Type-Safe Symmetric Composition of Metamodels Using Templates

Henning Berg and Birger Møller-Pedersen

Department of Informatics,
Faculty of Mathematics and Natural Sciences, University of Oslo
{hennb,birger}@ifi.uio.no

Abstract. Composition of models is a key operation in model-driven engineering where it is used for, e.g., elaborating models with additional concepts, acquiring a holistic system view, or making model variants. However, there are few state-of-the-art composition mechanisms that support type-safe symmetric composition of metamodels and their behavioural semantics. This hampers the flexible customisation and reuse of metamodels in model-driven engineering approaches. This paper presents a new mechanism for composing metamodels by defining metamodels as reusable templates. Composition of metamodels is achieved using template instantiations that allow customising the metamodel classes as part of the composition process. The work includes a prototypical metamodel composition tool that supports the ideas presented. The result is an approach for composing metamodels in a type-safe manner, where name conflict resolution, composition of behavioural semantics and reuse of tools are supported.

Keywords: Metamodelling, composition, reuse, behavioural semantics, metamodel templates, domain-specific languages.

1 Introduction

Metamodelling is a central aspect of *Model-Driven Engineering (MDE)* [1] where it is used to formalise languages, transformations and domain knowledge. Metamodels can be created in two different ways: directly from scratch or by some kind of model transformation where existing metamodel definitions are used. A model transformation is a process where a set of source models is used as basis for creating a target model. Metamodel composition can be seen as a specific kind of model transformation, with the purpose of elaborating a metamodel with additional concepts or semantics, or weave in variability as part of software product line development.

There exist many different kinds of model composition mechanisms/languages/tools, e.g., *Kompose* [2], *XWeave* [3], *Atlas Model Weaver (AMW)* [4], *Epsilon Merging Language (EML)* [5], *SmartAdapters* [6], *GeKo* [7,8], and *RAM* [9]. Unfortunately, most of these mechanisms are constrained to particular usage scenarios and/or they require a considerable initial effort to facilitate composition of a given set of models. For example, using AMW requires constructing

Ø. Haugen, R. Reed, and R. Gotzhein (Eds.): SAM 2012, LNCS 7744, pp. 160–178, 2013.
© Springer-Verlag Berlin Heidelberg 2013

a weaving model, composition in EML is described using a set of rules whose definition is demanding, SmartAdapters require creating a ConcreteAdapter that describes bindings between the constituent models of a composition. In addition, many composition mechanisms are designed explicitly for composing models rather than metamodels. Hence, composition of behavioural semantics is not addressed. Other limitations of current composition mechanisms are: no resolution of name conflicts, no support for composition of more than two models at the same time, and no support for symmetric composition - models typically take a base or aspect role. While there are a few composition mechanisms available that address these limitations, the work of this paper additionally discusses how existing tools can be reused. This is the main contribution of the paper.

Models are primary artefacts in MDE, whereas model transformations, including model compositions, are important operations on the models. Metamodels are models, yet their composition using state-of-the-art composition mechanisms, as we discuss in the related work, is not flexible enough to support the MDE philosophy. Specifically, composition of metamodels can not be performed in a simple, efficient, and context-free manner. The work of this paper addresses these issues. We will discuss how metamodel templates facilitate composition of metamodels' abstract syntax *and* behavioural semantics in a type-safe manner. Type safety is a requirement to be able to compose behavioural semantics. Our approach builds on the package template mechanism [10,11,12]. Specifically, we extend the package template mechanism with additional features that are particularly useful for metamodel composition, yielding metamodel templates. The ideas and examples have been validated by the construction of a metamodel composition tool[1].

The work is presented as follows. Section 2 gives an overview of our approach and introduces the metamodel template mechanism. In Sect. 3, we illustrate application of the composition tool by constructing a *Domain-Specific Language (DSL)*, while Sect. 4 delves into details on how metamodel templates work and how type safety is preserved with reference to the example application. Section 5 describes a set of new template features specifically designed for composition of metamodels; including the possibility of retyping class attributes. Section 6 presents and reviews several state-of-the-art composition mechanisms and discusses related work. Finally, Sect. 7 concludes the paper.

2 Metamodel Templates and Our Approach

The metamodel composition approach described in this paper is based on metamodel templates. Metamodel templates have taken some of the basic features from the package template mechanism [10,11,12]. A metamodel template (or template for short) comprises a class model that defines a metamodel or metamodel fragment. The class model is compatible with *Ecore/Essential MetaObject Facility (EMOF)* [13,14]. A template has to be instantiated in order to use the

[1] The metamodel composition tool can be found at this URL: http://swat.project.ifi.uio.no/software

classes defined within the template. An instantiation of a template within a given scope (a package or another template) will make the template classes available in this scope, as if they were defined there (unique class copies). The same template can be instantiated several times, both in the same scope and in different scopes. Template classes may be adapted for a specific purpose as part of the template instantiation. This is achieved by renaming classes and class properties (which also affects the types of operations and their parameters' types), by adding new properties to classes, and by merging of classes from different templates (in case more than one template is instantiated in the same scope). Overriding of operations and thereby dynamic binding are also supported. The resulting classes of one template instantiation are not type-compatible with those of other instantiations of that template. Templates can be type checked independently at development time. Type safety is also preserved after template instantiation, and this still holds when classes are customised and merged.

Type checking of classes is required for composing the behavioural semantics of metamodels. There are several environments that support defining behavioural semantics for metamodels, including *Eclipse Modeling Framework (EMF)* [13] and *Kermeta* [15]. In EMF, the behavioural semantics is known as model code and is separated from the abstract syntax of the metamodel. The model code is expressed using Java. In Kermeta, both the abstract syntax and behavioural semantics are defined in the class model, i.e., the class operations contain definitions of the behavioural semantics of the metamodel. The metamodel composition tool discussed in this paper is constructed as a Kermeta pre-processor. It accepts a mixture of Kermeta code and template instantiation code/directives. However, the ideas also apply to EMF and modelling environments like *MetaEdit+* [16] and *Generic Modeling Environment (GME)* [17].

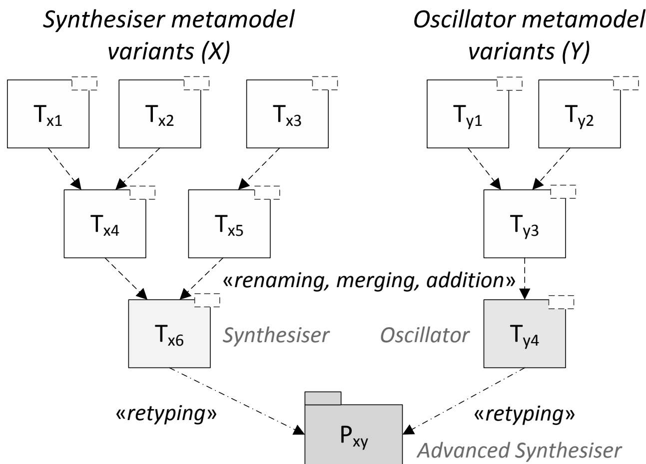

Fig. 1. Building template hierarchies

An overview of how templates can be organised in hierarchies is given in Fig. 1. The figure shows how an advanced synthesiser (P_{xy}) is made by utilising a template for modelling of synthesisers (T_{x6}) and a template for modelling of oscillators (T_{y4}). Each of these templates is created by instantiating other templates that contain metamodels with more basic concepts.

A major difference between our approach and other state-of-the-art composition mechanisms is that instantiation code can be expressed in the same modelling space as the templates are defined; the details of a composition is not defined in separate resources or models. In particular, templates themselves may contain instantiation code which supports complex hierarchical metamodel compositions.

Metamodel composition may result in tools that are no longer compatible with the resulting metamodel. This is unfortunate since the tools have to be manually refactored. The approach of this paper addresses how class attributes and references can be retyped as part of template instantiations according to predefined metamodel integration points (using superclasses), and thereby takes a first step towards tool reuse.

3 Application of the Metamodel Composition Tool

We will use audio processing as the example domain for illustrating the metamodel composition tool and the ideas presented in this paper. Today, there exist a vast number of virtual synthesisers. These synthesisers are realised in the form of software applications and are able to utilise the hardware of a standard computer. Virtual synthesisers replace traditional hardware synthesisers in many contexts. A synthesiser is usually implemented using a software development kit, with an appropriate API defined in a general purpose language such as C++. Programming a synthesiser requires knowledge in signal processing and is considered a challenging task. However, several companies have seen the potential in releasing modelling software that allows building synthesisers using a set of pre-defined building blocks. This allows users to build custom synthesisers without being an expert in signal processing.

Here, we will see how to define and utilise a set of (experimental) metamodels for building a DSL for modelling of synthesisers. There are three metamodels that will be used in the examples. These are named: *Synthesiser*, *Oscillator*, and *Filter*, and are given in Fig. 2. As supported by Kermeta, the behavioural semantics of the metamodels can be captured directly in the class operations. The metamodels are somewhat simplified, e.g., we do not consider all aspects of static semantics like model constraints (OCL).

The Synthesiser metamodel (language) in Fig. 2 can be used to model simple synthesisers. It comprises 11 classes[2]. A synthesiser is built using one or more layers each of which is composed of a sound source and processors. At this point, the sound source is a very simple oscillator. Different types of sound processing are performed by filters and amplifiers. Envelopes and *Low Frequency Oscillators*

[2] The MidiEvent class used in the definition of SoundSource is not included in the figure.

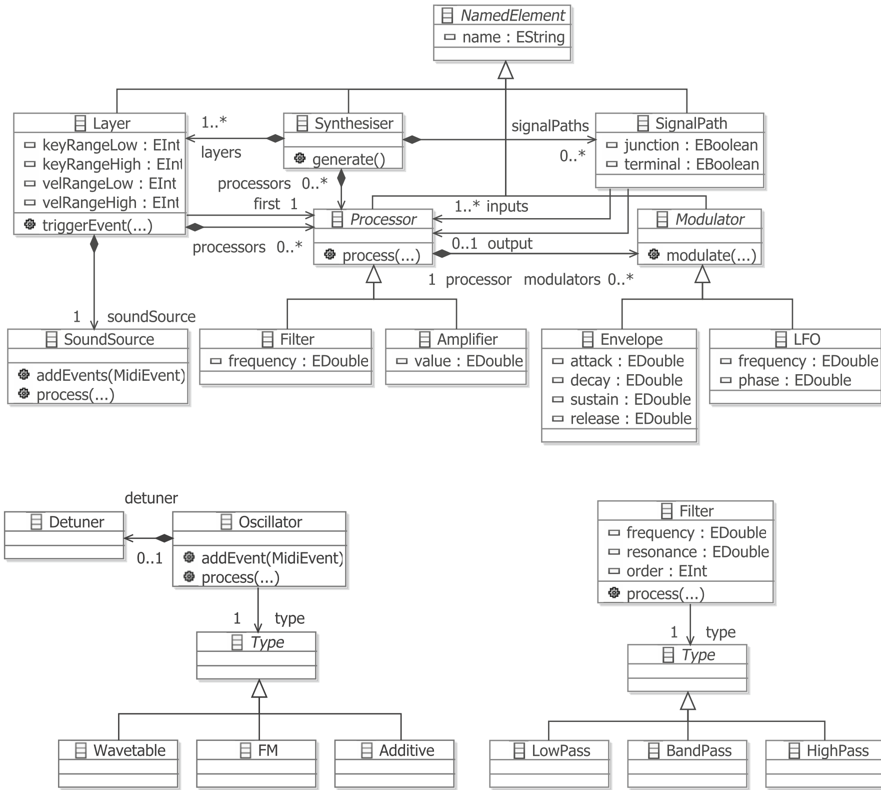

Fig. 2. Metamodels (languages) for modelling of synthesisers, oscillators, and filters

(LFOs) can be used to modulate parameters, for instance the cutoff frequency of a filter. The sound is generated by invoking the generate() operation of the Synthesiser class. We assume that this class contains logic for communicating with a USB musical keyboard. The behavioural semantics is not of interest for explaining the approach of this paper, and is thus excluded.

A synthesiser's sound depends heavily on the sound processing algorithms it uses. There are many different algorithms and methods for both generating sound (synthesis) and processing sound (filtering, effects, etc.). The ability to add/weave in variability to the Synthesiser metamodel is therefore desirable. Two metamodels for modelling of oscillators and filters, respectively, are found in Fig. 2. Only extracts of the metamodels are shown, due to size constraints.

We will now see how the described metamodels can be defined as metamodel templates and then combined to create an elaborated Synthesiser metamodel/DSL. A metamodel is converted to a metamodel template simply by defining it within a template scope, as designated by the keyword template. No other changes have to be done to the metamodel definition. The top of Fig. 3 shows an excerpt of the Synthesiser metamodel template.

```
template SynthesiserTemplate {
  abstract class NamedElement {
    attribute name : String
  }
  class Synthesiser inherits NamedElement {
    attribute layers : Layer[1..*]
    operation generate() is do ... end
    ...
  }
  class Layer inherits NamedElement {
    attribute processors : Processor[0..*]
    ...
  }
  ...
}


package advancedSynthesiser;
require "SynthesiserTemplate.kpt"
require "OscillatorTemplate.kpt"
require "FilterTemplate.kpt"
require "MidiEvent.kmt"

inst SynthesiserTemplate with
  SoundSource => Oscillator
    (addEvents() -> addEventsNative, process() ->  processNative),
  Layer
    (soundSource -> oscillator),
  Filter
    (process() -> processNative, frequency -> frq)
inst OscillatorTemplate with Type => OscillatorType
inst FilterTemplate with Type => FilterType

class Oscillator adds {
  operation addEventsNative( events : Bag<MidiEvent> ) is do
    addEvents( events )
  end

  operation processNative( left : Bag<Real>, right : Bag<Real> ) is do
    process( left, right )
  end
}

class Filter adds {
  operation processNative( left : Bag<Real>, right : Bag<Real> ) is do
    process( left, right )
  end
}
```

Fig. 3. Definition of the Synthesiser template and metamodel variant

In order to create the new Synthesiser metamodel variant, we want to refine the SoundSource and Filter classes of the Synthesiser metamodel. We do this by instantiating the metamodel templates in a package. Instantiation of a template is organised in three parts: the main instantiation statement, a renaming statement, and adds clauses in which additional properties and code can be added to the derived metamodel classes. The two latter parts are optional.

As can be seen in Fig. 3, the Synthesiser, Oscillator, and Filter metamodels are instantiated in the advancedSynthesiser package by the use of the keyword inst. (Alternatively, the instantiations could have been performed within a new template to define the new Synthesiser metamodel as a reusable module.) Arrows indicate atomic transformations, e.g., renaming. There are two types of renaming performed: renaming of classes (=>) and renaming of class properties (->). The SoundSource class is renamed to Oscillator. The Oscillator template contains a class Oscillator as well. Refer to the Oscillator metamodel of Fig. 2. The

semantics of the metamodel template mechanism yields a merge when two classes have the same name. Consequently, the SoundSource class of the Synthesiser template, now renamed to Oscillator, is merged with the Oscillator class of the Oscillator template. However, both the Oscillator classes in question contain equally named operations addEvents(...) and process(...), which introduces name conflicts. These are resolved by renaming the operations of the SoundSource class from the Synthesiser template. Similar considerations are made for the Filter classes originating from the Synthesiser template and the Filter template. The soundSource attribute (containment reference) of the Layer class is renamed to the more appropriate oscillator. The Type classes in the Oscillator and Filter templates are renamed to OscillatorType and FilterType, respectively.

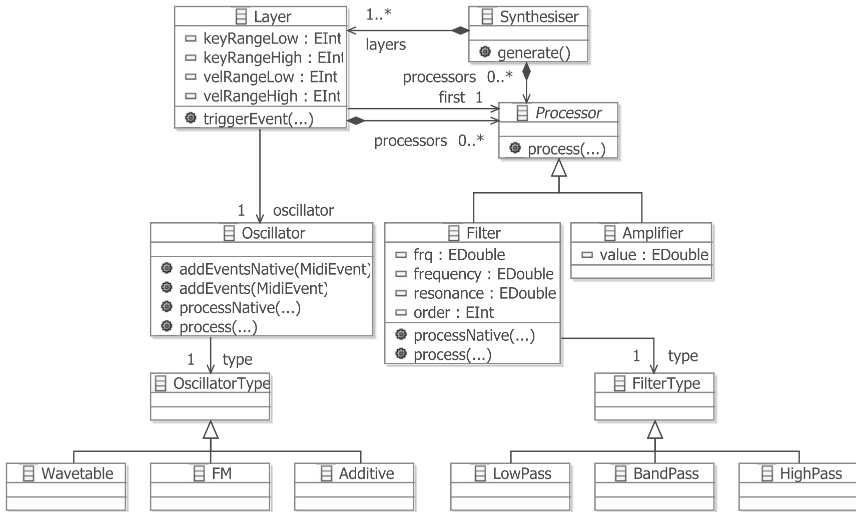

Fig. 4. Excerpt of the resulting metamodel after composition

The behavioural semantics of the Oscillator and Filter metamodels need to be integrated with the semantics of the Synthesiser metamodel. This is achieved using adds clauses, which allow adding new properties to classes and overriding operations. Operations named: addEvents(...) and process(...) are used repeatedly in the Synthesiser classes. These operations capture the behavioural semantics of synthesisers. To compose the semantics, the addEventsNative(...) operation of Oscillator (earlier the addEvents(...) operation of SoundSource) is overridden to invoke the addEvents(...) operation of the class (added from the Oscillator class of the Oscillator template as a consequence of merging). The same overriding is performed for the processNative(...) operation, whose new definition invokes the process(...) operation of the Filter class. As a result, the behavioural semantics of the Oscillator metamodel is used for sound synthesis, while the semantics of the

Filter metamodel replaces the native filter semantics. Fig. 4 shows an excerpt of the resulting metamodel for the Synthesiser variant. At this point tools have to be manually fitted for the new metamodel. We will later see how retyping addresses this.

4 How Metamodel Template Instantiation Works

4.1 Full Static Type Checking

Renaming of classes and class properties, class merging, and addition of class properties and code are performed during processing of the instantiation directives. The resulting classes from a template instantiation do not contain any template-specific code (like inst, adds and so forth).

Full static type checking is an important property of metamodel templates that differentiates this approach from, e.g., package extension [18]. There are two steps in ensuring type safety:

1. Type checking at template development time
2. Type checking of the resulting metamodel during template instantiations

Each template can be type checked at development time independently of other templates. Type safety is also checked when templates are instantiated and their classes customised and combined. Here, we will illustrate the mechanics of how type safety is preserved after processing of renaming transformations.

We have seen how classes and class properties can be renamed as part of a template instantiation. Giving a new name to a class or property is reflected in the derived class definitions resulting from the template instantiations including every place where the class or property is referred. The renaming transformations preserve type safety. We will explain this by using a subset of the Synthesiser metamodel template.

Let us see how renaming of the SoundSource class, its operations, and the soundSource attribute of the Layer class are reflected in the template classes (copies) that are made available in the advancedSynthesiser package. Figure 5 visualises this. First, the class SoundSource is renamed to Oscillator (1). This affects every piece of code that refers to this class (2). Renaming addEvents(...) to addEventsNative(...) (3) also affects the code within the triggerEvent(...) operation (4). Finally, renaming the soundSource attribute of the Layer class (5) is reflected by the code within triggerEvent(...) as well (6). Similar actions are taken for the other renaming statements in the production of the Synthesiser metamodel variant. For clarity, these are not illustrated in the figure. The classes of the package advancedSynthesiser have the definitions found in the lower part of Fig. 5 after template instantiation (and renaming).

Renaming works at more than one level. For example, the Synthesiser template could have been constructed by instantiating other templates within this template. As a consequence, renaming a property as part of instantiating the Synthesiser template could potentially lead to renaming across several levels of

templates. This "deep" kind of renaming is performed by the composition tool and ensures that all attributes, references, variables, operations, and parameters have the correct type when renaming of classes occur. The ability to rename classes and class properties supports reusing the same template multiple times (the classes within the template definitions are not changed; derived class copies are used). That is, the Filter template could have been instantiated twice or more in the package advancedSynthesiser if needed. This allows defining common metamodel patterns in the form of templates, which can then be used multiple times to construct a metamodel [19].

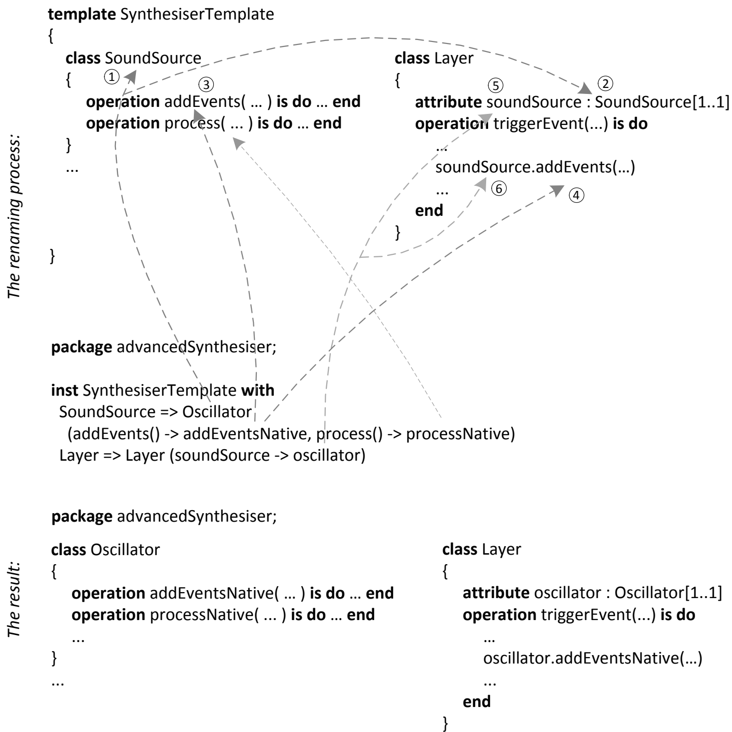

Fig. 5. Renaming of classes and class properties

4.2 Why Symmetry and Type Safety

Model composition comes in two variants: symmetric and asymmetric. In a symmetric model composition process, all constituent models are regarded equal with respect to roles (from the perspective of the composition mechanism). An asymmetric composition process identifies a base model and one or more aspect models. The approach discussed in this paper is symmetric; metamodel template instantiations are not differentiated in the composition process. Note that

the concepts of the Oscillator and Filter metamodels in the example represent two distinct aspects of a synthesiser. However, the composition process does not assign contrasting roles to these metamodels with respect to the Synthesiser metamodel. Thus, the composition process is still symmetric.

Symmetric composition is important to ensure flexibility and support agility. When composing an arbitrary number of models, it is not unlikely that these are required to be composed according to different schemes. Let us consider three metamodels: $\mathcal{MM}_b$, $\mathcal{MM}_{a1}$, and $\mathcal{MM}_{a2}$, and assume that $\mathcal{MM}_b$ takes the role as a base model while the other two metamodels are aspect models, as is the case when using an asymmetric composition mechanism. In such a case, both $\mathcal{MM}_{a1}$ and $\mathcal{MM}_{a2}$ have to be composed directly with $\mathcal{MM}_b$. However, this may not be possible if the aspects overlap, or at best difficult to achieve using an asymmetric composition mechanism. For example, let us assume that $\mathcal{MM}_{a1}$ and $\mathcal{MM}_{a2}$ have classes reflecting the same domain concepts. Instead, by using symmetric composition, $\mathcal{MM}_b$, $\mathcal{MM}_{a1}$, and $\mathcal{MM}_{a2}$ can be composed arbitrarily. Specifically, the metamodels can be composed where overlapping concepts are addressed explicitly in one single composition process.

Renaming and merging of template classes result in new classes that are unknown to other classes of the instantiated templates. As we have seen, the metamodel composition tool addresses this by updating references automatically as part of template instantiations and ensures that type safety is preserved. It also verifies that overriding of operations and addition of classes and behavioural semantics are type-safe[3]. Type checking is particularly important if the metamodel classes have an associated behavioural semantics, as type errors will completely break the integrity of the metamodel. In that case type checking ensures that composition of metamodels, and in particular composition of behavioural semantics, gives an expected result. Additional details on the consistency of the package template features can be found in [20].

5 Tailoring the Metamodel Template Mechanism for Metamodelling

So far we have only used features of metamodel templates that resemble those of package templates. That is: class merging, renaming of classes and class properties, and addition of properties/overriding of operations. These features work well if the required manual rewriting of tools is not an issue. We will here discuss a slightly different approach where we utilise a new set of features. These features may be used together with the basic metamodel template features discussed so far. We will mainly illustrate *retyping* and *namespaces*.

In Fig. 3, we merged the SoundSource class of the Synthesiser template with the Oscillator class from the Oscillator template and composed the behavioural semantics by renaming and overriding the native addEvents(...) and process(...) operations. In the following, we do not want to merge the SoundSource and Oscillator

[3] A simplified type checking is used by the prototype tool.

classes to compose the metamodels. Instead, we give the soundSource attribute of the Layer class a new type specified as an abstract class. This type represents an "interface" between the Layer class and different kinds of sound sources.

Figure 6 gives the definition of an abstract class that declares operations that need to be implemented by a sound source: addEvents(...) and process(...). Note how SoundSourceDef is specified in a separate resource (file) and can thus be acquired in several metamodel templates.

```
// SoundSourceDef.kmt
package ssd;
require "MidiEvent.kmt"

abstract class SoundSourceDef {
  operation addEvents( events : Bag<MidiEvent> ) is abstract
  operation process( left : Bag<Real>, right : Bag<Real> ) is abstract
}
```

Fig. 6. Defining the essential properties of sound sources

```
// SynthesiserTemplate.kpt
require "SoundSourceDef.kmt"

template SynthesiserTemplate {
  class Synthesiser inherits NamedElement { ... }
  class SoundSource inherits ssd :: SoundSourceDef {
    ...
  }

  class Layer inherits NamedElement {
    attribute soundSource : ssd :: SoundSourceDef[1..1]
    ...
  }
  ...
}

// OscillatorTemplate.kpt
require "SoundSourceDef.kmt"

template OscillatorTemplate {
  class Oscillator inherits ssd :: SoundSourceDef {
    method addEvents( events : Bag<MidiEvent> ) is do ...  end
    method process( left : Bag<Real>, right : Bag<Real> ) is do ... end
  }
  ...
}
```

Fig. 7. Using SoundSourceDef in the Synthesiser and Oscillator metamodels

Figure 7 shows how the Synthesiser and Oscillator metamodels are refactored to use SoundSourceDef. This is a design-time decision. For example, the Synthesiser metamodel is designed in a manner that later allows retyping the soundSource attribute. Thus, the soundSource attribute acts as an integration point. Several abstract superclasses could have been used to define additional integration points. These classes define/guide how the Synthesiser metamodel can be integrated with other metamodels using retyping. Notice how the Oscillator

class inherits SoundSourceDef[4]. Hence, the Oscillator metamodel is refactored to be compatible with the Synthesiser metamodel (from the perspective of SoundSourceDef). An important observation is that the Layer class can not utilise class properties in SoundSource that are not defined in SoundSourceDef. Otherwise there may potentially be references to properties that are no longer present in the target type of a retyping transformation.

Integration of the two metamodels can be achieved by giving a new type to the soundSource attribute of the Layer class as part of the template instantiation process. The only requirement for this retyping operation is that the new type is a subtype of SoundSourceDef. See Fig. 8.

```
package advancedSynthesiser;
require "SynthesiserTemplate.kpt"
require "OscillatorTemplate.kpt"

inst t1 : SynthesiserTemplate with
  Layer (soundSource :-> t2 :: Oscillator)
inst t2 : OscillatorTemplate
```

Fig. 8. Retyping the soundSource attribute to Oscillator

In Fig. 8, the soundSource attribute is retyped to the Oscillator class resulting from the instantiation of the Oscillator template. The Oscillator class is referenced using the namespace identifer t2. That is, all the resulting classes of a given template instantiation can be referenced using an identifier. Note the different kind of arrow used, which identifies retyping (:->) instead of renaming.

So what do we achieve by using retyping instead of class merging. First, integration of metamodels' behavioural semantics is achieved by implementing (or overriding) a set of operations as specified by the common supertype. The new operation definitions replace the previous definitions. In Fig. 3, we used adds clauses to combine the semantics. This is thus not required anymore. The inherited operations from the supertype represent an interface between two classes in the different metamodels being integrated, hence, simplifying the integration of the semantics. Second, retyping causes metamodels to be merged according to integration points defined by supertypes. This ensures that existing tools can still be used with minimal required configuration. By using basic metamodel template features and retyping in unison we achieve a powerful mechanism for composing metamodels. Retyping is resolved at template instantiation time. An overview of the integration of two metamodels using retyping is given in Fig. 9.

$\mathcal{MM}_1$ and $\mathcal{MM}_2$ are metamodel templates that are defined independently. Both templates contain classes that inherit from the X and Y classes. Let us focus on the x_1 reference of the M_1 class. The type of x_1 is specified to be of type X. Thus, when using $\mathcal{MM}_1$ without retyping (e.g., when $\mathcal{MM}_1$ is used

[4] Kermeta uses method as keyword for overridden operations.

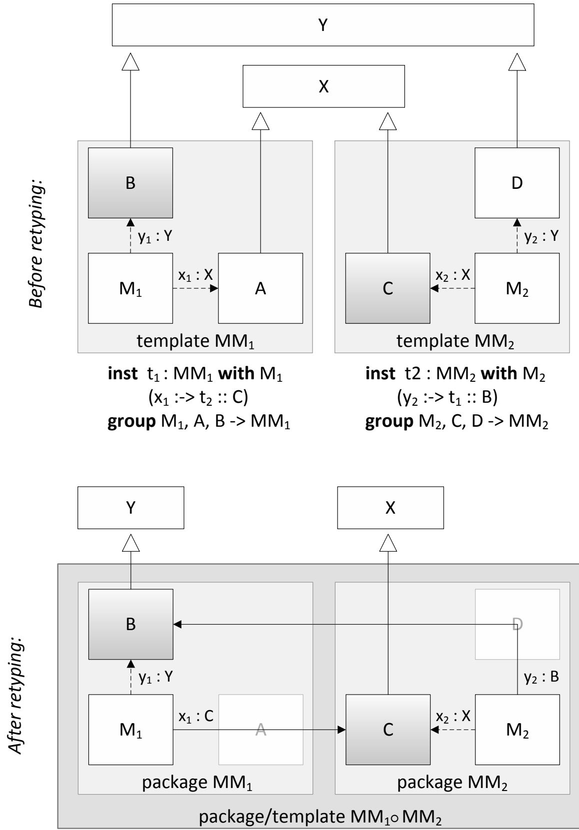

Fig. 9. The retyping process

standalone), the x_1 reference may relate objects of classes that are subtypes of X. There is only one such class in $\mathcal{MM}_1$, namely the A class (indicated with a dashed relationship symbol). Both templates are instantiated within a package (or template) named $\mathcal{MM}_1 \circ \mathcal{MM}_2$. The x_1 reference is retyped to the C class of $\mathcal{MM}_2$ as part of the template instantiations to make an explicit relationship between the M_1 class of $\mathcal{MM}_1$ and the C class of $\mathcal{MM}_2$. As a result, only objects of the C class can be related using the x_1 reference. The main point is that a class can still be utilised by a tool as long as it implements properties of a specified supertype (which the tool supports).

Notice how bi-directional integration is possible, as illustrated by retyping of y_2. Instantiating several templates causes all classes from the different templates to be mixed together in the same package. This is not desirable, and we use a *grouping* feature to maintain the existing separation of the different classes; the arguments of a group directive are added to a subpackage. One particular application of retyping is to allow using different concrete syntaxes for a DSL. Figure 10 illustrates how reuse of concrete syntaxes (modelling tools) is achieved.

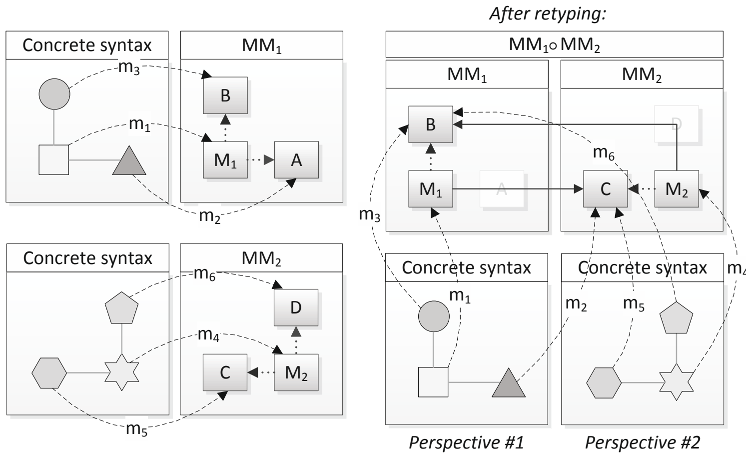

Fig. 10. Remapping of concrete syntax

Each of the two metamodels $\mathcal{MM}_1$ and $\mathcal{MM}_2$ has a unique concrete syntax. The concepts of the concrete syntaxes are mapped to the classes of the metamodels. For example, the square (m_1) is mapped to the M_1 class of $\mathcal{MM}_1$, denoted by the arrow m_1. The right part of the figure shows how the triangle (m_2) and pentagon (m_6) are remapped to the C and B classes, respectively. This is possible, since these types are subtypes of X and Y. Put differently, the triangle and pentagon concrete syntax concepts only relate to the properties declared in X and Y. Hence, it is possible to use several concrete syntaxes to describe different aspects of a composite language's problem domain. For example, $\mathcal{MM}_1$ and $\mathcal{MM}_2$ could have been two metamodel design patterns for describing typed relationships and state machines. Each of these design patterns ought to be represented with a distinct known concrete syntax. A model of the resulting metamodel will consist of two submodels expressed in the respective concrete syntaxes. Still, model objects can be shared between the submodels. For example, in the right part of Fig. 10, the circle (m_3) and the pentagon (m_6) both map to the same object of B. The triangle (m_2) and the hexagon (m_5) relate to the same C object. The objects of the B and C classes are contained by an M_1 object.

An alternative to retyping is using *Object Constraint Language (OCL)* constraints on the resulting metamodel. In the synthesiser example, it would be possible to add a constraint that restricts the type of objects contained by the soundSource attribute: the constraint would restrict objects to be instances of the Oscillator class. There are two reasons why retyping is the preferred approach. First, templates may be instantiated within other templates. In Fig. 1 we saw how this allows building template hierarchies. In particular, an attribute that has been retyped once may later be retyped again to a subtype of the current attribute type. For example, there could have been several kinds of oscillators

within the Oscillator metamodel that subtype the Oscillator class. Being able to retype the soundSource attribute to one of these special oscillators is required. Using OCL constraints makes this cumbersome as a constraint would have different value depending on where in the template hierarchy it is effective. More importantly, it would be required to support specifying OCL constraints within templates, as OCL can only be used in packages. Second, classes that are not used in a metamodel after retyping, e.g., SoundSource, are subject to be excluded from the metamodel. The tool supporting the work of this paper provides a naive exclusion feature known as *suppression*. The purpose of this feature is to remove classes from a metamodel that are no longer referred. This may in some cases result in OCL constraints that target removed classes. This is not desirable.

Retyping is achieved by changing the type of an attribute or reference. So far, we have only discussed the case where the source and target types are defined using classes. It is possible to use interfaces instead of classes. This is a special case of retyping as it is not possible to define required structure (other than operations) of a target type. For example, a transformation model may refer to attributes and references defined explicitly in a superclass, and thereby improve genericity of the transformation model.

In Fig. 6, SoundSourceDef was defined as a standalone resource. Instead, it would be possible to include this class in the Synthesiser metamodel. The Oscillator metamodel could then be composed with the Synthesiser metamodel by adding the Oscillator class as a subtype to SoundSourceDef and subsequently retyping the soundSource attribute to the Oscillator class as before. The difference following such approach is minimal, but it may make more sense from an organisational point of view.

6 Related Work

Atlas Model Weaver (AMW). The *AMW* [4] builds on the concept of using weaving models for expressing links between model elements. Weaving models can be used for several applications, e.g., model composition, where link types like *merge*, *override*, and *union* are relevant. A weaving model for a specific composition scenario is made by instantiating a core weaving metamodel. That is, a composition process is described as a model with specific links detailing the model composition (used by a merging tool). The primary subject for the AMW Model Weaver is composition of models and not metamodel composition as discussed in this paper. Creating a weaving model is only reasonable when this weaving model can be reused to compose several models that are instances of a common metamodel. Composing metamodels using weaving models is a cumbersome process. Neither is the initial cost of defining a weaving model for the composition of metamodels justified, since the weaving model is likely to be used only once.

The Epsilon Merging Language (EML). *EML* is a language for merging of models [5]. A model in the EML language comprises a set of rules that dictate

how model elements from source models are combined into a merged model. A central feature of EML is the ability to compare and match elements from the source models. This is achieved using match rules. Additional strategies can be applied to ensure that merging of models is carried out in the correct manner. EML is primarily intended for merging of terminal models. Creating an EML model with all required rules is not trivial. Creating rules for merging of metamodels that will only be used once is not a good approach. EML does not handle merging of models with conflicting elements well.

Weaving Metamodels using SmartAdapters. Weaving variability into metamodels can be seen as an asymmetric extension of a metamodel. An approach based on *SmartAdapters* is discussed in [6]. A SmartAdapter appears as a composition protocol that covers how an aspect model should be combined with a base model. The purpose of a SmartAdapter is to describe weaving of the aspect model separately from the base model definition, which supports reusing aspect models. Weaving is achieved by creating a ConcreteAdapter that specifies bindings between an aspect model and base model. SmartAdapters can be used to add new model elements to a base model, modify attributes and references and merge model elements. The approach is designed for defining software product lines and does not address metamodel composition in general. There are similarities between using SmartAdapters and the approach we discuss in this paper. Though, SmartAdapters do not support symmetric metamodel composition.

The GeKo Generic Aspect Model Weaver. *GeKo* is a weaver that supports weaving advice models into a base model [7,8]. The advice models and base model have to be instances of the same EMOF compatible metamodel. Compositions are described using pointcut models. A pointcut model consists of a set of objects that bridge elements in the base and advice models using morphisms (mappings). Elements from an advice model can either be added to or replace elements of the base model. The approach does not address adaption of behavioural semantics (as defined at the metamodel level), or how adding and replacing elements are reflected by a behavioural semantics.

Reusable Aspect Models (RAM). *RAM* is an aspect-oriented approach for integrating class, sequence, and state diagrams [9]. An aspect in RAM is a model of three constituent UML diagrams: a structural view, state view, and message view. Aspects are woven together as specified by instantiations. Instantiation directives describe how one aspect instantiates another and map incomplete entities in one aspect to entities of another aspect, regardless of view. The essence of RAM is support for model reuse, multi-view modelling, and view consistency checking. RAM resembles the work of this paper. One evident difference is that requirements for using an aspect are explicitly expressed in the aspect's definition; aspects may depend on each other as specified by instantiation parameters. In contrast, metamodel templates do not have parameters, though dependencies between templates are expressed using instantiation directives. RAM supports defining aspect dependency chains, where simpler aspects are combined to create

more complex aspects. In a similar manner, metamodel templates allow building template hierarchies. While RAM's focus is on reusable models, RAM does not explicitly address how tools can be reused on composed models. Metamodel templates take the latter into consideration.

Model Integration Using Mega Operations. An approach for weaving and sewing of metamodels and models is discussed in [21]. Weaving is based on using weaving operators, e.g., *overrides*, *references*, *prune*, and *rename*. These operators act as directives that govern a composition process. Specifically, weaving operators can be used to compose both metamodels and conformant models. There are several similarities between this work and the approach of this paper. For example, the operators resemble the template instantiation code (*with*, *adds*, etc.). However, the approach does not address integration of behavioural semantics. It is described how one of the operators, named *prune*, can be used to remove unnecessary elements from a metamodel. Such operation can only be performed safely if metamodels are considered purely as static structures; e.g., Kermeta and EMF both associate behavioural semantics to a metamodel's structure (abstract syntax). The composition tool backing the approach of this paper supports a naive operation of removing classes from metamodels - known as suppression. However, a thorough static analysis of the metamodel is required for such operation to be carried out safely. Several requirements for removal of a given class must be fulfilled. For example, objects of the class must be optional (multiplicities of the form [0..n]), the behavioural semantics of the metamodel can not contain code that instantiates the class, the class can not be a superclass with subclasses that should still be included in the resulting metamodel, the class can not participate in a bi-directional relationship that is not optional (from both sides), etc. The topic of excluding classes from metamodels is currently being studied.

Sewing is discussed as a way of integrating models loosely. The discussed advantages are autonomous models without entangled concepts, that can utilise, e.g., existing GUI. Two sewing operators are identified: *synchronizes* and *depends*. Synchronisation is used when model elements need to be synchronised, e.g., two attributes of two distinct models may be synchronised ensuring that the attributes always have the same value. Dependency indicates that existence of one model element is required for existence of another. The actual integration of models is realised using mediating entities, e.g., *Java Metadata Interface (JMI)*. In the approach discussed in this paper, we use object links for both synchronisation and dependency (using retyping). Thus, no mediators are required to integrate the models.

7 Conclusion and Future Work

Metamodels play an essential role in MDE, yet their efficient composition and reuse are hindered by limitations of many state-of-the-art model composition mechanisms. In this paper, we have discussed a template-based approach to

metamodel composition, which tackles some of the limitations of these mechanisms. We have introduced the concept of metamodel templates which promotes composition of both the structure and semantics of metamodels. A metamodel template comprises a class model whose classes can be customised for a specific usage by instantiating the template; including support for merging of classes, resolution of name conflicts, addition of semantics, overriding of operations and retyping of class properties. Specifically, retyping is not supported by any of the related approaches discussed. Hence, metamodels can be composed by utilising a set of powerful features, all of which can be fully type checked. The applicability of the approach has been demonstrated using a metamodel composition tool. Future work includes formalising how type checking is performed and elaborating the retyping and suppression concepts.

We argue that metamodel templates leverage how metamodels can be composed and address the increasing complexity and required agility in metamodel and language design.

References

1. Kent, S.: Model Driven Engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
2. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A Generic Approach for Automatic Model Composition. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 7–15. Springer, Heidelberg (2008)
3. Groher, I., Voelter, M.: XWeave – Models and Aspects in Concert. In: 10th International Workshop on Aspect-Oriented Modeling (AOM 2007), pp. 35–40. ACM Press (2007)
4. Didonet Del Fabro, M., Bézivin, J., Valduriez, P.: Weaving Models with the Eclipse AMW plugin. In: Eclipse Modeling Symposium, Eclipse Summit Europe 2006 (2006), http://ssei.pbworks.com/f/Del+Fabro.Weaving+Models+with+the+Eclipse+AMW+plugin.pdf
5. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging Models with the Epsilon Merging Language (EML). In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)
6. Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., Jézéquel, J.-M.: Weaving Variability into Domain Metamodels. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 690–705. Springer, Heidelberg (2009)
7. Morin, B., Klein, J., Barais, O.: A Generic Weaver for Supporting Product Lines. In: 13th International Workshop on Early Aspects (EA 2008), pp. 11–18. ACM Press (2008)
8. Kramer, M.E., Klein, J., Steel, J.R.H., Morin, B., Kienzle, J., Barais, O., Jézéquel, J.-M.: On the Formalisation of GeKo: a Generic Aspect Models Weaver. Technical Report, University of Luxembourg (2012) ISBN: 978-2-87971-110-2
9. Kienzle, J., Al Abed, W., Klein, J.: Aspect-Oriented Multi-View Modeling. In: 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009), pp. 87–98. ACM Press (2009)
10. Krogdahl, S., Møller-Pedersen, B., Sørensen, F.: Exploring the use of Package Templates for flexible re-use of Collections of related Classes. Journal of Object Technology 8(7) (2005), http://www.jot.fm/issues/issue_2009_11/article1/

11. Sørensen, F., Axelsen, E.W., Krogdahl, S.: Reuse and Combination with Package Templates. In: 4th Workshop on MechAnisms for SPEcialization, Generalization and InHeritance (MASPEGHI 2010), Article 3. ACM Press (2010)
12. Axelsen, E.W., Sørensen, F., Krogdahl, S., Møller-Pedersen, B.: Challenges in the Design of the Package Template Mechanism. In: Leavens, G.T., Chiba, S., Haupt, M., Ostermann, K., Wohlstadter, E. (eds.) Transactions on AOSD IX. LNCS, vol. 7271, pp. 268–305. Springer, Heidelberg (2012)
13. The Eclipse Foundation: Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf`
14. Object Management Group: Meta Object Facility (MOF) Core Specification, `http://www.omg.org/mof`
15. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
16. Tolvanen, J.-P., Kelly, S.: MetaEdit+ – Defining and Using Integrated Domain-Specific Modeling Languages. In: 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA 2003), pp. 92–93. ACM Press (2003)
17. Institute for Software Integrated Systems: GME – Generic Modeling Environment, `http://www.isis.vanderbilt.edu/projects/gme`
18. Clark, T., Evans, A., Kent, S.: Aspect-Oriented Metamodelling. The Computer Journal 46(5), 566–577 (2003)
19. Cho, H., Gray, J.: Design Patterns for Metamodels. In: SPLASH 2011 Workshops Proceedings, pp. 25–32. ACM Press (2011)
20. Axelsen, E.W., Krogdahl, S.: Package Templates: A Definition by Semantics-Preserving Source-to-Source Transformations to Efficient Java Code. In: 11th International Conference on Generative Programming and Component Engineering (GPCE 2012), pp. 50–59. ACM Press (2012)
21. Reiter, T., Kapsammer, E., Retschitzegger, W., Schwinger, W.: Model Integration through Mega Operations. In: Proceedings of the Workshop on Model-Driven Web Engineering (MDWE 2005) (2005), `http://www.lcc.uma.es/~av/mdwe2005/camera-ready/3-MDWE2005_MegaOperations_CameraReady.pdf`