Selection of all relevant items of att_src based on their applied stereotype or element type;
- Type-cast of the selected items to match the *type* and cardinality of the stereotype attribute att_st.

We generate all OCL expressions according to two patterns. The first pattern is used for stereotype attributes that have an upper cardinality one, while the second pattern is employed in all other cases:

(1) **self**.base_<MCB>.<att_src>−>any(<sel_exp>).<type−cast>
(2) **self**.base_<MCB>.<att_src>−>select(<sel_exp>)−><type−cast>

For example, when we employ Pattern (1) for the type attribute of the MC_{St} ComponentInstance in Fig. 7b, we obtain the following OCL expression for the corresponding attribute of <<ComponentInstance>> in Fig. 10b:

```
if self.base_Property.type
  −>one(isStereotypedBy('UP4TDL::ComponentType'))
then self.base_Property.type−>any(isStereotypedBy(
  'UP4TDL::ComponentType')).oclAsType(UML::Class)
else null endif
```

As the type attribute of the <<ComponentInstance>> stereotype is only single-valued, we evaluate in the first two lines of the generated OCL expression whether an element with applied <<ComponentType>> stereotype is assigned to the type attribute of the UML metaclass Property. If so, we type-cast and return the determined element.

4.6 Additional OCL constraints

Because the *type* properties of stereotype attributes are recomputed, we introduce additional OCL constraints to preserve the static semantics defined by MM_{Domain}. Due to Design Decisions 11 and 12, we distinguish between two categories of OCL constraints. The first is introduced to ensure the well-formedness of stereotype attributes that are not 'derived' and 'read-only'. The second ensures the well-formedness of UML metaclass attributes that are relevant for computing the *defaultValues* of 'derived' and 'read-only' stereotype attributes.

4.6.1 OCL constraints for stereotype attributes

As argued for Design Decision 11, a syntactically invalid value can be assigned to a stereotype attribute that is not defined as 'derived' and 'read-only'. Therefore, we introduce OCL constraints to ensure that only UML elements having applied particular stereotypes can be assigned to such stereotype attributes.

Assume an MC_{St} attribute att, where its *type* property refers to an MC_{St} A. This attribute is mapped to a corresponding attribute att', and a *Stereotype* A' is derived from A. In addition, this stereotype has an *Extension* to a UML metaclass MCB. Furthermore, the *type* property of att' is recomputed so that it refers to MCB. Thus, any kind of MCB instance can be assigned to att'. However, att' is only well formed if all elements assigned to it have A' applied. To ensure this, we create an OCL constraint that consists of the following parts:

- Navigation to stereotype attribute att_st;
- Verification that all items of att_st have stereotype ST applied.

Apart from the components above, the cardinality of the stereotype attribute must also be respected because different constraint kinds have to be used for single-valued and multi-valued attributes:

(1) **self**.<att_st> <> **null implies self**.<att_st>.isStereotypedBy(<ST>)
(2) **self**.<att_st>−>forAll(isStereotypedBy(<ST>))

For instance, the *type* property of the component attribute of the MC_{St} GateReference in Fig. 7b refers to the MC_{St} ComponentInstance. As we recompute the *type* property of stereotype attributes, that of the component attribute of <<GateReference>> refers to the UML metaclass Property. For this attribute, we introduce the following OCL *Constraint* by applying Pattern (1) above:

```
self.component <> null implies
self.component.isStereotypedBy('UP4TDL::ComponentInstance')
```

In Line 1, we evaluate whether a value is assigned to the stereotype attribute. If so, we employ the isStereotypedBy() operation in Line 2 to test whether the element assigned to the component attribute has applied the <<ComponentInstance>> stereotype. Consequently, only elements with applied <<ComponentInstance>> stereotype can be assigned to the component attribute.

4.6.2 OCL constraints for UML metaclass attributes

We introduce a second category of OCL constraints for ensuring the well-formedness of UML metaclass attributes that serve as value computation source for the *defaultValue* of 'derived' and 'read-only' stereotype attributes (Design Decision 12).

Suppose that the same excerpt of a metamodel MM_{Domain} (see Fig. 12) is given as we have employed to explain the generation of OCL expressions. After applying our derivation approach, we obtain a 'derived' and 'read-only' stereotype attribute `att_a'` that has an OCL expression as *defaultValue*. Moreover, an attribute `att_mcb` of a UML metaclass `MCB` serves as basis for computing this OCL expression. To preserve well-formedness, we have to ensure that only permitted elements can be assigned to metaclass attribute `att_mcb`. For the given example, only the assignment of elements with an applied stereotype `A'` is permitted. Hence, we introduce an appropriate OCL *Constraint* that constrains attribute `att_mcb`.

Apart from the given example, a UML metaclass attribute can also be used to compute several stereotype attributes. This is always the case if several MC_{St} attributes in the source model MM_{Domain} are subsetting or redefining the same MC_{AC} attribute. In such a scenario, we obtain a set of valid stereotypes that can be applied to items of the attribute `att_mcb`, and therefore, these also must be verified via a generated OCL constraint, which consists of the following parts:

- Navigation to attribute `att_mcb` of UML metaclass `MCB`, which is extended by *Stereotype* `ST`;
- Verification that each item of `att_mcb` has one of the permitted stereotypes applied.

We employ the following two patterns to generate an OCL *Constraint*. As in the case of the previous constraint category, we must obey the attribute cardinality here:

(1) **self**.base_<MCB>.<att_mcb> <> **null implies**
self.base_<MCB>.<att_mcb>.<verify_exp>
(2) **self**.base_<MCB>.<att_mcb>−>forAll(verify_exp)

For example, the `type` attribute of the MC_{St} `Component Instance` in Fig. 7b redefines an attribute of the MC_{AC} `AC_TypedElement`. As only one attribute of the MC_{St} `ComponentInstance` redefines the `type` attribute of the MC_{AC} `AC_TypedElement`, only one attribute of the `<<ComponentInstance>>` stereotype has an OCL expression for a value computation based on the `type` attribute of UML metaclass `TypedElement`. Consequently, according to Design Decision 12, only elements with applied `<<ComponentType>>` stereotype shall be assignable to this UML metaclass attribute. To ensure this, we create the following OCL constraint by employing Pattern (1):

```
self.base_Property.type <> null implies self.base_Property
  .type.isStereotypedBy('UP4TDL::ComponentType')
```

Line 1 tests whether a value exists at all. If this is the case, starting from an instance of `<<ComponentInstance>>`, we navigate in Line 2 to the `type` attribute and invoke the `isStereotypedBy()` operation to evaluate whether the element assigned to this attribute has the `<<ComponentType>>` stereotype applied.

5 Update of existing OCL expressions

A metamodel includes OCL expressions for different purposes. OCL expressions can be used to define *Constraints* on metaclasses, so that the static semantics of a computer language or DSL is captured. They can also serve as the basis for computing values of metaclass attributes and operations at runtime.

The automatic transfer of the static semantics of a metamodel MM_{Domain} to a derived UML profile UP_{Domain} is a key feature of our approach. Because *Stereotypes* and their extended UML metaclasses exist as separate instances in a model, an automatic transfer of OCL expressions from a metamodel to a UML profile is impossible without updating them at the same time. As argued in the Sec. 2.3, we do not consider the OCL update as a separate process but as an action of our UML profile derivation process.

First, we give an overview of the OCL metamodel as a basis for specifying our design decisions for the OCL update. Then, we discuss the details for updating OCL expressions, and finally, we elaborate on the most important aspects of our implementation.

5.1 The OCL metamodel

We briefly introduce the relevant parts of OCL's abstract syntax, which is defined by a MOF-compliant metamodel [45]. The OCL metamodel is divided into several packages, and the most important of them are:

- The 'Expressions' package, which specifies the different OCL expression types;
- The 'Types' package defines OCL's type system, which includes concepts for using predefined types as well as user-defined types introduced by a metamodel.

Because only the OCL metaclasses that define the abstract syntax of OCL expressions are of interest for our OCL update, we just treat some metaclasses of the 'Expressions' package. The basic structure of the abstract syntax of OCL

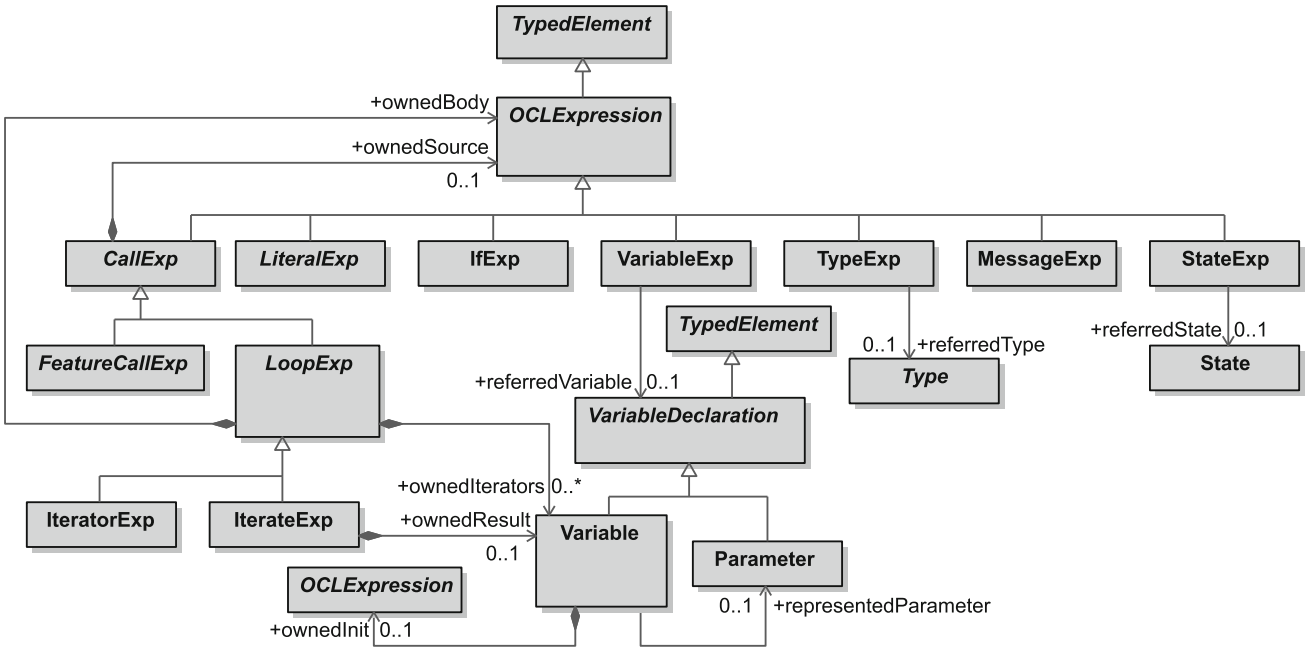


Fig. 13 Basic structure of OCL's abstract syntax

expressions is shown in Fig. 13. OCL is a 'typed language'. Therefore, the most general metaclass *OCLExpression* inherits from the *TypedElement*, so that all OCL expressions own the *type* property. Typically, an abstract syntax tree (AST) of an analysed OCL expression consists of any number of nested OCL expression instances. Each of these instances has a static type specified, which is determined by a recursive analysis of all nested expressions. The result value of an expression is determined by performing an evaluation, and the obtained value must conform to the static type of the expression.

The objectives of the metaclasses shown in Fig. 13 are explained below. Only the metaclasses *MsgExp* and *StateExp* are excluded, because they cannot be used for OCL expressions contained in metamodels.

CallExp is used to obtain the evaluation result of an operation or attribute of a classifier or the result of a collection type iterator. The purpose of a *CallExp* is defined by its concrete subtypes.

FeatureCallExp is employed to determine the evaluation result of an attribute or operation of a classifier. This abstract OCL metaclass is more precisely defined by further subtypes, see below.

LiteralExp represents a literal of a primitive type such as `Integer` or `String`.

TypeExp refers to a classifier in a model or a predefined type. Typically, a *TypeExp* is passed as argument to invoke one of the predefined OCL operations `oclAsType()`, `oclIsTypeOf()`, or `oclIsKindOf()`.

VariableDeclaration is a super-type of *Variable* and *Parameter*. *Variable* represents user-defined variables, and *Parameter* specifies operation parameters.

VariableExp is a reference to an explicitly declared variable, an operation parameter, or the implicitly introduced variables 'self' and 'result'.

LoopExp iterates over all items of a collection type. The invocation of a predefined collection iterator is represented by *IteratorExp*, which is a concrete subtype of *LoopExp*.

IfExp always consists of two mandatory alternative expressions and a `Boolean` condition. Depending on the condition's result, one of the alternative expressions is evaluated and the obtained result is returned.

The subtypes of the abstract OCL metaclass *FeatureCallExp* are shown in Fig. 14. Depending on the subtype, a *NavigationCallExp* represents a reference to a classifier attribute or an association class. Because the latter element type is not applicable to OCL expressions of metamodels, we do not consider the metaclass *AssociationClassCallExp*

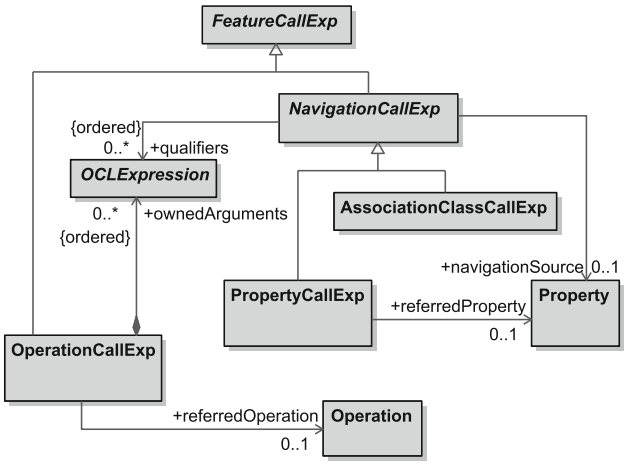


Fig. 14 Abstract syntax of OCL's *FeatureCallExp*

further. The *PropertyCallExp* is used to evaluate the value of a classifier attribute, and the *referredProperty* property is the reference to this attribute.

Another subtype of the *FeatureCallExp* metaclass is the *OperationCallExp* that is employed to evaluate the result of an operation invocation. The *referredOperation* property of this metaclass identifies the operation to be invoked, and the items of the *ownedArguments* property define the values that are passed to the operation's parameters.

5.2 Design decisions for updating OCL expressions

Based on our design decisions for deriving UML profiles specified in Sec. 4.1, we map MC_{St} metaclasses of the metamodel MM_{Domain} to corresponding *Stereotypes* of the UML profile UP_{Domain}, and MC_{St} attributes to stereotype attributes. Furthermore, we map *Operations* and *Constraints* of MC_{St} metaclasses to corresponding counterparts in *Stereotypes*. As the result of the mappings, all elements of an MC_{St} are owned by *Stereotypes*. The 'self' variable of an OCL expression contained in a *Stereotype* always refer to this *Stereotype* instead of a metaclass. If a feature of the extended UML metaclass is accessed based on the 'self' variable, the *'base_<metaclass>'* attribute must be accessed first when navigating to that metaclass feature.

Hence, we must update all OCL expressions that access an MC_{AC} attribute or MC_{AC} *Operation* based on a 'self' variable that refers to an MC_{St}. During this update, we introduce a *PropertyCallExp* that accesses the attribute *'base_<metaclass>'*.

Design Decision 13 *A PropertyCallExp that accesses the 'base_<metaclass>' attribute shall be introduced for OCL expressions that access an MC_{AC} attribute or MC_{AC} Operation via a 'self' variable that refers to an MC_{St}.*

The *type* properties of attributes are recomputed during the mapping, so that only *Classifiers* (*Classes* or *DataTypes*) of the MM_{UML} or MM_{Add} metamodel are referenced (Design Decision 9). Moreover, the same recomputation is applied to the *type* property of an operation *Parameter*. Thus, neither the *type* properties of stereotype attributes nor those of operation parameters refer to *Stereotypes*. This situation also applies to attributes and operation parameters of 'additional' metaclasses of MM_{Add}. Thus, accessing mapped stereotype attributes or operation parameters always returns a metaclass instance.

If a *Property* or *Operation* of a *Stereotype* shall be accessed based on the result returned by an OCL expression, then the *'extension_<stereotype>'* attribute has to be employed for navigating to the stereotype feature. For this reason, we update all OCL expressions that contain a navigation from one MC_{St} feature to another MC_{St} feature, and during this update we introduce a *PropertyCallExp* that accesses the *'extension_<stereotype>'* attribute.

Design Decision 14 *A PropertyCallExp that accesses the 'extension_<stereotype>' attribute shall be introduced for OCL expressions that contain a navigation from one MC_{St} feature to another.*

The predefined OCL operations `oclIsTypeOf()` and `oclIsKindOf()` are used to verify whether the evaluation result of an OCL expression is an instance of the *Classifier* defined as argument. The `oclIsTypeOf()` operation returns only value 'true', if the evaluation result exactly matches the type specified as argument, while the `oclIsKindOf()` operation returns 'true' also for subtypes.

Because the *type* property of stereotype attributes and operation parameters is recomputed during the mapping, OCL expressions that access these elements can never return a stereotype instance as result. Thus, we cannot utilize the aforementioned operations to verify whether a particular stereotype is applied. Instead, we employ the operations `isStrictStereotypedBy()` and `isStereotypedBy()`. We update an *OperationCallExp* that invokes the `oclIsTypeOf()` or `oclIsKindOf()` operation, passing an MC_{St} metaclass as argument. During this update, we introduce a new *OperationCallExp* that invokes the operation `isStrictStereotypedBy()` or `isStereotypedBy()`.

Design Decision 15 *An OperationCallExp that invokes the* `oclIsTypeOf()` *or* `oclIsKindOf()` *operation, passing an MC_{St} as argument, shall be replaced by an OperationCallExp that invokes the operation* `isStrictStereotypedBy()` *or* `isStereotypedBy()`.

The elements of MM_{Domain} are located in UP_{Domain}, MM_{Add}, and MM_{UML} after the mapping. Because a *Type-*

Exp refers to a particular *Classifier* of a metamodel, we have to update all OCL expressions of that type, so that they refer to elements in MM_{Add} or MM_{UML}. The *Stereotypes* of UP_{Domain} are not considered for this update, because the *type* properties of stereotype attributes and parameters are recomputed during the mapping so that *Stereotypes* are not referenced.

Design Decision 16 *All TypeExp shall be updated so that they refer to Classifiers of* MM_{UML} *or* MM_{Add}.

5.3 The OCL update in detail

We now detail our approach for updating OCL expressions contained in the metamodel MM_{Domain}. We assume that a textually specified OCL expression is parsed and the resulting abstract syntax tree (AST) is present. This AST is an instantiation of OCL's metamodel.

FeatureCallExp. By Design Decisions 13 and 14, an additional *PropertyCallExp* that refers to the *'base_<metaclass>'* or *'extension_<stereotype>'* attribute must be introduced for an existing *FeatureCallExp* in certain situations. However, this applies only to instances of *PropertyCallExp* and *OperationCallExp*, because only these subtypes of *FeatureCallExp* are applicable in the context of metamodels.

Following Design Decision 13, we create an additional *PropertyCallExp* to access an *'extension_<stereotype>'* attribute if all criteria below are met:

1. The *type* of the *ownendSource* expression of a *FeatureCallExp* refers to an MC_{St} metaclass;
2. The *referredProperty* of a *PropertyCallExp* is an MC_{St} attribute, or in case of an *OperationCallExp*, the *referredOperation* is an MC_{St} operation.

Assume that the *PropertyCallExp* shown below is used as input for the OCL update, and that the above criteria are met. After applying the update, we obtain the result shown in the second line, where the `<stereotype>` placeholder represents the name of the *Stereotype* that owns the *referredProperty*:

input: source.referredProperty
result: source.extension_<stereotype>.referredProperty

For example, the update defined by Design Decision 13 is applied to the constraint of the MC_{St} `StructuredDataType` in Fig. 7a. The `membersAreDistinguishable()` operation that is invoked by this constraint is inherited from MC_{AC} `AC_Namespace`. In UP_{TDL}, this operation is not inherited by the derived `StructuredDataType` stereotype, but we can invoke it via the extended UML metaclass. For this purpose, we introduce an *PropertyCallExp* for the attribute `base_DataType` based on Design Decision 13:

input: **self**.membersAreDistinguishable()
result: **self**.base_DataType.membersAreDistinguishable()

Due to Design Decision 14, another kind of update of an existing *FeatureCallExp* is required in order to access a UML metaclass feature based on a 'self' variable that refers to a *Stereotype* instance. We create an additional *PropertyCallExp* that refers to the *'base_<metaclass>'* attribute, so we can access the UML metaclass feature. We only conduct this update if all following criteria are met:

1. The *ownendSource* expression of a *FeatureCallExp* is a *VariableExp* that refers to the 'self' variable, and the *type* of this *VariableExp* expression refers to an MC_{St} metaclass;
2. The *referredProperty* of a *PropertyCallExp* is an MC_{AC} attribute, or in case of an *OperationCallExp*, the *referredOperation* is an MC_{AC} operation.

Assume that the *PropertyCallExp* shown below is used as input for the OCL update, and the above criteria are met. The result of this update is shown in the second line. The `<metaclass>` placeholder corresponds to the name of the UML metaclass that owns the *referredProperty*:

input: **self**.referredProperty
result: **self**.base_<metaclass>.referredProperty

For example, the MC_{St} `StructuredDataInstance` in Fig. 7a has defined the constraint below, which includes two sub-expressions that meet the discussed criteria of Design Decision 14. Hence, we introduce a *PropertyCallExp* for both attributes `extension_StaticDataUse` and `extension_Parameter`.

input: **self**.memberSpec.determinedType = **self**.member.dataType
result: **self**.memberSpec.extension_StaticDataUse.determinedType = **self**.member.extension_Parameter.dataType

OperationCallExp. The predefined OCL operations `oclIsTypeOf()` and `oclIsKindOf()` can be employed to determine whether the result type of an expression matches the expected type. By Design Decision 15, an update of an *OperationCallExp* is required if one of the operations is applied to an MC_{St} metaclass. We conduct this update as follows:

- The referred operation `oclIsTypeOf()` of an *OperationCallExp* is replaced by the `isStrictStereotypedBy()` operation, if the passed argument is an MC_{St} metaclass.
- The operation `oclIsKindOf()` that is referred by an *OperationCallExp* is replaced by the `isStereotypedBy()` operation, if the passed argument is an MC_{St} metaclass.

For instance, suppose the two OCL expressions below are used as input for the update, and both expressions meet the above criteria. In Case (A), we introduce the operation `isStrictStereotypedBy()`, whereas the operation `isStereotypedBy()` is employed in Case (B). The placeholder `<stereotype>` represents the stereotype name that is derived from the MC_{St} shown in the input expressions:

input:
(A) source.oclIsTypeOf(MCSt)
(B) source.oclIsKindOf(MCSt)
result:
(A) source.isStrictStereotypedBy(<stereotype>)
(B) source.isStereotypedBy(<stereotype>)

Below, for example, we apply the OCL update according to Design Decision 15 to a constraint of the MC_{St} `SimpleDataInstance` in Fig. 7a. As shown, the call of the operation `oclIsKindOf()` is replaced by an invocation of the operation `oclIsStereotypedBy()`. In addition, the MC_{St} `SimpleDataType` specified as argument is substituted with the qualified name of the corresponding *Stereotype*:

input: **self**.dataType.oclIsKindOf(SimpleDataType)
result: **self**.dataType.isStereotypedBy('UP4TDL::SimpleDataType')

TypeExp. According to Design Decision 16, we have to update all occurrences of *TypeExp* that refer to an MC_{AC} or an MC_{St} metaclass. A *TypeExp* that refers to an MC_{AC} metaclass is updated so that the 'matching' UML counterpart of the MC_{AC} is referenced. Furthermore, a *TypeExp* that refers to an MC_{AMC} metaclass is updated so that it refers to an 'additional' metaclass.

We explain the application of the update based on the `getTestDescription()` operation of the MC_{St} `AtomicBehaviour`. Line 3 of this operation invokes the operation `oclAsType(T)`, where a *TypeExp* is passed as argument `T` that refers to the MC_{St} `TestDescription`. After conducting our update, the UML metaclass `BehavioredClassifier` is passed as argument, as required by Design Decision 16.

input:
if self.allNamespaces()−>one(oclIsTypeOf(TestDescription))
then self.allNamespaces()−>any(oclIsTypeOf(TestDescription))
.oclAsType(TestDescription)
else null endif

result:
if self.base_InteractionFragment.allNamespaces()−>one(
isStrictStereotypedBy('UP4TDL::TestDescription'))
then self.base_InteractionFragment.allNamespaces()−>any(
isStrictStereotypedBy('UP4TDL::TestDescription')
).oclAsType(UML::BehavioredClassifier)
else null endif

6 Case study-based evaluation

We employ the Test Description Language (TDL) as a running example to explain the details of deriving a UML profile from a metamodel. Below, we detail our case study on TDL, where we fully automatically derive a UML profile from TDL's metamodel using our derivation approach. First, we investigate whether all elements of the UML profile UP_{TDL} are created as expected. Then, we compare our UP_{TDL} with the UML profile [6] standardized for TDL. In this way, we assess the quality of UML profiles generated via our approach against manually created ones.

6.1 Evaluation of the generated UML profile UP_{TDL}

In Table 2, we compare the metaclasses of MM_{TDL} with the stereotypes that we have derived for the UML profile UP_{TDL}. Because of Design Decision 1, we do not map MC_{AC} metaclasses to model elements of UP_{TDL}. Accordingly, the 13 MC_{AC} metaclasses have no counterparts in UP_{TDL}.

The `<<ToStereotype>>` stereotype is applied to 59 of the 80 MC_{St} metaclasses in MM_{TDL}, mainly to define the UML metaclasses to be extended by generated *Stereotypes* (Design Decision 9). Based on the number of MC_{St} metaclasses in MM_{TDL}, one could assume that UP_{TDL} contains a total of 80 *Stereotypes*. As shown in Table 2, only 79 *Stereotypes* are created for UP_{TDL}. This corresponds to the desired result, because we have defined the MC_{St} `MappableDataElement` in MM_{TDL} (see Fig. 7a) as not to be mapped.

Stereotype extensions and generalizations. To explain more clearly for which kinds of MC_{St} metaclasses *Stereotypes* are introduced either with *Extensions* or *Generalizations*, we distinguish two MC_{St} groups in Table 2:

For MC_{St} metaclasses that inherit directly from MC_{AC} metaclasses, we introduce *Extensions* between generated *Stereotypes* and UML metaclasses identified based on the MC_{AC} metaclasses (Design Decision 4). Because one of the 33 MC_{St} metaclasses that inherit directly from MC_{AC} metaclasses is specified as not mappable, we derive only 32 *Stereotypes* with an *Extension* to a UML metaclass. 29 of these MC_{St} metaclasses have the `<<ToStereotype>>` stereotype applied, so that the UML metaclasses to be extended are explicitly specified (Design Decision 9) and not determined based on MC_{AC} metaclasses.

The second MC_{St} group summarizes all MC_{St} metaclasses that inherit from other MC_{St} metaclasses by means of *Generalization*. This group comprises 47 MC_{St} metaclasses of which 30 have the `<<ToStereotype>>` stereotype applied. For all *Stereotypes* that are created from these 47 MC_{St} metaclasses, we introduce *Generalizations* to other *Stereotypes* (Design Decision 5). In addition, we create 30 *Extension* relationships from *Stereotypes* to UML

Table 2 Metaclasses in MM_{TDL} vs. stereotypes in UP_{TDL}

MM_{TDL}		UP_{TDL}	
MC_{AC} metaclasses	13	–	
MC_{St} metaclasses in total	**80**	**79**	***Stereotypes in total***
MC_{St} inheriting from MC_{AC} (29 MC_{St} with applied `<<ToStereotype>>` stereotype)	33	32	*Stereotypes* with *Extensions* to UML metaclasses
MC_{St} inheriting from other MC_{St}	47	17	*Stereotypes* inheriting from other *Stereotypes*
(30 MC_{St} with applied `<<ToStereotype>>` stereotype)		30	*Stereotypes* inheriting from other *Stereotypes* and *Extensions* to UML metaclasses

metaclasses, because the source MC_{St} metaclasses have applied the `<<ToStereotype>>` stereotype (Design Decision 9).

Stereotype attributes. Due to the higher number of relevant design decisions, the mapping of attributes is more complex than the one of *Stereotypes*. Accordingly, several types of attribute mappings are considered in Table 3:

'Derived' stereotype attributes: This group summarizes all mapped stereotype attributes that are defined as 'derived' and 'read-only'.

'Non-derived' stereotype attributes: This group captures all stereotype attributes that do not belong to the first group.

As shown in Table 3, MM_{TDL} contains a total of 126 metaclass attributes, of which 99 are considered as MC_{St} attributes that map to 52 'non-derived' and 47 'derived' stereotype attributes. The remaining 27 attributes are owned by MC_{AC} metaclasses and, therefore, are not considered when deriving UP_{TDL}.

To discuss the mapping of MC_{St} attributes to 'derived' or 'non-derived' stereotype attributes, we group them according to their type in Table 3. In addition, we specify how many of the MC_{St} attributes of a group have the `<<ToTaggedValue>>` stereotype applied. This is because the application of this stereotype overrides the default mapping for MC_{St} attributes, so that these attributes are mapped to 'derived' stereotype attributes.

The first group of MC_{St} attributes subsumes all MC_{St} attributes that are already defined as 'derived' and 'read-only' in MM_{TDL}. By default, these MC_{St} attributes are mapped to stereotype attributes also defined as 'derived' and 'read-only'. During the mapping, we copy an OCL expression available as *defaultValue* and subject it to our OCL update. However, if an MC_{St} attribute has the `<<ToTaggedValue>>` stereotype applied, we use the OCL expression defined by this *Stereotype* instead of the existing one in MM_{TDL}.

Overall, MM_{TDL} contains three MC_{St} attributes defined as 'derived' and 'read-only', of which one has applied the `<<ToTaggedValue>>` stereotype. Accordingly, existing OCL expressions are transferred to UP_{TDL} for only two MC_{St} attributes. In the case of the MC_{St} attribute with applied `<<ToTaggedValue>>` stereotype, the OCL expression defined by this *Stereotype* was copied to UP_{TDL}.

The second MC_{St} attribute group contains nine MC_{St} attributes that redefine other metaclass attributes. Eight of these MC_{St} attributes are mapped to 'derived' and 'read-only' stereotype attributes, where six attributes are mapped based on Design Decision 7 and two others based on the applied `<<ToTaggedValue>>` stereotype, as required by Design Decision 10. One further MC_{St} attribute is mapped to a 'non-derived' stereotype because it redefines an MC_{St} attribute and is thus not captured by Design Decision 7.

All attributes that subset other metaclass attributes are captured by the third MC_{St} attribute group. In total, this group comprises 43 MC_{St} attributes, of which 29 have the `<<ToTaggedValue>>` stereotype applied. Based on Design Decision 7, these 29 attributes are mapped to 'derived' and 'read-only' stereotype attributes. In addition, we introduce OCL expressions as *defaultValues*, which are used to compute the values of stereotype attributes at runtime.

One could now expect that all 14 remaining MC_{St} attributes would also be mapped to 'derived' and 'read-only' stereotype attributes according to the rules defined by Design Decision 7. However, this is not the case as we have supplemented these rules during the implementation of our derivation approach, as argued in Sec. 4.4. Therefore, only those MC_{St} attributes that redefine or subset MC_{AC} attributes not declared as 'derivedUnion' or 'read-only', are mapped according to Design Decision 7. Because the 14 remaining MC_{St} attributes do not meet this condition, they are mapped to 'non-derived' stereotype attributes.

The last MC_{St} attribute group comprises all attributes that are not captured by one of the three other groups. This applies to 44 MC_{St} attributes, of which seven have applied the `<<ToTaggedValue>>` stereotype, so they are mapped to seven stereotype attributes that are defined as 'derived' and 'read-only' (Design Decision 10). The 37 remaining

Table 3 Metaclass attributes in MM_{TDL} vs. stereotype attributes in UP_{TDL}

Metaclass attributes	**126**	**Stereotype attributes** 'non-derived'	'derived'
MC_{AC} attributes	27	—	—
MC_{St} attributes in total	**99**	**52**	**47**
MC_{St} attributes defined as 'derived' and 'read-only'	3	—	2
Attributes with <<ToTaggedValue>> stereotype	*(1)*	—	1
Redefining MC_{St} attributes	9	1	6
Attributes with <<ToTaggedValue>> stereotype	*(2)*	—	2
Subsetting MC_{St} attributes	43	14	—
Attributes with <<ToTaggedValue>> stereotype	*(29)*	—	29
Other kind of MC_{St} attributes	44	37	—
Attributes with <<ToTaggedValue>> stereotype	*(7)*	—	7

Table 4 OCL artefacts of MM_{TDL} vs. the ones of UP_{TDL}

MM_{TDL}			UP_{TDL}
OCL *Constraints*	**67**	**113**	**OCL *Constraints***
		67	Copied and updated from MM_{TDL}
		40	Introduced for stereotype attributes that refer to UML metaclasses with applied stereotypes
		6	Created for redefining MC_{St} attributes
OCL *Operations*	**10**	**10**	**OCL *Operations***

Table 5 Comparison of the UML profiles UP_{std} and UP_{TDL}

	UP_{std}	UP_{TDL}
Mappings for UML elements without stereotypes	23	—
Stereotypes	56	79
Stereotype attributes	52	99
OCL *Constraints*	2	113
OCL *Operations*	0	10

metaclass attributes of this group are mapped one-to-one to 'non-derived' stereotype attributes (Design Decision 6).

OCL-defined elements. The metamodel MM_{TDL} contains 67 *Constraints* and ten *Operations* defined via OCL, which we map to corresponding elements in UP_{TDL}. When comparing MM_{TDL} and UP_{TDL}, we find that UP_{TDL} comprises 113 OCL *Constraints* instead of 67. This is because we introduce additional OCL *Constraints* for *Stereotypes* of UP_{TDL} (Design Decisions 11 and 12).

To ensure that only UML elements with certain *Stereotypes* can be assigned to 'non-derived' stereotype attributes, we create appropriate OCL *Constraints*. Table 4 shows that UP_{TDL} contains 52 'non-derived' stereotype attributes, but only 40 OCL *Constraints* are created for these. This is because only 40 of the 52 'non-derived' stereotype attributes are derived from MC_{St} attributes whose *type* property refers to an MC_{St} (Design Decision 11).

In addition, further OCL *Constraints* are provided according to Design Decision 12 for 'derived' and 'read-only' stereotype attributes derived from subsetting or redefining MC_{St} attributes. These OCL *Constraints* ensure that only UML elements with matching *Stereotypes* can be assigned to UML metaclass attributes that serve as computation source for the values of the stereotype attributes in question. As six stereotype attributes in UP_{TDL} are derived from redefining MC_{St} attributes, an equal number of OCL constraints is introduced based on Design Decision 12.

6.2 The standardized versus the derived UML profile

In the following, we investigate whether an automatic generated UML profile is comparable to a manually created one. Therefore, we compare our derived UML profile UP_{TDL} with the one standardized for TDL [6]. We refer to the latter UML profile via the acronym UP_{std}. First, we analyse the number of model elements contained in UP_{std} and UP_{TDL}, so that we can draw initial conclusions. Afterwards, we review some *Stereotypes* of both UML profiles to obtain a detailed view on the quality of the UML profile UP_{TDL}.

Quantitative comparison. To compare the various elements in UP_{std} and UP_{TDL}, we summarize their quantities in Table 5. A particularity of UP_{std} is shown in the first table row. UP_{std} defines a mapping to TDL metaclasses for 23 UML element types without defining *Stereotypes*, i.e. the relevant UML model elements are mapped without TDL-specific *Stereotypes* applied to them. The disadvantage of this approach is that no OCL *Constraints* are defined for the affected UML element types due to the missing *Stereotypes*; thus, the static semantics of the relevant language concepts of TDL is not captured.

UP_{std} embraces a lower number of *Stereotypes* than UP_{TDL}, which is due to the above reason. In contrast and except of the `MappableDataElement` metaclass, UP_{TDL} provides a specific *Stereotype* for each MC_{St} metaclass of MM_{TDL}. In addition, the *Stereotypes* of UP_{TDL} have a larger number of attributes. One might assume that this is due to the larger number of *Stereotypes* in UP_{TDL}. However, the question arises whether the different numbers of *Stereotypes* is the only reason, because UP_{TDL} contains with 99 stereotype attributes almost twice as many as UP_{std}. Another reason might be that not each metaclass attribute has a corresponding *Stereotype* attribute in UP_{TDL}. This aspect is analysed in more detail below.

The compliance of UML elements with TDL's static semantics can only be evaluated if *Stereotypes* with appropriate OCL *Constraints* exist. Because UP_{std} has only two OCL *Constraints*, this statement also applies to those TDL language concepts for which corresponding *Stereotypes* are present. In contrast, UP_{TDL} provides 113 *Constraints* and 10 *Operations* that are defined via OCL. These include not only the *Constraints* transferred from MM_{TDL}, but also those introduced during the derivation of UP_{TDL}. Therefore, we conclude that UP_{std} cannot be used to ensure TDL's static semantics, whereas this drawback does not exist for UP_{TDL}.

Comparison of the syntactic structure. After we determined that UP_{std} has some limitations over UP_{TDL}, we now compare the structure of the *Stereotypes* in both UML profiles. For this purpose, we examine the *Stereotypes* that define the abstract syntax for TDL test descriptions. The relevant Stereotypes of UP_{std} are shown in Fig. 15a, and those defined for UP_{TDL} can be found in Fig. 15b.

Considering the *Stereotypes* and the UML metaclasses extended by them in Figs. 15a and 15b, we can identify two significant differences. The `<<ComponentInstance>>` stereotype in UP_{std} extends the UML metaclasses `EncapsulatedClassifier` and `Property`, whereas the one in UP_{TDL} extends only the latter. Both UML metaclasses extended by the stereotype of UP_{std} are fundamentally different language concepts. The UML metaclass `Property` represents a specialization of `StructuralFeature`, whereas `EncapsulatedClassifier` is a specialization of `Classifier` that can provide various `Features`, such as `StructuralFeature`.

If we consider the TDL metaclasses shown in Fig. 7b, we see that the `componentInstance` attribute of the `TestConfiguration` metaclass is a composition. Hence, instances of `ComponentInstance` are parts of `TestConfiguration` instances. In addition, `ComponentInstance` provides a `type` attribute that is employed to refer to a `ComponentType`. Thus, from a semantic point of view, a TDL `ComponentInstance` can be compared with a UML `Property` rather than a UML `EncapsulatedClassifier`. For this reason, we consider an extension of this UML metaclass by the `<<ComponentInstance>>` stereotype in UP_{std} as semantically incorrect, and therefore, this *Stereotype* should only extend the UML metaclass `Property`, as is the case in UP_{TDL}.

Comparing Figs. 15a and 15b, we find that the stereotypes of UP_{TDL} have six and those of UP_{std} have only three attributes defined by *Association* ends. Because stereotypes of UP_{TDL} are derived from MC_{St} metaclasses in MM_{TDL}, they have the same number of attributes as the corresponding metaclasses. In contrast, stereotypes in UP_{std} have a lower number of attributes than TDL metaclasses.

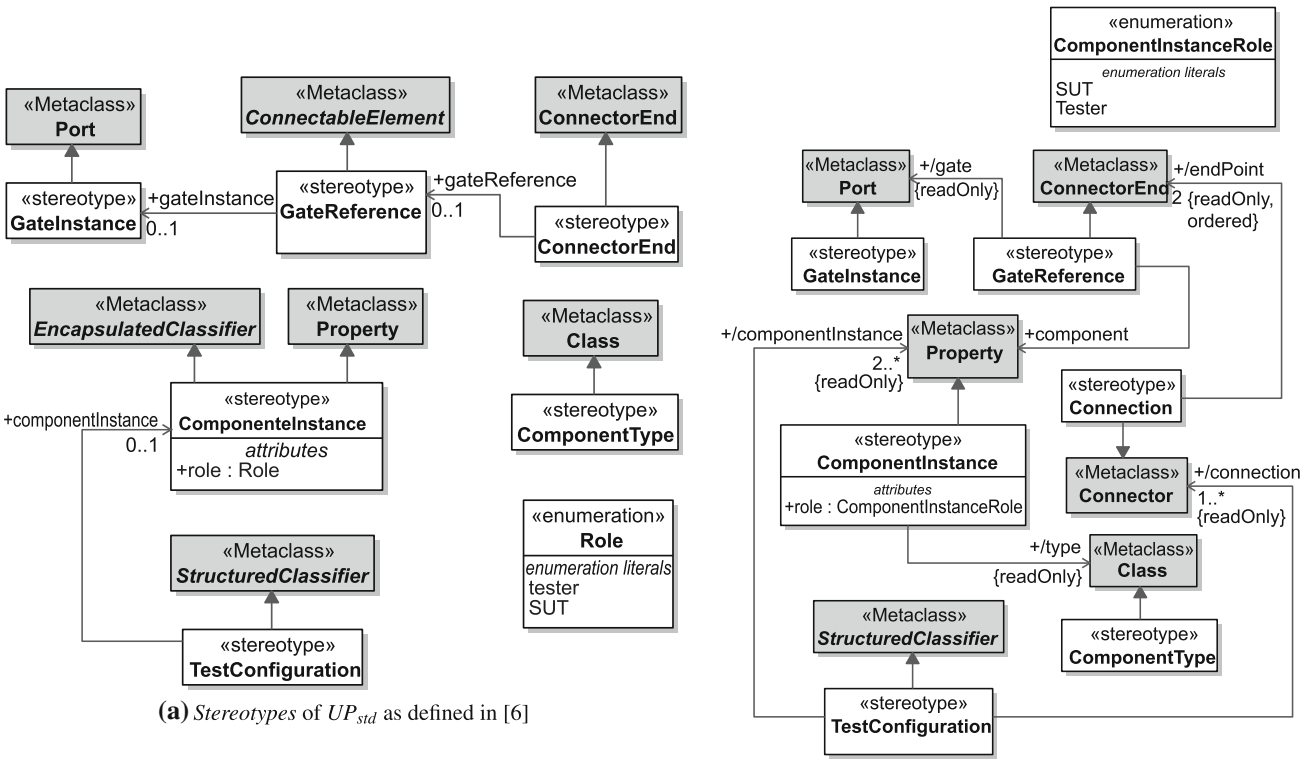
Instead of the non-existent stereotype attributes, a mapping of UML metaclass attributes to certain TDL metaclass attributes is defined in UP_{std}. This is a viable way, but due to the absence of OCL *Constraints* in UP_{std}, the values permitted for UML metaclass attributes are not constrained. Therefore, UP_{std} allows the specification of models that are invalid in relation to the static semantics of TDL. Because UP_{std} does not specify a corresponding stereotype attribute for each metaclass attribute, this also explains the difference in the number of attributes between UP_{std} and UP_{TDL}.

All stereotype attributes introduced by *Association* ends in UP_{TDL} refer to UML metaclasses extended by *Stereotypes*. In contrast, the stereotype attributes in UP_{std} (e.g. see Fig. 15a) always refer to *Stereotypes*. Because navigation from a *Stereotype* instance to the extended UML element or vice versa is possible, referencing *Stereotypes* or extended UML metaclasses is comparable from a syntactical point of view. However, the latter option requires the use of appropriate OCL *Constraints* to ensure that only UML elements with a specific applied *Stereotype* can be assigned to a stereotype attribute. As discussed above, exactly such OCL *Constraints* are automatically introduced by our derivation approach.

Another difference between the stereotype attributes of UP_{TDL} and UP_{std} shown in Fig. 15 is their cardinality. If we compare the stereotype attributes in Fig. 15b with the corresponding metaclass attributes in Fig. 7b, we find that their cardinalities match. In contrast, the stereotype attributes shown in Fig. 15a have different cardinalities than their corresponding metaclass attributes of MM_{TDL}. Thus, a syntactical difference between UP_{std} and the TDL metamodel exists. The TDL standard [6] does not indicate that the deviating attribute cardinalities are an explicit design decision for the creation of UP_{std}. We therefore assume that these are issues that result from the manual creation of UP_{std}.

6.3 Discussion of the evaluation results

We investigated the applicability of our approach to derive a UML profile and used the *Test Description Language (TDL)* [6] to evaluate the applicability of our derivation approach to new DSLs. In contrast, the *Specification and*



(a) *Stereotypes* of UP_{std} as defined in [6]

(b) *Stereotypes* of UP_{TDL} that are derived from the metaclasses shown in Fig. 7b

Fig. 15 *Stereotypes* in UP_{std} and UP_{TDL} used to define the syntax of TDL test descriptions

Description Language (SDL) [21] was subject of another case study [27,28] to evaluate the applicability of our approach for existing DSLs with available production rules.

In Sects. 4 and 5, we already employed TDL as running example to detail our approach for deriving a UML profile and updating the associated OCL-defined static semantics; therefore, here we only quantitatively investigated whether UP_{TDL} was derived from MM_{TDL} according to our Design Decisions 1–12. The comparison of UP_{TDL} with MM_{TDL} showed that exactly the number of model elements we expected, such as stereotypes, was generated for UP_{TDL}. Hence, we conclude that our derivation approach conforms to our Design Decisions. However, this does not answer whether a UML profile automatically derived by our approach is comparable to a manually created one.

Thus, we compared TDL's standardized UML profile UP_{std} with our automatically derived UML profile UP_{TDL} quantitatively and qualitatively. This comparison showed that some of TDL's language concepts in UP_{std} are not represented as *Stereotypes* but as UML elements, for which mapping rules but no OCL constraints exist. Furthermore, we also noticed that the *Stereotypes* in UP_{std} have fewer attributes than the respective TDL metaclasses. Moreover, the cardinalities of some stereotype attributes did not match those of the corresponding metaclass attributes.

When compared to UP_{std}, our automatically derived UP_{TDL} showed a certain correlation regarding the *Stereotypes* and extended UML metaclasses. Due to the one-to-one derivation of *Stereotypes* from TDL metaclasses, the highlighted syntactic drawbacks of UP_{std} are not present in the automatically derived UP_{TDL}. Furthermore and because of our automatic OCL update, UP_{TDL} enables an evaluation of the static semantics of TDL, which is infeasible with UP_{std} due to the absence of OCL *Constraints*.

As mentioned before, we conducted another case study [27, 29] where the *Specification and Description Language (SDL)* [21] served us to evaluate our derivation approach for grammar-based DSLs. The focus of our investigation was on the semi-automatic creation of an SDL metamodel based on production rules and on the automatic derivation of a UML profile and 'additional' metaclasses. In the following, we briefly summarize the results of our SDL case study.

By comparing our generated UML profile UP_{SDL} with the one standardized for SDL [20], we observed that the *Stereotypes* in the latter have a much smaller number of attributes. Furthermore, we noticed that many *Stereotypes* of the standardized UML profile represent not just one but several SDL language concepts. Both identified differences result in more complex mapping rules and OCL constraints in the standardized variant. In contrast, the *Stereotypes* of our UP_{SDL} have

a one-to-one relationship to SDL's language concepts they represent. Consequently, we could decrease the complexity of mapping rules and OCL constraints.

Furthermore, we found that some of SDL's well-formed rules were not captured by OCL constraints in the standardized UML profile, while in our UP_{SDL} all rules are considered. Furthermore, we identified differences regarding OCL constraints that ensure the syntactic structure: OCL constraints of the standardized UML profile capture less syntactic aspects than those of our UP_{SDL}. This is because our derivation approach automatically introduces OCL constraints in order to ensure syntactic aspects, e.g. the application of specific *Stereotypes* to attribute items. In contrast, this kind of OCL constraints may be ignored when manually creating a UML profile, because they are often the result of implicit requirements imposed by structural aspects. Hence, we consider the automated derivation of UML profiles to be less error-prone when compared to manual creation.

The above comparison clearly shows that a UML profile generated with our derivation approach captures all language concepts of a DSL as dedicated *Stereotypes*. This is the essential prerequisite for an automatic transfer of the OCL-defined static semantics from a metamodel to a UML profile and enables the evaluation of the static semantics for UML models with applied UML profile. For manually created UML profiles, an OCL-defined static semantics is either not available or not all well-formedness rules are captured. Furthermore, our automatic derivation prevents manual modelling errors such as false multiplicities of stereotype attributes.

7 Related work

Apart from manual approaches (e.g. [30,50]) for creating UML profiles, the most closely related works to ours are [13,14,48,56], which also automatically derive UML profiles from existing DSL metamodels. Their commonality is that, in addition to the metamodel, mapping rules must be provided as input for the profile derivation. Depending on the approach, this is realized in terms of so called 'Integration Metamodels' or 'Mapping Models'. In contrast, our approach expects CMOF-based metamodels as input for the derivation of UML profiles, which reuse 'Abstract Concepts' as proposed in [10,49].

Giachetti et al. propose an approach [13,14] that can be applied to generate UML profiles and mapping rules for model transformations. To derive these artefacts, they employ a certain type of EMOF-based metamodel called 'Integration Metamodels' [15]. Such a metamodel is initially created as a copy of a metamodel for the DSL of interest. In the next step, the copy must be altered manually by adding additional metaclasses, until all rules defined for 'Integration Metamodels' are met. In addition, the structure and semantics of the UML metaclasses that are defined as a mapping target and the ones of the DSL metamodel have to be considered during the rework. Because of the creation of new metaclasses, a revision of the OCL constraints contained in the 'Integration Metamodel' is required. Furthermore, the mappings to UML metaclasses also have to be specified in the 'Integration Metamodel'. Finally, a UML profile and the mapping rules can be generated based on the revised 'Integration Metamodel'. Giachetti et al. state in [12,13] that the OCL constraints contained in an 'Integration Metamodel' must be included in a derived UML profile. Therefore, all references in OCL constraints to elements of the 'Integration Metamodel' are modified so that corresponding UML metaclasses or stereotypes are referenced after UML profile generation.

In contrast, our approach for deriving UML profiles can be applied to CMOF-based metamodels that are created by reusing 'Abstract Concepts'. Due to the correlation between 'Abstract Concepts' and UML metaclasses, we do not have to modify a metamodel so that its structure matches that of the UML metamodel, as is required for an 'Integration Metamodel'. In most cases, we can use these correlations to automatically determine the information required for deriving the different elements of a UML profile. The explicit definition of mapping information is only necessary if 'additional' metaclasses shall be derived, or if a derived *Stereotype* shall extend a UML metaclass that does not have a matching 'Abstract Concept' counterpart.

Another advantage of our approach is the possibility to define 'alternative' OCL expressions so that the values of stereotype attributes are computed at runtime without manual assignment. This is also useful if the value of an attribute can only be determined based on UML model elements that do not have stereotypes applied, or if several model elements must be accessed to obtain the required values.

Equally important and in contrast to our work, the approach of Giachetti et al. [13,14] does not consider the generation of OCL expressions and constraints for subsetting and redefining stereotype attributes. While they discuss a transfer of OCL constraints of an 'Integration Metamodel' towards generated stereotypes by adapting the referenced element types, this is insufficient to obtain valid OCL constraints for generated UML profiles. Because stereotypes and UML metaclasses are instantiated separately, additional attribute navigations must be inserted in OCL expressions, as supported by our approach.

Another category of related works covers the derivation of metamodels based on existing UML profiles, as proposed in [33,35], which is exactly the inverse of our approach. The creation of a UML profile from scratch may be an option for a new DSL with low complexity, but for an existing DSL with higher complexity this can be difficult. This is because

not only the static semantics of the DSL but also UML's static semantics must be taken into account. Therefore, the manual creation of a metamodel followed by an automatic derivation of a UML profile should be preferred for more complex DSLs such as SDL [21]. This variant should also be chosen if the production rules for a DSL are given, because then existing tools [16] (e.g. EMFText [17] or xText [1,5]) can be employed for automatically deriving metamodels.

8 Summary and Conclusions

In addition to our overview on all aspects of our derivation approach given in [28] and our SDL case study [27], which evaluates the applicability of our approach to grammar-based DSLs, the present article details the fully automated derivation of UML profiles and the transfer of static semantics from metamodels to these profiles. In addition, we present another case study on TDL, where we evaluate the applicability of our approach for DSLs to be created from scratch. Our DSL-MeDeTo tool chain, which implements our approach, and the case studies on TDL and SDL are available via our homepage [58].

While only syntactic constructs for UML profiles can be automatically derived from EMOF-based metamodels using existing works [13,14,48,56], our approach also supports CMOF-based metamodels. Particularly noteworthy is the mapping of redefining or subsetting metaclass attributes to 'derived' and 'read-only' stereotype attributes, whose values are computed at runtime via automatically generated OCL expressions. The advantage of such stereotype attributes is that a manual value assignment at runtime is not required, which also reduces the modelling effort.

The automated update and transfer of OCL-defined attributes, *Operations* and *Constraints* to a UML profile is a further contribution of our approach when compared to existing works. This enables us to automatically transfer the static semantics of a DSL metamodel to a derived UML profile. Consequently, we make it possible to automatically evaluate the static semantics for UML models that have applied a UML profile for a particular DSL.

As highlighted in Introduction to this article, a manual creation of UML profiles is often error-prone and, if present, well-formedness rules of UML profiles are usually specified in natural language, which can lead to ambiguities. We confirmed these statements with our TDL case study, where we identified several shortcomings regarding the syntax and static semantics of TDL's standardized UML profile [6]. On the one hand, we found that no corresponding stereotypes exist for some TDL metaclasses, not all metaclass attributes are captured by stereotype attributes, and the cardinalities of metaclass and stereotype attributes do not match. On the other hand, the standardized UML profile for TDL has only two OCL constraints, while the metamodel has 62. Consequently, TDL's static semantics is not captured by the UML profile. In contrast, the aforementioned shortcomings are fixed in the UML profile for TDL that we derived via our DSL-MeDeTo toolchain. In particular, a stereotype is present for each TDL metaclass, and all metaclass attributes are captured by stereotype attributes. Furthermore, all OCL constraints of TDL's metamodel are updated and transferred to the derived UML profile. Therefore, TDL's static semantics can be automatically validated for UML models that have the profile applied.

Because our work has overcome essential shortcomings of existing approaches for the derivation of UML profiles, especially the support of CMOF language concepts and the automatic transfer of static semantics, we anticipate that there is no demand for substantial future work in this area. However, some limitations regarding the derivation of CMOF-based metamodels and transformations for model interoperability exist, because we only create these artefact types semi-automatically based on a single metamodel. As the main goal of our work is the derivation of UML profiles, a semi-automatic generation of the other artefact types is sufficient for us. The consideration of certain approaches presented in the literature, e.g. in [9,31,56]), could be of interest for the fully automatic generation of model transformations.

Acknowledgements We thank Gerald Lüttgen of the Software Technologies Research Group at the University of Bamberg, Germany, for his many valuable remarks on this article. Furthermore, we also thank Richard Paige of the Department of Computing and Software at the McMaster University, Canada, for several discussions on the article's topic.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Bergmayr, A., Wimmer, M.: Generating metamodels from grammars by chaining translational and by-example techniques. In: Model-driven Engineering by Example, CEUR Workshop Proceedings, vol. 1104, pp. 22–31. CEUR-WS.org (2013)
2. Boulet, P., Amyot, D., Stepien, B.: Towards the generation of tests in the test description language from use case map models. In:

SDL 2015: Model-Driven Engineering for Smart Cities, LNCS, vol. 9369, pp. 193–201. Springer (2015)
3. Clark, T., Sammut, P., Willans, J.S.: Applied metamodelling: A foundation for language driven development (3rd ed.). Computing Research Repository (CoRR) abs/1505.00149 (2015)
4. D'Souza, D., Sane, A., Birchenough, A.: First-class extensibility for UML - packaging of profiles, stereotypes, patterns. The Unified Modeling Language: Beyond the Standard. LNCS, vol. 1723, pp. 265–277. Springer, Berlin (1999)
5. Efftinge, S., Völter, M.: OAW xText: A framework for textual DSLs. In: Modeling Symposium at Eclipse Summit, vol. 32, pp. 118–121. eclipsecon.org (2006)
6. ETSI: ES 203 119-1: The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics, V1.3.1. European Telecommunications Standards Institute (2016)
7. ETSI: ES 203 119-2: The Test Description Language (TDL); Part 2: Graphical Syntax, V1.2.1. European Telecommunications Standards Institute (2016)
8. ETSI: ETSI ES 201 873-1: The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, V4.10.1. European Telecommunications Standards Institute (2018)
9. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. In: Model Driven Engineering Languages and Systems, LNCS, vol. 5301, pp. 326–340. Springer (2008)
10. Fischer, J., Piefel, M., Scheidgen, M.: A metamodel for SDL-2000 in the context of metamodelling ULF. In: System Analysis and Modeling, LNCS, vol. 3319, pp. 208–223. Springer (2004)
11. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to UML profiles. Europ. J. Inform. Prof. **5**(2), 6–13 (2004)
12. Giachetti, G., Albert, M., Marín, B., Pastor, O.: Linking UML and MDD through UML profiles: a practical approach based on the UML association. J. Univ. Comput. Sci. **16**(17), 2353–2373 (2010)
13. Giachetti, G., Marín, B., Pastor, O.: Integration of domain-specific modelling languages and UML through UML profile extension mechanism. J. Comput. Appl. **6**(5), 145–174 (2009)
14. Giachetti, G., Marín, B., Pastor, O.: Using UML as a domain-specific modeling language: A proposal for automatic generation of UML profiles. In: Advanced Information Systems Engineering, LNCS, vol. 5565, pp. 110–124. Springer (2009)
15. Giachetti, G., Valverde, F., Pastor, O.: Improving automatic UML2 profile generation for MDA industrial development. In: Advances in Conceptual Modeling – Challenges and Opportunities, LNCS, vol. 5232, pp. 113–122. Springer (2008)
16. Goldschmidt, T., Becker, S., Uhl, A.: Classification of concrete textual syntax mapping approaches. In: Model Driven Architecture – Foundations and Applications, LNCS, vol. 5095, pp. 169–184. Springer (2008)
17. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-based language engineering with EMFtext. In: Generative and Transformational Techniques in Software Engineering IV, LNCS, vol. 7680, pp. 322–345. Springer (2013)
18. ITU-T: Rec. Z.109: Specification and Description Language – SDL-2000 combined with UML. International Telecommunication Union (2007)
19. ITU-T: Rec. Z.120: Message Sequence Chart (MSC). International Telecommunication Union (2011)
20. ITU-T: Rec. Z.109: Specification and Description Language – Unified Modeling Language profile for SDL-2010. International Telecommunication Union (2013)
21. ITU-T: Rec. Z.100: Specification and Description Language – Overview of SDL-2010. International Telecommunication Union (2016)
22. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S.: Design guidelines for domain specific languages. In: Domain-Specific Modeling. arxiv.org (2009)
23. Kobryn, C.: UML 2001: A standardization odyssey. ACM **42**(10), 29–37 (1999)
24. Kraas, A.: Open issues of the SDL-UML profile. Tech. Rep. N-106519, Fraunhofer ESK, Munich, Germany (2009)
25. Kraas, A.: The SDL-UML profile revisited. In: System Analysis and Modeling: About Models, LNCS, vol. 6598, pp. 108–123. Springer (2011)
26. Kraas, A.: Towards an extensible modeling and validation framework for SDL-UML. In: System Analysis and Modeling: Models and Reusability, LNCS, vol. 8769, pp. 255–270. Springer (2014)
27. Kraas, A.: Automated tooling for the evolving SDL standard: From metamodels to UML profiles. In: SDL 2017: Model-Driven Engineering for Future Internet, LNCS, vol. 10567, pp. 1–21. Springer (2017)
28. Kraas, A.: On the automated derivation of domain-specific UML profiles. In: Modelling Foundations and Applications, LNCS, vol. 10376, pp. 3–19. Springer (2017)
29. Kraas, A.: On the automated derivation of domain-specific UML profiles. Ph.D. thesis, Department of Information Systems and Applied Computer Sciences, University of Bamberg, Germany (2019). https://fis.uni-bamberg.de/handle/uniba/45648
30. Lagarde, F., Espinoza, H., Terrier, F., Gérard, S.: Improving UML profile design practices by leveraging conceptual domain models. In: Automated Software Engineering, pp. 445–448. ACM (2007)
31. Langer, P., Wimmer, M., Kappel, G.: Model-to-model transformations by demonstration. In: Theory and Practice of Model Transformations, LNCS, vol. 6142, pp. 153–167. Springer (2010)
32. Makedonski, P., Adamis, G., Käärik, M., Kristoffersen, F., Zeitoun, X.: Evolving the ETSI test description language. In: System Analysis and Modeling: Technology-Specific Aspects of Models, LNCS, vol. 9959, pp. 116–131. Springer (2016)
33. Malavolta, I., Muccini, H., Sebastiani, M.: Automatically bridging UML profiles to MOF metamodels. In: Software Engineering and Advanced Applications, pp. 259–266. IEEE (2015)
34. Marroquin, A., Gonzalez, D., Maag, S.: A novel distributed testing approach based on test cases dependencies for communication protocols. In: Research in Adaptive and Convergent Systems, pp. 497–504. ACM (2015)
35. Noyrit, F., Gérard, S., Selic, B.: FacadeMetamodel: Masking UML. In: Model Driven Engineering Languages and Systems, LNCS, vol. 7590, pp. 20–35. Springer (2012)
36. OMG: MOF Model to Text Transformation Language – Version 1.0. Object Management Group (2008)
37. OMG: Software Systems Process Engineering Metamodel (SPEM) – Version 2.0. Object Management Group (2008)
38. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification – Version 1.1. Object Management Group (2011)
39. OMG: OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1. Object Management Group (2011)
40. OMG: OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. Object Management Group (2011)
41. OMG: OMG Unified Modeling Language (OMG UML), Version 2.5. Object Management Group (2011)
42. OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1. Object Management Group (2011)
43. OMG: Service oriented architecture Modeling Language (SoaML) – Version 1.0.1. Object Management Group (2012)
44. OMG: Business Process Model and Notation (BPMN) – Version 2.0.2. Object Management Group (2013)
45. OMG: Object Constraint Language – Version 2.4. Object Management Group (2014)
46. OMG: UML Profile for BPMN Processes – Version 1.0. Object Management Group (2014)
47. OMG: OMG Meta Object Facility (MOF) Core Specification – Version 2.5. Object Management Group (2015)

48. Pastor, O., Giachetti, G., Marín, B., Valverde, F.: Automating the interoperability of conceptual models in specific development domains. In: Domain Engineering: Product Lines, Languages, and Conceptual Models, pp. 349–373. Springer (2013)
49. Scheidgen, M.: Description of languages based on object-oriented meta-modelling. Ph.D. thesis, Humboldt-Univ., Berlin, Germany (2009)
50. Selic, B.: A systematic approach to domain-specific language design using UML. In: Object and Component-Oriented Real-Time Distributed Computing, pp. 2–9. IEEE (2007)
51. da Silva, A.R.: Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures **43**(C), 139–155 (2015)
52. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
53. Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. Software: Practice and Experience **39**(15), 1253–1292 (2009)
54. Tenbergen, B., Bohn, P., Weyer, T.: Ein strukturierter Ansatz zur Ableitung methodenspezifischer UML/SysML-Profile am Beispiel des SPES 2020 Requirements Viewpoints. In: Software Engineering 2013, Lecture Notes in Informatics, vol. 215, pp. 235–244. GI (2013)
55. Ulrich, A., Jell, S., Votintseva, A., Kull, A.: The ETSI Test Description Language TDL and its application. In: Model-Driven Engineering and Software Development (MODELSWARD 2014), pp. 601–608. SCITEPRESS (2014)
56. Wimmer, M.: A semi-automatic approach for bridging DSMLs with UML. J. Web Inform. Syst. **5**(3), 372–404 (2009)
57. EMFText concrete syntax mapper homepage. https://marketplace.eclipse.org/content/emftext (Accessed: 04.02.2020)
58. Homepage of the SU-MoVal and DSL-MeDeTo toolchains. http://www.su-moval.org (Accessed: 04.02.2020)
59. xText homepage. http://www.eclipse.org/Xtext/ (Accessed: 04.02.2020)