Towards Non-intrusive Composition of Executable Models

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

Keywords: Model Composition, Languages, Domain-specific Modelling, Runtime.

Abstract: An essential operation in model-driven engineering is composition of models and their metamodels. There exist several mechanisms for model composition. However, most of these only consider composition of either models or metamodels and not both kinds of models simultaneously, and do not address how the composition impacts modelling artefacts like editors, transformations and semantics. Moreover, model composition mechanisms typically deal with model structure and do not consider operational semantics. In this paper, we discuss a novel approach for the composition of both models and metamodels in a virtually non-intrusive manner. We achieve this by utilising a placeholder mechanism where classes in one metamodel may represent classes of another. The ideas presented have been validated by the construction of a framework. We will illustrate how non-intrusive composition allows linking the operational semantics of different languages without rendering existing modelling artefacts inconsistent. This increases the flexibility in how languages can be combined, and reduces the amount of necessary changes of tools and other modelling utilities.

1 INTRODUCTION

Metamodel composition is an important operation in metamodelling. In the course of the last decade, several mechanisms for composition of models and metamodels have been devised, e.g. (Morin et al., 2009)(Fleurey et al., 2008)(Kolovos et al., 2006)(Groher and Voelter, 2007). However, most of the mechanisms work either on the model or on the metamodel level, and do not consider composition of both kinds of models simultaneously. Specifically, metamodel composition mechanisms do not address how existing models can be combined as their metamodels are composed. A consequence of this is that the models are rendered invalid as they are not valid instances of the composite metamodel resulting from the composition process. Other artefacts in the metamodelling ecosystem like editors (concrete syntax), semantics (including constraints), transformations and code generators are also impacted when metamodels are composed. The reason is that these artefacts are defined relatively to a metamodel. Typically, each of these artefacts needs to be refactored to comply with a composed metamodel. Ideally, artefacts in the ecosystem should be aligned with new metamodel variants automatically by utilising information from the composition process. Similar considerations are discussed in (Di Ruscio et al., 2012)(Garca et al., 2013)(Demuth et al., 2013).

A natural evolution of composition mechanisms is the ability to consider model composition both at the metamodel and model levels. Put differently, a composition mechanism should govern the composition of metamodels *and* their models. This requires the composition mechanism to work explicitly on two abstraction levels where composition-specific directives, as required to compose (meta)models, are propagated and utilised for composition of models.

Meta Object Facility (MOF) (OMG, 2014) is the most prominent architecture for classification of models according to abstraction levels. In this architecture, metamodels reside on the M_2 level, whereas the models reside on the M_1 level. A model contains objects which are instances of classes in its metamodel. Thus, a model is classified by its metamodel. In the literature, this is typically referred to as the *conformsTo* relationship. A mechanism that supports composition of metamodels and the conforming models would work on both the M_2 and M_1 levels, respectively.

In this paper, we discuss how metamodels and models may be composed virtually *non-intrusively*. What this means is that metamodels and models are kept separate, yet they are composed through a set of implicitly defined mappings which allow specifying proxy classes. A proxy class in one metamodel represents a class in another metamodel. The approach only requires a minimal refactoring of existing arte-

Berg H. and Møller-Pedersen B..
Towards Non-intrusive Composition of Executable Models.
DOI: 10.5220/0005242401110121
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 111-121
ISBN: 978-989-758-083-3

facts which may be perfomed automatically or semi-automatically. The essence of our approach is the ability to link the operational semantics of languages, so that models/programs of different languages can interact at runtime. The ideas presented are applicable to metamodelling environments that allow expressing the operational semantics of a metamodel/language using an object-oriented action language, e.g. the *Eclipse Modeling Framework (EMF)* (EMF, 2014) and *Kermeta* (Muller et al., 2005). The approach has been validated by the construction of a prototypical framework on top of EMF. We will use this framework and show how the behaviour of classes/objects in a *General-Purpose Language (GPL)* can be defined by models expressed in a domain-specific *State Machine Language (SML)*.

The paper is organised as follows. In Section 2 we define our non-intrusive composition mechanism. We then introduce an example in Section 3 and use it to illustrate the ideas and the mechanics of the framework in Section 4. An evaluation of the composition mechanism follows in Section 5, and in Section 6 we discuss our approach in relation to existing work in the field. Finally, in Section 7 we summarise and conclude the paper.

2 DEFINITIONS

Traditional (meta)model compositon mechanisms are *explicit* in the sense that structural elements from different constituent models are interwoven, e.g. two classes from different metamodels may be merged or a class may be made a subtype of another class. As motivated, this has consequences with respect to other entities in the metamodelling ecosystem. The mechanism presented in this paper works by establishing mappings between metamodels and models in a practically non-intrusive manner. This means that the composition is lifted away from the modelling space and established using a separate specification. There are two types of mappings: M_2-mappings and M_1-mappings. The names reflect the MOF level on which the mappings occur. That is, M_2-mappings are created between metamodel structures, whereas M_1-mappings are created between model structures (objects of the classes in the metamodels). Both types of mappings can be expressed by non-injective partial functions. M_2-mappings are described in a *Unification Model (UM)*, while M_1-mappings are described in a *Linking Model (LM)*.

Definition 1. *An M_2-mapping is a uni-directional or bi-directional binding between two structural elements, τ_x and π_y, in two different metamodels. A structural element is a package, class, attribute, operation or parameter. A bi-directional binding may be decomposed into two uni-directional bindings. For uni-directional bindings, τ_x is the source element and π_y is the target element. For bi-directional bindings, τ_x and π_y are both source and target elements corresponding to decomposition of the bi-directional binding into two uni-directional bindings.*

$$\begin{array}{ll} : & \langle \tau_1 \mapsto \pi_1 \rangle (uni-directional) \\ : & \langle \tau_1 \leftrightarrow \pi_1 \rangle \ (bi-directional) \end{array}$$

where τ_x and π_x are on either of the forms (N being a name):

$$\begin{array}{ll} : & \langle N_{package}, N_{class} \rangle \\ : & \langle N_{package}, N_{class}, N_{attribute} \rangle \\ : & \langle N_{package}, N_{class}, N_{operation} \rangle \\ : & \langle N_{package}, N_{class}, N_{operation}, N_{parameter} \rangle \end{array}$$

Definition 2. *A unification point is a collection of M_2-mappings between two classes in two different metamodels. A unification point is either asymmetric or symmetric. The source class of an asymmetric unification point is referred to as a proxy as it represents a placeholder for the target class. The two classes of a symmetric unification point represent compatible types. A unification point is only valid if both classes share a common equivalent structure. A class may be part of an arbitrary number of unification points. A unification point χ can be modelled as a set of mappings:*

$$\begin{array}{lll} \chi_a & : & \langle \tau_1 \mapsto \pi_1, \tau_2 \mapsto \pi_2, ..., \tau_n \mapsto \pi_n \rangle \ (asymmetric) \\ \chi_s & : & \langle \tau_1 \leftrightarrow \pi_1, \tau_2 \leftrightarrow \pi_2, ..., \tau_n \leftrightarrow \pi_n \rangle \ (symmetric) \end{array}$$

Definition 3. *The (partial) equivalent structure of a unification point is a set of attributes, operations and operation parameters that both related classes of the unification point need to have. Two classes C_1 (proxy) and C_2 (classes are considered as sets of attributes, references and operations) of two metamodels may be unified if:*

$$\forall s_1 \in C_1 \cdot \exists s_2 \in C_2 \ \rightarrow \ s_1 \sim s_2$$

where $\sim : \mathcal{S} \times \mathcal{S} \rightarrow Bool$ is a recursive partial function that is true if its two arguments have an identical structure. For symmetric unification points, C_1 and C_2 have to be equipotent, that is: $\mid C_1 \mid = \mid C_2 \mid$. $\mathcal{S}$ is a set of structural elements: $\mathcal{S} = \{Class, Attribute, Reference, Operation, Parameter\}$.

Definition 4. *A Unification Model (UM) unifies an arbitrary number of metamodels. It consists of one or more unification points. Two given metamodels may be unified with one or more unification points. The Unification Model comprises a set of unification points:*

$$UM : \langle \chi_1, \chi_2, ..., \chi_n \rangle$$

Definition 5. *An M_1-mapping φ is a bi-directional binding between two model elements (objects), with identifiers i and j, in two models x and y. An identifier encompasses both the class name for which the object is an instance and a unique integer designator.*

$$\varphi \quad : O_i^x \leftrightarrow O_j^y,\ i,j \in I$$

where I is a set of tuples on the form: $\langle ClassName, NaturalNumber \rangle$. Inheritance of attributes and operations (including overriding) is supported (e.g. binding to an operation specified in a superclass to ClassName).

Definition 6. *The Linking Model comprises a set of M_1-mappings between an arbitrary number of models:*

$$LM \quad : \quad \langle \varphi_1, \varphi_2, \ldots, \varphi_n \rangle$$

Figure 1 illustrates non-intrusive composition. The figure has three metamodels and models of these. The C_4 (proxy) class in MM_1 is unified asymmetrically with the D_1 class in MM_2, whereas the (proxy) class D_3 in MM_2 is unified asymmetrically with the class E_1 in MM_3. The classes C_3 and D_4 are unified symmetrically. That is, C_3 and D_4 represent a structure-wise identical type[1]. A key feature of the mechanism is the ability to specify the exact models (and objects) that should be linked, e.g. the M_1 model of MM_2 is linked with the M_2 model of MM_3.

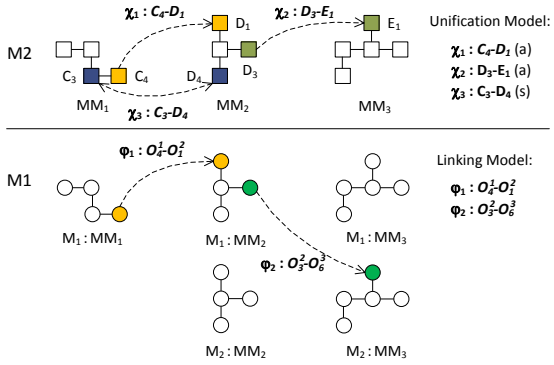

Figure 1: Conceptual overview of non-intrusive composition (class names are omitted for object identifiers).

The χ_1 and χ_3 unification points are illustrated in Figure 2.

[1]More precisely, D_4 has the required structure of C_3.

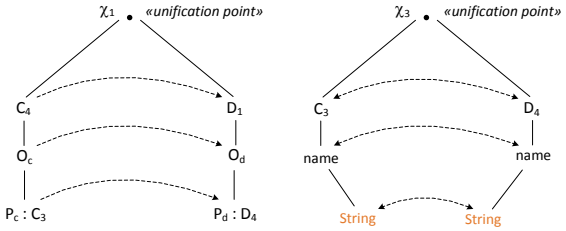

Figure 2: The χ_1 (asymmetric) and χ_3 (symmetric) unification points.

For the sake of the illustration, we assume that the C_4 class contains an operation O_c with a parameter P_c. The operation and its parameter are unified with equivalent structure in D_1. M_2-mappings bind instance names of metamodel structure, e.g. the parameter named P_c is bound to the parameter named P_d. However, for structural equivalence, attribute and parameter names are irrelevant. What is relevant is the types of attributes or parameters (and their sequence). Hence, the types of P_c and P_d need to be structural equivalent for the two classes to form a valid unification point.

By looking at the χ_3 unification point in the Unification Model it is clear that the types C_3 and D_4 are structural equivalent, as both only contain a String attribute, as can be seen by the rightmost unification point. (The two classes represent an identical concept as seen from an ontological perspective.) At runtime this implies that an object of C_3 may be converted to an object of D_4 and used in a type-safe invocation of the O_d operation. By construction, all unification points can be represented as tree structures as seen in the figure.

The idea of non-intrusive composition is based on the principle of partial representation. Structural commonalities allow for a proxy class to represent another class, as long as both classes have a set of common structure (as dictated by the proxy class). Hence, the proxy class mimics the structure and meaning of another class.

3 EXAMPLE

We will illustrate the approach by exemplifying how the behaviour of classes/objects in a GPL may be defined using a state machine. Specifically, we will see how the state of an object can be maintained by a state machine and how methods in the class of the object may be invoked as a consequence of a state change in the state machine model. To the best of our knowledge, no composition mechanism available can handle simultaneous composition at both the metamodel

and model levels in the situation where the behaviour of classes/objects are defined by a state machine.

Figure 3 gives an overview of the metamodel of the GPL. The metamodel allows creating very simple programs. We have kept the number of statement types to a bare minimum, yet they suffice to model interesting enough programs for the purpose of illustrating our framework. We will not consider other language artefacts like concrete syntax in detail and merely use such to visualise models.

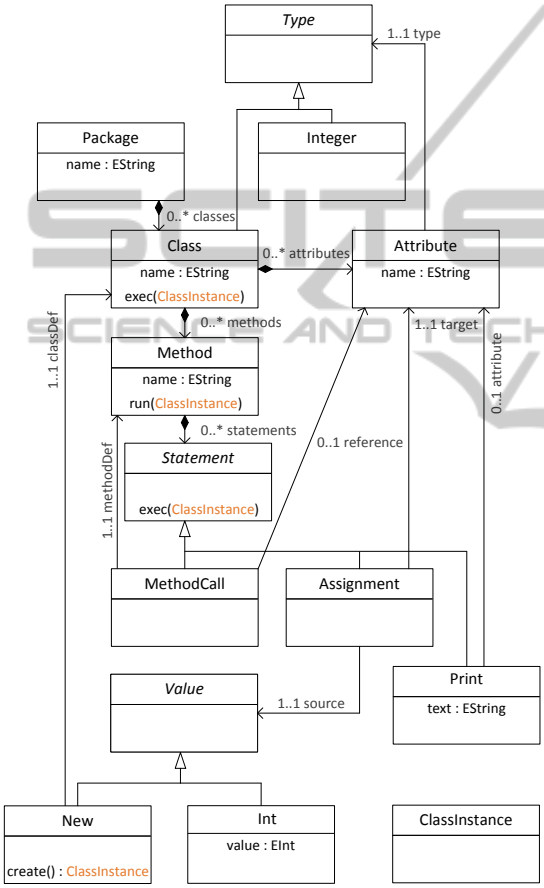

Figure 3: Metamodel of a simplified General-Purpose Language (GPL).

Briefly explained, a program consists of a package with an arbitrary number of classes. The classes may have attributes and methods. An attribute is either of the primitive type Integer or class-typed. Methods have no return type or parameters. They may be defined using a combination of assignments, print statements and method calls. A method may either invoke other methods defined in the same class or methods in any other class defined within the same package by using references (class-typed attributes). ClassInstance represents a (runtime) object of a class. Notice how Method and Statement both have operations with a ClassInstance parameter. This allows invoking methods and executing statements for one particular object (class instance). A ClassInstance is constructed by invoking the operation create() in the New class.

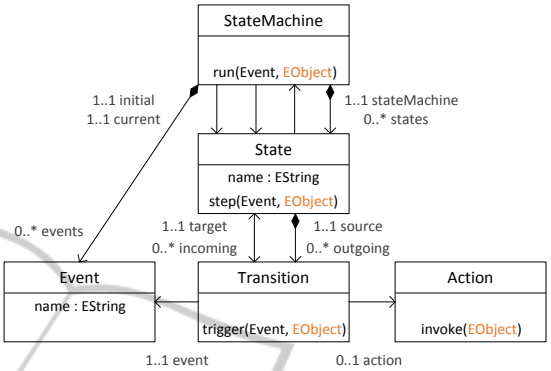

Figure 4: Metamodel of a State Machine Language (SML).

The metamodel for the SML is given in Figure 4. The StateMachine class has an operation named run that takes two arguments. The first parameter is of type Event, the other is of the more generic type EObject. The latter allows sending an object of an EObject subtype as an optional argument when a new event is raised. The Transition class has an operation named trigger which causes a state change if the event received matches the event associated with any of the outgoing transitions. The metamodel includes a class Action with an operation named invoke. The invoke() operation has to be overridden in a subtype of Action in order to provide a custom action semantics (when using the language by itself).

Both metamodels are defined in EMF. The operational semantics of the classes has been added by building on the code generated using the built-in code generator of the EMF framework. Thus, the models/programs of the two languages may be run standalone. In order to model the behaviour of objects we need to extend the GPL metamodel with concepts for sending signals/events. In particular, we want to add a new kind of statement that allows sending signals from within methods. Figure 5 shows how this can be achieved.

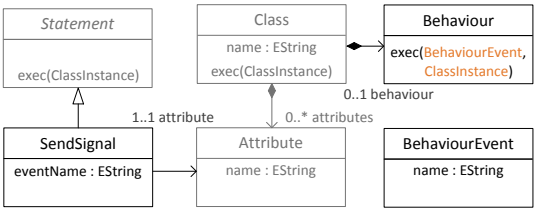

Figure 5: Additional classes for modelling of signals/events.

Behaviour is intended to be a proxy class for the StateMachine class in the SML metamodel, whereas BehaviourEvent is intended to represent the Event class. Notice that using send signal statements and creating Behaviour objects are optional, which means that existing models and tools are not impacted significantly, e.g. existing models of the GPL metamodel are still valid. That is, classes in the extended GPL language may have behaviour externally defined if required. The only required update to e.g. a model editor is to support creating instances of a proxy class, if such is added to a metamodel to facilitate composition (existing classes in a metamodel can also be used as proxy classes).

```
package Example {
  class C1 {
    attribute c2 : C2

    main() {
      c2 = new C2();
      c2.print();
      sendSignal( "On", c2 );
      c2.print();
      sendSignal( "Off", c2 );
      c2.print();
      sendSignal( "On", c2 );
      c2.print();
    }
  }

  class C2 [behaviour = true] {
    attribute a : Int

    setA0() { a = 0; }
    setA1() { a = 1; }
    print() { _print( "Value:" + a ); }
  }
}
```

Figure 6: An example GPL program.

The next step is to compose the metamodels and models in the two languages. The behaviour of the objects of a class may be modelled by a state machine. An object's state thus depends on the current state of the state machine and what signals (events) that have been received. The source of such signals is irrelevant in this case, but we assume there is some kind of sensor. An example program in the extended GPL is given in Figure 6. A send signal statement takes two arguments; an event name and the object on which to send the signal. The signalling concept could have been defined to wait for an external event from e.g. a sensor. A behaviour is associated to the C2 class by setting a flag named behaviour (resulting in the creation of a Behaviour object). To keep things manageable, what we set out to do can be summarised by these main points:

- Program execution starts by invocation of main
- An "On" signal/event is sent to an object of the C2 class
- The event is forwarded to the specified SML model (utilising the framework)
- The current state of the state machine changes from Idle to On
- The state change causes invocation of setA1() in C2 (utilising the framework)
- The value of a is printed to screen

The simple scenario above suffices to underline the mechanics of our framework.

4 THE FRAMEWORK

We have implemented a prototypical framework that realises the concepts of this paper. The framework builds on top of EMF. This means that it works with all EMF-compatible metamodels and their operational semantics (model code). Non-intrusive composition using the framework corresponds to the five phases illustrated in Figure 7. We will discuss each phase in detail using the example.

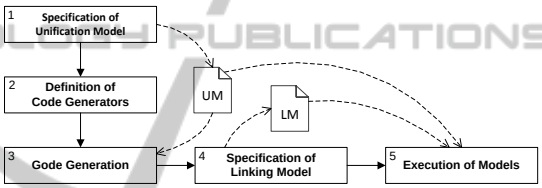

Figure 7: The five phases of non-intrusive model composition.

4.1 Specification of Unification Model

In order to specify the behaviour for the Class concept using a state machine model we have to create a set of unification points that relate the classes of the two metamodels. The Behaviour class in the GPL metamodel is going to represent the StateMachine class in the SML metamodel, whereas Action will be representing Method. Furthermore, the BehaviourEvent class of the GPL metamodel will be unified with the Event class of the SML metamodel. Four unification points are required to achieve this.

Behaviour is intended to be a proxy class for StateMachine. Hence, all the structure of Behaviour needs to be matched by equivalent structure in StateMachine, since code generated for the Class concept will implicitly refer to such structure. Behaviour contains an operation with two parameters; one of type BehaviourEvent and one of type ClassInstance. StateMachine contains an operation as well with parameters of type Event and EObject. The only way that Behaviour and StateMachine can be unified is if the BehaviourEvent class can be unified with the Event class and the ClassInstance class can be unified with the

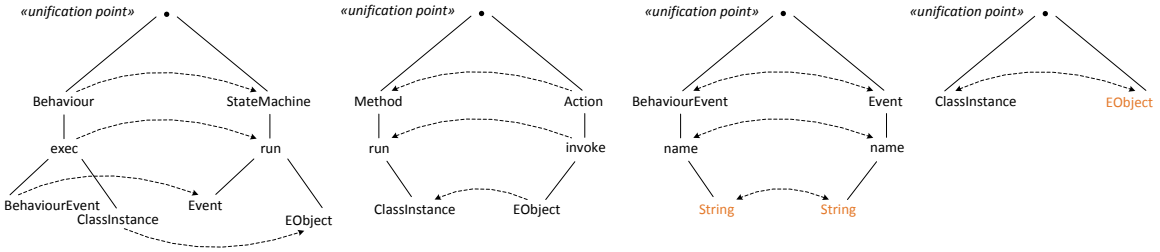

Figure 8: The four unification points between classes in the GPL and SML metamodels.

EMF built-in EObject interface. BehaviourEvent consists of an attribute with a String type. Event has a String attribute as well. Hence, BehaviourEvent and Event can be unified by a symmetric unification point since BehaviourEvent and Event are structurally equivalent. Moreover, ClassInstance can be unified with EObject. ClassInstance is not a subtype of EObject. However, the Java counterpart of ClassInstance, ClassInstanceImpl, will be generated as a subtype of the EObjectImpl Java class by the built-in EMF code generator. EObjectImpl implements the EObject interface. As a result, Behaviour and StateMachine may indeed be unified by an asymmetric unification point. The purpose of the EObject parameter in the run() operation is to allow sending an optional argument. In the example, a ClassInstance object of the C2 class, whose behaviour is modelled by the state machine model, is passed as an argument. It is later used to know what object on which to invoke the setA1() operation. Action contains an operation named invoke with an EObject parameter, whereas Method contains an operation run() with a parameter of type ClassInstance. We have already seen that ClassInstance and EObject are type-compatible, thus, Action and Method can be unified. Notice that additional structure in StateMachine with respect to Behaviour, and additional structure in Method with respect to Action, would have been ignored as it is not required for the partial representation whose requirements are specified by the proxy classes (Behaviour and Action). Figure 8 gives the complete Unification Model. The nodes representing parameters are replaced with types instead of parameter names to improve the readability of the figure.

4.2 Definition of Code Generators

The framework supports the automatic generation of reflective code for linking the operational semantics of two metamodels non-intrusively. By also using subtyping and overriding of methods we ensure that the existing operational semantics does not need to be changed.

In the example, the added Behaviour class in the GPL metamodel is intended to represent the StateMachine class in the SML metamodel. This means that the operational semantics of the Class concept of the GPL metamodel needs to be revised to reflect this. This is achieved by creating a code generator which outputs a subtype of ClassImpl. ClassImpl is the Java class generated by EMF which defines the Class concept's operational semantics. The subtype will contain the necessary code for interacting with the operational semantics of the StateMachine class. The code will be defined in an operation named signal. The signal() operation has a parameter typed with BehaviourEvent and one typed with ClassInstance. It will be invoked from the operational semantics of the SendSignal class, i.e. an overridden version of the exec() operation as defined in Statement. See Figure 9.

```
public class ClassImplCustom extends ClassImpl
{
  public void signal( BehaviourEvent be,
    ClassInstance ci )
  {
    ...
  }
}
```

Figure 9: The subtype of ClassImpl.

Similarly, Action is intended to represent Method. Thus, a code generator that generates a subtype of the TransitionImpl class needs to be constructed. The code generators may be simple Java classes or utilise emitter templates. Code generators are used as tools for simplifying specification of glue code. It is possible to define the subtypes manually and utilise the framework directly, though this requires writing a lot more code.

4.3 Code Generation

The code generators are run to generate code required for linking the operational semantics of two or more metamodels. The new code is added to the existing code defining a metamodel's operational semantics by using inheritance. A build script also ensures that the EMF factories are updated to create objects of the generated subtypes. The subtypes are still compatible with EMF. The generated subtypes contain manually

defined domain-specific code and automatically produced reflective code. There are two types of reflective code that may be generated: code for accessing attributes and/or code for invoking operations. The reflective code may be tailored by using a set of simple options/flags, e.g. it may be required to clone a set of objects at runtime or send objects as operation arguments that will later be returned in a callback fashion. The framework will take care of forwarding invocations between two metamodels' operational semantics and convert parameter types at runtime (corresponding to symmetric unification points). That is, it eliminates the need for a common type (used in the definitions of both metamodels) when sending non-primitive values.

4.4 Specification of Linking Model

The Linking Model allows pin-pointing what (proxy) object of a given model that should be linked to a (target) object in another model. The Linking Model is built by referring to objects of different models and building pairs of two objects. In principle, there is no restriction to what objects that may be linked. However, only pairs that reflect unification points in the Unification Model are valid.

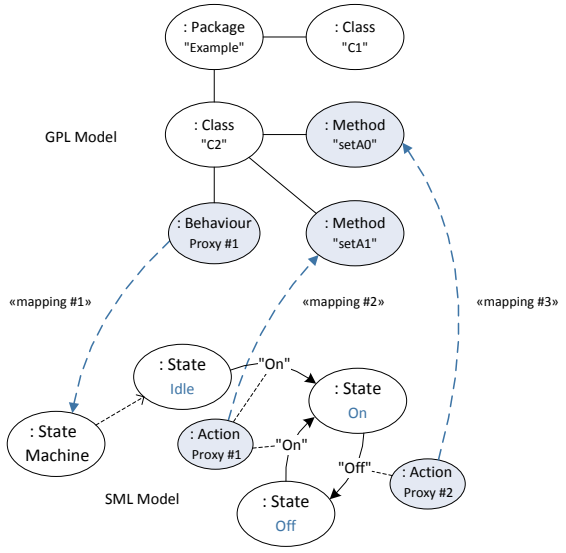

Figure 10: Linking of the objects in the GPL and SML models (graphical representation of the Linking Model).

Figure 10 gives a state machine model whose definition captures the intended class/object behaviour, and the objects of the GPL program previously introduced. Notice the event names that trigger the transitions, e.g. "On" for the transition between the Idle and On states. What we want to achieve is that each signal/event from the GPL program is forwarded to the state machine model. Hence, we need to create an M_1-mapping between the Behaviour object and the object of the StateMachine class. An important point is that each instance of the C2 class needs a unique set of runtime objects representing the state machine model, since their behaviour is independent. This is achieved by setting a flag when utilising the generateOperationCall() method of the framework[2]. However, it is still possible to specify the Linking Model with only one state machine model (at M_1). The state machine model contains two Action objects. These objects need to be linked to the Method objects representing the setA0() and setA1() methods. By creating these mappings, we complete the definition of the Linking Model.

4.5 Execution of Models

By linking model objects we are able to ensure that the corresponding runtime objects (Java objects) are linked as well. This is essential for the operational semantics of different metamodels to work in concert. The framework forwards operation invocations and deals with conversion of runtime objects used as arguments. It utilises the Unification Model to find the correct pairing of classes and creates objects using reflective code. Figure 11 details how the operational semantics of the two languages work together. It also summarises how non-intrusive composition works. To avoid confusion we use the term *object* for an M_1-object, e.g. the object C1 resulting from instantiating the Class concept in the GPL metamodel. For a runtime object of C1 we use the term *instance*. Furthermore, invocations of operations (language semantics) are in a regular font, whereas invocations of methods (in GPL programs) are in an italic font.

The execution starts by invoking the exec() operation on the Class object that represents C1 in the GPL model (1). A instance of C1 (represented by a ClassInstance object) is passed as argument to the operation (manually chosen). The semantics of the exec() operation invokes the main() method as defined in the C1 class (2). The print() method of the C2 class is then invoked after instantiating this class (3). The next statement of the main() method is a send signal statement (4). The semantics of this statement creates a BehaviourEvent instance and invokes the signal() operation on the C2 object (5). The BehaviourEvent and C2 instances are passed as arguments to the operation. signal() invokes the exec() operation on the associated Behaviour object (6). More precisely, the invocation

[2]The method generates reflective code for invocation of an operation.

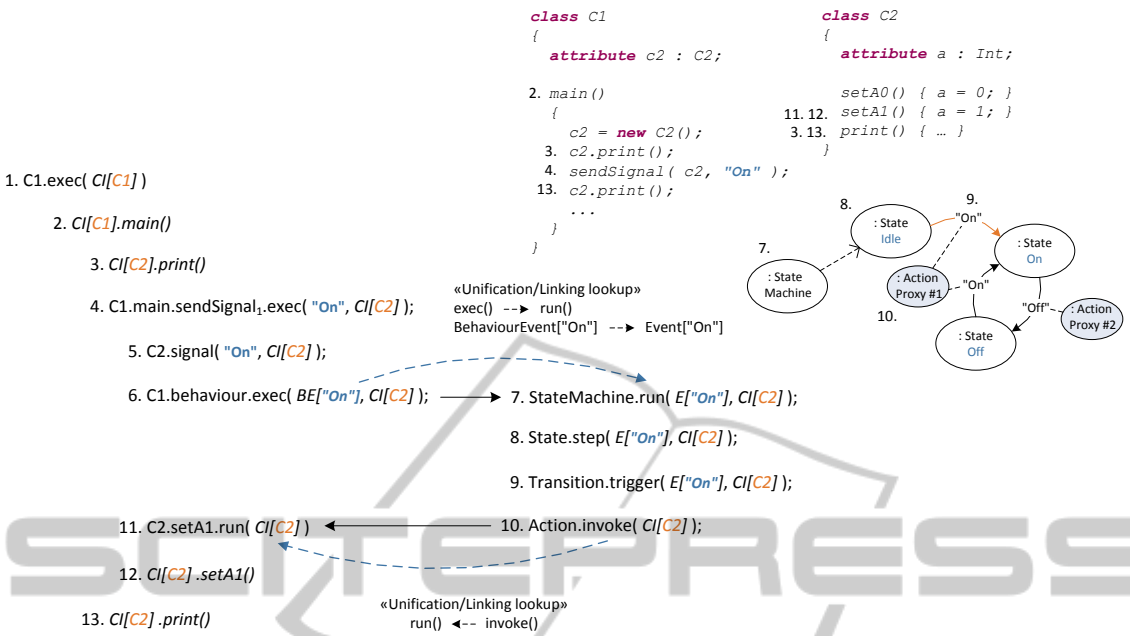

Figure 11: Illustration of how a object's behaviour is executed at runtime. Acronyms used: ClassInstance (CI), BehaviourEvent (BE), Event (E).

of exec() does not result in the actual operation in the Behaviour class to be invoked. Instead the operation invocation is sent to the framework where it is resolved and forwarded as an invocation of the run() operation on the StateMachine object in the SML model (7). This includes converting the BehaviourEvent instance passed as argument to an Event object. Notice how the ClassInstance object, representing the instance of C2, is sent as the second argument. The Event object causes the transition between the Idle and On state to be triggered (9). This results in the invoke() operation on the Action object to be invoked (10), or more precisely, the invocation of the operation is resolved by the framework, which forwards the invocation to the run() operation on the designated Method object (11). The argument to run() is the ClassInstance object that was initally passed via the call to the run() operation on the StateMachine object. The setA1() method is then invoked on the ClassInstance object, i.e. the instance of the C2 class. Finally, the print() method is invoked and the value 1 is printed to screen. Notice how the Unification Model and Linking Model are queried for information at runtime.

5 EVALUATION

The key problem arising when composing metamodels is how existing tools and models are rendered invalid. The approach of this paper addresses this. Specifically, we wanted to show that it is possible to combine two languages' operational semantics by defining mappings between the languages' concepts and objects in the models of these languages in a separate manner. By using reflective code we are able to forward operation invocations and convert parameter types. The key advantage with our approach is that the metamodels' structures are not woven together. Hence, there are minimal impacts on modelling artefacts that are defined relative to the metamodels. Even though the approach is flexible, it may be required to design a metamodel in a way that allows it to be composed with another metamodel. As an example, it may be required to add a proxy class to an existing metamodel in order to create the structural bridge between metamodels. That said, an object of such a proxy class would be optional and therefore not impact existing tools or models significantly. The proxy class is only instantiated when the metamodel is intended to be composed with another metamodel. An interesting observation is that the object of a class changes role depending on whether the class is part of a unification point or not. Consider the Method class of the GPL metamodel in Figure 3. Let us assume that we have a metamodel for modelling of methods - an expression language. The Method class may thus represent a proxy for the top node class of such an expression language. In such case, an object of the

Method class does now have a different purpose. It is merely used to establish an M_1-mapping in the Linking Model. This means that its contained statement objects would have been ignored. A class may have several methods. Each method object would thus be linked to a distinct expression language model.

Evolution of metamodels requires evolution of the Unification Model and Linking Model. However, the complexity concerned with changing these models are significantly lower than addressing co-evolution of models and other modelling artefacts.

The mechanism described in this paper focuses on how the operational semantics of different languages can work together. The M_1- and M_2-mappings may also be used by tools, e.g. by editors, to present integrated views of different languages.

Reflective code is known to be slower than non-reflective code. The mechanism discussed in this paper uses reflective code to decouple different metamodels' operational semantics. However, the reflective code is only active when the operational semantics of the metamodels interact. For the example in this paper, this adds up to two operation invocations (with instantiation of arguments) per new signal/event received. That is, most of the operational semantics being executed is based entirely on non-reflective code.

In the example, we saw how the Behaviour class is a placeholder for the StateMachine class. Consequently, it appears that an object of Class contains a StateMachine object via the behaviour containment reference. This is an important point, because the containment reference dictates the type of composition between the GPL and SML metamodels. That is, the association represents a composition relation.

We have not discussed tool support for creating the Unification Model and Linking Model. In an industrial context, a reflective graphical editor (created using e.g. GMF) may greatly simplify the process of specifying these models by allowing to draw the M_2- and M_1-mappings directly between metamodel and model structure. The graphical models can then be used to generate the Unification Model and Linking Model automatically. A graphical editor may also provide a simple way of specifying the manual code required to realise unification of two classes' operational semantics. An editor may also implement functionality for addressing co-evolution of the models.

6 RELATED WORK

We have not been able to find much related work concerned directly with the implicit nature of our composition approach. That is, the literature mainly covers explicit model composition and adaptation strategies using migration techniques and transformations. Most of the available related work addresses composition of structure and does not directly consider composition of operational semantics.

Methods for automatic co-evolution of metamodels and models are necessary to further model-driven engineering. The work of (Herrmannsdoerfer et al., 2009) describes how models may be migrated as a consequence of metamodel adaptations. This is achieved using coupled transactions. A coupled transaction is constructed using a set of primitives. There are two types of primitives: primitives for querying a metamodel or model, and primitives for modifying such artefacts. A coupled transaction preserves both metamodel consistency and metamodel-model conformance. The work has been validated by implementing a language on top of EMF. With respect to the work of our paper, the method for coupled co-evolution works on metamodels and models by changing these explicitly. Hence, other modelling artefacts are impacted by the changes, i.e. co-evolution of tools are not addressed.

Another similar approach for metamodel-model co-evolution is discussed in (Wachsmuth, 2007). The work is based on the application of transformations both on the metamodel and model level. Co-evolution of metamodels is described using a set of relations between metamodels, in which are used to ensure semantics and instance preservation for the transformations.

The authors of (Cicchetti et al., 2008a)(Cicchetti et al., 2008b) discuss how model migration steps can be generated directly from a difference model that encorporates information on the evolutionary changes of a metamodel. The difference model is used as basis for a higher-order transformation in which produces a transformation that is capable of re-establishing conformance between models and their metamodel. The work addresses concerns related to parallel dependent metamodel manipulations which may cause conflicts as they work on the same metamodel elements. These can be resolved in an iterative process, which yields a set of parallel independent modifications.

The work of (Herrmannsdoerfer et al., 2011) presents a catalogue of (coupled) operators for achieving automated migration of models as a consequence of evolving metamodels. The operators are classified according to several dimensions: language preservation, model preservation and bi-directionality. The work discussed in our paper does not utilise coupled operators. However, the M_1-mappings carefully reflect the M_2-mappings. Specifi-

cally, only objects whose classes are related using M_2-mappings may safely be related using M_1-mappings.

Another approach utilising higher-order transformations is discussed in (Hoisl et al., 2014). The approach is based on defining bi-directional transformations between modelling artefacts, and uses higher-order transformations on the specifications of the bi-directional transformations. This ensures that also the transformations between the modelling artefacts co-evolve correctly.

An approach for defining reusable metamodel behaviour is discussed in (de Lara and Guerra, 2011). The approach is based on generic concepts which allow adding the same behaviour to unrelated metamodels. This is achieved by using pattern matching according to the parameters and requirements of the concept. A similar approach, in the form of model types, is discussed in (Steel and Jzquel, 2007).

7 CONCLUSION AND FUTURE WORK

Composition mechanisms that work on both the metamodel and model level are important to ensure consistency in the metamodelling ecosystem. In this paper, we have illustrated how metamodels and models can be composed in a practically non-intrusive manner in order for their operational semantics to be linked together. Non-intrusive composition is achieved by utilising a set of mappings, both at the metamodel level and at the model level. By building on the principle of partial representation we are able to specify proxy classes. A proxy class is a placeholder for another class. Its attributes and operations represent structural requirements that need to be supported by the class for which the proxy class is a placeholder. Non-intrusive composition allows for metamodels and models to be composed without rendering models, editors and other modelling artefacts invalid.

An interesting next step is to see whether the mappings may be realised in a different form and incorporated more closely into a language's definition, and to study whether non-intrusive composition brings value also for non-executable models. Future work also includes solidification of the framework to industry standard, with the inclusion of a graphical editor.

REFERENCES

Cicchetti, A., D. Ruscio, D., Eramo, R., and Pierantonio, A. (2008a). Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference (2008)*.

Cicchetti, A., D. Ruscio, D., Eramo, R., and Pierantonio, A. (2008b). Meta-model differences for supporting model co-evolution. In *Proceedings of the 2nd Workshop on Model-Driven Software Evolution.*

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

Demuth, A., Lopez-Herrejon, R., and Egyed, A. (2013). Supporting the co-evolution of metamodels and constraints through incremental constraint management. In *Model Driven Engineering Languages and Systems, LNCS vol. 8107, pp.287-303. Springer (2013)*.

Di Ruscio, D., Iovino, L., and Pierantonio, A. (2012). Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems. In *Graph Transformations, LNCS vol. 7562, pp.20-37. Springer (2012)*.

EMF (2014). Eclipse modeling framework (emf).

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)*.

Garca, J., Diaz, O., and Azanza, M. (2013). Model transformation co-evolution: A semi-automatic approach. In *Software Language Engineering, LNCS vol. 7745, pp.144-163. Springer (2013)*.

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)*.

Herrmannsdoerfer, M., Benz, S., and Juergens, E. (2009). Cope: Coupled evolution of metamodels and models for the eclipse modeling framework. In *ECOOP 2009 - Object-Oriented Programming, LNCS vol. 5653, pp.52-76. Springer (2009)*.

Herrmannsdoerfer, M., D. Vermolen, S., and Wachsmuth, G. (2011). An extensive catalog of operators for the coupled evolution of metamodels and models. In *Software Language Engineering, LNCS vol. 6563, pp.163-182. Springer (2011)*.

Hoisl, B., Hu, Z., and Hidaka, S. (2014). Towards co-evolution in model-driven development via bidirectional higher-order transformation. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development. Springer (2014) [to appear]*.

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

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 (2014). 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)*.

Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *ECOOP 2007 - Object-Oriented Programming, LNCS vol. 4609, pp.600-624. Springer (2007)*.