Service-Oriented Integration of Metamodels' Behavioural Semantics

Henning Berg

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

Abstract. Metamodel composition is a central operation in model-driven engineering approaches. Composition of metamodels is not trivial. The essence of the problem is that metamodels are not defined as reusable artefacts. Moreover, most composition mechanisms focus on the structural aspects of metamodels without considering how metamodels may be composed semantically. Hence, models of different metamodels can not exchange data directly during execution at runtime. In this paper we investigate a new approach for integrating metamodels and their models by considering metamodels as reusable services at a conceptual level. In particular, the behavioural semantics of metamodels can be coupled in a loosely manner, without entanglement of semantically different concepts. This allows creating complex metamodel architectures where separation of concerns is high.

Keywords: Metamodelling, Model Composition, Behavioural Semantics, Aspect-orientation, Service-oriented Architecture, Domain-specific Language.

1 Introduction

Metamodels have an important role in *Model-Driven Engineering (MDE)* [1] where they are used, e.g. as formalisations in language and tool design. In most MDE environments, metamodels are realised as class models. Class models do not have other structure than what can be realised using simple packages, inheritance, composition and association relationships. This means that all metamodel concepts, regardless of purpose, are reified in the same modelling space without the ability to differentiate one type of concept from another. The lack of additional metamodel structure is not critical for metamodels consisting of a limited number of classes. However, as metamodels become larger and more complex, as a consequence of increasing maturity in model-driven approaches, several troubling issues emerge.

Model composition is a commonly used approach for elaborating a model or metamodel with additional concepts, e.g. [2][3][4][5][6]. Model composition is also a prerequisite for generating holistic system code, combining system views, verifying system views consistency and addressing software evolution. Model composition is performed by combining a set of models in an asymmetric or symmetric manner. Composition of metamodels is typically achieved using a variant of class merging or aspect-oriented weaving. Regardless of method, the result is a composite metamodel containing all classes from the source metamodels. There are some evident issues with many of the current model composition approaches. First, classes reflecting concepts of different concerns are all blended without inclusion of any additional metainformation describing from what source models the concepts in question originate, i.e. traceability is not semantically backed up. Second, composition of models induces conflicts that have to be resolved. E.g. class merging implies that the constituent classes do not contain equally named properties of different types, etc. Third, composition of models requires that the source models are altered intrusively. In particular, such alteration is required to integrate the constituent models' behavioural semantics. Fourth, integration of models requires explicit knowledge in metamodel design and insight into the specific environment used to realise the metamodels, e.g. *Eclipse Modeling Framework (EMF)* [7], *MetaEdit+* [8], *Generic Modeling Environment (GME)* [9], *Kermeta* [10] or similar. Fifth, the resulting models become large which makes reuse more challenging. The main problem combining proprietary metamodels is that these are not structured as reusable artefacts. In particular, there are no apparent ways metamodels should be composed. This gives a lot of flexibility since the metamodels can be combined in many different ways. However, this also induces several problematic issues as motivated.

A metamodel typically contains concepts related to one particular problem domain. By combining metamodels it is possible to increase expressiveness by extending the set of concepts that can be used in the conformant models. Composing metamodels belonging to different domains results in different concerns being tangled. This is not practical as metamodels become difficult to grasp and reason about. Even more critical is the inability to differentiate between concerns in associated tooling and editors. E.g. a *Domain-Specific Language (DSL)* made on the basis of three combined metamodels requires an associated concrete syntax where language constructs pertaining to three different concerns are all mixed together. We believe that the ability to consider one concern at the time is important to support increasingly more complex metamodels and associated tooling.

In this paper, we present the novel idea of considering metamodels as services. Specifically, we will discuss how the behavioural semantics of metamodels can be combined in a service-oriented manner, and thereby support loosely coupled integration of metamodels. Note that we do not consider every aspect of services in this paper, but use the concept of service-orientation as inspiration for defining loosely coupled metamodel components.

The paper is organised as follows. Section 2 explains the concept of metamodel components and uses *SoaML* [11] to illuminate how metamodel components are connected/composed. Section 3 delves into details on how metamodel components can be realised, while Section 4 presents an example where two metamodel components are used in concert to construct an e-commerce solution. Section 5 discusses related work, and Section 6 conludes the paper.

2 Metamodels as Services

A metamodel formalises the structure and semantics of models. We consider both static and behavioural semantics as parts of the metamodel. E.g. EMF allows defining behavioural semantics, referred to as model code, in methods of plain Java classes. Alternatively, Kermeta is a metalanguage that allows defining behavioural semantics within the operations of the metamodel classes using an action language. Hence, the conformant models are executable programs. We will not go into details on how the abstract syntax and behavioural semantics are mapped, and consider the behavioural semantics to be defined in operations within the metamodel classes.

A metamodel is constrained to a particular problem domain, and may conceptually be thought of as a service that provides structure and semantics for expressing and solving problems in this domain; in particular, behavioural semantics for performing some kind of processing. A system may be defined by using an arbitrary number of metamodels, each providing concepts for modelling of one particular system view. Metamodels are typically not related. Thus, their conformant models/programs can not exchange data at runtime in a generic manner. In this paper, we discuss how exchange of data between models can be supported by treating metamodels as services that can be connected. This allows models to send messages to each other during execution regardless of the platform on which the models execute.

Service-Oriented Architectures (SOAs) is a software engineering branch that deals with services and how they interact to realise a software system. A service is a reusable set of functionalities that provides value to its clients, e.g. other services. SOA is a broad field. In this paper, we will only use a small subset of the SOA terms and concepts to describe our approach. Specifically, we will use a service-oriented approach for integrating models at runtime. The intention is not to elaborate on all aspects of services nor give a complete definition of such.

To integrate models at runtime we need some kind of framework. Specifically, we need to formalise how the models should integrate. As mentioned, the behavioural semantics of a model can be specified as a set of operations in the model's metamodel. Hence, by creating *mappings* between metamodels' operations, we are able to formalise how their conformant models can interact. A mapping is created by using two types of interfaces: *consumer* and *provider* interfaces. A metamodel may be mapped to an arbitrary number of other metamodels through interfaces. The interfaces can be seen as an extension of the metamodel. We define a *metamodel component* as an entity consisting of three elements: a metamodel (abstract syntax and static/behavioural semantics), tools like concrete syntax and editors (optional), and one or more *service contracts*. A service contract is a SoaML concept for service specification; it specifies an agreement detailing how participants of a service fulfill roles as described by interfaces. A realised service contract is a pair of provider and consumer interfaces. SoaML is an *Object Management Group (OMG)* standardised modelling language for describing services architectures. It provides us with the modelling tool for describing *metamodel architectures*. That is, two or more metamodel components that are connected through interfaces. Note that we will not follow the SoaML specification strictly. Some additional terminology is used. A visual representation of a metamodel component is given in Figure 1. A metamodel component has a name and a platform descriptor identifying the platform on which the component is defined. In this case, the generic descriptions: *Metamodel Component* and *Platform*, respectively. Examples of platforms are EMF and Kermeta.

In SoaML, a service contract is modelled as a consumer and provider role linked by a service channel. Each role is typed with an interface that defines the role's behaviour. In our case, the roles of a service contract are fulfilled by metamodel components.

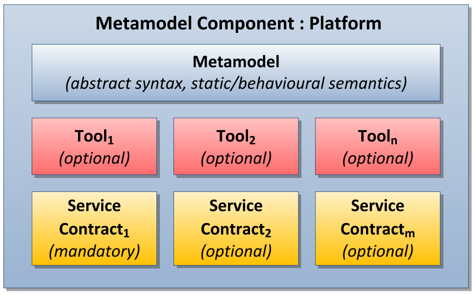

Metamodel Component : Platform
Metamodel
(abstract syntax, static/behavioural semantics)
Tool₁ (optional)
Tool₂ (optional)
Toolₙ (optional)
Service Contract₁ (mandatory)
Service Contract₂ (optional)
Service Contractₘ (optional)

Fig. 1. Overview of a metamodel component

A metamodel architecture is created by choreographing a set of components. That is, each metamodel is bound to one or more roles of service contracts. Binding a metamodel to a service contract role is achieved by mapping each of the interface's operations to an operation found in any of the metamodel's classes. A component whose metamodel is bound to a provider role of a service contract is regarded as an *aspect* component (provider component) from the perspective of this particular service contract. Or more precisely, the component's metamodel is an aspect model since it reflects one particular aspect or concern that is utilised by a *base* component. The component whose metamodel is bound to the consumer role of the same service contract is a base component (consumer component). A metamodel component can take both an aspect and base role simultaneously, and be composed with several components in parallel. This is achieved by utilising several service contracts/interface pairs. Refer [4] for details on how the terms *aspect* and *base* are used to describe model compositions. Figure 2 gives an example where three metamodel components are composed yielding a metamodel architecture.

As can be seen, all components are connected with each other. E.g. $\mathcal{MCA}$ is connected to $\mathcal{MCB}$ through the two interfaces specified by the $\mathcal{P}rovide\,\mathcal{B}_1$ service contract of the $\mathcal{MCB}$ component. Consequently, models conforming to $\mathcal{M}etamodel\,\mathcal{A}$ may invoke operations (operation instances) on models conforming to $\mathcal{M}etamodel\,\mathcal{B}$. The same architecture given as a SoaML services architecture model is given in Figure 3. The metamodel components are here participants that are related through service contracts. SoaML also operates with *service interfaces*. A service interface is a revised service contract, where the provider and consumer interfaces are elaborated with message types. We will limit the use of this term to avoid confusion. Let us focus on the $\mathcal{P}rovide\,\mathcal{B}_1$ service contract and see how it is defined. The $\mathcal{P}rovide\,\mathcal{B}_1$ service contract specifies two roles, here named: baseModel and aspectModel, which are linked through a service channel. The roles are associated with a consumer and provider interface, respectively. Figure 4 gives the service contract as a SoaML model.

The definitions of RequiredInterface and ProvidedInterface are given in Figure 5. The consumer interface specifies an operation named operation_1(...), whereas the provider interface consists of the two operations: operation_1(...) and operation_2(...). We ignore types for now. Both the latter operations are mapped to operations of the B_1 class. A consumer interface may be empty if bi-directional messaging is not required.

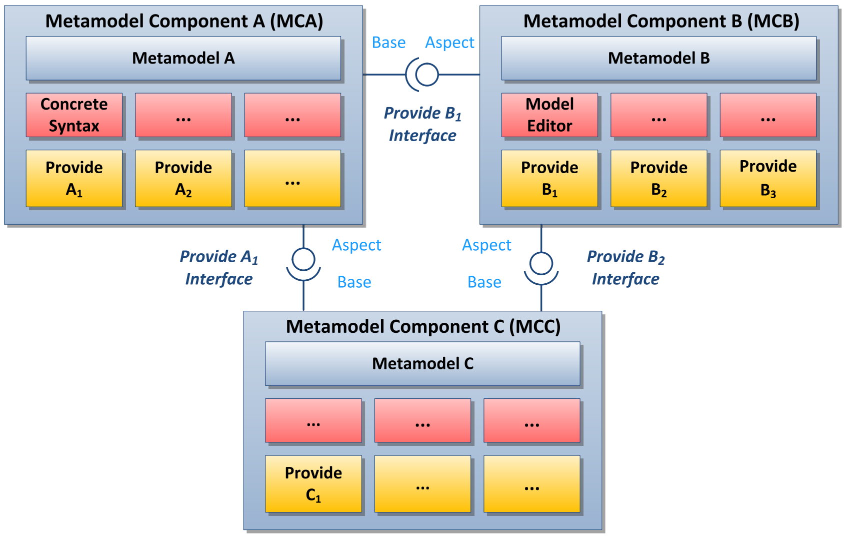


Fig. 2. Example metamodel architecture (using a simplified notation)

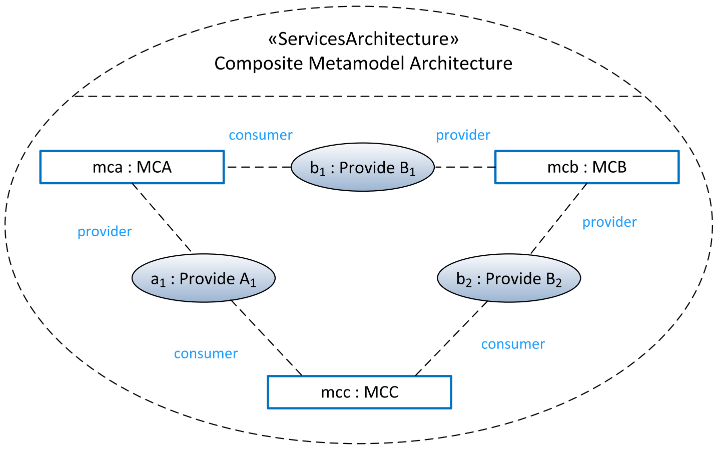


Fig. 3. Services architecture consisting of three participating metamodel components

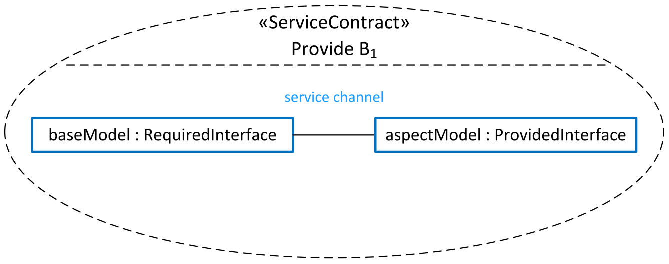

«ServiceContract»
Provide B1
service channel
baseModel : RequiredInterface
aspectModel : ProvidedInterface

Fig. 4. The $\mathcal{P}rovide\,\mathcal{B}_1$ service contract

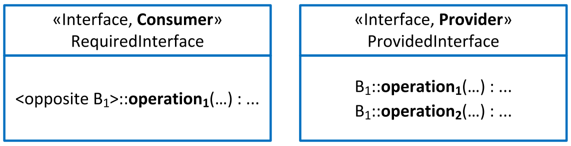

«Interface, Consumer»
RequiredInterface
<opposite B1>::operation1(...) : ...
«Interface, Provider»
ProvidedInterface
B1::operation1(...) : ...
B1::operation2(...) : ...

Fig. 5. The $\mathcal{P}rovide\,\mathcal{B}_1$ consumer and provider interfaces

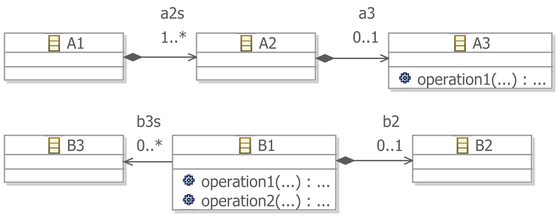


Fig. 6. Excerpts of representatives for $\mathcal{M}etamodel\,\mathcal{A}$ (top) and $\mathcal{M}etamodel\,\mathcal{B}$ (bottom)

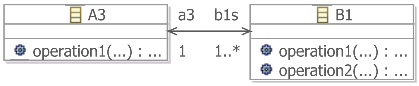


Fig. 7. Realised relationship between two metamodels' classes

Let us see two metamodels that may be connected through these interfaces, and thus fulfill the requirements of the $\mathcal{P}rovide\,\mathcal{B}_1$ service contract. As may be expected, the minimum requirement is a class in each of the metamodels that has the operation(s) specified. Figure 6 shows two excerpts of compatible metamodels. The operations of the provider interface are mapped to the operations of B_1, whereas the operation in the consumer interface is mapped to the operation in A_3. The names of the class operations do not have to be identical to those of the interface operations. Mapping of interface operations to operations in a metamodel's classes is performed by manual specification as part of the service choreography.

Hence, each service contract specifies a service channel between classes of two metamodels. The service channel represents a set of relationships between the classes that realise operations of the interfaces. In this case, there is only one relationship between classes of the two metamodels. That is, there are only two classes in the metamodels that are related. The type of relationship between classes is either an association (non-containment) or composition (containment) reference. According to the $\mathcal{P}rovide\,\mathcal{B}_1$ service contract, there will be a bi-directional relationship between A_3 and B_1. This is because the consumer interface is non-empty. The type of relationship is decided as part of the service choreography.

Operations in the two interfaces associated by a service contract may additionally relate in callback chains, e.g. where one operation in one of the interfaces is invoked as a consequence of invoking an operation in the other interface (reflected in the operation definition). Let us assume that the desirable relationship between A_3 and B_1 is an association reference. Though the metamodels are not composed, we practically end up with the scenario as illustrated in Figure 7.

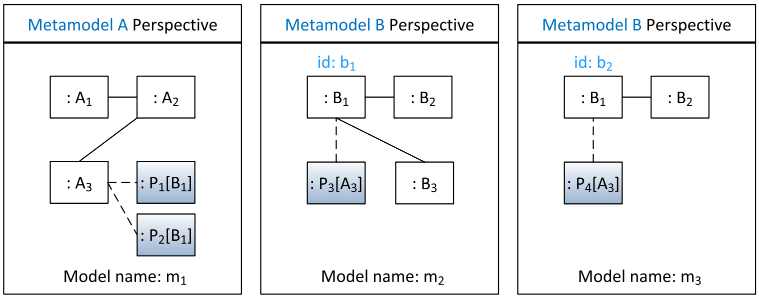


Fig. 8. Modelling separate concerns in different perspectives

3 Realising Metamodel Components

3.1 Modelling Using Proxies

So far we have seen how metamodels' operations can be related through interfaces (service contracts). Thus, the metamodels are loosely coupled which allows creating and processing of models using any kinds of proprietary tools and editors. Also important is the ability to model *A* and *B* concerns independently of each other. E.g. it is still possible to use tools compatible with $\mathcal{M}etamodel\,\mathcal{A}$ to create models of this metamodel. Typically, this is difficult when metamodels are composed since the associated tools need to be refactored. Figure 8 illustrates how a model of $\mathcal{M}etamodel\,\mathcal{A}$ and a model of $\mathcal{M}etamodel\,\mathcal{B}$ can be modelled in separate perspectives (views). Notice that the A_3 object in the left perspective relates two B_1 *proxy objects*, as specified using square brackets. The B_1 object in the right perspective relates a proxy object representing the A_3 object.

A service contract specifies a connection or mapping between two metamodels' operations. As discussed, a service contract's service channel represents a set of class relationships. To realise such relationships it must be possible to navigate the operations accessible through the relationships. This is achieved using proxies. When modelling, it should be possible to refer the B_1 concept from within a model conforming to $\mathcal{M}etamodel\,\mathcal{A}$ (and vice versa), since $\mathcal{M}etamodel\,\mathcal{B}$ is an aspect model with respect to $\mathcal{M}etamodel\,\mathcal{A}$. However, the metamodels are not composed together. To address this, a placeholder/proxy representing a B_1 object can be used in models of $\mathcal{M}etamodel\,\mathcal{A}$. The proxy is linked to an actual object of the B_1 class at runtime using XML-based messages. The object of the B_1 class, as represented by the proxy, is selected from a set of previously created models conforming to $\mathcal{M}etamodel\,\mathcal{B}$, as found in a model repository. That is, all models created using a metamodel architecture are stored in a model repository for later reference. Two B_1 proxies are used in the m_1 model of Figure 8. Each proxy represents a unique B_1 object (as found in m_2 and m_3). The proxies P_3 and P_4 represent the A_3 object in m_1.

Figure 9 shows how the B_1 proxies are linked to the m_2 and m_3 models (and objects) of $\mathcal{M}etamodel\,\mathcal{B}$. The A_3 proxies are linked to the m_1 model in a similar manner (not shown in the figure). The four proxies realise the $\mathcal{P}rovide\,\mathcal{B}_1$ service contract at

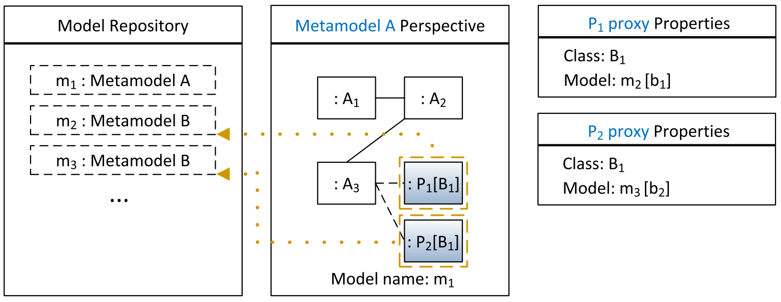


Fig. 9. Linking proxies to models/objects

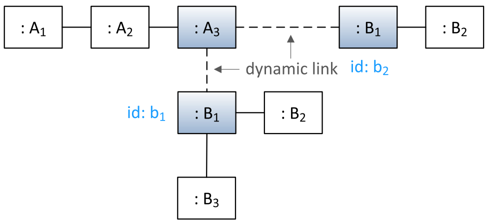


Fig. 10. The resulting model(s) as used at runtime

runtime. We will return to how the interface operations are mapped to class operations. Figure 10 shows what is achieved at runtime when executing the models.

As can be seen, links are established between the A_3 object and the B_1 objects. The links are *dynamic* since they only exist at runtime (realised using proxy runtime objects linked by XML-based messages). Dynamic links are established and maintained by a *metamodel component runtime environment*. The runtime environment acts as a superstructure on top of a metamodelling environment, like EMF. It is out of scope to go into details on how the proxies are managed by the runtime environment.

3.2 Service Choreography

Metamodel components are combined into architectures using *service choreography*. Choreography of metamodel components comprises two steps: mapping operations of service contracts' associated interfaces to class operations and selecting relationship types that the service channels represent. This includes choosing the relationship multiplicities (some constraints apply). Choreography can either be performed textually or graphically. We will illustrate choreography using an XML-based format. Figure 11 gives an excerpt of the service choreography yielding the architecture of Figure 2.

Recall that a service contract is defined using a provider and consumer interface. A service interface is a refined service contract that utilises both the provider and consumer interface to specify a service port type on a component (aspect). The conjugate service interface (defined using the same provider and consumer interfaces) specifies the type of a request port (base). We will only focus on the $\mathcal{P}rovide\,\mathcal{B}_1\,\mathcal{I}nterface$ here. The interface is a refinement of the $\mathcal{P}rovide\,\mathcal{B}_1$ service contract. The $\mathcal{MCA}$ and $\mathcal{MCB}$ components are composed by filling out three pieces of information. First, the operations

```
<interface name="Provide A1 Interface">...</interface>
<interface name="Provide B1 Interface">
  <provider component="MCB">
    <operation name="operation1" type="..." classOperation="B1::operation1" />
    <operation name="operation2" type="..." classOperation="B1::operation2" />
  </provider>
  <consumer component="MCA">
    <operation name="operation1" type="..." classOperation="A3::operation1" />
  </consumer>
  <channel baseClass="A3" aspectClass="B1" type="non-containment"
    bidirectional="true" multiplicityBase="1..1" multiplicityAspect="1..*" />
</interface>
<interface name="Provide B2 Interface">...</interface>
```

Fig. 11. Service choreography using a textual format

of the ProvidedInterface must be mapped to the operations of B_1. Second, the operation of the RequiredInterface has to be mapped to the operation in A_3. Third, the type and multiplicity of the relationship between the A_3 and B_1 classes need to be specified.

4 An E-Commerce Solution

In this section, we will illustrate metamodel components using a more pragmatic example in the domain of website design. We will use two DSLs for modelling of two different concerns: website structure and queries. Excerpts of the metamodels for the DSLs are given in Figures 12 and 13. We will refer to the metamodels as $\mathcal{W}ebsite$ and $\mathcal{Q}uery$, respectively.

As seen in Figure 12, a website comprises one or more pages that contain an arbitrary number of elements. In particular, a page may contain forms realised within a table structure. An example of a form is a list of products or similar, that can be selected by the end user. The Form behavioural semantics includes an operation addObjects(...) which accepts a list of (deserialised) objects. The operation populates a form constructed using a table element. The number of rows and columns in the table is determined automatically by the number and type of objects used as argument. We assume that the metamodels are defined in EMF, thus the behavioural semantics would be written in Java.

A simple query language is given in Figure 13. It captures concepts for expressing queries that can be used for acquisition of objects, e.g. from a database abstraction. A query consists of one or more object identifiers. An object identifier is composed of a set of property name-value pairs which is used to identify a custom set of objects. For instance, an e-commerce solution for selling computer hardware may utilise a domain model including the class Product with data fields for storing information such as product name, manufacturer, version, description, price, etc. Different types of products have unique values for the data fields. Querying for a given type of product would then be performed by using object identifiers and property name-value pairs. Describing queries that are issued to a database is a natural part of designing an e-commerce solution. However, designing the website and programming the queries represent two different concerns. It is likely that different stakeholders would model these concerns. A graphical designer could construct the website, while a programmer would define the backbone business logic including database queries. We have identified two DSLs that allow modelling these concerns. A possible approach would be to compose the metamodels of the

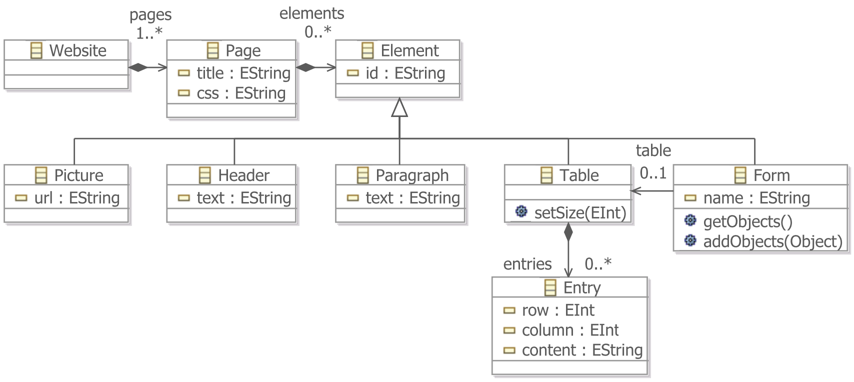


Fig. 12. Metamodel for the website design language

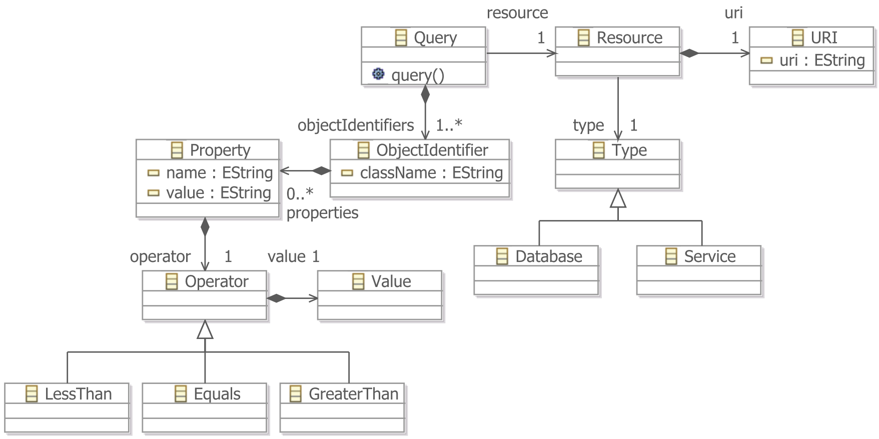


Fig. 13. Metamodel for the query language

DSLs to create a richer language that can be used to both model the website and express database queries, e.g. the Form and Query classes could be merged. First, combining Form and Query is awkward, since these classes are not semantic coherent. Second, the composition process clearly results in entanglement of concepts for expressing different concerns. A graphical concrete syntax for the composite language would yield a palette of language constructs for the entire language, whereas a textual syntax would provide the user with code completion suggestions for all the constructs. A graphical website designer would not be interested in the language constructs for performing queries as used by the programmer, and vice versa. One alternative is to manually program the concrete syntax of the composite language to differentiate the two sets of language constructs, yet the resulting model of a website and associated queries would still be expressed in the same modelling space. Providing two sets of concrete syntax concepts would require in-depth technical knowledge, which reduces the reuse value of the languages/metamodels. Additionally, the website language would typically be implemented with a graphical concrete syntax, whereas the query language is better designed using a textual syntax. Combining different kinds of syntaxes is not a trivial task.

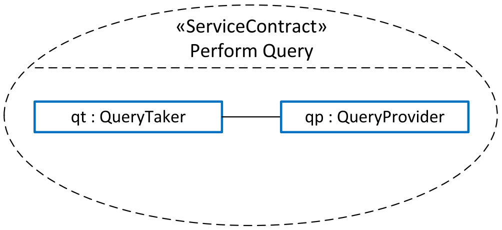


Fig. 14. The $\mathcal{P}erform\ \mathcal{Q}uery$ service contract

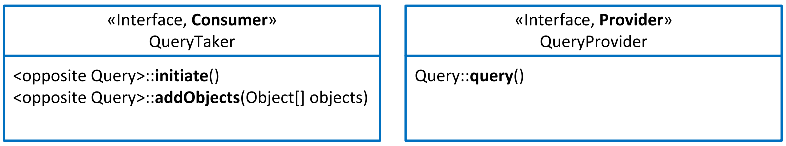

«Interface, **Consumer**»
QueryTaker
<opposite Query>::**initiate**()
<opposite Query>::**addObjects**(Object[] objects)
«Interface, **Provider**»
QueryProvider
Query::**query**()

Fig. 15. The $\mathcal{P}erform\ \mathcal{Q}uery$ consumer and provider interfaces

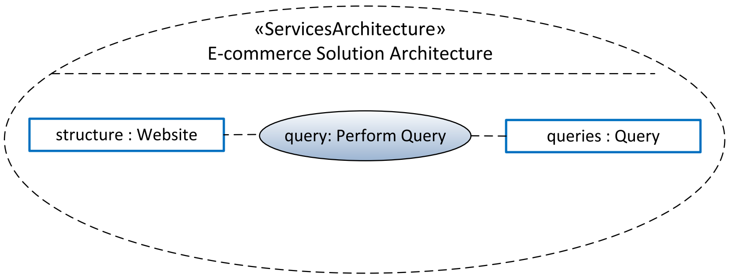
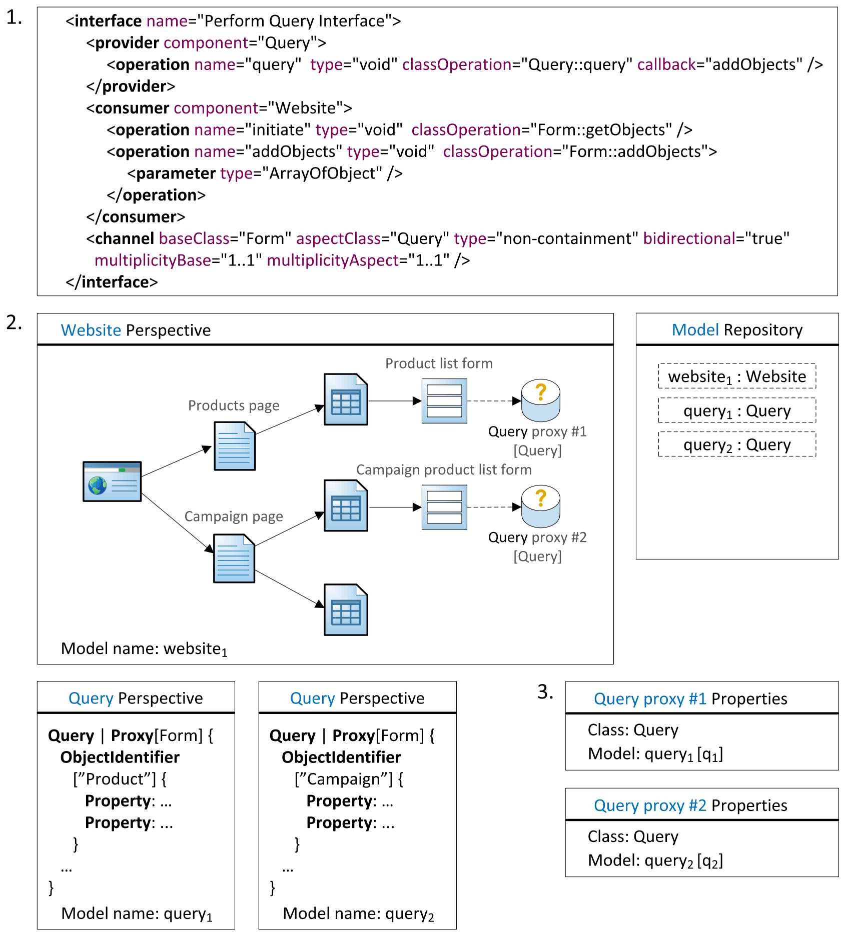


Fig. 16. The e-commerce modelling solution services architecture

Let us see how metamodel components tackle the same scenario. The behavioural semantics of Form in Figure 12 comprises the operations getObjects() and addObjects(...). The semantics of Query in Figure 13 consists of the operation query(). The operations could either be a natural part of the classes' semantics or be defined explicitly in order to construct the metamodels as reusable components. The three operations will reify the consumer and provider interfaces associated by a service contract named $\mathcal{P}erform\ \mathcal{Q}uery$.

The purpose of the example is to illustrate how models of the $\mathcal{W}ebsite$ and $\mathcal{Q}uery$ metamodels may communicate by defining the metamodels as components. The components are named $\mathcal{W}ebsite$ and $\mathcal{Q}uery$ as well. Only the $\mathcal{Q}uery$ component will feature a service contract. The service contract of the $\mathcal{Q}uery$ component is given in Figure 14. It specifies two roles, each typed with an interface. The interfaces are given in Figure 15.

As can be seen, the provider interface has one operation named query(), while the consumer interface specifies the operations initiate() and addObjects(Object[] objects). The names of the class operations that fulfill the service contract do not need to have identical names as the interface operations, however, the class operations' signatures and return types are required to match those of the interface operations. Verification of



```
<interface name="Perform Query Interface">
   <provider component="Query">
      <operation name="query"  type="void" classOperation="Query::query" callback="addObjects" />
   </provider>
   <consumer component="Website">
      <operation name="initiate" type="void"  classOperation="Form::getObjects" />
      <operation name="addObjects" type="void"  classOperation="Form::addObjects">
         <parameter type="ArrayOfObject" />
      </operation>
   </consumer>
   <channel baseClass="Form" aspectClass="Query" type="non-containment" bidirectional="true"
    multiplicityBase="1..1" multiplicityAspect="1..1" />
</interface>
```

```
Query | Proxy[Form] {
  ObjectIdentifier
    ["Product"] {
      Property: ...
      Property: ...
    }
  ...
}
Model name: query1
```

```
Query | Proxy[Form] {
  ObjectIdentifier
    ["Campaign"] {
      Property: ...
      Property: ...
    }
  ...
}
Model name: query2
```

Fig. 17. Choreography and modelling of an e-commerce solution

operation mappings is out of scope of this paper[1]. We assume that each service contract has a description that informally specifies the intended semantics of the associated interfaces' operations. The services architecture describing the e-commerce modelling solution is given in Figure 16. The choreography and modelling process of the e-commerce solution consists of three steps:

1. Service choreography
2. Modelling of each concern in distinct modelling perspectives
3. Linking the base model proxies with the aspect model proxies

Figure 17 shows the three steps of choreographing and modelling of the e-commerce solution (with imagined tool support). The Form class of the $\mathcal{Website}$ metamodel and

[1] Assuring that an operation does what its use requires is also out of scope of this paper.

the Query class of the $\mathcal{Query}$ metamodel realise the consumer and provider roles of the $\mathcal{Perform\ Query}$ service contract (implicitly referred to by the $\mathcal{Perform\ Query\ Interface}$) (1). The initiate() operation of the consumer interface is mapped to getObjects() in the Form class ($\mathcal{Website}$ metamodel), while the addObjects(...) operation is mapped to the equally named operation in the same class. The query() operation of the provider interface is mapped to query() in the Query class ($\mathcal{Query}$ metamodel). I.e. the service contract is fulfilled.

The website and queries are modelled separately (2). The website model contains two forms, thus two queries have to be programmed. The website model is named website_1, while the query models are named query_1 and query_2. All models are stored in the model repository (when saved). Two proxies representing Query objects are used in the website model. A proxy representing a Form object is used in each of the query models. The proxies are linked to the respective models in properties panes/views (3) (only properties for the Query proxies are shown). Several proxies can be assigned to clones of the same model. E.g. if both forms required the same type of query, they could both be linked to, e.g. query_1. At runtime, the operations specified in the interfaces are invoked to exchange data between the models (website_1, query_1 and query_2) using a serialised XML-based message format. Population of a form is initiated when the behavioural semantics of the website language invokes getObjects(). This invocation is resolved by the component runtime environment and results in invocation of query() in the associated query model. Consequently, a set of objects are acquired from the database and returned to the website model via the addObjects(...) operation. addObjects(...) is specified as a callback operation for query(). This means that the query() operation's code invokes the addObjects(...) operation on the associated Form proxy, which in turn invokes the addObjects(...) operation on the actual Form object. The models are linked dynamically. That is, proxies are linked to model objects at runtime. Service choreography can to some extent be pre-defined, where information concerning consumer components is filled in according to a fixed scheme. I.e. the mappings of provider interface operations to class operations are known at design-time. These mappings are immutable properties of the provider components.

The example shows how two metamodels can be used together without using model composition. Here, only one service contract was fulfilled. A metamodel component can feature an arbitrary number of service contracts. This allows creating complex architectures with many metamodels. In addition, it is possible to connect a metamodel with other metamodels in several ways depending on what service contracts that are fulfilled. Figure 18 gives an overview of the resulting website system. We assume that the components are defined in different metamodelling environments.

5 Related Work

The work of [12] discusses how metamodel components can be realised using a graph transformation-based formalisation of MOF. In essence, a metamodel component provides export and import interfaces. Each interface identifies a submodel. A submodel of an export interface can be bound to the submodel of an import interface using graph morphisms, and thereby combining the metamodels. The work resembles the approach

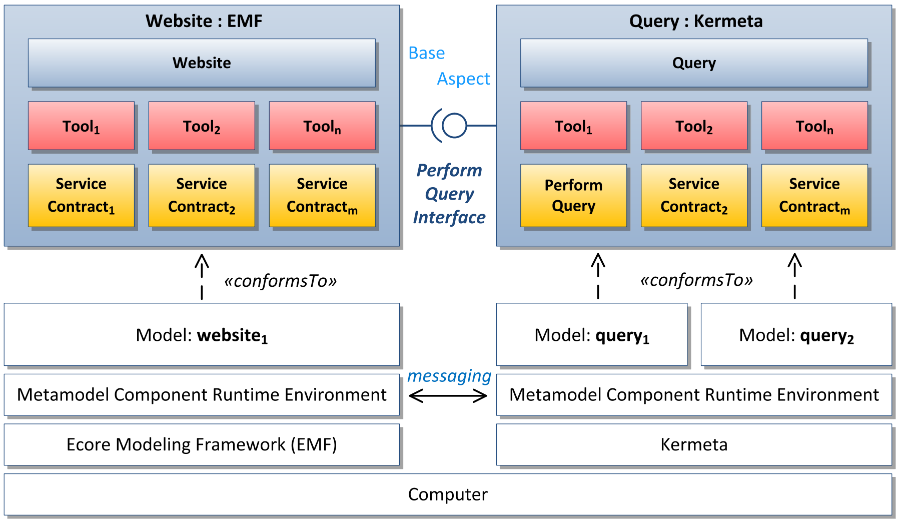


Fig. 18. Overview of the resulting website system

of this paper. The main difference is that our approach allows a higher degree of decoupling, since metamodels are connected as services.

An approach for enabling generic metamodelling is elaborated in [13]. The paper investigates how C++ concepts, model templates and mixin layers can be used to specify generic behaviour and transformations, create model component and pattern libraries and extend metamodels with new classes and semantics. A concept can be bound to models that fulfill a set of requirements specified by the concept. The binding is performed using pattern matching. Consequently, generic behaviour can be reused for instances of the compatible models. Model templates allow defining reusable patterns and components which can be instantiated with actual parameters. The parameters comprise models and model elements. Finally, mixin layer templates facilitate extending metamodels with new classes and semantics in a non-intrusive manner.

Package extension is a mechanism that allows merging equally named classes of metamodels that reside in packages [14]. A package can be defined by extending other packages. The paper also describes a package template concept. A package template is a package that can be parameterised with string arguments. The arguments support renaming of several package elements simultaneously.

An approach for loose integration of models, in the form of model sewing, is discussed in [15]. Model sewing is an operation that allows models to be both synchronised and depend on each other without utilising model composition. The discussed advantages are the ability to utilise existing GUI for the constituent models of a sewing operation, and avoidance of entanglement of concepts from different models. The approach identifies the need of mediating entities that bind the models together. The work resembles the approach of this paper. The main difference is that we utilise interfaces and treat metamodels as components that are combined in a service-oriented manner.

6 Discussion and Conclusions

Metamodel components allow using metamodels in unison without composing these explicitly together. This has apparent advantages. First, it is possible to model different concerns separately, and still support exchange of data between the resulting models at runtime by utilising model links that are maintained dynamically. This ensures a loosely coupled integration. Second, models expressing different concerns can be validated and tested independently one at the time. Specifically, the proxies can communicate with mock-ups that represent models (simulation mode). Third, choreography of metamodel components can be achieved by non-technical stakeholders since the metamodels do not need to be altered in order to connect these. The service contracts formalise the agreement between the metamodels. Fourth, a model or model fragment (clone) can be acquired from a model repository and reused, which simplifies the modelling process.

Metamodel composition usually requires that classes are merged. However, it is not always reasonable to merge two classes, particularly when the classes represent concepts of different problem domains. Using the approach of this paper, an aspect model class is instead used to type the relation between this class and a base model class (and vice versa). This resembles class refinement as discussed in [16].

A consequence of composing the abstract syntax of metamodels is the need of combining the concrete syntaxes as well. This is avoided by using components since each component independently may provide its distinct textual or graphical concrete syntax. Components also address evolution issues. *Model conformance* is a term that indicates whether a model is compatible with a metamodel. Composing metamodels breaks model conformance, which requires using model transformations to create a conformant composite model from the basis of pre-existing models. Components address this by defining a sand box/scope for each metamodel. Changing or revising the metamodel of one component will only break conformance with the existing models of this metamodel. Models of the other components' metamodels in the services architecture will still conform to their metamodels.

Two important aspects of service-oriented approaches are service repositories and service discovery, which support service reuse and availability. These concepts may be adapted for metamodel components. In particular, reusable generic metamodel patterns can be stored in searchable, distributed repositories and used as language building blocks by language engineers. A metamodel pattern may describe an aspect or requirement that is common for several metamodels/languages, e.g. a state machine or similar [17]. Analysis and validation of services are important parts of service-oriented engineering methodologies and required to ensure high quality architectures and systems. This has not been addressed in the paper.

An interesting application of metamodel components is connecting metamodels and languages (and their models) defined in different metamodelling environments. This is possible since the behavioural semantics of each metamodel can be run separately, yet integrated as specified in the service contracts. E.g. models defined in EMF could integrate with models defined in Kermeta, or similar. This is one particular application of metamodel components that justifies the high degree of separation provided by a service-oriented metamodel integration.

We believe that combining metamodels in a service-oriented manner addresses many of the limitations of model composition by increasing decoupling of models. This in turn increases reusability and scalability of metamodels and models.

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. Fabro, M.D.D., Bézivin, J., Valduriez, P.: Weaving Models with the Eclipse AMW plugin. In: Eclipse Modeling Symposium (2006)
3. 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)
4. Groher, I., Voelter, M.: XWeave: Models and Aspects in Concert. In: Proceedings of the AOM Workshop 2007 (2007)
5. 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)
6. Morin, B., Klein, J., Barais, O.: A Generic Weaver for Supporting Product Lines. In: Proceedings of the Workshop on Early Aspects (EA 2008) (2008)
7. Eclipse Modeling Framework (EMF) (2012), `http://www.eclipse.org/modeling/emf`
8. Tolvanen, J.-P., Kelly, S.: MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In: Proceedings of OOPSLA 2009 (2009)
9. Institute for Software Integrated Systems. Generic Modeling Environment (GME) (2012), `http://www.isis.vanderbilt.edu/projects/gme`
10. 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)
11. Object Management Group (OMG). Service-Oriented Architecture Modeling Language (SoaML) (2012), `http://www.omg.org/spec/SoaML`
12. Weisemöller, I., Schürr, A.: Formal Definition of MOF 2.0 Metamodel Components and Composition. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 386–400. Springer, Heidelberg (2008)
13. de Lara, J., Guerra, E.: Generic Meta-modelling with Concepts, Templates and Mixin Layers. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 16–30. Springer, Heidelberg (2010)
14. Clark, T., Evans, A., Kent, S.: Aspect-Oriented Metamodelling. The Computer Journal 46(5) (2003)
15. 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)
16. Emerson, M., Sztipanovits, J.: Techniques for Metamodel Composition. In: proceedings of the 6th OOPSLA Domain-Specific Modeling Workshop (DSM 2006) (2006)
17. Cho, H., Gray, J.: Design Patterns for Metamodels. In: Proceedings of the 11th SPLASH Domain-Specific Modeling Workshop (DSM 2011) (2011)