Service-oriented Design of Metamodel Components

Henning Berg
Department of Informatics, University of Oslo, Oslo, Norway

Keywords: Metamodelling, DSLs, Model Weaving, Model Integration, Aspect-orientation, Service-oriented Architecture.

Abstract: Integration of models is an important aspect of Model-Driven Engineering. Current approaches of model integration rely on model weaving and model transformations. In particular, weaving of metamodels allows extending a metamodel with additional concepts, and thereby supporting a larger problem space. Unfortunately, weaving of metamodels is not trivial and requires in-depth knowledge both of the problem domains of the constituent metamodels and the modelling environment. In addition, name conflicts have to be resolved in an intrusive manner. Another disadvantage of many model weaving approaches is that concepts describing different concerns are mixed together without the possibility for semantically tracing the origin of the concepts. In this paper, we propose a new approach for combining metamodels by defining these as reusable services at a conceptual level. We will show that this approach both addresses the issues that arise when models are woven, and illustrate how metamodel components simplify modelling.

1 INTRODUCTION

Metamodels have a central role in *Model-Driven Engineering (MDE)* (Kent, 2002) where they e.g. are used as formalisations for 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 inheritance, composition and association relationships, and simple packages. 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 in 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 weaving/composition is a popular approach of elaborating a metamodel with additional concepts, e.g. (Fabro et al., 2006) (Kolovos et al., 2006) (Groher and Voelter, 2007) (Morin et al., 2009) (Morin et al., 2008). Model weaving is achieved by combining a set of models in an assymmetric or symmetric manner. Regardless of method, the result is a composite model containing all classes from the source models. There are some evident issues with this approach. First, the resulting metamodels become large which makes it difficult to relate to the models. Second, classes reflecting concepts of different concerns are all blended without adding any additional metainformation describing from what source models the concepts in question originated, i.e. traceability is not semantically backed up. Third, weaving 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. Fourth, weaving of models requires that the source models are altered intrusively. In particular, such alteration is required to integrate the constituent models' dynamic semantics. Fifth, 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)* (EMF, 2012), *MetaEdit+* (Tolvanen and Kelly, 2009), *Generic Modeling Environment (GME)* (GME, 2012) or similar. 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 is domain-specific, in the sense that it contains concepts related to one particular problem domain. Combining metamodels is primarily performed to increase expressiveness by extending the

Berg H..
Service-oriented Design of Metamodel Components.
DOI: 10.5220/0004082900700079
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 70-79
ISBN: 978-989-8565-19-8

set of concepts that can be used in the conformant models. Weaving heterogenous metamodels belonging to different domains results in different concerns being tangled. This is not practical as metamodels become difficult to grasp. Even more critical is the disability to differentiate between the different concerns in associated tooling and editors. E.g. a *Domain-Specific Language (DSL)* made on the basis of three combined heterogenous metamodels yields an associated concrete syntax where language constructs pertaining to three different concerns are all mixed up. We believe that the ability to consider one concern at the time is important to facilitate more complex metamodels and associated tooling.

In this paper, we present the novel idea of considering metamodel components as services. Specifically, we will discuss how the dynamic semantics of metamodels can be integrated in a service-oriented manner, and thereby support loosely coupled integration of metamodels. Service-oriented design of metamodel components yields an alternative to model weaving, where the metamodels are kept separated. Still, the metamodels and their conformant models are integrated and can be utilised in unison to fulfill modelling requirements. 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 (SoaML, 2012) to illuminate how metamodel components are connected. Section 3 delves into details on how metamodel components can be realised, while Section 4 presents a case study where 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

Service-Oriented Arhitectures (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. We will use *SoaML* (SoaML, 2012) in our examples. SoaML is an *Object Management Group (OMG)* standarised modelling language for describing services architectures. It provides terminology for bridging the business and IT levels of service design. Note that we will not follow the SoaML specification strictly, but use it as a platform for describing the concepts of this paper. Some additional terminology is used.

A metamodel formalises the structure and semantics of models. We consider both static and dynamic semantics as part of the metamodel. E.g. EMF allows defining dynamic semantics, referred to as model code, in methods of plain Java classes. Alternatively, *Kermeta* (Muller et al., 2005) is a metalanguage that allows defining dynamic semantics within the operations of the metamodel classes. We will not go into details on how the abstract syntax and dynamic semantics are mapped, and consider the dynamic semantics to be defined in operations within the metamodel classes.

A metamodel is constrained to a particular problem domain, and can be observed as a service that provides structure and semantics for expressing problems in this domain; in particular, dynamic semantics for performing some kind of processing. We will use the term *metamodel component* as an entity that comprises four elements: the metamodel (abstract syntax and static/dynamic semantics), concrete syntax (optional), one or more service contracts and an informal description of the metamodel's domain and purpose, see Figure 1.

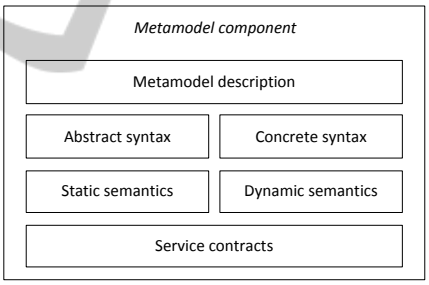

Figure 1: Overview of a metamodel component.

Each service contract defines an integration point between the referenced metamodel and another compatible metamodel. In SoaML, this can be modelled as a service channel between a consumer and a provider role. A composite metamodel is created by orchestrating a set of metamodel components by binding these to the roles of the service contracts, and thereby fulfilling the service contracts. Metamodel components are combined in an asymmetric manner. Thus, the metamodel of a component bound to a consumer role of a service contract implies a base model, while the metamodel of a component bound to the provider role of the contract implies an aspect model. A metamodel can act as both model types depending on whether its component takes a consumer or provider role. It can also fulfill several roles in parallel. Refer (Groher and Voelter, 2007) for details on

how these terms are used in model weaving. An example of a generic services architecture for a composite metamodel is given in Figure 2. Notice that a composite metamodel is more of a conceptual notion, as its constituent metamodels are not woven together.

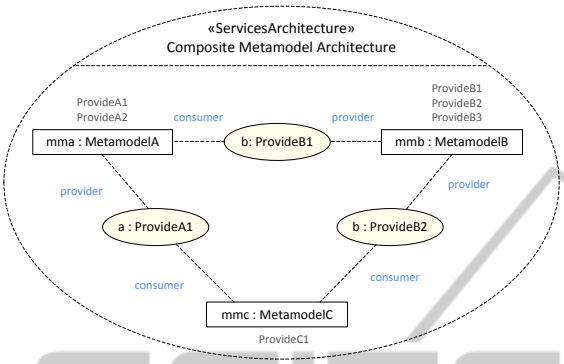

Figure 2: Services architecture consisting of three participating metamodel components.

A services architecture consisting of the three metamodel components: MetamodelA, MetamodelB and MetamodelC, is given in Figure 2. The three metamodel components specify various service contracts. E.g. MetamodelB defines the three service constracts: ProvideB1, ProvideB2 and ProvideB3, whereas the two first are fulfilled as part of realising the composite metamodel. We have included fragments of the abstract syntax for two example metamodels that fulfill the requirements of the ProvideB1 service contract. See Figure 3.

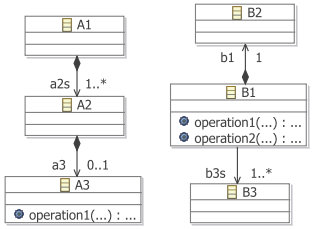

Figure 3: Excerpts of compatible metamodels for the MetamodelA (left) and MetamodelB (right) components.

Let us focus on the ProvideB1 service contract and see how it is defined. The ProvideB1 service contract consists of two roles: baseModel and aspectModel, which are connected through a service channel. See Figure 4. The roles are associated with a consumer and provider interface, respectively, as described by the SoaML specification.

Each service contract specifies an integration point between two classes: one class from the base model and one from the aspect model. The connection between the classes can be realised e.g. as an association or composition relationship. We will later come back to how this connection can be realised. The interfaces for the ProvideB1 service contract are given in Figure 5. The ProvidedInterface specifies two operations: operation1(...) and operation2(...). Both operations are defined within the B1 class. The service contract is part of the MetamodelB component. Thus, the B1 class represents the aspect model part of an integration point. RequiredInterface specifies one operation: operation1(...). This operation has to be implemented by the base model class that relates the aspect model B1 class, and indicates a bidirectional relation between the two metamodels' dynamic semantics. See Figure 6.

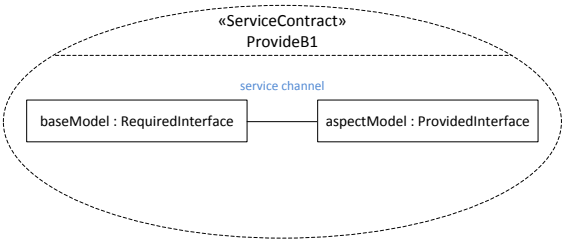

Figure 4: The ProvideB1 service contract.

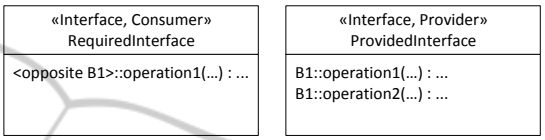

Figure 5: Consumer and provider interfaces.

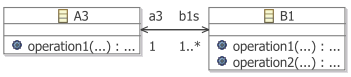

Figure 6: Realised integration point.

Figure 6 shows how the ProvideB1 service contract is realised by the class A3 in the metamodel of the MetamodelA component and B1 in the metamodel of the MetamodelB component. In the remainder of this paper, we will only differentiate between a metamodel and metamodel component where this is required. E.g., we may use 'MetamodelA' to denote the metamodel of the MetamodelA component.

3 REALISING METAMODEL COMPONENTS

So far we have deliberately avoided discussing how metamodel components can be realised. The reason for this is that metamodel components can be realised

in various manners using several supporting mechanisms. Here we discuss some general issues in realising metamodel components and present a brief overview of the most promising approaches.

Weaving metamodels together gives a resulting composite metamodel where all the source metamodels are tightly integrated. Creating models that conform to the composite metamodel takes the same form as creating models of either of the source models, i.e. using an editor that provides representations for all the composite metamodel concepts. Modelling using metamodel components orchestrated as services works slightly different. Specifically, it is still possible to model using concepts from each of the source metamodels separately, see Figure 7. This allows modelling different concerns one at the time. Let us see how this is achieved.

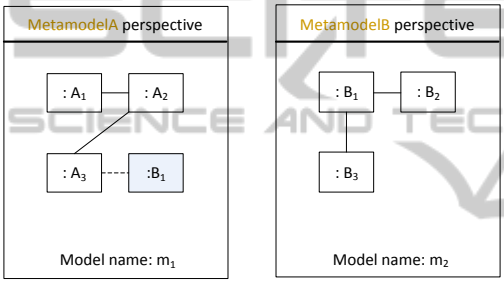

Figure 7: Modelling separate concerns in different perspectives.

A service contract defines an integration point between two metamodels. An integration point is realised as a (possibly bidirectional) relation between two classes and two sets of operations that can be accessed (navigated) using the relation. The operations are specified by the associated consumer and provider interfaces. The relation can take many forms, but would typically be an association or composition. The type of relation is selected as part of the service orchestration.

Let us return to the running example. When modelling, it should be possible to refer the B1 concept from within a model of MetamodelA, since MetamodelB is an aspect model that increases the expressiveness of MetamodelA. However, the metamodels are not woven together. To address this, a placeholder/proxy representing B1 is used in the model of MetamodelA, see Figure 7. The proxy is linked to an actual object of the B1 class at runtime in a loosely manner using XML-based messages. The object of the B1 class is selected from a repository of models conforming to MetamodelB.

Modelling using the metamodel of a metamodel component results in a model that is automatically accessible via a model repository. This facilitates linking the proxy of a model object in one perspective to a model of another perspective. Additionally, the same model (clones) can be used several times. Figure 8 illustrates this. We assume there are five models available in the repository.

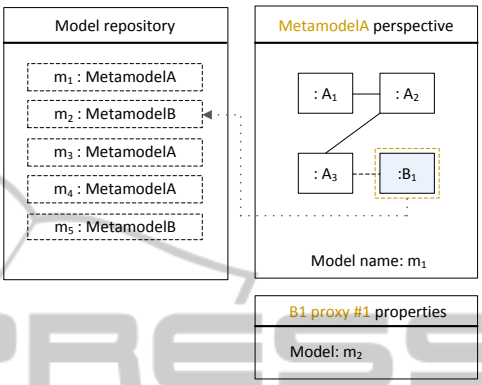

Figure 8: Linking a proxy to a model.

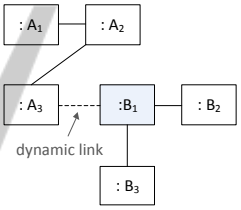

Figure 9: The resulting model as used at runtime.

Message-based links between models are established and maintained by the runtime environment. The runtime environment acts as a superstructure on top of a modelling environment, like EMF. The resulting (apparent) model at runtime is given in Figure 9.

3.1 Service Orchestration

Orchestration of metamodel components comprises two steps: instantiating service contracts and selecting relation type with multiplicity. Orchestration can be performed graphically. Specifically, a relation is made between a base model class and a service contract of the aspect model. In Figure 10, the A3 class of MetamodelA is related to MetamodelB through the ProvideB1 service contract [1]. MetamodelC is integrated with both MetamodelA and MetamodelB. The C1 class has an assosiation to the class A1 (via the ProvideA1 service contract). Objects of the C2 class are composed of B2 objects, as indicated by the composition

[1]The name of the relation indicates base model class.

relation between C2 and the ProvideB2 service contract. Notice how MetamodelA is both a base model and aspect model, depending on its role of the fulfilled service contracts.

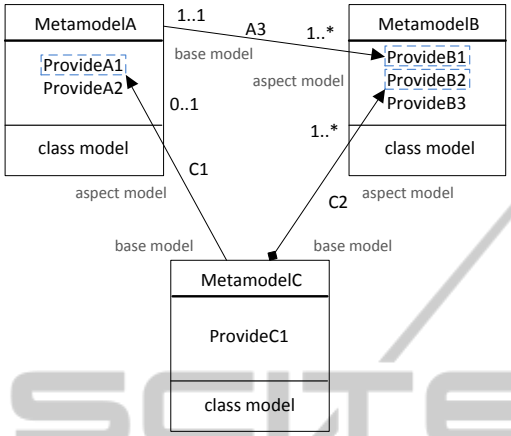

Figure 10: Orchestrating metamodel components.

3.2 Mechanisms for Realisation of Metamodel Services

There are several mechanisms that may support realisation of metamodel components as discussed in this paper. Model types (Steel and Jézéquel, 2005), class nesting and mechanisms used in component technologies are all promising alternatives.

One potential application of model types is for definition of service contracts. A model type can be used to specify one or more classes of the aspect model that form the basis for integration points. This approach would allow defining generic service contracts that can be reused by several metamodels.

Class nesting may be used to identify different perspectives or subsets of classes within a metamodel. In particular, an inner nested class may represent an interface for interacting with other metamodels.

There exist several component technologies used for distributed systems. Some may provide mechanisms that can be used to realise metamodel services. An interesting topic is how to serialise models which can be distributed over a network. This would allow different stakeholders to model a given system without being at the same location. The metamodel components and respective models could later be combined to form the composite model of the system.

4 CASE STUDY: AN E-COMMERCE SOLUTION

In this section, we will illustrate metamodel components using a practical example in the domain of web design. We will use two DSLs for modelling of two different concerns: web site structure and queries. The metamodels of the DSLs are given in Figures 11 and 12. We will refer to the metamodels as Website and Query, respectively.

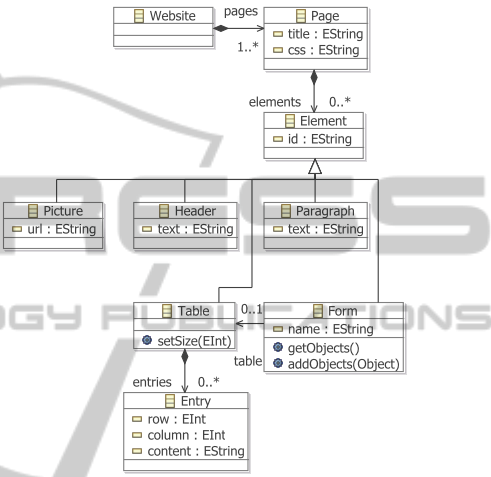

Figure 11: Metamodel for the website design language.

As seen in Figure 11, 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 dynamic semantics include an operation addObjects(...) which accepts a list of (deserialised) objects. The operation populates a form 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 dynamic semantics would be written in Java.

A simple query language is given in Figure 12. 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 are used to identify a given set of objects. For instance, an e-commerce solution for selling computer hardware may utilise a domain model including the domain class Product. An excerpt of the definition of such a class is given in Figure 13.

Distinct products have different values for the at-

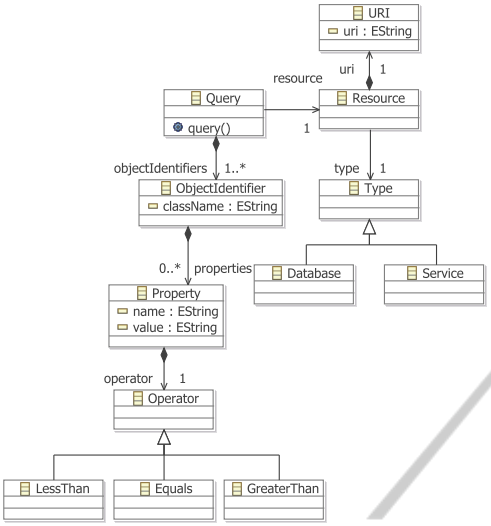

Figure 12: Metamodel for the query language.

```
class Product
{
  private String manufacturer;
  private String name;
  private int version;
  private String description;
  private double price;
  ...
}
```

Figure 13: The Product domain model class.

tributes. Querying for a given type of products is performed by using object identifiers and property name-value pairs. An example of a statement that identifies a set of products from a couple of target manufacturers is given in Figure 14 in a textual concrete syntax. The query statement of Figure 14 returns all products that are manufactured by Toshiba or Acer and cost under $300.

Describing queries that are issued to a database is a natural part of designing an e-commerce solution. 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 address each of these concerns. The traditional approach would be to weave the metamodels of the DSLs to create a richer language that can be used to both model the website and express database queries, i.e. the Form and Query classes would be merged. First, combining Form and Query is awkward, since these two classes are not semantic coherent. Second, the weaving process clearly mixes two concerns. E.g. a graphical concrete syntax

```
Query {
  ObjectIdentifier["Product"] {
    Property: "manufacturer" = "Toshiba"
    Property: "manufacturer" = "Acer"
    Property: "price" < 300
  }

  Resource[Database] {
    URI: "jdbc://..."
  }
}
```

Figure 14: An example query in a textual notation.

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. The 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.

Let us see how metamodel components tackle the same scenario. The dynamic semantics of Form in Figure 11 comprises the operations getObjects() and addObjects(...). The semantics of Query in Figure 12 consists of the operation query(). These 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 of a Query service contract. For this example, only the query component will feature a service contract.

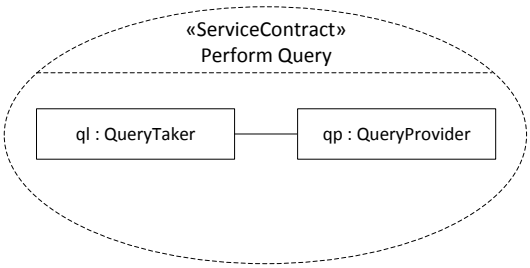

Figure 15: The Perform Query service contract.

The service contract of the query component is given in Figure 15. It specifies two roles, each typed with an interface. The interfaces are given in Figure 16.

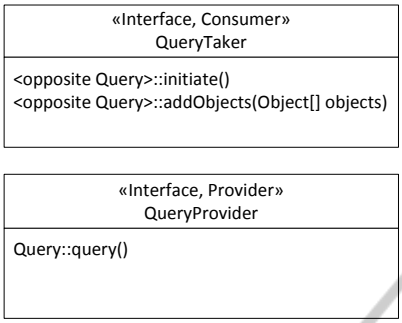

Figure 16: The consumer and provider interfaces associated with the Perform Query service contract.

As can be seen, the provider interface comprises one operation named query(), while the consumer interface specifies the operations getObjects() and addObjects(Object[] objects). The actual operations of the class that fulfill the consumer interface do not have to match the operation names in the interface, however, the target operations' signatures and return types are required to match those of the interface operations. How to ensure that the correct operations are chosen is out of scope of this paper[2]. We assume that each service contract has a description that informally specifies the required semantics of its operations.

The services architecture describing the e-commerce modelling solution is given in Figure 17.

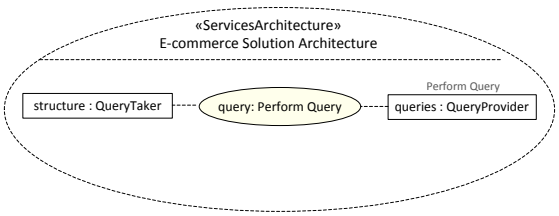

Figure 17: The e-commerce modelling solution services architecture.

The modelling process of the e-commerce solution consists of three steps:

- Service orchestration.
- Modelling each concern in distinct perspectives.
- Linking the base model proxies by acquiring aspect models from the model repository.

[2]Assuring that the operation does what it should is also out of scope of this paper.

Figure 18 shows the three steps of modelling the e-commerce solution (with imagined tool support). The Form class of the Website metamodel is related to the Query metamodel via the Perform Query service contract using an association (1). This is possible since the Form class contains operations with the signatures and return types as specified by the service contract (required operations). The initiate() operation of the consumer interface is realised as getObjects() in the Form class, while the addObjects(...) interface operation is realised by the equally named operation. I.e. the service contract is fulfilled. A simplified description of the functionality offered by the query component is given in XML format. The website and queries are modelled separately (2). The website contains two forms, thus two queries have to be modelled. 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). Note the two proxies in the website model representing the Query class objects. The two proxies are linked to the query models in properties panes/views (3). Consequently, each proxy is bound to the respective Query object of the specified query model. The class of the model object represented by the proxy, and the details on how the proxy object connects to the aspect model, are determined by the service contract and associated interfaces. Thus, there is no need for explicitly designating the proxy to a specific model object. It is already defined in the interfaces. Several proxies can be assigned clones of the same model. E.g. if both forms required the same type of query, they could both be linked to e.g. $query_1$. Models are cloned automatically by the tooling. At runtime, the different operations specified in the interfaces are invoked to exchange data between the constituent models ($website_1$, $query_1$ and $query_2$) using an XML-based message format. Population of a form is initiated when the dynamic semantics of the website language invokes getObjects(). This invocation is resolved by the 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. The models are linked dynamically. Thus, proxies are linked to model objects at runtime. As pointed out, the runtime link is realised as messages sent as serialised XML data (loose coupling).

The example shows how two metamodels can be used together without using model weaving. 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

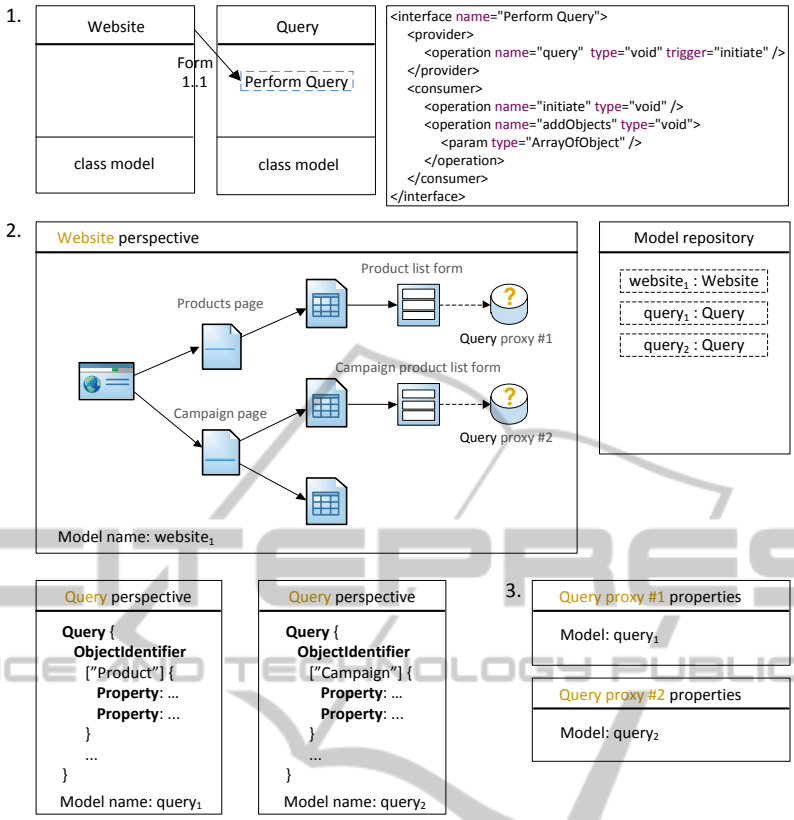

Figure 18: Orchestration of metamodel components and modelling of an e-commerce solution.

integrate a metamodel with other metamodels in several ways depending on what service contracts that are fulfilled.

5 RELATED WORK

The work of (Weisemöller and Schürr, 2008) 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 with the submodel of an import interface using graph morphisms, and thereby combining the metamodels. The work resembles the approach of this paper. The main difference is that our approach allows a higher degree of decoupling, since metamodels are integrated through proxies.

An approach for enabling generic metamodelling is elaborated in (Lara and Guerra, 2010). 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 bounding 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 (Clark et al., 2003). 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 facilitate renaming of several package elements simultaneously.

An approach for loose integration of models, in the form of model sewing, is discussed in (Reiter et al., 2005). Model sewing is an operation that allows models to be both synchronised and depend on each other without utilising model weaving. 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 weaving these explicitly together. This has apparent advantages. First, it is possible to create models that express different concerns in a separate fashion. The models of different concerns are still integrated by the modelling environment using model links that are maintained dynamically. This ensures a loosely coupled integration. Second, models of 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, orchestration of metamodel components can be achieved by non-technical stakeholders since the metamodels do not need to be altered in order to combine these. The service contracts formalise the integration points. Fourth, a model or model fragment (clone) can be acquired from the model repository and reused, which simplifies the modelling process. Thus, it is not required to model the same thing twice.

Model weaving 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 (Emerson and Sztipanovits, 2006).

A consequence of weaving the abstract syntax of metamodels is the need of combining concrete syntaxes as well. This is avoided by using components, since each component independently provides 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 its metamodel. Weaving metamodels breaks model conformance. This 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 component's 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 facilitate service reuse and availability. Metamodel components may follow a similar scheme. 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 (Cho and Gray, 2011). 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 this paper.

An interesting application of metamodel components is for integrating metamodels and languages (and their models) defined in different modelling environments. This is possible since the dynamic semantics of each metamodel component can be run separately, yet connected as specified in the service contracts. E.g. a metamodel and conformant models defined in EMF could utilise models defined in GME, 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 current limitations of model weaving by simplifying integration of models and the modelling process, and thereby increasing the reusability and value of metamodels and models.

REFERENCES

Cho, H. and Gray, J. (2011). Design patterns for metamodels. In *Proceedings of the OOPSLA/SPLASH DSM'11 Workshop*.

Clark, T., Evans, A., and Kent, S. (2003). Aspect-oriented metamodelling. *The Computer Journal*, 46(5).

Emerson, M. and Sztipanovits, J. (2006). Techniques for metamodel composition. In *The 6th OOPSLA Workshop on Domain-Specific Modeling*.

EMF (2012). Eclipse modeling framework (emf). http://www.eclipse.org/modeling/emf.

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

GME (2012). Generic modeling environment (gme). http://www.isis.vanderbilt.edu/projects/gme.

Groher, I. and Voelter, M. (2007). Xweave: Models and aspects in concert. In *Proceedings of AOM Workshop '07*.

Kent, S. (2002). Model driven engineering. In *Proceedings of IFM'02*.

Kolovos, D. S., Paige, R. F., and Polack, F. A. (2006). Merging models with the epsilon merging language (eml). In *Proceedings of MODELS 2006*.

Lara, J. and Guerra, E. (2010). Generic meta-modelling with concepts, templates and mixin layers. In *Proceedings of MODELS'2010*.

Morin, B., Klein, J., and Barais, O. (2008). A generic weaver for supporting product lines. In *Proceedings of the Workshop on Early Aspects (EA'08)*.

Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., and Jézéquel, J.-M. (2009). Weaving variability into domain metamodels. In *Proceedings of MODELS 2009*.

Muller, P., Fleurey, F., and Jézéquel, J. (2005). Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS 2005*.

Reiter, T., Kapsammer, E., Retschitzegger, W., and Schwinger, W. (2005). Model integration through mega operations. In *Proceedings of the Workshop on Model-Driven Web Engineering MDWE 2005*.

SoaML (2012). Service-oriented architecture modeling language (soaml). http://www.omg.org/spec/SoaML.

Steel, J. and Jézéquel, J.-M. (2005). Model typing for improving reuse in model-driven engineering. In *Proceedings of MODELS 2005*.

Tolvanen, J.-P. and Kelly, S. (2009). Metaedit+: Defining and using integrated domain-specific modeling languages. In *OOPSLA 2009*.

Weisemöller, I. and Schürr, A. (2008). Formal definition of mof 2.0 metamodel components and composition. In *Proceedings of MODELS 2008*.