Typing and Subtyping of Metamodels

Henning Berg and Birger Møller-Pedersen
Department of Informatics, University of Oslo, Oslo, Norway

Keywords: Metamodelling, Typing, Subtyping, Domain-specific Modelling, Languages.

Abstract: In model-driven engineering, models are considered first-class entities. Model-driven engineering has been around for over a decade. Still, there has not been much work on how to type models or metamodels, which is important to realise true model-driven software development. In this paper, we discuss how a metamodel can be typed by means of an enclosing class whose state can be utilised by tools such as editors and interpreters. This allows using established object-oriented mechanisms on the metamodel level and supports specialisation of metamodels.

1 INTRODUCTION

Model-Driven Engineering (MDE) (Kent, 2002) is a collective term for a number of approaches and methodologies for software development in which models are first-class entities. MDE can be seen as a natural progression of object-orientation by raising the abstraction level from the class level to the model level. A model is a set of interconnected objects, whose descriptions are formalised by a metamodel (class model). In spite of MDE's model-centric view on software design and development, most MDE technologies and tools do not have native support for typing models or metamodels. This has consequences with respect to reuse of models, model transformations and interpreters. The notion of polymorphism at the metamodel level is also unclear, as the type of a metamodel is not well defined. The work of (Khne, 2010) motivates strongly why model substitutability is a valuable property to aim for in MDE, whereas (Khne, 2006) discusses using inheritance, in the form of subtype specialisations, as a basic relationship between models.

The work of (Steel and Jzquel, 2007) presents one approach for realising model typing in MDE. In particular, the work adresses concerns related to reuse of model transformations and interpreters, or in general, situations where externally defined code should be applicable to a number of different models all sharing a minimum set of properties as specified by a reference model type. However, there are still many open questions on how to support cases where model types also cover behavioural semantics, and not just model structure, software evolution and how to define functional model types. That is, model types whose definition also cover behavioural semantics in addition to structure.

The success of object-orientation is to a large extent a consequence of its powerful mechanisms, e.g. specialisation, polymorphism and composition (using object references). The MDE philosophy supports the idea that such mechanisms should also be available at the model and metamodel levels. Many mechanisms address these aspects of (meta)model usage and evolution, e.g. (Fabro et al., 2006)(Kolovos et al., 2006)(Groher and Voelter, 2007)(Fleurey et al., 2008)(Morin et al., 2009)(Morin et al., 2008). However, these mechanisms require the use of additional frameworks. Furthermore, composition and variability directives are described in separate resources (files) in the form of either a weaving model, pointcut model or composition/variability rules. Such additional resources complicate reuse. They also pose certain challenges in maintaining files to reflect alterations of the metamodels.

The notion of *specialising* (as in subtyping) metamodels has not received the same attention as model transformations and composition. In this paper, we discuss how metamodels can be typed by nesting them within an enclosing class. We will see how the enclosing class may indeed represent the type of the enclosed metamodel. The enclosing class can be subtyped which allows us to specialise metamodels. We will focus on metamodels whose behavioural semantics is defined in methods or operations, as supported by the *Eclipse Modeling Framework (EMF)* (Eclipse-

Berg H. and Møller-pedersen B..
Typing and Subtyping of Metamodels.
DOI: 10.5220/0004713901110118
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 111-118
ISBN: 978-989-758-007-9

Foundation, 2013a), *Kermeta* (Muller et al., 2005) and *Epsilon Object Language (EOL)* (EclipseFoundation, 2013b).

The paper is organised in five main sections. Section 2 discusses the basic mechanics of using class nesting to type metamodels, and the purpose of typing metamodels. Section 3 delves into the matter of using class nesting for defining metamodel types, whereas Section 4 presents related work. Section 5 concludes the paper.

2 METAMODEL TYPES

A metamodel defined in the *Essential MetaObject Facility (EMOF)* (OMG, 2013) architecture comprises a set of classes contained in one or more packages, whose objects constitute one or more models. The classes are related by association and specialisation relationships (in the form of class properties). A metamodel can be uniquely identified by the name and namespace of its containing package. However, a package is not a semantically powerful concept (Monperrus et al., 2009), and can not be used as a type specification for the contained metamodel. Specifically, it is not clear how different packages relate and which operations that can globally be applied on the data (model objects) described by the package contents (classes).

A class specifies the type of its instances. The type is defined by the attributes and operations of the class. In addition, non-static nested classes (inner classes) contribute to the type, as they can be considered class-valued attributes. In this paper, we pursue the idea of defining a metamodel within an enclosing class. The enclosing class is contained in a package. The purpose of defining a metamodel within an enclosing class is that the class explicitly describes a type for the metamodel. Hence, using an enclosing class allows us to take advantage of established principles of object-orientation - at the metamodel level. The metamodel's type is that of the enclosing class. In the context of this paper, we will simply refer to a class that encloses a metamodel as a *metamodel type*. A specific instance of a metamodel type represents a specific metamodel/language, and models of this metamodel/language will be in terms of objects of the classes enclosed in the specific instance. The instances of a given metamodel type will be generated by tools, while modellers will only be concerned with making models using the enclosed classes.

2.1 Definitions and Example

Definition 1. *A metamodel type is an enclosing class containing an arbitrary number of non-static nested classes, attributes and operations. The nested classes constitute an EMOF-compatible metamodel. A metamodel type τ_m can be described as the sequence $\langle name, c, a, o \rangle \cdot s$, where $c \subset \mathcal{C}$ - a finite set of EMOF-compatible (nested) classes, $a \subset \mathcal{A}$ - a finite set of attributes (or references), $o \subset \mathcal{O}$ - a finite set of operations and $s \subset \mathcal{T}$ - a finite set of super metamodel types.*

An example of a state machine metamodel type, named TStateMachine, is given below:

$\tau_m = \langle TStateMachine, \{StateMachine, State, Transition, Event\}, \{sm : StateMachine, events : Event\}, \{transitionTable : String\} \rangle \cdot nil$

Definition 2. *A metamodel type instance is an object of the class defining a metamodel type. The metamodel classes of a metamodel type can be accessed and instantiated via a metamodel type instance. Several models can be created using the same instance. A metamodel type instance may have attributes (according to the metamodel type definition) whose values can be used to customise the metamodel.*

We will use Kermeta to illustrate the idea of metamodel types[1]. However, we will only use a subset of Kermeta to avoid complicating the picture. Kermeta is an object-oriented language for creating EMOF-compatible metamodels. It allows specifying the behavioural semantics of metamodels within class operations. The operations are invoked at runtime when executing a model/program. We do not discuss static semantics (OCL) in this paper.

Figure 1 gives the metamodel type TStateMachine implemented in Kermeta syntax. The metamodel type encloses a metamodel/language for modelling of state machines. The metamodel comprises the four classes: StateMachine, State, Transition and Event. StateMachine is the top node class of the metamodel from which all other classes are reachable through relationships. There is a reference typed with this class in the enclosing TStateMachine class. This reference is used to access the current model being processed by a tool (editor, interpreter, etc.). The metamodel details are not of interest, therefore three consecutive dots are used to represent content.

As seen in Figure 1, the enclosing class has a reference typed with the metamodel's top node class. It also has a global operation named transitionTable() that

[1]Note that the current version of EMOF/Kermeta does not support nesting of classes as discussed in this paper.

returns a textual description of all possible state transitions captured by a state machine model (i.e. the model referenced by the sm reference). A step of the state machine is performed by invoking step(...) in the State class. The step is carried out if the current state has a transition whose event value is equal to the operation argument.

```
package state_machine;

class TStateMachine {
  // Reference typed with the top node metamodel class
  reference sm : StateMachine[1..1]

  // Events that may occur
  attribute events : Event[1..*]

  //  Operation global to all the metamodel classes
  operation transitionTable() : String is do
    sm.states.each{ s | ... }
    ...
    result := ...
  end

  // Metamodel classes
  class StateMachine {
    attribute states : State[0..*]
    reference currentState : State[1..1]
    reference initialState : State[1..1]
    ...
  }

  class State {
    attribute name : String
    reference incoming : Transition[0..*]
    attribute outgoing : Transition[0..*]#source
    operation step( event : Event ) is do ... end
    ...
  }

  class Transition {
    reference source : State[1..1]#outgoing
    reference target : State[1..1]
    reference event : Event[1..1]
    operation trigger() is do ... end
  }

  class Event { ... }
}
```

Figure 1: A simple metamodel type for state machines (language).

A weighted state machine is a variant of the basic state machine that supports the description of a probabilistic aspect of events: how likely that a given transition should be triggered. This aspect can be applied to the basic state machine using specialisation. A first attempt to define this special kind of state machine would be to define a subclass of StateMachine. This seems obvious, as objects of the class StateMachine represent state machine models, and as such the class StateMachine appears to be the type of all these models. However, this would not work as intended, as the addition should be in the Transition class. Hence, a next step would be to create a subclass of Transition (in addition to the existing Transition class). However, this would imply that even simple state machine models might have weighted transitions. Instead, by defining the additional properties of a weighted state machine within a subclass of TStateMachine, we are able to specialise the state machine metamodel as a holistic entity and clearly differentiate the state machine variants while still being able to use tools defined according to the general variant of the state machine metamodel (tools e.g. editors will not be able to instantiate new classes that have been added in subtypes). The existing models of the initial state machine metamodel are still valid, as changes and added properties are given in the metamodel type variant. Existing tools may then invoke redefined virtual operations in the nested classes of the metamodel subtype. A metamodel type for a weighted state machine is given below (the arrows indicate inheritance):

$\tau_{mw} = \langle TWeightedStateMachine, \{State \uparrow State, Transition \uparrow Transition\}, \{\}, \{\}\rangle \cdot \{TStateMachine\}$

Figure 2 illustrates the new metamodel type in Kermeta. Notice that TWeightedStateMachine is as a specialisation of TStateMachine. Specifically, the classes Transition and State are given additional properties.

```
package weighted_state_machine;
require "state_machine.kmt"

class TWeightedStateMachine inherits TStateMachine {
  class State inherits TStateMachine.State { ... }
  class Transition inherits TStateMachine.Transition {
    attribute probability : Real
  }
}
```

Figure 2: A metamodel type for weighted state machines.

Alternatively, a state machine that supports composite states can be defined as:

$\tau_{mc} = \langle TCompositeStateMachine, \{CompositeState \uparrow State, \}, \{\}, \{\}\rangle \cdot \{TStateMachine\}$

Figure 3 shows the metamodel type in Kermeta syntax.

```
package composite_state_machine;
require "state_machine.kmt"

class TCompositeStateMachine inherits TStateMachine {
  class CompositeState inherits State {
    attribute stateMachine : StateMachine[0..1]
  }
}
```

Figure 3: A state machine metamodel type with simple and composite state constructs.

Note that this metamodel type includes two classes for modelling of states: State and CompositeState.

2.2 Specialisation and Polymorphism

Changing or refining an artefact of a system is not trivial since changes may impact other parts of the system, or even other systems. Changes made to artefacts at higher abstraction levels typically are more severe when it comes to impacting other parts of a

software system. A metamodel is a model describing a set of models, i.e. the language of realisable models (Favre, 2004). Changing a metamodel impacts all the conformant models. That is, the language of valid models that can be recognised is changed. In most cases, changing a metamodel may render its existing conformant models incompatible. This in turn requires manually changing the models or automating this process by creating a model transformation that incorporates knowledge about the changes to apply. Changes to a metamodel also impact its tools. By using specialisation based upon metamodel types it is possible to create new metamodel variations without render existing tools unusable. That is, the tools will still work by invoking redefined virtual operations. Hence, some of the challenges of software evolution (additions) can be tackled.

One of the main contributions using metamodel types is the ability to use polymorphism. In the example, the metamodel type TWeightedStateMachine may be used as a substitute for TStateMachine. Importantly, this can be achieved without type casting. External code defined according to basic state machines can still be used with weighted state machines.

Metamodel type hierarchies may form a type system that facilitates reuse of commonly occurring metamodel structure and semantics (Cho and Gray, 2011). Using an enclosing class to specify a metamodel type yields a high degree of encapsulation; a tool that is built to be compatible with the TStateMachine type can also operate on subtypes of this, e.g. TWeightedStateMachine.

3 IMPLICATIONS OF METAMODEL TYPES

3.1 Interpretation and Code Generation

In Kermeta, the metamodels' behavioural semantics are defined in class operations. For instance, the semantics for stepping/triggering a new state in the state machine metamodel is defined in the step(...) and trigger() operations in the State and Transition classes. The exact definition of this semantics is not interesting. However, it is clear that the default stepping/triggering semantics will not suffice for a weighted state machine. By subtyping State and Transition, the semantics of these classes can be redefined for the new state machine type. The main point here is that a framework for interpretation of state machine models will still work on the redefined semantics due to dynamic binding, e.g. such a framework may invoke the step(...) and trigger() operations. Figure 4 shows a skeletal of such a framework. This code will work regardless of state machine variants (a consequence of virtual operations).

```
class StateMachineInterpreter
{
  operation interpret( TStateMachine.StateMachine sm ) is do
    var initial : State init sm.initialState
    ...
    initial.step( event )
    ...
  end
}
```

Figure 4: Excerpt of an interpreter for execution of state machine models.

A typical approach for generating executable DSLs is to use code generators that work on purely structural metamodels/models.

However, the operations may also be used to implement a code generator (where each class contains code for a target language). Since we allow subtyping both the enclosing class and the inner classes of a metamodel type, we are able to redefine the code generator using virtual operations (and types).

3.2 Analysis Tools

The operations in the classes may also be used to generate information about the conformant models. In particular the enclosing class may contain operations that work on models as a whole. We have given one such operation in the TStateMachine, namely transitionTable(). This operation contains semantics that is not part of of the metamodel/language for creating state machines. Yet, it allows calculating information about a state machine that can be presented to the modeller during the modelling process. By using subtyping it is possible for analysis tools to work on metamodel variants, since the access points of the analysis tools are predefined as global operations in the enclosing class.

3.3 Type Safety

In this paper, we relate variants of metamodel types using the subtyping relation (Liskov and Wing, 1994). Subtyping imposes certain restrictions on subtypes: the parameter types of an operation are required to be contravariant, whereas the return type needs to be covariant. What this means is that virtual operations of subtypes can be invoked type-safely in place of their supertype equivalents. As pointed out in (Steel and Jzquel, 2007), types of class properties are required to be invariant in *MetaObject Facility (MOF)* (OMG, 2013) metamodels. This latter requirement is difficult to fulfill, as additions of attributes and references

to a class is common when creating metamodel variants. Such additions change the type of the containing class, which in turn results in a covariant redefinition of attributes and references that are typed with the class. Let us see how this affects metamodel types as presented in this paper.

There are two places where subtyping occurs in creating a metamodel type variant. First, the enclosing class is subtyped. Second, the inner classes constituting the metamodel may be subtyped selectively depending on which specialisations that are required. Recall how the Transition class needs an additional attribute probability to create a weighted state machine metamodel from the basic state machine metamodel. Let us assume that this would be the only required addition to the basic state machine metamodel. What we have now is a situation of covariant type redefinition. The attributes incoming and outgoing of the State class (see Figure 1) are typed with Transition. The new Transition class variant contains an additional attribute. Hence, the types of the incoming and outgoing references in the State class of the metamodel type variant (TWeightedStateMachine) are not invariant, but covariant. A potential problem would occur when related metamodel types are mixed, e.g. when a model contains instances of both state machine and weighted state machine metamodel classes. However, these situations do not occur since e.g. a model editor has to instantiate either of these types. Hence, it is not possible to instantiate the subtype of the Transition class when creating a basic state machine.

Attributes and references have multiplicities which indicate the possible amount of values/objects the properties may take. We consider attributes and references to be immutable.

An important ability in metamodels is to define bi-directional relationships. Using bi-directional relationships in the nested metamodel of a metamodel type does not induce any new challenges.

According to (OMG, 2011), classes in MOF do not define XMI namespaces. Nesting of classifiers may thus yield name collisions. However, the idea presented in this paper utilises a constrained form of class nesting, where the enclosing class has a special role. We do not discuss arbitrary nesting of classes.

3.4 Multiple Inheritance

We have seen how metamodel types can be used to represent metamodel patterns or fragments, e.g. a state machine. We have also discussed how a metamodel can be considered and interpreted from the perspective of one specific metamodel type. Seen in the light of this, a metamodel may have several types. Put differently, a metamodel can be constructed by combining an arbitrary number of patterns. This means that a metamodel can also be considered from several perspectives depending on situation and purpose (e.g. a tool may present several viewpoints to the user, where each viewpoint correspond to a metamodel type).

```
package metamodel;
require "state_machine.kmt"
require "game.kmt"

class TMetamodel inherits TStateMachine, TGame
  ...
}
```

Figure 5: Using multiple inheritance to relate metamodel types (yielding a composite metamodel).

Regardless of specialisations of the inner classes, the enclosed metamodel of TMetamodel can still be considered from two perspectives/aspects: state machines and game. That is, tools defined for TStateMachine and TGame can still be used. For example, it is convenient to analyse a model conforming to the composite metamodel from the state machine perspective alone, e.g. by writing out the state transition table or similar. This is possible regardless of the added properties in specialisations of the inner classes.

Using multiple inheritance may potentially require resolution of name conflicts. There are several approaches to improve the applicability of multiple inheritance, e.g. Kermeta allows the modeller to explicitly specify the operation to override in ambigious situations. We will not go into details on this subject.

3.5 Defining Metamodel Types

A metamodel type can be created automatically by enclosing the metamodel in a class. Additional attributes and operations may then be added by the metamodel developer if required. An enclosing class may also be created implicitly by tools, e.g. if a metamodel is not defined as a metamodel type. This works as long as no additional elements are required in the enclosing class.

3.6 Using Virtual Classes and Generic Parameters

The classes of a metamodel type can both be defined as virtual and utilise generic type parameters. Let us return to our example. If the language for making metamodels (e.g. Kermeta) supports virtual classes (Madsen and Mller-Pedersen, 1989), then the Transition could be defined as a virtual class and then redefined in TWeightedStateMachine (by extending the Tran-

sition class). This allows code in TStateMachine to generate Transition objects with the additional property, given that the context in which this code is executed is an object of TWeightedStateMachine. For an in-depth discussion on this topic, see (Berg et al., 2011). Virtual classes also allows existing tools like editors to instantiate redefined classes in subtypes.

3.7 Metamodel Customisation

In its simplest form, an enclosing class does not contain other elements than the nested metamodel classes. The enclosing class may also contain attributes and operations. This adds a new dimension to metamodels. Specifically, the state of a metamodel type object can be used to customise the behavioural semantics of its encapsulated metamodel. As an example, the behavioural semantics of a metamodel (as defined in operations) may use different algorithms depending on context. These algorithms may share basic properties (attributes) whose values can be changed with the intention of tuning the behavioural semantics for a specific usage or context. Being able to adjust these properties simultaneously for all the algorithms allows customising the semantics easily without changing the actual models. The properties with their values are in the object of the enclosing class. Hence, this object's state captures an (execution) configuration for models of a given metamodel/language. Changing such values for a metamodel/language would change the meaning for all conformant models/programs. It would also be possible to maintain several objects of the enclosing class and thereby facilitate execution profiles (serialised to files). However, this will give rise to an additional level of polymorphism as different object states give different execution results.

4 RELATED WORK

There are several mechanisms that address model composition and variability. Some of these are discussed in (Berg and Mller-Pedersen, 2013). Common to these mechanisms is their external definition from the language used to define the models and/or metamodels. Moreover, most mechanisms use some kind of merging techniques to combine the metamodels which compromises the principle of encapsulation.

We have used Kermeta to illustrate metamodel types. Kermeta features a mechanism known as *static introduction*. This mechanism allows specifying partial class definitions using *aspects*. Several aspect definitions are combined (or woven) at runtime to form the definition of a class. The mechanism allows defining new aspects that are combined with an existing class definition. Aspects allow creating metamodel variants. However, they can not be used to type a metamodel - the resulting classes are contained in a regular package.

Model types, as described in (Steel and Jzquel, 2007) resembles the work of this paper. There are some differences and similarities that we will discuss. First, a model type can be seen as a type-safe set of an arbitrary number of model object types. The model type mechanism defines a conformance relation between model types, which allows reusing code or transformations. Specifically, code for manipulating or executing models (interpretation) can be defined according to a reference model type. All models that are typed with a model type conformant to the reference model type can be manipulated or executed by the same code. A model type is created by referring to classes of an arbitrary number of existing metamodels. This is a powerful ability, since classes defined in different packages can be "extracted" to constitute a model type.

A metamodel type, as discussed in this paper, allows typing metamodels as holistic MDE structures. We have used the notion *metamodel* type instead of *model* type because of a significant difference between the approaches. An instance of a metamodel type (object of the enclosing class) represents one particular metamodel. The object can be used to access the metamodels' classes and thereby create model objects. The object of the enclosing class also references one model whose semantics is e.g. intended to be executed at runtime. Conversely, in the work of (Steel and Jzquel, 2007), an instance of a model type can not be used to instantiate the classes of the model type. Instances of these classes are instead added to the model type instance. The model type instance acts as a filter where only objects of the model type's classes can be added to the model type instance successfully. The similarity in this respect between the approaches is that both a metamodel type instance and a model type instance can be used to reference a conformant model. The capabilities of the two model typing approaches differ. Model types are designed to simplify reuse of code from an external perspective, e.g. from the perspective of an interpreter or transformation. Conversely, metamodel types allow creating metamodel variants. By design, metamodel types are functional types. Reuse of code is achieved by creating type variants in the form of subtypes. Model types, on the other hand, are structural. They can not be combined, or be related to create variations.

The inability to use substitution in model typing

is addressed by (Guy et al., 2012). The paper discusses four subtyping mechanisms, and how these allow defining relations between model types as defined in (Steel and Jzquel, 2007). The work differentiates between total and partial subtyping relations, that are either isomorphic (model type matching with respect to properties and operations) or non-isomorphic. According to the definitions of (Guy et al., 2012), using an enclosing class as type specification for metamodels can be seen as a total isomorphic subtyping relation that is declared explicitly. The explicitly declared subtyping relation allows reusing structure of supertypes through inheritance. And, as we have seen, the inherited classes can be redefined. Using an enclosing class supports compile-time checking of the subtyping relations between types. It is stated in (Guy et al., 2012) that it is not possible to achieve type group substitutability using object subtyping. The work on metamodel types shows that this is in fact possible when a metamodel is a property of an object - as realised using class nesting. Hence, we are able to achieve type group substitutability based upon established object subtyping principles. This includes the ability to reuse existing type checking algorithms. The drawback of our approach is that substitutability of metamodels can not be defined partially. However, as illustrated, a metamodel can be typed according to several metamodel types, which addresses this concern to some extent.

An approach for generic specification of metamodel's behaviour is discussed in (de Lara and Guerra, 2011). The approach relies on the use of generic concepts to define behaviour that is applicable to a family of unrelated metamodels. Concepts allow specifying details of models' structure by utilising parameters. A concept can be bound to metamodels that satisfy the concept's requirements using pattern matching. There is no dependency between a metamodel and a concept. Hence, utilising concepts is non-intrusive. Conversely, we have seen how metamodel types are related using subtyping in two levels. In other words, we utilise a typical object-oriented typing scheme that allows defining metamodel variants explicitly using subtyping.

Specialisation relationships between models are carefully discussed in (Khne, 2010). The work formalises two relations for specifying forward- and backward-compatibility between models and applies these relations to, e.g. models related using subtyping. The compatibility relations are defined using a definition of conformance. Forward-compatibility is achieved if instances of a submodel conform to the supermodel, and vice versa for backward-compatibility. One important point discussed is the desire to maximise the forward-compatibility of a language since this allows reusing existing tools on new models (via redefined virtual operations). The subtyping relation ensures a high degree of forward-compatibility. Subtyping also guarantees mutator forward-compatibility. This supports a round-trip between new instances of submodels and support of these by existing tools. That is, the submodel instances appear like supermodel instances. Subtyping is the most restrictive specialisation relationship, with strict behaviour conformance of subtypes. We believe that this type of relationship is the best suited to relate metamodel types.

5 CONCLUSIONS

The work discussed in this paper presents a novel way of typing a metamodel by defining it within an enclosing class. The purpose of the enclosing class is to facilitate object-orientation at the metamodel level that can be utilised by tools. Hence, we achieve a manner of defining functional types for metamodels that can be related using subtyping. An object of the enclosing class is not part of a model, but used by tools to manage the enclosed metamodel. Subtyping ensures substitutability between metamodel types. That is, tools defined according to a metamodel type can be reused on subtype variations of this metamodel type (possible by redefinition of virtual operations). Another important aspect of using subtyping is the ability to maintaining conformance between models and their metamodels.

To the best of our knowledge, there are currently no metamodelling tools or environments that support nesting of classes. Formalising and implementing a complete type system with inner classes is a very time-consuming endeavour. As a first iteration, we have therefore focused on the theoretical approach to the subject.

REFERENCES

Berg, H. and Mller-Pedersen, B. (2013). Type-safe symmetric composition of metamodels using templates. In *7th International Workshop on System Analysis and Modelling (SAM '12), LNCS vol. 7744, pp.160-178. Springer (2013).*

Berg, H., Mller-Pedersen, B., and Krogdahl, S. (2011). Advancing generic metamodels. In *SPLASH '11 Workshops Proceedings, pp.19-24. ACM Press (2011).*

Cho, H. and Gray, J. (2011). Design patterns for metamodels. In *SPLASH '11 Workshops Proceedings, pp.25-32. ACM Press (2011).*

de Lara, J. and Guerra, E. (2011). From types to type requirements: Genericity for model-driven engineering. In *Software and Systems Modeling. Springer (2011)*.

EclipseFoundation, T. (2013a). Eclipse modeling framework (emf).

EclipseFoundation, T. (2013b). Epsilon object language (eol).

Fabro, M. D. D., Bzivin, J., and Valduriez, P. (2006). Weaving models with the eclipse amw plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006*.

Favre, J.-M. (2004). Towards a basic theory to model model driven engineering. In *3rd International Workshop on Software Model Engineering (WISME '04)*.

Fleurey, F., Baudry, B., France, R., and Ghosh, S. (2008). A generic approach for automatic model composition. In *Models in Software Engineering, LNCS vol. 5002, pp.7-15. Springer (2008)*.

Groher, I. and Voelter, M. (2007). Xweave - models and aspects in concert. In *10th International Workshop on Aspect-Oriented Modeling (AOM '07) pp.35-40. ACM Press (2007)*.

Guy, C., Combemale, B., Derrien, S., Steel, J., and Jzquel, J.-M. (2012). On model subtyping. In *Modelling Foundations and Applications, LNCS vol. 7349, pp.400-415. Springer (2012)*.

Kent, S. (2002). Model driven engineering. In *Integrated Formal Methods, LNCS vol. 2335, pp.286-298. Springer (2002)*.

Khne, T. (2006). Matters of (meta-) modeling. In *Software and Systems Modeling, vol. 5, no. 4, pp.387-394. Springer (2006)*.

Khne, T. (2010). An observer-based notion of model inheritance. In *Model Driven Engineering Languages and Systems, LNCS vol. 6394, pp.31-45. Springer (2010)*.

Kolovos, D., Paige, R., and Polack, F. (2006). Merging models with the epsilon merging language (eml). In *Model Driven Engineering Languages and Systems, LNCS vol. 4199, pp.215-229. Springer (2006)*.

Liskov, B. and Wing, J. (1994). A behavioral notion of subtyping. In *ACM Transactions on Programming Languages and Systems, vol. 16, no. 6., pp.1811-1841. ACM Press (1994)*.

Madsen, O. L. and Mller-Pedersen, B. (1989). Virtual classes - a powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89*.

Monperrus, M., Beugnard, A., and Champeau, J. (2009). A definition of abstraction level for metamodels. In *16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '09), pp.315-320, IEEE Computer Society (2009)*.

Morin, B., Klein, J., and Barais, O. (2008). A generic weaver for supporting product lines. In *13th International Workshop on Early Aspects (EA '08), pp.11-18. ACM Press (2008)*.

Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., and Jzquel, J.-M. (2009). Weaving variability into domain metamodels. In *Model Driven Engineering Languages and Systems, LNCS vol. 5795, pp.690-705. Springer (2009)*.

Muller, P.-A., Fleurey, F., and Jzquel, J.-M. (2005). Weaving executability into object-oriented meta-languages. In *Model Driven Engineering Languages and Systems, LNCS vol. 3173, pp.264-278. Springer (2005)*.

OMG (2011). Omg issue 7603, received 27th of july 2004, closed 27th of may 2011.

OMG (2013). Meta object facility (mof) core specification.

Steel, J. and Jzquel, J.-M. (2007). On model typing. In *Software and Systems Modeling, vol. 6, no. 4, pp.401-413. Springer (2007)*.