Integration of Operational Language Semantics using Exported Namespaces

Henning Berg
Research report 462, February 9, 2017

ISBN  978-82-7368-427-1
ISSN  0806-3036
Abstract

In model-driven engineering approaches, models are considered first-class entities. The composition of models is a necessity for modelling software systems. Structural composition of models has been studied thoroughly during the course of the last decade. However, composition or integration of executable models’ operational semantics is still an open problem. In this report we propose a mechanism for integrating the operational semantics of programming languages. The mechanism is highly flexible and provides the developers with a powerful tool for adapting languages for a specific software development project. Importantly, integration of operational semantics can be achieved without breaking the conformance between existing models and their metamodel whose operational semantics is integrated with the operational semantics of another metamodel.

1 Introduction

Model-Driven Engineering (MDE) [1] approaches are becoming increasingly popular in the software industry. One such approach is Language-Driven Development (LDD) [2]. LDD focuses on using multiple languages to model a software system. These languages are typically domain-specific and capable of modelling a certain aspect or part of the system. An essential operation in LDD is the integration or composition of the languages. This is required to e.g. produce holistic code for the system, verification of consistency and the combination of viewpoints. Composition of languages has been thoroughly studied during the course of the last ten years. In particular, several mechanisms for the structural composition of models have been invented, e.g. [3][4].

A language is either purely structural or it may support execution of its models. In some situations, languages with executable models are referred to as programming languages. There exist several approaches for providing behaviour for a language. Common approaches are translating or mapping non-executable models to alternative representations that may be executed, e.g. translating non-executable models into models of a programming language (with pre-defined behaviour) using model-to-model transformations. This allows using heterogeneous modelling languages to model the different parts of a system, and then mapping these models to one or more homogeneous model(s)/program(s) for (simultaneous) execution [5]. A related approach is to generate low-level (byte) code from the non-executable models. Interpreting the models directly is also an option.

A Domain-Specific Language (DSL) is a language whose solution space is within one specific domain. A common approach for defining DSLs is to use a development environment and/or a metalanguage specifically designed for this purpose. Examples of such metalanguages and environments are the Eclipse Modeling Framework (EMF) [6], Kermeta [7] and the Epsilon Object
These environments and metalanguages can be used to define both the abstract syntax and the operational semantics of DSLs. The operational semantics is specified using an action language. A question arising when using several (executable) DSLs in unison to model a system is how the operational semantics of the languages can work in concert. Specifically, being able to rapidly combine the operational semantics of DSLs for a given development project is required. This may require writing glue code/adapting the operational semantics of the concepts of the different languages that are unified. One way the operational semantics can be integrated is for each language to provide an API (e.g. as a foreign function interface) of operations which other languages’ operational semantics may invoke at runtime. However, this constrains how the operational semantics of a given language may interact with other languages’ semantics since integration is limited to invoking pre-defined operations. That is, it is not possible to redefine the semantics of the abstractions provided by a language. Also, defining an API for each language clutters the language specifications with an excessive number of API operations. It may also be difficult to know what operations to provide in such an API. In the general case, a critical point when integrating DSLs’ operational semantics is that each DSL’s proprietary tools and editors are at the risk of being rendered inconsistent. This may happen as language specifications are altered, e.g. by composing abstract syntaxes. Furthermore, models/programs made using a DSL may suddenly be invalid as well since they do no longer conform [9] to the metamodel defining the abstract syntax of the DSL due to structural changes. Being able to integrate languages’ operational semantics without rendering tools and models inconsistent is a strong motivational point behind our work. Similar concerns are thoroughly discussed in [10].

In this report we discuss a novel approach for integrating the operational semantics of programming languages which do not render existing tools, editors or models invalid. We achieve this by using a type-safe integration mechanism which allows exporting and importing namespaces. A namespace contains a subset of the attributes, references and operations found in a language’s metamodel. We refer to these entities as exported structure. The exported structure acts as an integration point towards a language’s operational semantics. An exported namespace can be imported by another language which gains access to the exported structure. The mechanism allows integrating languages’ operational semantics in a highly flexible way, yet ensuring that encapsulation is still preserved since only deliberately exported structure is accessible by other languages. That is, integration of operational semantics is achieved by defining/redefining operations, which allows for better compositibility. We are not aware of any other mechanism that allows integrating operational semantics directly as we do, e.g. without transforming or mapping the models to a semantic domain (that is, another executable language or formalism). The main contribution is an approach
for integrating languages’ operational semantics in a highly flexible manner without rendering other artefacts, that are defined relative to the metamodel, invalid.

The mechanism has been defined formally and validated by implementation of a framework. The framework is built on top of EMF. The implemented framework supports a prototype metamodeling language resembling Kermeta and KM3 [11]. All examples and code in the report are expressed in this language.

Section 2 introduces the integration mechanism. Section 3 puts theory into practice and describes the mechanism using an example. A discussion and evaluation of the mechanism then follows, before the implemented framework is briefly described. Section 4 puts our work in context with related work in the field. Finally, Section 5 concludes the report and points out extensions and future work.

2 The Integration Mechanism

The integration mechanism supporting the approach discussed in this report allows integrating languages’ operational semantics in a highly flexible manner. The mechanism allows exporting entities from a language’s metamodel by the use of namespaces. A namespace is exported by one metamodel and imported by another metamodel within an operation of a metamodel class. A namespace contains attributes, references and operations, known as exported structure. The exported structure/entities is decided by the designer of a language. A language may export an arbitrary number of namespaces, and may also import an arbitrary number of namespaces that have been exported by other languages.

The mechanism is designed for metamodeling languages that allow defining metamodels’ operational semantics by using an action language. Kermeta is an example of such a metamodeling language (or metalanguage). In Kermeta, operational semantics is defined directly in the operations of the metamodel classes using a promoted action language [7]. This means that a Kermeta metamodel defines both the abstract syntax and the operational semantics. EMF uses a slightly different approach; the metamodel structure and action language are defined separately in different modelling spaces [6]. The abstract syntax is defined in Ecore models, whereas the operational semantics is described using Java. In principle, the mechanism discussed in this report may be applied to both Kermeta and EMF. As a first iteration, we have focused on metamodels whose operational semantics is defined directly in operations of classes (e.g. the Kermeta approach). Hence, a namespace is imported using an import statement within a class operation’s definition. The structure of an imported namespace can be used type-safely together with the structure in the class (i.e. attributes, references and operations)
and by the code in the operation that imports the namespace.

The high level of flexibility of the mechanism requires special effort in order to ensure that a given language’s runtime properties are preserved when its operational semantics is integrated with other languages’ operational semantics. For instance, the state of all objects has to be valid. This translates to ensuring that the values of instance variables (class variables) with a numerical type are within the correct range. Moreover, the values for operation arguments and the result/return value of operations have to be valid, i.e. within the correct ranges for numerical types. This has been addressed in two ways. First, only explicitly exported structure is available for utilisation by other languages. This allows defining integration points at the discretion of the language’s designer. Second, invariants, pre- and post-conditions (contracts) allow specifying constraints that are enforced at runtime. The designer of these contracts would be the language designer.

The underlying idea of the approach is that DSLs’ operational semantics may be defined and tuned by utilising shared structure. This allows composing the operational semantics of languages with as few constraints as possible since there is virtually no limitations on how the structure can be combined with the operational semantics/code of the importing metamodel. Specifically, entities of a namespace may be used arbitrarily as long as type-safety is preserved (which can be checked statically). A conceptual overview of how our approach integrates two metamodels’ operational semantics can be seen in Figure 1. Notice how namespaces are exported at the class level and imported at the operation level. This is a deliberate choice. First, exporting local variables from within operations does not make any sense since these are first put on the runtime stack when an operation is invoked (and they do not influence the state of an object). Hence, attributes, references and operations are the only entities that are reasonable to export. Second, importing a namespace within the definition of an operation both makes adaptation of a language construct highly flexible and avoids breaking the conformance between existing models and the metamodel. We will later elaborate on the latter point.

![Figure 1: Conceptual overview of using exported namespaces](image-url)
2.1 Name Binding, Scoping and Masking

Importing a namespace makes the structure/entities of the namespace accessible in the operation in which the namespace is imported into, as if the structure was actually defined within the operation or at the class level. A namespace import statement can be used wherever a regular statement is legal. For example, a namespace may be imported at the outer block of an operation or within an if-statement. The entities of an imported namespace can be used directly by other statements in the operation. Name resolution follows lexical scoping rules. This means that an identifier is bound to the first attribute, reference, variable, parameter or operation that is accessible by tracing upwards through the blocks/scopes regardless of whether the resolved entity is imported or not. This implies that imported structure can mask non-imported structures. It is also possible that imported structure may be shadowed by other imported structure (e.g. several namespaces can be imported within a single operation). Clearly, this may induce name conflicts. To address this, all entities of a namespace can be given an alias. A name alias replaces the original name of a namespace entity (e.g. the name of an attribute) in a given scope, and in all of the inner scopes.

We will use a simple example to illustrate how the mechanism works. Figure 2 shows an excerpt of a metamodel containing a class C. The class contains an attribute named a and an operation named o. The operation imports the two namespaces na and nb that have been exported from another metamodel identified as example. The namespaces are imported in different nested levels.

```plaintext
import metamodel "..." as example

class C
{
    attribute a : Integer

    operation o(){
        <statements_1>
        import:i1 example.na, example.nb
        {<statements_2>
            if( true )
            {
                var b : Integer;
                import:i2 example.na ( a as aa ), example.nb
                {<statements_3>}
            }
        }
    }
}
```

Figure 2: Importing namespaces

For the sake of the illustration, the na namespace contains an attribute named a, whereas the nb namespace contains an attribute named b. Refer Figure 3 for an excerpt of the metamodel/class exporting the two namespaces.
class D
{
    attribute a : Integer, export to na
    attribute b : String, export to nb
    ...
}

Figure 3: Exporting namespaces

For <statements_1>, all occurrences of the identifier a bind to the attribute a in the C class. (We assume that <statements_1> does not include variable declarations. Also nesting of operations is not allowed.) In <statements_2>, a and b bind to the imported attributes. Hence, the imported attribute a from the na namespace masks the a attribute in the class. Both namespaces are imported again using a new import statement within the if-branch. Clearly, this is not required for the na namespace since the content of this namespace is still accessible within the if-branch. However, importing the namespace again illustrates the mechanism in greater detail. In <statements_3>, a still refers to a from the first import of the na namespace, since there is no variable declaration, alias or imported attribute/reference from another namespace that masks the imported attribute. To illustrate name aliases, notice how the a attribute from na is now also accessible in the inner block using the name aa (from the second import of the na namespace). aa is an alias for the a attribute in the na namespace (the alias only represents the a attribute for the second import of na - a and aa now refer to the same attribute). In <statements_3>, b binds to the b attribute of the nb namespace available by the inner namespace import statement for nb (as the variable declaration masks b imported during the outer import of the nb namespace). The flexibility in how namespaces can be nested simplifies the integration process between existing code in a class and imported structure, as the structure of namespaces may be imported where it is required in the existing code. Also, this reduces the number of name conflicts that may emerge if a namespace is imported only once at the class or metamodel level, e.g. as a package import. The current version of the framework imports all entities in a namespace during namespace import. We foresee that being able to select or filter what entities to import may be beneficial. This is identified as future work.

2.2 Static Type-Checking

With respect to our mechanism, static type-checking has four tasks: 1) ensuring that an imported attribute, reference or operation has the correct type according to the expression in which it is used, 2) ensuring that arguments used during invocation of an imported operation have the proper types, 3) detecting type errors that occur as a consequence of identifiers being masked during namespace import and 4) issuing warnings about namespaces that have identical structure names.

Exporting references or operations with non-primitive parameters and result types requires class types to be exported as well. Class types are not
explicitly exported to a namespace. Instead, a class type is implicitly available in a block defined by an import statement when the type is referenced. This can be done automatically. An example will illustrate this in detail.

class C
{  
  operation o( p : D ), export to n { ... }  
}

class D
{  
  reference e : E[0..1]  
}

class E { ... }

Figure 4: Exporting class types

Figure 4 shows three classes of a metamodel. The C class has an operation o() with one parameter. The parameter is typed with the D class. Being able to type-safely invoke the operation requires the argument to be an object of the D class (or any of its subtypes). Hence, the class D must be available in the scope in which the operation is imported. This in turn may require that the E class is imported in the scope as well, since an object of the D class may link to an object of the E class. Our implemented framework solves this by adding all classes of a metamodel to a repository that may be queried at runtime when a class is needed, e.g. when a local variable in an operation in one metamodel is typed with a class of another metamodel.

2.3 Interpretation

The mechanism induces dependencies between models. Specifically, interpreting a model of one metamodel depends on accessing one or more models of other metamodels as dictated by how namespaces are exported and imported. The specific models that are intended to communicate need to be identified manually. The reason for this is that there may exist an arbitrary number of models of a given metamodel. Moreover, each model may consist of an arbitrary number of objects of a given class (in a metamodel). Therefore, it is also required to specify the links between objects of the models that will exchange messages at runtime. The links could be specified e.g. by using a graphical editor (not implemented). The result is a linking model containing a set of bi-directional mappings between model objects. Figure 5 illustrates such a model.

Figure 5: Visualisation of a linking model
Each namespace import statement has an identifier (\texttt{id}) which makes it possible to pin-point exactly the objects that are accessed within such a statement at runtime, see Figure 6. This is required since several import statements may be used within one operation, and they may be nested. The identifiers are utilised when creating the linking model, as is illustrated in Figure 5. Only communicating objects are specified in the linking model.

```plaintext
operation o()
{
    // Namespace import statement with id: i1
    import:i1 (...)
    import:i2
    {
        import:i3 {...}
    }
}
```

Figure 6: Namespace import statements with ids

The linking model is queried by the framework at runtime (with respect to the export and import statements in the metamodels and the import statement ids). A runtime object is either mutable or immutable. This is specified manually. The state of an immutable object cannot be changed, whereas the state can be changed if the object is a mutable object. An object may be both mutable and immutable with respect to different namespace imports (synchronisation/thread safety has not been addressed in the current version of the framework).

As in Kermeta, interpretation of a model starts by invocation of a (main) operation. The program control then propagates depending on the structure of the operational semantics. Invocation of an operation that imports a namespace may affect the state of an object in another model. For instance, executing an assignment statement may change the value of an attribute on a mutable object.

An operation being invoked via a namespace import is run in the context of the class in which the operation is defined. Operation invocations adhere to static scoping conventions, i.e. exporting and importing an operation resembles how closures are used in programming languages. This means that the execution context for each statement in the operation is oblivious of where the operation is invoked from.

Interpretation also deals with evaluating invariants, pre- and post-conditions. It is possible to introduce cycles, i.e. when two operations invoke each other directly or transitively through a number of exported namespaces. Cycles may be detected by the type checker (currently not supported).

### 2.4 Semantic Interoperability

Combining two metamodels’ operational semantics means that two computational domains are mixed. As described, only deliberately exported structure may be imported and accessed. Moreover, invariants, pre- and post-conditions ensure that values for e.g. attributes and parameters are always
within pre-defined ranges. However, these constraints can not ensure that integration of metamodels’ operational semantics is sound. Also, a construct (class) of one metamodel does not necessarily have the same semantics and meaning as a construct of another metamodel; which may introduce semantic inconsistencies. Clearly, writing the glue code has to be performed with caution.

One way of increasing the chance of a sound semantical integration between metamodels is to only allow exporting operations. An operation has a pre-defined task and is likely not to render a model state invalid (as may be the case by updating the values of class attributes individually). By enforcing this limitation, we are able to mimic the invocation of API operations. We may relax things further by allowing exporting attributes as well and stating that their values can not be changed from within the operation that imports them; only operation invocations may change a model’s state.

Further restrictions may be enforced regarding how the imported structures can be utilised, conditions for when the value of attributes may be changed and similar. In the general case, being able to automatically check whether integration of a set of metamodels is semantically sound is very difficult, if even possible. It requires an in-depth reasoning about a metamodel’s properties and semantics.

3 Example and Discussion

We will use an example to illustrate how the mechanism can be used. The example will also provide the foundation for the discussion of the mechanism. State machines are used in modelling of simulations. Computer games can be seen as a kind of simulation. A typical scenario is when some kind of entity operates in a virtual world. Figure 7 gives an excerpt of a metamodel for modelling of simple behaviour for a spaceship.
namespace "http://game/v1.0"
import namespace "http://statemachine/v1.0" as sm

class Spaceship {
    reference containment armor : Armor[1..1]
    reference containment shield : Shield[1..1]
    reference containment weapon : Weapon[1..1]

    operation receiveEvent( se : SensorEvent ) {
        ...
        armor.updateArmor( value );
        ...
        shield.updateShield( value );
        ...
        weapon.updateArmor( value );
    }
}

class Armor {
    attribute armor : Integer, export to spaceship [armor > 0, armor < 100]
    operation updateArmor( value : Integer ) { armor = value; }
}

// Shield and Weapon are defined similarly as Armor

class Shield { ... }
class Weapon { ... }

Figure 7: An excerpt of a metamodel for modelling of spaceship behaviour

The spaceship has an armor, a shield and a weapon. The effect and power of these components vary depending on whether the ship has collected items, e.g. more power or a new weapon. The status of each component is described with an integer value, see the Armor class. Sensors (e.g. when the ship collects a new weapon) invoke the receiveEvent(...) operation on a Spaceship object. The three consecutive dots represent additional code, e.g. code that decides what value each component should have based on the recently received sensor input. The metamodel exports class attributes (e.g. armor of the Armor class) to a namespace called spaceship. It also imports a namespace from a state machine metamodel. How this state machine is used will be discussed later (refer Figure 9).

It is clear that modelling the behaviour becomes complex as the number of sensors increases. Hence, using a state machine would simplify this task drastically. Figure 8 gives the metamodel for a state machine language (some references are excluded for brevity). The metamodel imports the namespace defined in the game metamodel.

namespace "http://statemachine/v1.0"
import namespace "http://game/v1.0" to game

class StateMachine {
    reference containment states : State[0..*]
    operation run( event : Event ), export to statemachine [ ... ]
}
class State {
    reference incoming : Transition[0..*]
    reference outgoing : Transition[0..*]
    operation step( event : Event ) { ... }
}
class Transition {
    reference source : State[1..1]
    reference containment action : Action[1..1]
    operation trigger( event : Event ) {
        action.invoke();
    }
}
class Action { operation invoke() {} }
class Event {}

Figure 8: Simplified state machine metamodel
By using two namespace export/import pairs, the behaviour of the spaceship can now be modelled by a state machine model. Specifically, by overriding the `receiveEvent(...)` operation in a subclass of `Spaceship`, the operation may import the `statemachine` namespace exported from the state machine metamodel. `receiveEvent(...)` can invoke the `run()` operation on the `StateMachine` class directly (the `Event` class is imported implicitly since the parameter of the `run()` operation is typed with this class.) By importing the `spaceship` namespace within an overridden version of the `trigger()` operation in a subclass of `Action`, the correct values for each of the class attributes (`armor`, `shield` and `weapon`) can now be set explicitly for each transition. See Figure 9 for the overridden operations.

```java
// From the game metamodel
class Spaceship extends Spaceship {
  operation receiveEvent( se : SensorEvent ) {
    ... import:i1 sm.statemachine
    var event = init new Event();
    ... run( event );
  }
}

// From the state machine metamodel
class Action extends Action {
  operation invoke() {
    import:i2 game.spaceship
    armor = ...
    shield = ...
    weapon = ...
  }
}
```

Figure 9: Integration of the operational semantics using two exported namespaces

So what do we achieve? Let us see how a similar interaction between the two languages’ operational semantics would be achieved without using exported namespaces, i.e. by utilising a set of operation invocations. First, in order to run the state machine, the `receiveEvent(...)` operation needs to have access to an object of the `StateMachine` class. This implies that the entire state machine metamodel with all its classes needs to be imported by the game metamodel. Furthermore, the state machine object needs to be stored by an attribute in the `Spaceship` class so that the same state machine object is used for each event received. As can be seen, this tangles the language specifications, as the reference to a state machine does not logically belong to the spaceship concept. More importantly, adding a reference to the class breaks the conformance between existing spaceship models and the spaceship metamodel, as the reference is unknown to the models. This may also render existing language tools and artefacts like editors, concrete syntaxes and model transformations invalid. This is avoided by using our mechanism because shared entities of one metamodel are imported within an operation of another metamodel. Conformance is a relation that ignores the content of
operations [9]. Hence, adding namespace import statements to an operation does not break conformance or compromise tool support, as the artefacts are oblivious of the new code. (The proof for this is trivial.) Second, in order for the `invoke()` operation to set values for the attributes in the component classes (e.g. `Armor`), the `invoke` operation needs to have access to objects of these classes. This means that all `ActionSub` objects need to refer the same objects (using added references) which is cumbersome if the state machine has a lot of states and transitions. Third, values for the attributes in the classes may be set directly on objects of these classes, e.g. `armorObject.armor = 3;` however, this is harmful since there is no way to ensure that the values adhere to the value ranges of these attributes (on the contrary, notice how invariants are used in the example for the `Armor` attribute). The solution is to invoke getter or setter operations, or defining derived properties where the properties have explicitly defined setter code [7]. This requires greater changes to the metamodels, and also clutters language specifications. Fourth, when metamodels contain cross-references, e.g. when the `Spaceship` class contains a `StateMachine` reference, the models conforming to these metamodels are not separate entities anymore. That is, the models of each metamodel are combined to a composite model. By using exported namespaces this is instead addressed by linking what model objects that should communicate, which increases separation of concerns. For the spaceship example, this means that the `Spaceship` object needs to be linked to the `StateMachine` object, whereas each `ActionSub` object needs to be linked to the `Armor`, `Shield` and `Weapon` objects. The links can be described externally from the models, e.g. in a linking model which is simple to maintain. As has been illustrated, this means that attributes, references and operations of a class in one metamodel can be accessed from within an operation in a class of another metamodel without explicitly using an object of the first class. The resolution of objects happens "behind the scenes" according to the mappings of the linking model. Hence, models of different languages can be created separately by different stakeholders and later combined by specifying a simple linking model.

The metalanguage we have defined provides constructs for exporting namespaces directly from within a metamodel’s definition. An alternative approach is to specify the namespaces in an external model (lifted away from the metamodel definition). This allows defining (additional) namespaces completely non-intrusively without using inherent namespace export statements. Also by having the namespaces specified externally it is possible to decide what structures to export when it is clear what languages that are going to be integrated. This can be difficult to foresee when designing a language. Furthermore, pre-decided exported namespaces may be removed for a specific integration scenario.

The implemented framework is built on top of EMF using the `Textual`
Modeling Framework (TMF) [12]. It reads metamodels (defined in the metamodel using a metamodel editor) and builds a representation of them as Java objects which are added to a repository. A model of a specific metamodel is created programmatically in Java by querying the repository for a specific metamodel which gives access to all the classes of the metamodel. Objects of these classes can then be created to construct the model. The framework also supports creating models using an XMI editor. The mechanism is realised by keeping record of what metamodels that are integrated by export and import statements. The correct order of execution is achieved by the framework by giving each statement a unique identifier which is used to create a correct execution sequence (the statements are ordered according to their identifiers). As an example, importing a namespace in-between statements of an existing operation changes the desirable execution sequence of the existing code in the operation. That is, the code that now comes after the import namespace statement should be executed after the code within this statement. Models conforming to metamodels defined using the metalanguage are interpreted directly at runtime (recursively). The current versions of the implemented framework and the metalanguage have some limitations. For example, an object is specified as mutable or immutable for all import namespace statements. That is, it is not currently possible to specify this specifically for a given import namespace statement. Moreover, the metalanguage does not have constructs for iterating a list of objects, e.g. when a reference has a multiplicity whose upper bound is greater than one. With our prototype metalanguage we have shown that the mechanism can potentially be implemented in Kermeta (the metalanguage closely resembles Kermeta). Refer the implementation for further details.

4 Related Work

There exist several approaches for semantically integrating models and metamodels of different domains, e.g. by using name-based references, weaving models and model mappings [5]. These approaches rely on using either code generation or model transformations for reifying low-level integration. This means that information for realising integration is incorporated into a code generator or transformations. In this report we follow another approach by exporting namespaces. The mechanism is based on the principle that structures/identifiers can be shared between metamodels. In particular, the structures can be utilised when defining the operational semantics using an action language. Using exported namespaces resemble weaving models to some degree. A weaving model describes a set of links between elements in different models, e.g with the purpose of expressing equivalence or composition [19]. The authors of [5] discuss a related approach where ontologies are used to establish semantic links between models. However, the work does
not go into details on how a low-level implementation of the links can be derived or constructed. We take this a step further by defining exactly how language constructs (classes) are related/integrated at a low-level - not only how they are related at a high-level.

Code generators and transformations allow reuse. However, manual glue code always has to be written to support integration of custom semantics. By using exported namespaces this glue code is specified directly in class operations (within import blocks).

The authors of [14] discuss how action semantics modules can be combined to gradually develop a language. Action semantics use context-free grammars to define abstract syntax trees, which are mapped to an action using a semantic equation. An action specifies a computational-independent representation of behaviour. The work does not discuss how independent languages may be combined, and whether this is even possible using action semantics. The aim is to facilitate reuse of behavioural specifications and not integration of languages’ semantics.

Aspect-oriented approaches allow combining different code bases. The main motivation for aspect-oriented approaches is separation of concerns. Kermeta [7] supports defining aspects and weaving these together using static introduction. Specifically, the definition of a class may be divided into several aspects which are combined at runtime. Static introduction does not allow dividing the definition of operations between aspects, and is therefore not very flexible nor suitable for integrating the semantics of languages.

[10][15][16][17][18] discuss approaches for managing co-evolution of metamodels, models and other artefacts in the metamodelling ecosystem. All approaches are intrusive, in the sense that the metamodels are changed in a manner which requires models and tools to be updated. The approach discussed in this report focuses on integration of operational semantics and utilises that changing and adding content to class operations do not impact models or tools.

The authors of [19] discuss using concepts for specifying structural and behavioural requirements. A concept can be bound to metamodels that meet the requirements of the concept. This way, behaviour can be specified in a generic manner and be applied to several metamodels. Our approach resembles that of [19]. However, our focus is on integrating the operational semantics of arbitrary metamodels and not application and reuse of predefined semantics.

Semantic anchoring using model transformations is discussed in [20]. The approach allows mapping the abstract syntax of a DSL to the abstract syntax of a semantic unit which allows specifying the behaviour of the DSL. Using a namespace import statement can in principle be seen as mapping a metamodel construct to a semantics unit (via the exported structure). The semantic unit in this case is the class (and metamodel) that exports the structure. As with semantic units [20], this allows reusing behaviour for
several metamodels.

Interoperability by the means of information exchange across models is identified as one of three possible relationships for supporting interactions by modelling languages [21]. The mechanism for exporting namespaces builds on using information exchange in an interoperable manner; the information shared between models is in terms of values and objects as made possible by the exported (shared) structure.

To a certain extent, our approach resembles how object files are linked in C/C++. However, there is an important difference. Being able to quickly prototype and combine languages requires an agile mechanism that also supports traceability. Our approach does not require compiling object files/libraries that are later linked. In contrast, we can integrate an arbitrary number of languages (in a random manner) and verify/utilise their integration straight away.

In [22] we discuss a similar approach for integration of operational semantics by defining mappings between structures of different metamodels and using proxy classes. The mappings define unification points between two metamodels, e.g. a unification of a proxy class with a target class. Hence, a proxy class in one metamodel represents a target class in another metamodel. Invoking operations on a proxy object is resolved as invoking operations on an object of the target class. In a similar way, it is possible to set or get the values of attributes in an object of the target class by setting or getting values on the proxy object. The operational semantics of two or more metamodels can be integrated by using an arbitrary number of proxy classes (and proxy objects). The approach is practically non-intrusive since the mappings between the structures are defined externally from the metamodels. Objects of different metamodels are connected by using a linking model in the same way as discussed for the approach in this report.

5 Conclusion and Future Work

This report addresses how the operational semantics of metamodels/languages may be integrated in a language-driven development setting, i.e. when several smaller languages are used to model a software system. The main contribution is a novel approach for integrating the operational semantics by using exported namespaces. An exported namespace comprises attributes, references and operations that are exported from one metamodel and imported by other metamodels. The imported structure can be used directly within code to define the operational semantics of a metamodel in a type-safe manner. Using exported namespaces increases the flexibility in how the operational semantics of metamodels can be integrated with respect to other alternatives like utilising APIs and aspect-oriented mechanisms. Importantly, using exported namespaces does not break the conformance re-
lation between existing models and their metamodels. Also, the impact on existing language tools is low. Future work includes solidifying the framework and evaluating the mechanism on industrial use cases. Integration of languages defined using heterogeneous metalanguages is also an interesting research topic. We believe this can be achieved in the same way as discussed in this report; as long as the operational semantics of these languages are defined using an imperative/object-oriented form of code. More work is required to evaluate whether integration of other kinds of dynamic semantics is possible using a similar approach, e.g. enabling integration by transferring simple typed values between models of different languages.
References


