Mixins and Extenders for Modular Metamodel Customisation

Conference Paper · January 2016

DOI: 10.5220/0005761102590270

2 authors, including:

Srdjan Zivkovic
BOC Information Technologies Consulting GmbH
17 PUBLICATIONS **54** CITATIONS

SEE PROFILE

Mixins and Extenders for Modular Metamodel Customisation

Srđan Živković and Dimitris Karagiannis
Faculty of Computer Science, University of Vienna, Vienna, Austria
{srdjan.zivkovic, dimitris.karagiannis}@univie.ac.at

Keywords: Metamodelling, Metamodel Composition, Metamodel Customisation, Metamodelling Tools

Abstract: Metamodelling is a practical yet rigorous formalism for modelling language definition with a metamodel being its pivotal engineering artifact. A multitude of domain-specific modelling languages (DSML) are engineered to cover various modelling domains. Metamodels of such languages evolve over time by introducing changes and extensions and are further customised to suite project-specific needs. While majority of DSML development techniques provide concepts for creating metamodels from scratch, composition concepts for metamodel customisation beyond class inheritance are sought towards more flexibility and reuse. In this paper, we introduce a modular approach for metamodel customisation based on the idea of mixins and extenders. While mixins allow for defining self-contained metamodel modules for reuse, extenders enable non-intrusive composition of such reusable modules on top of existing metamodels. We show how this approach can be applied in a metamodelling tool such as ADOxx and demonstrate its usefulness by customising the BPMN language. The benefit of the modular metamodel customisation is twofold. On the language engineering level, our approach significantly promotes reuse, flexibility and overall efficiency in language definition and customisation. On the modelling level, the approach leverages engineering flexibility to provide custom modelling languages that better suits enterprise modelling needs.

1 INTRODUCTION

Model-based engineering approaches encourage the usage of modelling languages to analyse, design and develop increasingly complex systems and software. A multitude of standard and domain-specific languages are being engineered to cover various modelling domains. Independently of an application domain, modelling languages, like other software deliverables, evolve over time. New versions are released that introduce various changes and extensions (compare UML versions from 1.0 to 2.4.1 or BPMN versions from 1.0 to 2.0.2). Furthermore, released language versions are further adapted and customised to suite problem and project-specific needs. For example, a company may adopt BPMN 2.0 (OMG, 2013) as a standard for business process modelling, but it further requires company-specific extensions for process-based risk management. Such customisation may involve introduction of additional risk-related properties to existing language entities, creation of new entities or even integration with proprietary languages to build a custom hybrid solution. Ideally, such custom extensions should be portable to the upcoming version of the base language. The *evolving* nature of languages, the need for *customised* languages and the *complexity* that arises when combining evolution and customisation phenomena together, call for systematic, flexible and modular approaches for language design and customisation.

Metamodelling has been recognised as a practical yet rigorous formalism for modelling language development. In metamodel-based approaches, a metamodel is used to define the abstract syntax of the language. As a pivotal element in language definition, metamodel defines language concepts for which precise semantics and one or more concrete syntaxes may be defined (Selic, 2011). Nowadays, a multitude of mature *metamodelling languages* exist such as the standard MOF (OMG, 2014), or tool-specific meta-languages such as Eclipse EMF Ecore (Steinberg et al., 2008), MetaEdit+ GOPPRR (Kelly et al., 1996), ADOxx Meta^2-Model (Junginger et al., 2000; Kühn, 2010), or GME MetaGME (Ledeczi et al., 2001). In metamodelling languages, we may distinguish between *core* and *supporting* metamodelling capabilities. *Core constructs* are used to define fundamental elements of a metamodel. Constructs such as class, property or reference are examples of core constructs as they contribute to the core expressive

Table 1: Overview of core and supporting capabilities of selected metamodelling languages

Capability	ADOxx Meta^2-Model	EMF Ecore	GME MetaGME	MetaEdit+ GOPPRR	MOF 2.0
Core capabilities					
Class	Class	EClass	Atom	Object Type	Class
Attribute	Attribute	EProperty	Attribute	Property	Property
Relation	Relation Class	EReference	Connection	Relationship	Association
Relation End	Endpoint	-	Connection Role	Role, Port	Property
Model Type	Model Type, Mode	EPackage	Model, Aspect, Role	Graph Type	Package
Supporting capabilities					
Modularisation	Library, Fragment	Package	Project	Graph Type	Package, Profile
Extensional Composition	Single Inheritance, Aggregation	Multiple Inheritance	Multiple Implementation Inheritance, Interface Inheritance	Single Inheritance, Inclusion	Multiple Inheritance, Package Merge, Stereotype, Extension, Tag

power of a metamodelling language. Complementary, *supporting constructs* contribute to the efficiency of metamodelling. They provide support for better structuring of metamodel artefacts and promote reuse of core metamodel artefacts. Modularisation constructs such as packages that are used to encapsulate metamodel elements, or composition constructs such as class inheritance which enables reuse of structural features of classes, are examples of such constructs.

While comprehensive support for the core metamodelling concepts is common to all metamodelling languages, the opposite is true for the supporting metamodelling constructs (see Table 1). Here, a positive exception, however, is the metamodelling standard MOF. Through the common UML2 infrastructure, MOF provides a set of mature mechanisms for metamodel customisation and incremental metamodel refinement such as the *Profile* mechanism and the *packageMerge*. Nevertheless, profiles have been, until now, exclusively used to refine only metamodels based on UML. In (Langer et al., 2012) the idea of UML profiles has been applied to Ecore, in order to enable profiles more broadly for DSMLs. Furthermore, while the package merge supports modular and incremental metamodel customisation, it operates on the level of packages and relies on the name-based element matching to apply merge, which is not always a desired approach for metamodel composition and customisation. Finally, the inheritance, in some of its forms as a single, multiple, interface or implementation-like, is supported by all metamodelling languages. Inheritance is most widely used technique for metamodel composition and customisation. However, while reusability of structural features by subclassing is one of the main advantages of inheritance, it may, at the same time, be its major drawback. Sublcassing as a way to extend a parent class with additional structural features may often end in complex class hierarchies and over-engineered metamodels. Furthermore, extending a class by subclassing may in some cases not be possible (single inheritance restriction, "sealed" base classes) or not desired (the base class is already in use, i.e. instances exist that would require tool recompilation and model migration).

Modular, incremental development has been one of the major drivers for the shaping of object-orient programming languages (OOPL). Single and multiple inheritance, mixins, traits, extension methods and templates in OOPLs are some of the key mechanisms that boost efficiency and flexibility in programming.

In this paper, inspired by some of the known extensibility concepts from OOPLs, we introduce a modular approach for metamodel customisation. In particular, the approach introduces the notions of *mixins* and *extenders* with appropriate composition operators that complement existing techniques for metamodel composition and customisation. Mixins allow for defining reusable self-contained metamodel extensions that can be combined by arbitrary modules without the creation of multiple class hierarchies. On the other hand, extensions allow for non-intrusive injection of custom metamodel extensions on top of existing metamodels eliminating the need for creating derived types. This way, with mixins we increase the overall potential for reuse in metamodelling beyond inheritance, whereas with extensions, we contribute to greater flexibility when extending metamodels.

The paper is structured as follows. In Section 2, we introduce a running example related to the customisation of the BPMN metamodel. In Section 3, after we've revisited the limitations of inheritance, we

introduce Mixins and Extenders and two new metamodel composition operators, Mixin Inclusion and Extension. Section 4 elaborates on the application of the approach based on the ADOxx metamodelling language. In Section 5 we discuss the related work. Finally, Section 6 concludes the paper.

2 RUNNING EXAMPLE: CUSTOMISATION OF BPMN

Let us consider the previously mentioned example of metamodel customisation, in which, the industry standard business process modelling language BPMN 2.0 is extended by a business-oriented extension for process-based risk management (RM) and by a process simulation extension (PS) for business process simulation (Herbst et al., 1997). The necessity of extending the BPMN with further business aspects for enterprise modelling and analysis has been discussed in (Rausch et al., 2011).

Let us suppose we follow a modular approach to metamodel development, where metamodels are encapsulated into reusable, stand-alone modules, focusing on single aspects and concerns. In that case, we will have three metamodel modules, *BPMN*, *RM* and *PS* as depicted in Figure 1. In module *BPMN*, the class *Task* is the central entity for modelling business process flows, which we aim at extending with other related concepts. In module *RM*, the class *Risk* is used to model company risks. A risk may be assigned to various business entities, which is represented by the abstract class *RiskHolder*. In the *PS* module, the abstract class *SimulationActivity* represents an abstract activity containing attributes *Time* and *Costs* necessary to run the process simulation algorithm. Note that this is a very simplified view of the simulation aspect. Such activity may have an assigned performer (*Performer*) that executes the activity during the simulation. Our goal is to customise BPMN in a way that the task becomes connectable to risks, and that it contains simulation features. In other words, we want from class *Task* to have characteristics of both classes, *RiskHolder* and *SimulationActivity*. Furthermore, metamodel composition should be *non-intrusive*, i.e. both BPMN as well as RM and PS modules must not be modified. In addition, no new derived entities should be defined, in order to retain the compatibility with existing mechanisms and model bases.

3 METAMODEL COMPOSITION BASED ON MIXINS AND EXTENDERS

In this section, we first discuss current limitations of the inheritance in metamodelling languages based on the running example, since inheritance, in some of its variants, is commonly supported composition mechanism by all metamodelling languages (see Table 1). Afterwards, we introduce the concepts of Mixin and Extension, two new metamodel composition operators for flexible, modular metamodel customisation.

3.1 Inheritance is Not Enough

In a nutshell, the intention of inheritance is to reuse structural features of classes such as properties and references by creating parent-child class hierarchies. A subclass inherits all features of either one super class (single inheritance) or of more than one super class (multiple inheritance). While metamodelling languages such as Ecore, MetaGME and MOF support multiple inheritance, languages such as ADOxx and GOPPRR are restricted to single inheritance. Multiple inheritance has been discussed controversially since its introduction in programming languages (Bracha, 1992), as well as, more recently, in metamodelling (Selic, 2011). Multiple inheritance is criticised for an increased unanticipated complexity and ambiguity in class design, allowing for antipatterns such as “diamond inheritance problem” and over-generalisation.

Figure 1 illustrates how both single and multiple inheritance can be applied to extend the *BPMN* module with *RM* and *DM* modules. We summarise major deficiencies in the context of customisation in two categories, *singleness* and *subclass imperative*.

Singleness. Given the single inheritance restriction by a metamodelling language, we introduce new extension module *extBPMNa* which contains one derived class *DerivedTask* as a subclass of *Task* from module *BPMN*. Since multiple inheritance is not allowed, we cannot inherit additionally from classes *RiskHolder* and *SimulationActivity*, but need to define two new references *assignedRisks'* and *assignedPerformer'* to classes *Risk* and *Performer* and to remodel properties from the class *SimulationActivity* such that the class *DerivedTask* includes the semantics of classes *Risk* and *Performer*. Obviously, this approach is not flexible enough as it doesn't allow for reuse of additional structural features other than those that are inherited from the single super class. If a metamodelling language allows for multiple inheritance, the class *DerivedTask* in module *extBPMNb*

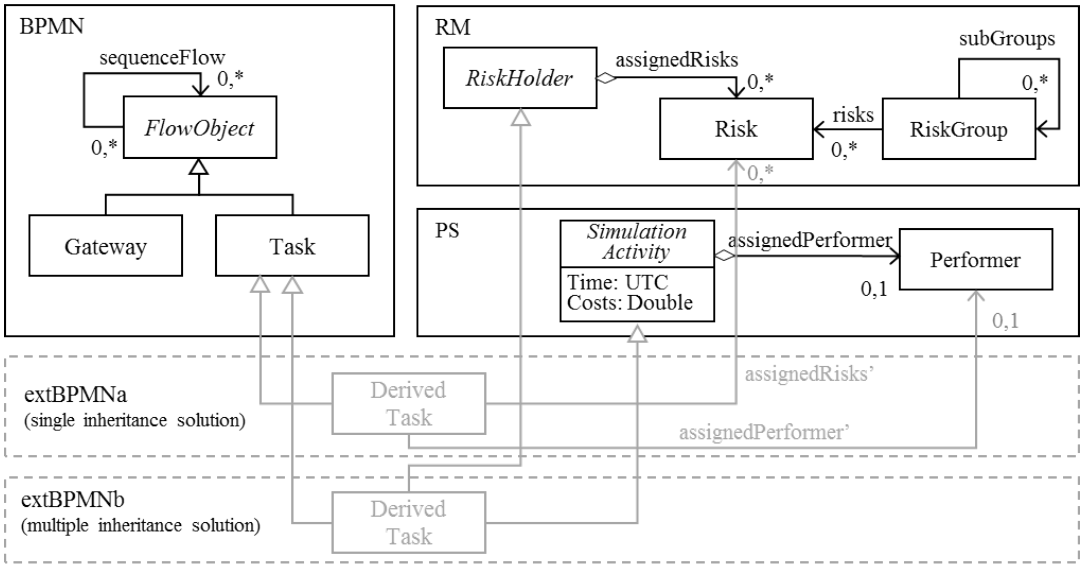

Figure 1: Customisation of the BPMN metamodel using reusable RM and DM modules.

may specialise the class *Task* and also inherit from the abstract classes *RiskHolder* and *SimulationActivity* to accompany all required features. This solution appears to be more elegant, as it allows for reuse by inheritance from multiple superclasses.

Subclass imperative. Although multiple inheritance overcomes the problems of singleness, in both inheritance-based solutions, however, we are forced to introduce an explicit derived type in order to extend a class without directly modifying it. In our case, the class *DerivedTask* inherits in both cases from the class *Task* and must be used if extended process modelling with risks and simulation is desired. This kind of customising by subclassing may be an undesirable when applying metamodel customisation. It forces the introduction of a new modelling class in the language, even though, conceptually, only an adaptation of an existing class was required. Furthermore, as an effect of a new derived type, both functionality and models conforming to the base BPMN metamodel have to be upgraded to the new metamodel version, i.e. the instances of class *Task* need to be converted to the subclass *DerivedTask*, in order to apply the extension. We call this problem the *subclass imperative deficiency*.

3.2 Mixin-based Metamodel Composition

In order to mitigate the singleness problem of inheritance and complex and potentially ambiguous multiple inheritance hierarchies, while increasing the potential of reuse, we propose the usage of mixins in metamodelling. Adopting the general idea of mixin-based inheritance (Bracha and Cook, 1990) in metamodel composition, mixins are said to allow for the definition of independent metamodel element parts (Mixins) that may be reused, i.e. *mixed*, by other elements. Mixins usually bundle some common set of features that may be shared among other metamodel elements. To allow for the mixin-based metamodel composition, a *parent element*, a *mixin metamodel element* and a *mixin inclusion composition operator* are needed.

- *Parent element*. A parent element in the mixin composition may be any element of a compound type, i.e. an element that contains other elements. For instance, a class is a compound metaclass that may contain structural features such as properties and references.
- *Mixin element*. A mixin element is a compound element type that contains features to be shared among other elements. We define mixin element as a non-instantiable, abstract element, in order to denote its pure *supporting metamodelling capability*. The mixin element must be of the same type as the base element. For example, an abstract element of type Class may be defined that contains a set of common properties and/or references that may be shared between various other classes. In our case, appropriate candidates for mixin classes are *RiskHolder* and *SimulationActivity*.
- *Mixin inclusion*. Mixin inclusion operator is a relation that takes a parent element and a mixin element as an input, and includes (“mixes in”) the

child elements of the mixin element to the parent element. A parent element may mix in many mixin elements. In turn, a mixin element may be reused by arbitrary parent elements. Hence, mixin operator allows for an increased flexibility in appending features to an element without distracting the inheritance hierarchy. On the other side, it enables the definition of self-contained aspectual modules which can be flexibly combined, thus fostering reuse and clear separation of concerns. In our example, the class *DerivedTask* may now include mixin classes *RiskHolder* and *SimulationActivity*, in order to obtain their features.

3.3 Extension-based Metamodel Composition

While mixins solve the problem of inheritance singleness and complex inheritance hierarchies, mixins do not tackle the issue of the subclass imperative, when it comes to extending an already existing base metamodel. We still need to create a derived type, when extending the very base element (in our example, class *Task*). In order to address this issue, we introduce the concept of extension and extender-like elements. The extenders may be thought of as inverse mixins. They allow for extensively injecting the features into a parent element without creating a derived element. The extension mimics the semantics of the inheritance, however without a need for a derived type.

The extension-based metamodel composition picks up on the initial idea of invasive software composition (Aßmann, 2003), in which program fragments such as methods and properties may invasively be injected into existing program code by operating on their implicit interfaces using transformative techniques. However, instead of intrusively changing the code, we introduce a native metamodelling language operator that allows to add features to an existing element without the need to modify it. Applied on metamodels, *implicit interfaces* allow for controlled variation points where metamodel elements may be extended. An implicit interface represents an extension point of a metamodel element, which is implicitly defined by the inherent semantics of the underlying metamodelling language. Each element type may have different implicit interfaces. For example, implicit interfaces of the metaclass *Class* are its structural feature sets. Using the extension operator, another element may access such an implicit interface and extend that particular class by injecting e.g. additional member attributes or references. Implicit interfaces are crucial in metamodel customisation, where extensional composition should take place on previously not explicitly defined extension points, or on non-modifiable elements. To combine elements based on extension, a *base element*, an *extender element* and an *extension operator* are required.

- *Base element*. A base element may be any compound metamodel element, for which at least one implicit interface exists. For example, it doesn't make sense to extend elements such as attribute types that do not aggregate other elements and features. In our case, the base element is the class *Task*.
- *Extender element*. An extender element is a compound element, that holds extensions that should be injected to the base element. Since it is a pure *supporting metamodelling construct*, it is a non-instantiable, abstract element. In addition, the extender element must be of the same type as the base element. This is required to implicitly constrain only extensions that are possible for that specific element type.
- *Extension composition operator*. Extension operator is a relation that takes a base element and an extender element as input and extends the base element by injecting extensions based on well-defined implicit interfaces. Like in inheritance, but inversely, the structural features of the extender element are propagated to the base element without any syntactic modification of the base element. An extender element may extend many base elements. In turn, a base element may be extended by arbitrary extender elements. Hence, the extension operator diminishes the necessity of the subclassing imperative, since base elements may be extended via direct feature injection.

Figure 2 illustrates the revisited customisation of the BPMN module now using the mixin inclusion and extension operators. The modules *RM* and *PS* become self-contained, reusable "mixin" modules, having classes *RiskHolderMixin* and *SimulationActivityMixin* as their central mixin elements. Instead of having the explicit derived class *DerivedTask*, the extension module *extBPMNc* defines the extender class *TaskExtender*, which, on the one side, includes the mixin classes, and, on the other side, extends the class *Task* by injecting its structural features, that of mixins. Mixin and extension operators, when used in combination, allow for flexible, non-intrusive, and modular metamodel customisation by composition. While mixins can be used to define self-contained, reusable modules applicable for arbitrary metamodels, extenders play the role of the *composition glue logic*, i.e. they allow for injecting mixins into existing meta-

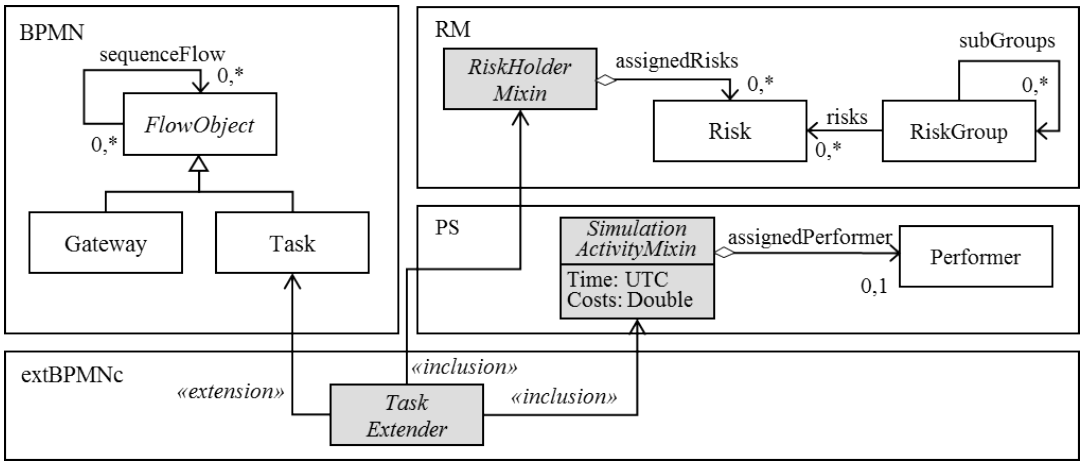

Figure 2: Metamodel customisation based on mixin inclusion and extension composition operators.

model fragments.

4 APPLICATION IN ADOXX

This section elaborates on the application of the introduced metamodel composition concepts within the metamodelling language of the metamodelling tool ADOxx (Junginger et al., 2000; Kühn, 2010; OMILab, 2015). Even though the concepts introduced in the following are defined considering the characteristics of ADOxx metamodelling language, we believe that ideas presented may be translated to other metamodelling languages due to comparable metamodelling expressiveness (see Section 1, Table 1).

4.1 ADOxx Metamodelling Language

ADOxx Meta^2-Model is the meta-metamodel of ADOxx. In the following, we explain its main concepts that will serve as a basis for the further discussion regarding the introduced compositional extensions. Figure 3 illustrates the ADOxx meta-metamodel. In ADOxx, all metamodel constructs are identified by IDs and names. This is represented by the top-level abstract metaclass *AObject* and its subclass *ANamedObject*. Further, various meta-constructs in ADOxx may have attributes. The metaclass *AttributeDefinition* represent an attribute construct that may have a default value, a set of constraints and may be of some simple or complex attribute type. Attributable constructs are generalised by an abstract metaclass *AObjectWithAttributes*. A *ALibrary* is an attributable construct which consists of model types and implicitly of other constructs such as classes and relations. As such, a library represents a bundle of different diagram types. To define diagram types, the concept of model type is used. A *AModelType* is an attributable construct that typifies models and consists of classes and relations. In addition, a model type may have *AModes*, which further subset a model type with respect to available classes and relations. A *AClassDefinition* is an abstract metaclass that can hold attributes, and can be contained by model types. Class definitions may be classes or relations. A *AClass* is the central metamodelling construct used to specify entities of a modelling language. ADOxx supports single inheritance for classes. The construct *ARelationClass* connects classes and/or model types. A relation class connects to other elements indirectly using the concept of endpoint definition. An *AEndpointDefinition* allows classes and model types to be target types of a relation. The number of endpoints defines the arity of the relation, however ADOxx restricts relations to be binary. To be directed, a relation must have at least one endpoint of type From and one of type To. Hence, an endpoint specifies which elements may participate in the relation and how (multiplicity). Furthermore, ADOxx features an additional reuse mechanism by aggregation to increase the support for *intra-level reuse*. Reuse by aggregation is a Cartesian product aggregation function, such that any allowed child element may be aggregated by any allowed parent element. For example, a globally defined attribute definition may be reused by any attributable element and vice versa. Similarly, classes and relations may be reused by model types and modes, endpoints by relations etc. Finally, the central modularisation construct for encapsulating metamodel elements into reusable, modular units is a *AFragment*. A fragment may contain owned or imported elements. Owned elements are existential members of that fragment. Imported

elements are those referenced from other fragments. Imported elements contribute to inter-fragment reuse and provide a basis for the application of arbitrary composition operators.

4.2 Mixins and Extenders in ADOxx

While introducing the idea of mixins and extenders for metamodelling in Section 3, we also defined on which type of metamodelling constructs mixin inclusion and extension composition operators can be applied. In particular, we defined what the parent element, the base element, as well as, what the mixin and the extender elements may be. For all elements applies that they must be of the compound type, i.e. that they must contain an extensible set of child elements. Furthermore, mixin and extender elements must be abstract elements. Therefore, in ADOxx we define both mixin inclusion and extension operator as relationships on the most general level of attributable constructs, i.e. at the metaclass *AObjectWithAttributes* (see Figure 3). By saying that an attributable construct may mixin and/or extend another attributable construct, we allow for the application of these operators for all compound subtypes such as the class, the model type, the relation class, the endpoint, and the library. This is desirable as all of the compound elements have at least attributes as child elements and, in addition, other contained elements, too. For example, an endpoint has attributes, and a set of target classes or target model types. However, there is a number of constraints that need to be obeyed. In the following, we mention the most important ones:

- **Constraint 1**: *Same type mixin composition*. Mixin inclusion relationship can only connect elements of the same metatype. For example, a class can mix in another class, but cannot mix in a model type.
- **Constraint 2**: *Same type extension composition*. Extension relationship can only connect elements of the same metatype. For example, a model type can extend another model type, but cannot extend a class.
- **Constraint 3**: *Abstract mixin element*. The target element of the mixin inclusion relationship must be declared as abstract.
- **Constraint 4**: *Abstract extender element*. The source element of the extension relationship must be declared as abstract.
- **Constraint 5**: *Acyclic mixin dependency*. The mixin element cannot mix in itself, neither directly nor indirectly.
- **Constraint 6**: *Acyclic extender dependency*. The extender element cannot extend itself, neither directly nor indirectly.
- **Constraint 7**: *Acyclic mixin/extender dependency*. A parent element cannot mix in a mixin element, if it at the same time extends it, neither directly nor indirectly.

The semantics of both operators are common for each instantiable compound metaclass (class, relation class, endpoint definition, model type, mode, library), with regard to inclusion and extension of attributes. Moreover, the semantics of the inclusion operator is very similar to the inheritance of attributes, whereas for extension, it acts as a kind of inverse inheritance of attributes. Hence, in ADOxx, we implement the semantics for mixin inclusion and extension relationship based on the following definitions. First, we introduce the common semantics for all metaclasses that are attributable elements (all subclasses of the metaclass *AObjectWithAttributes*).

- **Definition 1**: *Inclusion of Objects with Attributes*. Given the parent element Ep with the set of attributes Sp, and the mixin element Em with a set of attributes Sm, Ep *includes* Em by aggregating all attributes of Sm into Sp.
- **Definition 2**: *Extension of Objects with Attributes*. Given the base element Eb with the set of attributes Sb, and the extender element Ee with a set of attributes Se, Ee *extends* Eb by aggregating all attributes of Se into Sb.

However, since each metaclass (subclass of objects with attributes) has a different compound structure, i.e. the set of containable structural features on which the operators are applied, the semantics of operators vary for each such metaclass. For example, model type mixin inclusion aggregates all classes of a model type to the parent model type. Inversely, a model type extender inserts all its class members into the base model type. Since listing of all additional definitions for each construct would exceed the limits of the underlying work, we focus only on those elements which, as we will see, we also use in our running example - classes and endpoint definitions. While for classes no further structural containment exists, for the endpoint we define mixin inclusion and extension as follows:

- **Definition 3**: *Inclusion of Endpoint Definitions*. Given the parent endpoint EPp with the set of target classes Sc_1 and the set of target model types Sm_1, and the endpoint mixin EPm with a set of target classes Sc_2 and the set of target model types Sm_2, EPp *includes* EPm by aggregating all target

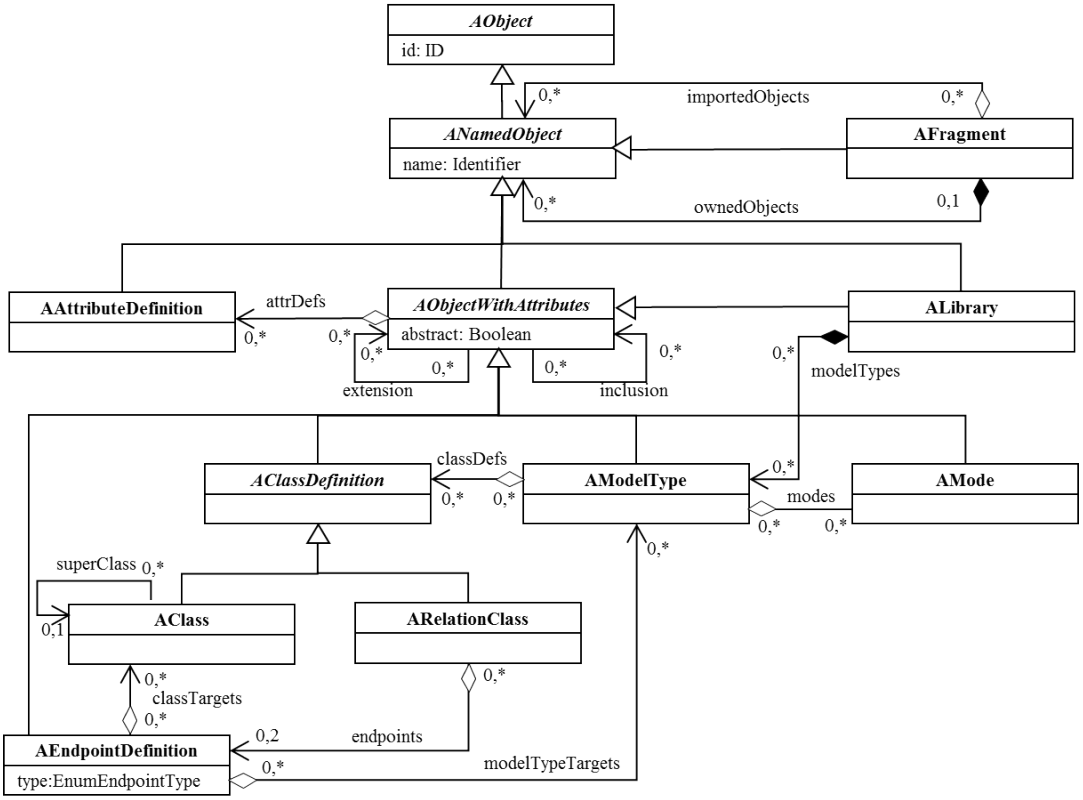

Figure 3: Meta-metamodel of ADOxx featuring Mixins and Extensions

classes of Sc_2 into Sc_1 and all target model types of Sm_2 into Sm_1.

- **Definition 4**: *Extension of Endpoint Definitions*. Given the base endpoint *EPb* with the set of target classes Sc_1 and the set of target model types Sm_1, and the endpoint extender *EPe* with the set of target classes Sc_2 and the set of target model types Sm_2, *EPe extends EPb* by aggregating all target classes of Sc_2 into Sc_1 and all target model types of Sm_2 into Sm_1.

Finally, both inclusion and extension relationships are applied transitively.

4.3 Applying Mixins and Extenders

In the following, we revisit the running example from Section 2 and the conceptual solution from Section 3, in order to exemplify the application of mixins and extenders in ADOxx. Figure 4 illustrates the revisited solution. We now apply ADOxx metaclasses to implement metamodel fragments *BPMN*, *RM*, *PS* and *extBPMNc*. In doing so, we use UML stereotypes to denote corresponding metaclasses from ADOxx. Note that we explicitly model cross-package relationships, instead of re-modelling the imported classes, in order to save the space in the diagram. We define the abstract class *TaskExtender*, which, on the one side, includes the mixin class *SimulationActivityMixin*, and on the other side, extends the class *Task* by inserting the structural features (that of the included mixin). Note that in ADOxx, only attributes *Time* and *Costs* will be propagated as the only member features of the class *SimulationActivityMixin*. Since relations between classes in ADOxx are defined over endpoints as first-order constructs, a relation of a class is syntactically not an inherent feature of that class. Instead, a class is a target, a structural feature of a relation endpoint. Hence, to extend the class *Task* with the relation *AssignedPerformer*, we define the endpoint extender *FromAPExtender*, which, on the one side, targets the class *Task*, and on the other side, extends the corresponding endpoint *FromAP* of the relation *AssignedPerformer*. Similarly, we define the endpoint extender *FromARExtender* for the endpoint *FromAR* of the relation *AssignedRisks*, in order to allow for tasks to connect to performers. As a result of composition, the class *Task* contains both risk-related and simulation features.

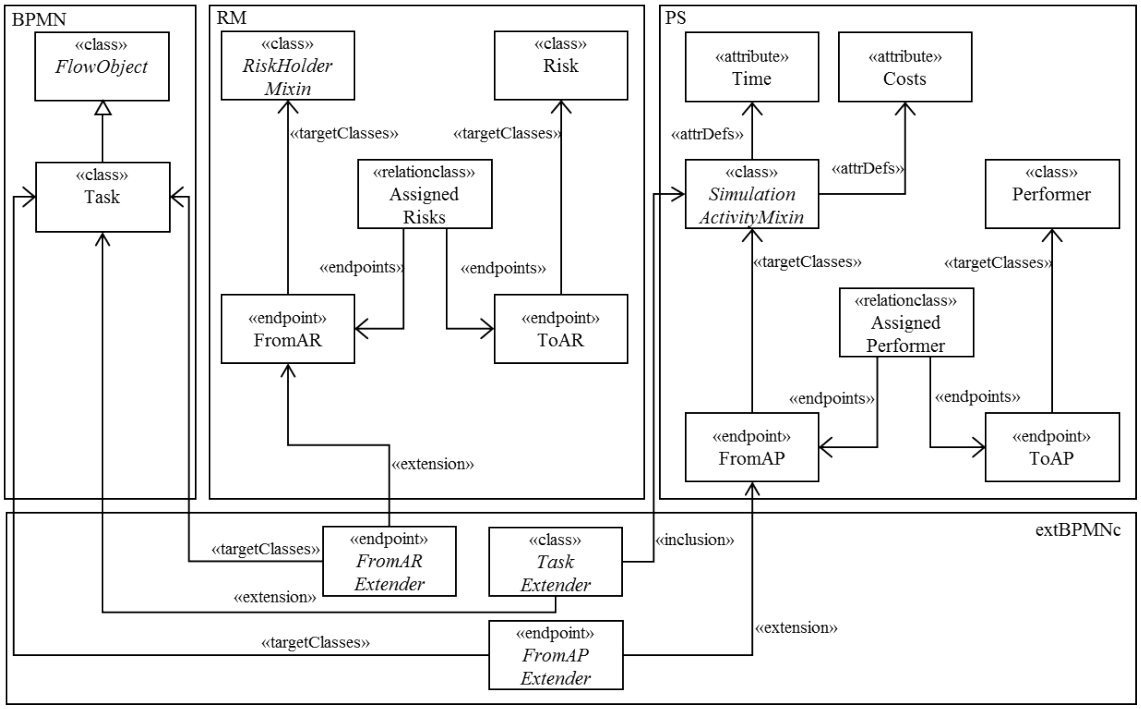

Figure 4: Application of mixin inclusion and extension composition operators in ADOxx.

4.4 Application Evaluation

One may argue that introduced concepts such as mixins, extenders and, in general, modular thinking, although powerful, bring an additional level of complexity in metamodelling. While this argument is true for a one-time metamodel customisation project, the true benefit of modular customisation with mixins and extenders becomes visible with repeated use. In order to evaluate this thesis, we conducted a survey among experienced ADOxx language and metamodel engineers. As part of the survey, we asked engineers to provide two effort estimations for the sample metamodel customisation project from the introduced running example. Given the introduction of the new modular customisation concepts in ADOxx, they provided effort estimations to conduct the work 1) based on basic metamodelling constructs, 2) based on new modular metamodel customisation constructs. The following customising tasks have been considered: 1) initial (from scratch) customisation, 2) migration of the customisation to a new base metamodel version, 3) reuse of customisation for another metamodel. The results are summarised in the Table 2.

As expected, the initial customisation effort estimation in average was slightly higher when using new constructs (mixins and extenders). This was explained by the fact, that this kind of customisation required upfront design in modules for future reuse.

Table 2: Comparison of customisation effort with basic metamodelling constructs and with mixins and extenders for modular metamodel customisation (estimation in story points (SP))

Customisation effort	Basic ADOxx constructs	Mixins and Extenders
Initial customisation	3 SP	3.5 SP
Migration of customisation	2 SP	0.5 SP
Reuse of customisation	2 SP	0.5 SP

However, the estimated effort for the migration was significantly lower when using mixins and extenders. This was mainly due to the fact that the extension could be ported as a mixin module to the new version and injected via extenders without needing to migrate data (refer to the subclassing imperative issue of inheritance). Regarding the reuse of customisation in another project, the effort was clearly lower, since the mixin extension module could be reused as-is, with the only effort of defining new extender class.

5 RELATED WORK

In the area of programming languages, the idea of mixins has been around for years. The term was

coined in the language Flavors (Moon, 1986), however, mixins have been initially defined as a formal language construct for language CLOS (Bracha and Cook, 1990). Mixins found usage in OOPLs such as Smalltalk (Bracha and Griswold, 1996), and Scala (Odersky et al., 2004). GPLs such as C++, that do not support mixins natively, aim at emulating the behaviour of mixins based on parameterised inheritance and template classes (Smaragdakis and Batory, 2001). Similarly, in (Ancona et al., 2000) an extension for Java has been proposed called *Jam*, to allow for mixin-based class composition. As for extensions, the initial idea of an *Extend* operator that injects program code fragments into an existing program code at implicit interfaces was proposed in (Aßmann, 2003) as a part of the broader approach of invasive software composition. Similar approach to extend existing types exists in C#, in which so-called *Extension Methods* allow for injecting methods into an existing base type without the need to create a new derived type, recompile, or otherwise modify the base type (MSDN, 2015).

In modelling language engineering, several techniques for metamodel and DSML composition have been proposed (Vallecillo, 2010). We focus on those that allow for metamodel customisation beyond inheritance. UML 2 provides the profile mechanism for metamodel customising, particularly applicable for UML 2 family of languages. In *UML Profiles*, selected concepts of the UML metamodel may be extended using stereotypes. With UML 2, profiles have been improved from a lightweight customisation approach to a sound mechanism for both metamodel customisation and for the design of new languages (Selic, 2007). In the current UML version 2.4.1 (OMG, 2011), the concept of the *Stereotype* is a full metaclass that specialises the *Class* concept and extends it through the explicit association *Extension*. As a subclass of the *Class*, a stereotype may own properties that extend the base class. Furthermore, stereotypes allow for the creation of new associations between stereotypes and other metamodel elements. The notion of a stereotype is comparable to our extender element, and the extension association with our extension operator. However, instead of introducing a separate metaclass for it, we add extensibility capability to the corresponding metaclass itself, with the only constraint that such extender element must be abstract. Hence, our extender element is simply a supporting metamodelling construct that extends an existing element while not being instantiable on the model-level. Since the extension occurs in design and compile time, no further model-level mechanisms are required to correlate the instances of a stereotype with instances of a base element. Although, it is claimed that profiles are made generic and compatible for any MOF-based language (Selic, 2007; OMG, 2011), to our best knowledge, we are not aware of any other profile applications than those for UML.

In (Langer et al., 2012), the idea of profiles is applied on Ecore, however, not on the meta-metamodel level but on the metamodel level through metalevel lifting. The so-called *EMF Profiles* help to customise arbitrary DSMLs that are based on EMF Ecore. Since the same idea of UML Profiles is applied, similarities and differences to our approach apply as mentioned before for UML profiles. Furthermore, two additional mechanisms are added that increase profile reuse, *Generic Profiles* and *Meta Profiles*. Generic profiles are based on generic types. This is inherently supported in our approach through the usage of abstract elements when defining mixin and extender modules. Meta Profiles aim at applying extensions to the constructs of the meta-metalevel, such that extensions are applicable for all DSMLs. This is an interesting approach for massive extensions, we do not yet support.

In (Braun and Esswein, 2015) a similar idea of adapting the UML Stereotype concept towards a mechanism for generic metamodel extensions, however only on the conceptual level, is proposed. Unlike in (Langer et al., 2012) and similar to our approach, the authors, propose an extension on the meta-metamodel level, with an application focus on enterprise modelling languages. As in the UML/MOF stereotype mechanism, the stereotype construct is defined as a separate metaclass, however not only for the Class construct, but multiplied for each metaclass type (model type, property etc.). Each stereotype construct has a limited set of extension possibilities. In our approach, we fully reuse existing metaclasses as abstract constructs to capture extensions, and, instead, define precise extension semantics on the extension operator.

Another metamodel extensibility concept common to UML and MOF is the *PackageMerge*. Package merge is used to merge elements and the content of two packages. As noted in (Selic, 2011), package merge allows for incremental metamodel refinement, because one can define an extending element (increment) with the same name as the base element, add additional properties to it and merge it with the base. The semantics of the package merge is similar to that of generalisation, with a difference that the derived element has the same name as the base. In comparison to our work, package merge has similarities to both mixin and extension. However, the

difference is that we do not rely on name matching algorithm but on explicit relationships defined by language engineer, which contributes to an increased soundness in metamodel composition. Furthermore, both mixins and extensions are applied on the level of metamodel elements to allow for a fine-grained and precise extension definition, whereas package merge operates on the level of packages, which potentially opens the door for uncontrolled reuse and the need for metamodel pruning techniques such as *Package Unmerge* (Fondement, Frédéric and Muller, Pierre-Alain and Thiry, Laurent and Wittmann, Brice and Forestier, Germain, 2013). Another MOF extensibility capability is the *Extension*, a lightweight approach to annotate existing metamodel elements with *Tags* that, however, solely represent simple name-value pairs.

Apart from the mainstream metamodelling approaches, in (de Lara and Guerra, 2013), generic programming techniques such as *concepts*, *templates* and *mixin layers* are applied for metamodelling in order to increase the support for abstraction, modularity, reusability and extensibility of (meta)models and corresponding model management operations. Focusing on the usage of mixins, mixin layers rely on templating technique that allows for defining templated metamodel extensions (mixin layers), that can be applied on metamodels that conform to template parameters. The basic idea is to use the parameterised inheritance to realise a generic mixin. The "instantiation" of the template binds the mixin layer to a concrete metamodel that is the subject to extension and that conforms to the structure defined by the parameter type (concept). While the authors introduce templates and template instantiation to realise generic mixins and their application, we rely on the abstract metaclasses, and mixin and extension operators. Instead of applying the parameterised inheritance to achieve the flexibility of mixin applications, we define extenders that insert mixins into arbitrary elements. However, we believe that the two approaches are complementary. While acknowledging the power of templates for specifying the generic concepts that promote abstraction, modularity and reuse, our mixin inclusion, and, in particular, *extension* operator may be applied on the level of templates to allow for mixing and extending of template definitions themselves.

Finally, in (Jézéquel et al., 2013) an approach to design domain-specific languages based on multiple meta-languages within the Kermeta language workbench is proposed. The composition operators *aspect* and *require* are introduced that allow for the composition of language elements such as abstract syntax, static semantics, and behavioral semantics. Inspired from the open class concept, the aspect allows to reopen a previously created class and to add features. In what it does, the aspect is similar to our extension operator. Differently to our approach, the focus is on modular composition of language concerns, whereas we concentrate on the pivotal language aspect, the metamodel. The authors state that the modularisation does not apply for the metamodel aspect (for which definition the EMOF is used).

6 CONCLUSIONS

This work represents a contribution to the field of metamodel composition and customisation, in the context of metamodel-based modelling language engineering. Our approach is based on the notions of Mixins and Extenders and appropriate composition operators, that allow for flexible, modular metamodel customisation. Mixin and extension-based metamodel composition facilitate reuse and contribute to more flexibility and overall efficiency in metamodel definition. Mixins allow for the creation of reusable aspectual metamodel element extensions that can be combined by arbitrary metamodel elements using the mixin inclusion operator. Furthermore, the extension operator allows for the injection of structural features into otherwise non-modifiable base metamodel elements in a non-intrusive way by relying on their implicit interfaces. While mixin inclusion complements single inheritance, and, at the same time, represents a lightweight alternative to multiple inheritance, extension resolves the issue of subclass imperative, an important issue in metamodel customisation. We illustrated the usefulness of the approach based on a running example of the simplified BPMN metamodel customisation. We also elaborated on the application of our approach within the metamodelling tool ADOxx. Although we explained the syntax and semantics of the operators based on ADOxx, the concepts may be mapped to other metamodelling languages and tools, as well.

In this work, we focused primarily on the metamodel as a pivotal part of language definition. A part of our future work at OMILab [1] will focus on investigating how modular approach based on mixins and extensions may be applied on the composition of other language elements such notation and semantics. Furthermore, with an increased usage of mixins and extenders in metamodel composition, we will work on identifying common patterns for modular metamodel engineering.

[1] http://omilab.org

REFERENCES

Ancona, D., Lagorio, G., and Zucca, E. (2000). Jam-a Smooth Extension of Java with Mixins. In *ECOOP 2000ŮObject-Oriented Programming*, pages 154–178. Springer.

Aßmann, U. (2003). *Invasive Software Composition*. Springer.

Bracha, G. (1992). *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah.

Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *ACM SIGPLAN Notices*, volume 25, pages 303–311. ACM.

Bracha, G. and Griswold, D. (1996). Extending Smalltalk with Mixins. In *Workshop on Extending Smalltalk*.

Braun, R. and Esswein, W. (2015). Designing Dialects of Enterprise Modeling Languages with the Profiling Technique. In *19th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2015, Adelaide, Australia, September 21-25, 2015*, pages 60–67.

de Lara, J. and Guerra, E. (2013). From Types to Type Requirements: Genericity for Model-Driven Engineering. *Software & Systems Modeling*, 12(3):453–474.

Fondement, Frédéric and Muller, Pierre-Alain and Thiry, Laurent and Wittmann, Brice and Forestier, Germain (2013). Big Metamodels are Evil. In *Model-Driven Engineering Languages and Systems*, pages 138–153. Springer.

Herbst, J., Junginger, S., and Kühn, H. (1997). Simulation in Financial Services with the Business Process Management System ADONIS. In *Proceedings of the 9th European Simulation Symposium*.

Jézéquel, J.-M., Combemale, B., Barais, O., Monperrus, M., and Fouquet, F. (2013). Mashup of Metalanguages and its Implementation in the Kermeta Language Workbench. *Software & Systems Modeling*, pages 1–16.

Junginger, S., Kühn, H., Strobl, R., and Karagiannis, D. (2000). Ein Geschäftsprozessmanagement-Werkzeug der nächsten Generation - ADONIS: Konzeption und Anwendungen. *Wirtschaftsinformatik*, 42(5):392–401.

Kelly, S., Lyytinen, K., and Rossi, M. (1996). Metaedit+ a Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Advanced Information Systems Engineering*, pages 1–21. Springer.

Kühn, H. (2010). The ADOxx Metamodelling Platform. In *Workshop on Methods as Plug-Ins for Meta-Modelling, Klagenfurt, Austria*.

Langer, P., Wieland, K., Wimmer, M., Cabot, J., et al. (2012). EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):1–29.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17.

Moon, D. A. (1986). Object-oriented programming with Flavors. In *ACM Sigplan Notices*, volume 21, pages 1–8. ACM.

MSDN (2015). Extension Methods (C# Programming Guide). https://msdn.microsoft.com/en-us//library/bb383977.aspx.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the Scala programming language. Technical report, EPFL.

OMG (2011). UML 2.4.1 Infrastructure Specification. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/.

OMG (2013). Business Process Model and Notation (BPMN) Version 2.0.2. http://www.omg.org/spec/BPMN/2.0.2/PDF.

OMG (2014). Meta Object Facility (MOF) Version 2.4.2. http://www.omg.org/spec/MOF/2.4.2/.

OMILab (2015). ADOxx Metamodelling Platform. http://www.adoxx.org.

Rausch, T., Kuehn, H., Murzek, M., and Brennan, T. (2011). Making BPMN 2.0 Fit for Full Business Use. *BPMN 2.0 Handbook Second Edition*, page 189.

Selic, B. (2007). A Systematic Approach to Domain-specific Language Design using UML. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. 10th IEEE International Symposium on*, pages 2–9. IEEE.

Selic, B. (2011). The Theory and Practice of Modeling Language Design for Model-Based Software Engineering - A Personal Perspective. In *Generative and Transformational Techniques in Software Engineering III*, pages 290–321. Springer.

Smaragdakis, Y. and Batory, D. (2001). Mixin-based Programming in C++. In *Generative and Component-based Software Engineering*, pages 164–178. Springer.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

Vallecillo, A. (2010). On the Combination of Domain Specific Modeling Languages. In *Proceedings of European Conference on Modelling Foundations and Applications, 2010. (ECMFA 2010)*, volume 6138 of *LNCS*, pages 305–320. Springer.