On Language-Independent Model Modularisation

Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
{florian.heidenreich|jakob.henriksson|
jendrik.johannes|steffen.zschaler}
@tu-dresden.de

Abstract. As model-driven software development covers additional parts of the development process, the complexity of software models increases as well. At the same time, however, many modelling languages do not provide adequate support for modularising models. For this reason there has been an increasing interest in the topic of model modularisation, often under the heading of aspect-oriented modelling (AOM). The approaches range from techniques that closely mimic concepts from aspect-oriented programming (AOP), such as AspectJ, to very powerful composition techniques for specific types of models—for example, state machines.

We believe that AOM is more than just copying the concepts of AOP at the modelling level and should rightly include a large number of other model-composition techniques. Developing model composition techniques and tooling is costly, however. To minimise the effort required, this paper presents a generic technique for model composition. The technique is based on invasive software composition and our Reuseware tooling and can be used with arbitrary modelling languages. The basic technique itself is language independent, but it can be adapted to construct language- and purpose-specific composition techniques for specific modelling languages and situations. Hence, it can be used both as a tool for developing specific model-modularisation techniques and as an instrument of research for studying basic properties and concepts of model modularisation. The paper gives a detailed description of our approach and evaluates it using a number of examples.

1 Introduction

Model-driven development (MDD) [1] is increasingly viewed as one way of dealing with the complexity of modern-day software. Its promise is that by making models our primary development artefacts and generating the final application code from them, we can achieve a higher level of abstraction in development and, thus, achieve an improved understanding of more complex systems. MDD requires all models to completely describe the specific part and property of a system for which they have been constructed. This completeness requirement leads to an increasing size of models used. Therefore, it is often no longer possible to provide and use one single monolithic model of a system. Rather, we need to be able to split complex models into less complex, partial models that can be independently developed, maintained, and studied.

Modern modelling notations used in the context of MDD, such as the Unified Modeling Language (UML) [2], already provide some modularisation support. This support—for example UML's packages, hierarchical classifiers and hierarchical state machines—however, typically follows a dominant decomposition of the system to be developed. For some formal specification techniques—for example, state machines [3] or Petri nets [4]—other decomposition techniques are defined, but these are not typically supported by modern modelling languages. Even worse, in the context of MDD often domain-specific modelling languages (DSMLs) for very specific purposes are developed in one or a number of projects.

This situation has led to a lot of interest in model-modularisation techniques apart from the dominant decomposition of a system. As the model modules studied in this context often cross-cut the dominant decomposition this research has typically been performed under the heading of *aspect-oriented modelling* (AOM) [5, 6]. AOM covers a quite large range of approaches, from those, for example [7], mimicking aspect-oriented programming (AOP) approaches (such as, for example, AspectJ [8]) to those, for example [9, 10], providing very powerful composition techniques for specific purposes and languages using specific properties of modelling languages.

Developing such modularisation techniques and the supporting technology is costly and error prone. At the same time, it needs to be repeated for every new DSML to be enriched with such concepts. Therefore, a generic approach is required that can be applied efficiently to realise different specific model modularisation techniques. We discuss existing approaches that support modularisation techniques for DSMLs in Sect. 7.

To close this gap, in this paper we present such a generic approach based on Invasive Software Composition (ISC) [11] and implemented in our tool Reuseware [12]. ISC is a generic, grey-box composition technique based on rewriting source code. It was formalised in the Reuseware approach [11, 13] to be applicable to arbitrary context-free, textual languages. In this paper, we extend that work to cover graph-structured, possibly graphical languages. This paper is an extension of [14]. In that paper, we enhanced Reuseware by introducing the notion of *fragment queries* to group model or source code fragments. This enabled us to implement the concept of quantification [15], which is at the heart of aspect orientation. Fragment queries and standard Reuseware composition concepts, however, provide a rather crude set of tools for expressing model compositions. This paper extends and refines these simple concepts by adding the notions of ports, port groups and composition steps, greatly reducing the complexity of composition programs and enhancing the flexibility and expressiveness of the language-independent composition technique. Furthermore, we discuss how arbitrary modelling languages can be extended to integrate with our approach and how such an extension can be designed such that existing tooling for the language can still be used.

The rest of this paper is structured as follows: We begin in the following section by giving two motivating examples. Section 3 discusses requirements on a language-independent solution for model modularisation. This is followed by a presentation of our proposed solution in Sect. 4. In Sect. 5, we briefly discuss our tool Reuseware, which implements the concepts presented in this paper. Section 6 shows how our initial examples can be solved by our approach to support our claim of language independence.

2 Motivating Examples

This section introduces two examples that we will use in the paper to explain our approach. The first example is based on UML activity diagrams, outlining a real-world scenario. The second example is based on a toy DSL, explaining modularisation issues in DSLs in an illustrative manner. It should be noted that we not only present examples of concrete models but also of languages in which they are written. Our approach can be applied to any language expressible by a metamodel.

2.1 Business process extension

Business processes can be described by behaviour modelling—for instance using UML activity diagrams. Often, general processes (e.g., a process for ordering goods in a shopping system) can be defined once and specialised for a concrete system with special requirements.

Although UML activity diagrams can be modularised into partitions in single models, reusing and combining parts of activities modelled separately is not well-supported by UML itself. We would like to define general processes with activity diagrams and keep them extensible with specific activities for concrete application use-cases.

As an example, we look at the order processing activity modelled in Fig. 1. The process contains a checking activity (the *CustomerDataCheck* action together with the decision node below) that determines whether certain data (here customer data) is consistent. We want to keep the order processing activity extensible such that additional checks can be inserted in parallel to the customer data check.

To perform the extension, a developer should not need to know anything about the ordering process, but that check activities can be inserted. What this developer needs to know is that a *check activity* has to have one incoming control flow (from the *checkFork* node) and two outgoing flows (to the *checkMerge* and *checkJoin* nodes). With this knowledge, the developer can design additional check activities—for instance, the one from Fig. 2 that determines the customer's credit card liquidity.

Such extensibility can be realised by thinking about models as components. Treating the ordering process model (cf. Fig. 1) as a model component, almost the whole activity should be encapsulated. Only the *checkFork*, *checkMerge*, and *checkJoin* nodes (grey boxes), to which the incoming and outgoing flows of additional checks can connect, should be reflected in the composition interface. Looking at the credit card check (cf. Fig. 2) as a model component, we can again hide the internal activity. We only think of the initial (*InitialNodeCREDIT*) and final nodes (*FINISH* and *CHANCEL*) as *open spots* in the model which need to be manipulated through the composition interface.

Our approach will enable us to look at UML models as model components by utilising UML language specifics to define composition interfaces on UML models. With our language-independent composition tooling we can then easily define and execute a composition of both presented activities resulting in the model shown in Fig. 3, where only parts belonging to the composition interface of the model components (grey boxes) were manipulated.

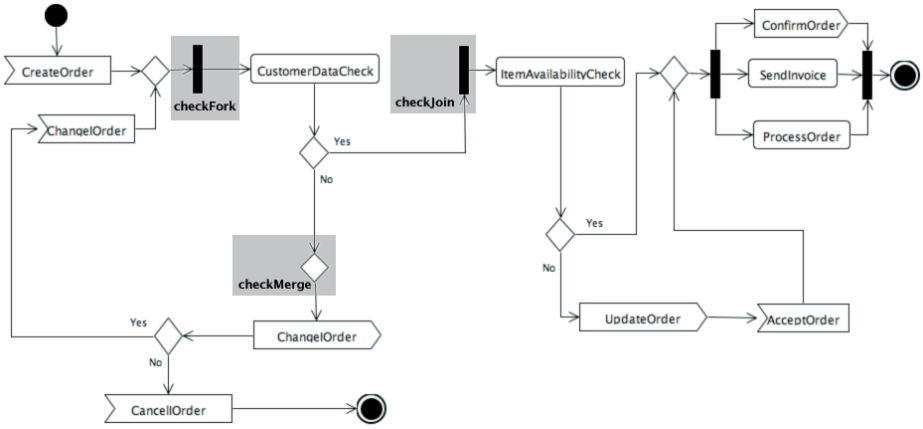

Fig. 1. An activity diagram for the control flow of an order process

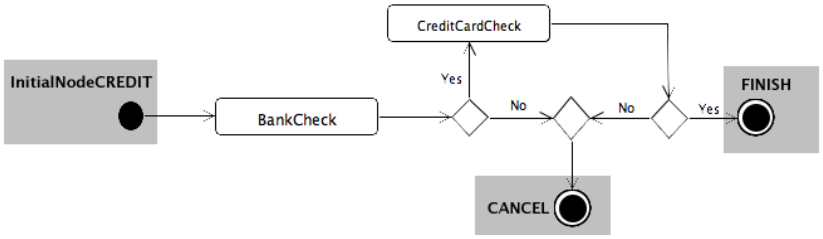

Fig. 2. An activity diagram for credit-card checks that can extend the order process with an additional check.

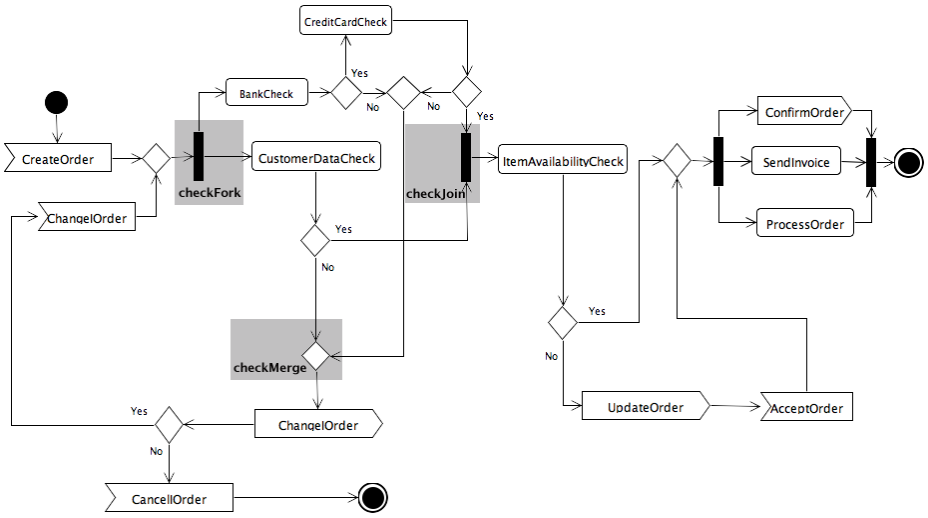

Fig. 3. A composition of the order process and the credit card check activity

2.2 Modular Ship and Cargo Distribution

In this example we utilise the toy DSL *TaiPan*[1] that was created to demonstrate features of the Eclipse Graphical Modeling Framework [16]. Figure 4 shows a model defined in the TaiPan language. The language can be used to model a configuration of an *Aquatory* consisting of *Ports*, *Routes* between ports, and *Ships* that may hold *Items* as cargo. Ships travel on a *Route* and have a *Port* as destination. A special kind of ship is a *Warship* that has the additional ability to execute *EscortOrders* (escorting another ship on its route) or *AttackOrders* (besieging a port).

Let us assume that *Ports*, *Ships*, and *Items* are complex model parts that consist of several model elements and that there are many relations (*Routes*) between ports and many relations (*EscortOrders*) between *Ships*. Then it becomes obvious that certain parts of a model can be reused in other models: the ports on the sea are always the same, while the ships on it can be different. The part that models a certain item can be reused everywhere it represents the cargo of a ship. We identified three different model parts in a TaiPan model which can be individually reused in other TaiPan models.

1. *Port model* (Fig. 5) Here the ports and routes between them are modelled. The number of ports and routes and their names seldom change in this model. However, details of the ports (e.g., its size and capacity) can change over time.
2. *Flotilla model* (Fig. 6) This models a flotilla of ships with their specifics and relations between them. Again, the number of ships and their names do not change so often in one flotilla, while the escort orders between ships might do so often.

[1] Available from: http://wiki.eclipse.org/index.php/GMF_Tutorial#Quick_Start

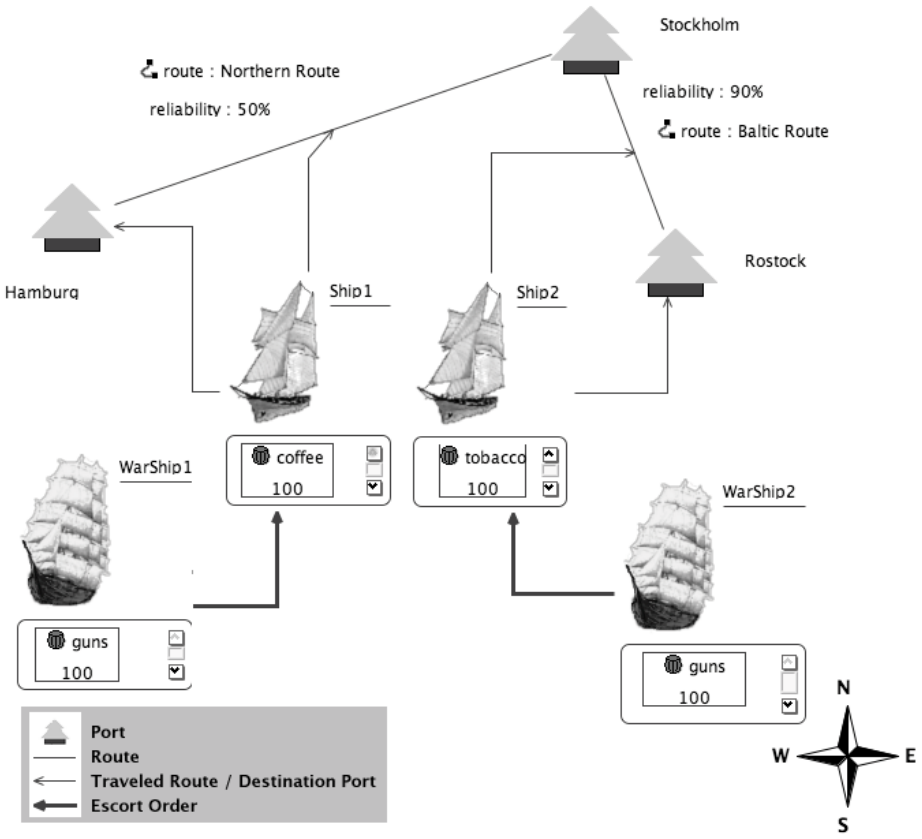

Fig. 4. A TaiPan model

3. *Cargo model* (Fig. 7) Here individual items of cargo are modelled. Assuming that the model of a single item is not simple, this separation makes sense: only one item type needs to be modelled once and can be reused for several ships.

We want to look at these partial models as model components and compose them into a single TaiPan model, like the one in Fig. 4. We identify the following three components:

1. The port model should encapsulate details about the ports and routes, such as the size of the ports. It should offer an interface which allows access to the port and route names, such that they can be assigned to ships.
2. The flotilla model should encapsulate details about the ships and the relations between them (e.g., which war ship escorts which cargo ship). It should offer a composition interface which allows modification of port and route assignments, as well as a composition interface to fill the load of a cargo ship.
3. The cargo model should encapsulate details about the different cargo types. It should offer a composition interface that allows access to extract specific items from the cargo model.

Through such a component-oriented way of thinking about the models, benefits are gained over conventional model integration approaches (like model transformations). Through encapsulation two advantages are gained: First, it ensures that certain elements are not changed during composition which increases the understanding of relations between the components and the composition: The author of a component knows and can control which elements are changed during composition through defining the composition interface together with the component. Second, by knowing the encapsulated details, the component author can change them without risk of breaking the composition. Through the clearly defined interfaces, composition becomes easier. One only has to bother about the interface and not the internals of the models (in contrast to model transformations, where extensive knowledge about the details of the involved models is often required).

Our model composition approach will allow to easily introduce the notion of composition interfaces into the TaiPan language and use our language-independent composition tooling to quickly define compositions of TaiPan model components. Thus, we can extend the language with new features that were not originally supported.

Were we to implement support for the two kinds of model compositions presented above manually into the activity-diagram and the TaiPan language, we would face a daunting task indeed. For each language, we would have to design a modularisation mechanism, manually adjust the language metamodel, and implement the transformations necessary to make it work. Having done so for one language, we would not be able to reuse the effort spent once we move on to another language. Furthermore, whenever the original language changes (for example, when the representation of ships in Taipan is modified to give more details of the internal structure of the ships), we would have to manually redo the implementation of the modularisation mechanism—a sheer maintainance nightmare. In this paper, we will provide generic concepts that can be used to easily implement a variety of composition approaches for arbitrary languages including for the TaiPan and activity-diagram examples.

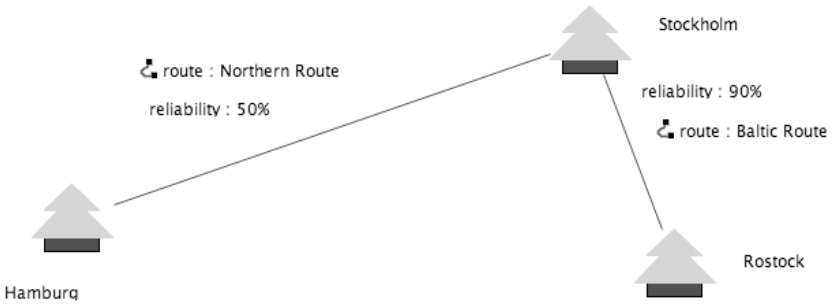

Fig. 5. A port model

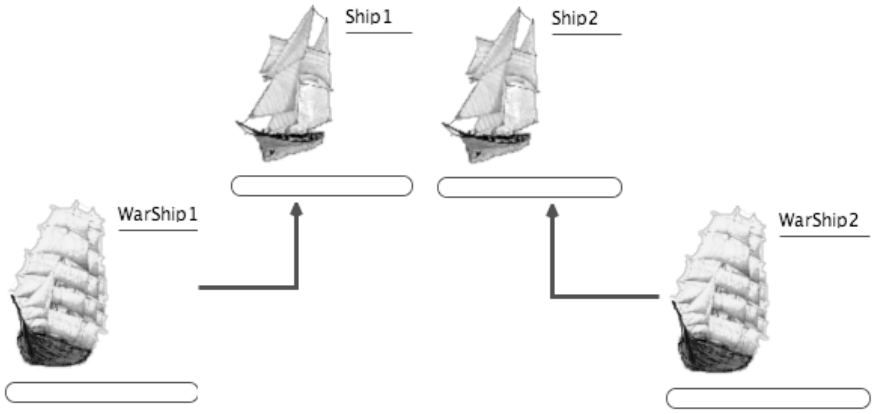

Fig. 6. A flotilla model

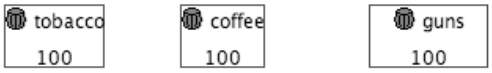

Fig. 7. A cargo model

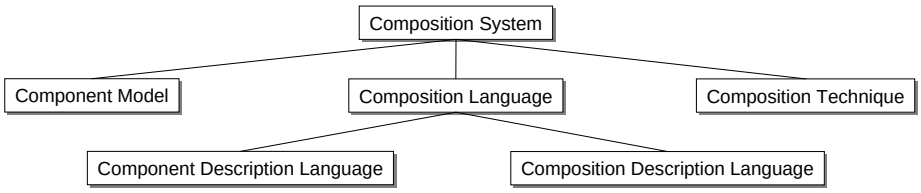

Fig. 8. Terms relevant in describing composition systems

3 Requirements for a Language-Independent Modularisation Technique

We can clearly see the need for generic support for implementing modularisation techniques for arbitrary domain-specific languages. This section discusses the requirements that such a generic support system needs to fulfil. We base our discussion on a classification defined in [11], where a composition system is sub-divided into a *component model,* explaining what components and their interfaces are, a *composition technique,* determining how components can be composed, and, finally, a *composition language,* that allows composition programs to be formulated and concrete compositions to be described. Following this classification—an extended overview of which can be seen in Fig. 8—we will consequently discuss our requirements on the component model, the composition technique and the composition language.

3.1 Requirements on the Component Model

The purpose of a component model in a composition system is to define the units of composition that should be usable for modularising a program or model. This requires that the component model defines the notions of components, composition interfaces, and consistent compositions.

A definition of the term *component* identifies the units of composition. This can range from a notion of binary, pre-compiled and immutable "black-box" components as defined, for example, for Enterprise Java Beans [17] and CORBA Components [18] to freely modifiable pieces of (structured) text ("white-box" components) as is, for example, the case for some hypermedia document components [19]. As, in this paper, we are looking for a composition system that is independent of specific component description languages[2], we, of course, require a component definition that is independent of the specific language used in expressing components. In particular, components should be editable, analysable, and maintainable by tools already available for the language in which they are expressed, while also being recognizable as components to the composition system.

A *composition interface* makes explicit what parts of a component can be accessed during composition; that is what information about the internal structure of a component

[2] A component description language is a language used to write components.

can be used when describing and executing compositions and how, if at all, this internal structure can be adjusted. This is very much related to the different types of components ranging from black-box to white-box. In particular, the composition interface of a component defines whether the component is black-box or white-box or somewhere in-between. For example, the interface of the black-box components of EJB is essentially an operational interface; that is, a list of signatures of operations that can be invoked on the component to interact with it plus a technique for resolving a component name into a component reference. For white-box, structured-text components the interface is defined by the structure of the text and the possibility to freely edit this text. We believe that a completely language-independent composition system must of necessity be more open than a black-box system. Note that systems such as CORBA components are language independent only to some degree in that they still restrict components to those being expressed in programming languages that can be hidden behind an operational interface. We want to be more generic and also include modelling languages and other techniques. At the same time, completely white-box systems give too little control to component authors so that we are requiring a "grey-box" approach; that is, an approach where the structure of components can be inspected and manipulated during composition, but where the component author can control, through explicitly defined composition interfaces, the amount of inspection and manipulation possible. A similar idea has recently been advocated in aspect-oriented programming (AOP) through the concept of explicit pointcut interfaces (XPIs) [20].

Consistent composition refers to conditions that must be fulfilled for two components to be composable in a certain manner. In effect this refines the constraints imposed by composition interfaces, expressing not only what parts of a component's structure can be manipulated, but also how much these parts can be manipulated. We can distinguish syntactic and semantic consistency. For example, the structure of text in the white-box hypermedia systems discussed above provides syntactic consistency by asking that the result of any composition (however much the individual components are modified) must respect the structural constraints of the hypermedia language; that is, the composition result must be syntactically well formed. Semantic consistency requires that semantic constraints induced by one component must not be weakened when the component is used in a composition. For example, for black-box operation components (such as CORBA components) we require that no behaviours that are not acceptable for a component in isolation become acceptable simply by the component being used in a composition. Semantic consistency very much depends on the specific semantics of the component language. As semantics can differ very widely, we will restrict ourselves to syntactic consistency in this paper.

3.2 Requirements on the Composition Language

The composition language of a composition system provides means of expressing *composition programs;* that is, of describing concrete compositions of concrete components. Therefore, it needs to provide syntactic constructs denoting components and their interfaces as well as for denoting individual steps in a composition. As other formal languages, composition languages can be either declarative or imperative in nature.

Because they allow more freedom in actually executing the composition, we prefer declarative composition languages.

Composition languages can be sub-divided into two parts:

1. *Component Description Language:* This is a language used to describe components and their interfaces. It can be used either in addition to the component language or it can be used as an extension of the component language. In any case, because we are looking for a technique independent of the component language, it must provide its constructs with minimum impact on the component language. In particular, if tools exist for analysing, editing, or compiling components, these must not be affected by the component description language.
2. *Composition Description Language:* This language is used to describe compositions of existing components. It should provide generic constructs for referencing components and their externally visible interfaces, and for expressing their composition. When using a composition description language, it should not be necessary to know what language components are expressed in. As a relaxation of this requirement, in this paper, we restrict ourselves to compositions of components that are all written in the same language. In the future it should also be possible to compose components of different languages.

3.3 Requirements on the Composition Technique

The purpose of a composition technique in a composition system is to provide semantics to the composition language. The composition technique defines the basic composition operators that can be used in the composition description language and explains their effects in terms of the composition result. Furthermore, it explains how composition programs are interpreted.

We need a composition technique that can be defined independently of the language in which components are written. We find that, in this case, it is easiest to provide a composition technique based on rewriting of components.

4 Extending Invasive Software Composition for Model Composition

This section presents our solution to the requirements for a language-independent system for model composition described in the previous section.

4.1 A Language-Independent Component Model for Model Composition

As indicated in Sect. 3.1, a component model needs to define what the components and their interfaces are. Additionally, a *generic* component model needs to do so independently of the language used for expressing the individual components. Our component-model definition is based on concepts from ISC [11–13]. Figure 9 gives a graphical overview of the main concepts of our component model. In the following, we will present and discuss these concepts. We will begin by discussing our notion of components—fragments or fragment components—followed by a discussion of the interface of such components.

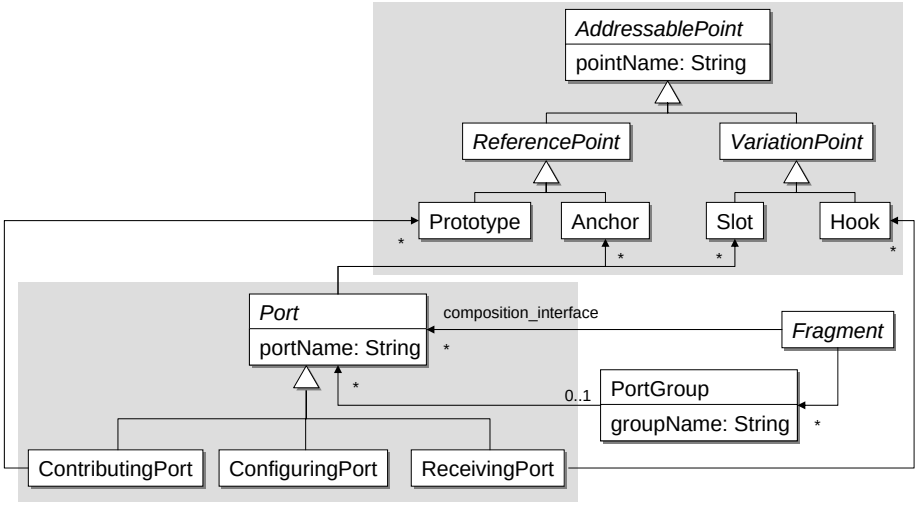

Fig. 9. Component model. These are the concepts available for describing components and their interfaces.

Fragment Components We introduce the concept of a *fragment component*—or *fragment* for short—as our notion of a component. A fragment is a partial expression in some formal language. This underlying language is called the *core language.* A fragment can be partial in two ways:

1. A fragment can be *incomplete.* For textual languages, this means the fragment is derived from a non-terminal other than the start symbol of the core-language grammar. For a graphical language with a metamodel whose instances are not necessarily trees, incompleteness means that the fragment only represents a sub-graph of a valid metamodel instance. In our activity-diagram example, any combination of activities and transitions would be an incomplete fragment unless they were also embedded in an activity diagram and had at least one start and one stop activity.
2. A fragment can be *generic.* This means that some part of the fragment (whether incomplete or not) is (intentionally) missing. For textual core languages, we can potentially leave open any non-terminal defined in the core-language grammar. For graphical languages, we can make almost any metamodel class generic.

To express genericity of a fragment, we introduce *variation points.* These are elements within a fragment that can be used as place-holders for other fragments leaving some part of a fragment unspecified. Further, we also need to be able to address fragments or parts thereof. To this end, we introduce fragment *reference points.* Reference points address fragments or sub-fragments and give them a name so that they can be used in compositions. Hence, in general a composition occurs when a variation point in one fragment is replaced by another fragment addressed through some reference point. Thus, variation points can be likened to formal parameters of procedures in imperative

programming, while reference points are similar to actual parameters for a composition. Taking this analogy one step further, we need to distinguish between two cases that are very similar to the passing of parameters 'by value' or 'by reference' that we often find in imperative programming: We introduce the following two pairs of variation and reference points:

1. *Hook–Prototype* This corresponds to the intuitive notion of binding a fragment to a variation point: The fragment, addressed by a *prototype* reference point, is copied and then replaces the *hook* variation point.
2. *Slot–Anchor* In analogy to the concept of passing by reference, no new copies of any fragment are created when an *anchor* is bound to a *slot.* Instead, references to the *slot* are replaced by references to the *anchor*.

It should be noted that when we say 'replace' above, this does not necessarily imply that the variation point is removed from its fragment. Whether a variation point is removed after a composition step depends only on the maximum multiplicity of the references pointing at it. A variation point is only removed after its maximum multiplicity has been reached. Therefore, variation points can be bound multiple times as long as their maximum multiplicity allows it.

Composition Interfaces A fragment is addressed during composition through its *composition interface*. Before we describe the details of fragment composition interfaces, we have to be aware that these interfaces are seen from two different perspectives.

1. *Fragment developer viewpoint* The fragment developers (persons who write fragments) look at the interfaces from "inside" of the fragment components. They define the interfaces and link them to the fragment's contents.
2. *Fragment user viewpoint* The fragment user (persons who reuse fragments defined earlier) look at the composition interfaces from the "outside". They address the fragments in composition programs, without looking at the internal details of the fragments.

A fragment composition interface is a collection of *port*s. Fragment developers define the ports and assign them unique names. Furthermore, they link each port to a set of variation and reference points in the fragment. Fragment users can then write composition programs in which they describe a composition by linking ports of different fragments.

In addition to the grouping of addressable points into ports, ports can be organised in port groups. A port group indicates to the fragment user that a set of ports should be addressed together in a composition program. Fragment developers can decide how they apply the two abilities of grouping addressable points according to the task at hand. The more addressable points are grouped into a port, the more abstract the interface becomes: details are hidden from the fragment user. If the grouping is shifted into port groups with several ports, the interface becomes less abstract and more responsibility is transferred to the fragment developer, but also more flexibility.

The two important properties of a fragment's composition interface are that it is *quantifying* and *typed*. The former supports the high abstraction of interfaces, the latter the consistent composition:

- *Quantifying* refers to the fact that a port collects a set of variation and reference points that are handled together during composition. The linkage between ports and addressable points can be expressed by explicitly assigning points to a port but also by giving a quantifying query expression over the set of addressable points in a fragment. This grouping is independent of any structure dictated by the fragment's content.
- The *typing* of a port is determined by the typing of its associated addressable points. These are typed in two dimensions: First, as discussed above, we distinguish four different types of addressable points; namely hooks, prototypes, slots, and anchors. Since these types are given by the generic component model we call this the *language-independent type* of the addressable point. Second, each variation point represents a specific metaclass and each reference point references an instance of a specific metaclass; hence, all addressable points are also typed by the metaclass they are associated with. Such a metaclass is given by a concrete language and is therefore the *language-dependent type* of the addressable point. The language-independent (language-dependent) type of a port is the set of all the language-independent (language-dependent) types of its associated addressable points.

Ports are categorised into three different categories by their language-independent types:

1. *Configuring Ports* contain only slots and anchors. They are used to configure a fragment (by re-routing references from slots to anchors) and not to extend it with additional model elements. Notice that using only configuration ports in a composition description makes no sense, as they can only be used to configure the composition result.
2. *Contributing Ports* contain prototypes (but no hooks). They offer content (i.e., addition model elements) as extension to a fragment. They may also contain slots or anchors for configuration.
3. *Receiving Ports* contain hooks (but no prototypes). They allow a fragment to be extended with additional model elements. They may also contain slots or anchors for configuration.

The distinction made above has conceptual and technical reasons. Conceptually, the fragment user can easily recognise whether a port contributes new elements or expects a contribution of elements without looking at the addressable points behind the port. Since slots and anchors cannot be used to add new model elements to fragments, they do not influence the contributing/receiving character of a port. The technical reason for the distinction is that our composition technique needs to know, at each point of the composition process between two fragments, which fragment is contributing and which one is receiving model elements. The reasons for this will be explained in the context of our composition technique in Sect. 4.3.

Configuring ports should be grouped with receiving or contributing ports into port groups. The reason is that configuration (i.e., cross-referencing) makes no sense before fragments are actually integrated. Thus, addressing configuring ports independently in a composition program is not useful. Furthermore, the more abstract the interface, the

less often configuring ports are required—the slots and anchors are contained directly in the contributing and receiving ports.

4.2 A Generic Composition Language for Model Composition

As stated in Sect. 3.2 the composition language can be split into the *Component Description Language* and the *Composition Description Language*. A component description language is used by the *fragment developer* to describe fragments and their interfaces. A composition description language is used by the *fragment user* to define fragment composition programs. Thus, both languages inherit concepts of the component model defined in the last section.

In our setting, the component description language can be any existing (or just developed) modelling language (a *core language*), extended to support the definition of fragment composition interfaces. We call such an extended language a *reuse language*. If this extension is done following the same formalism for any language the composition description language can be defined independently of any specific component description language. This is because the composition description language only relies on the extended part of the reuse language, which is based on the concepts of the language-independent fragment component model.

This section first describes the possibilities and the formalism to extend a core language by extending its metamodel to make it usable as a component description language. It then describes the (language-independent) composition description language.

Component Description Language To turn an arbitrary core language into a reuse language, which is usable as a component description language in our approach, we need to perform a language extension. Two methods are applicable for this:

1. *Extending the Core Language Metamodel.* This method can be used to inject constructs for variation and reference point definitions into the core language. This enables the fragment author to declare the interface of each fragment individually by defining variation points and marking model elements as reference points. A drawback, in some cases, is that tools which are already implemented on basis of the original metamodel might break.[3] For example, models containing variation points may not be accepted by tooling based on the original metamodel. Alternatively such new metamodel constructs may be ignored by the tooling, or in the worst case even removed from the model.
2. *Defining OCL Expressions over the Component Language Metamodel.* In this approach, we define how the composition interface is extracted from fragments defined in the core language. This approach can be used for two purposes: First, it avoids the need for language extension because original language concepts can be selected to represent addressable points (e.g., through naming conventions). Second, it can be used to define a default interface for fragments which does not require explicit declaration by the fragment developer. Note that structural queries

[3] As we will explain below, the extension is restricted and does not harm existing language constructs. If the language tools are build openly and allow for extension, the language extension approach may still be feasible.

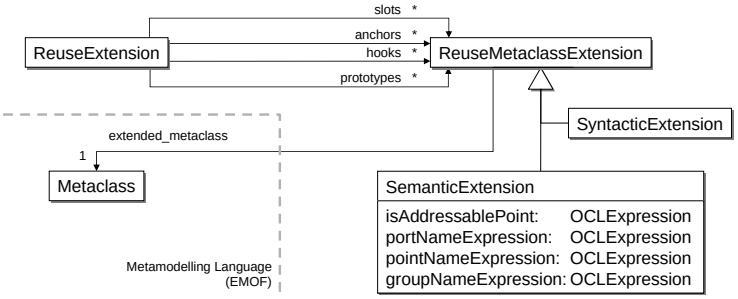

Fig. 10. Metamodel of the reuse-extension language.

over metamodel instances, as enabled by OCL, are sufficient because we only consider static compositions of development artefacts.

Effectively, both approaches extend the core language. The first one syntactically and semantically—by introducing new meaningful constructs into the core language. The second one only semantically—by giving additional meaning to existing constructs. In the following, we will refer to the first approach as a *syntactic language extension* and to the second approach as a *semantic language extension*. Both concepts can also be combined when extending a metamodel.

We call all such metamodel extensions *reuse extensions*. Figure 10 shows the metamodel of a small language we use to describe reuse extensions. Reuse extensions can be performed to provide constructs for expressing each of the four types of addressable points. Hence, a `ReuseExtension` collects four sets of `ReuseMetaclassExtensions`, one for each type of addressable point (`hook`, `prototype`, `slot`, and `anchor`). Each `ReuseMetaclassExtension` defines the extension on the basis of a metaclass of the core metamodel.[4]

For each type of metamodel extension, there is a specific subclass of `ReuseMetaclassExtension`. Semantic language extensions are captured by the `SemanticExtension` metaclass. The specific extension is defined by the following four attributes, of which two are mandatory and two optional:

1. *isAddressablePoint (required)* This is a constraint expressed on instances of the core metaclass. It results in true for those instances which should be interpreted as addressable points. Note that the language-independent type of the addressable point is already determined by the association from `ReuseExtension`.
2. *portNameExpression (required)* If the *isAddressablePoint* constraint holds, this expression is used to extract the name of the port this addressable point belongs to.
3. *pointNameExpression (optional)* If the addressable point has a specific name, this expression is used to extract it.

[4] Throughout the paper we refer to the core metamodel as the metamodel representing the core language

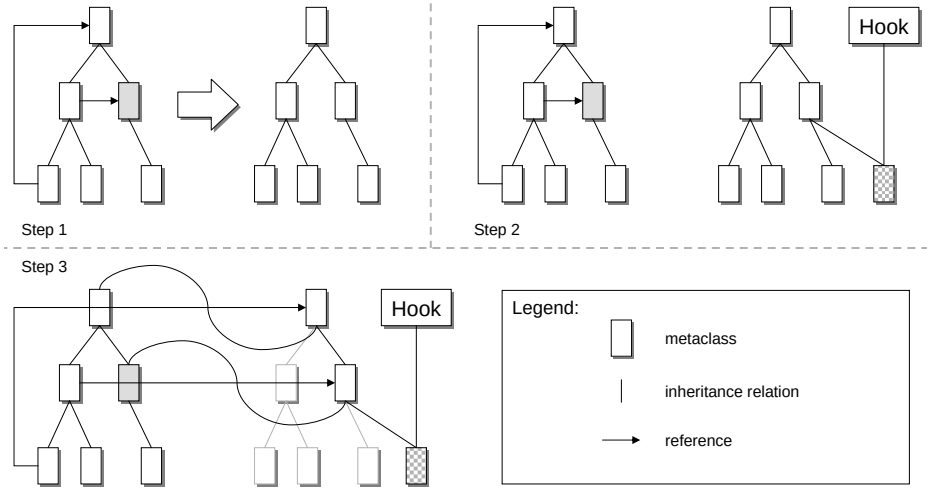

Fig. 11. Steps of the language extension algorithm.

4. *groupNameExpression (optional)* If the addressable point and its port should belong to a specific port group, this expression is used to extract the name of the group.

Syntactic language extensions are more involved, because we also need to modify the metamodel of the core language itself. Specifically, we need to introduce specific metaclasses to be used for expressing addressable points. Such addressable-point metamodel instances should be substitutable for their corresponding core metaclass everywhere in the extended language. For example, in the TaiPan example, we want to be able to place a cargo hook wherever we would be able to place a specific piece of cargo. At the same time, however, addressable points should not share any features specific to core language elements. For example, for a cargo hook we should not need to express a cargo name and amount. Typically, in metamodels, we use inheritance to express substitutability. However, we also use inheritance to share features between language elements. Therefore, simply inserting new elements into the inheritance structure of the core language would not fit our requirements. Instead, we need to use a slightly more involved algorithm. Figure 11 illustrates the steps of this algorithm using a symbolic example. In the following, we explain each step in turn:

1. *Type hierarchy extraction* To separate the use of inheritance for substitutability from its use for feature sharing, we duplicate the hierarchy of metamodel elements of the core metamodel. For each core metaclass, we create a new abstract metaclass and call it the *type metaclass* of the core metaclass. For each inheritance relationship between two core metaclasses, an inheritance relationship between the corresponding type metaclasses is introduced (This is represented in Fig. 11 by simple lines connecting two boxes. The upper box represents the super class of the lower box). In Step 1 of Fig. 11, we show a symbolic core metamodel on the left and the derived

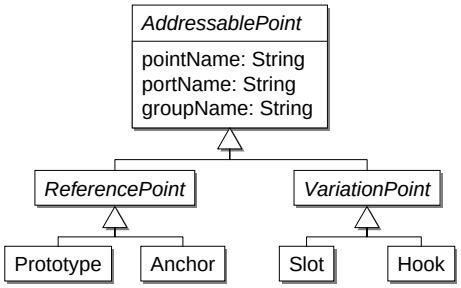

Fig. 12. Common metamodel that is integrated into reuse languages with syntactical extensions.

corresponding type hierarchy on the right-hand side. Notice that the type hierarchy contains only classes and inheritance relations, but no references or other features within these classes.

2. *Addressable point introduction* For each `ReuseMetaclassExtension` we introduce an *addressable point metaclass* into the type hierarchy. The addressable point metaclass inherits from the type metaclass of the core metaclass that is extended. In the figure, we show how one core metaclass (the grey one) is extended. To this end, we introduce a new class in the type hierarchy, represented by a hashed box in the figure.

   Additionally, we integrate into each reuse language the metamodel from Fig. 12 introducing the different types of addressable points available. Depending on the type of addressable point that is to be introduced for the core metaclass, the new class in the type hierarchy also inherits from the corresponding class from Fig. 12. For our TaiPan example from above, this would be the `Hook` metaclass. For prototype and anchor metaclasses, additionally, a reference (*content*) to the core metaclass is added to hold the actual model element that is referenced.

3. *Reference redirection* Finally, the two hierarchies of metaclasses need to be integrated, so that the new metaclasses from the type hierarchy are used instead of the original core metaclasses. To this end, every reference to an extended core metaclass (or to a superclass of an extended core metaclass) is redirected to the corresponding type metaclass. Each core metaclass to which references existed that have been redirected is made a subclass of its type metaclass. As a result, instances of the extended core metaclass are substitutable by addressable points.

After the metamodel extension for a specific core language was performed, fragments can be written and their interfaces can be defined in the extended language. A composition system can extract interfaces from the fragments to make them explicit by analysing the model elements of the fragment. All elements that are either instances of a subclass of a metaclass from Fig. 12 or on which an *isAddressablePoint* constraint holds, define the composition interface of the fragment.

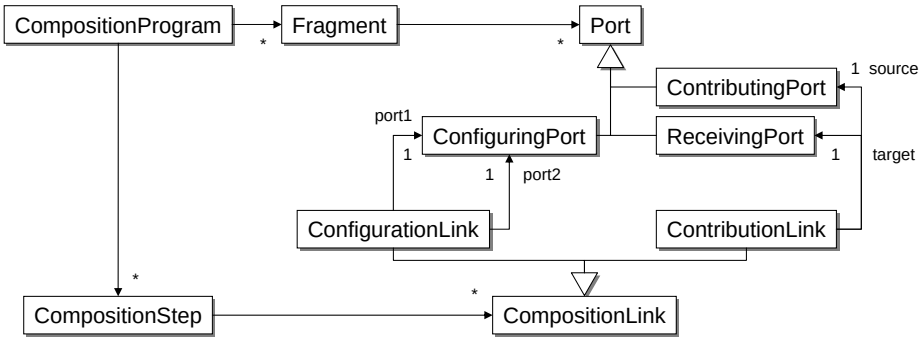

Fig. 13. Metamodel of the Composition Description Language.

Composition Description Language Figure 13 depicts the concepts of our Composition Description Language. A *CompositionProgram* consists of several *Fragment*s and their composition interface that is represented by *Port*s. A Port can either be a *ConfiguringPort*, a *ContributingPort*, or a *ReceivingPort*. Composition is realised through different *CompositionStep*s where each CompositionStep consists of *CompositionLink*s between two ConfiguringPorts (a *ConfigurationLink*) or one ContributingPort and one ReceivingPort (a *ContributionLink*).

We defined a graphical syntax for expressing compositions of fragments that includes concepts for representing fragments, their composition ports and composition links between those. Additionally, means for defining composition steps are provided. This syntax is supported by a graphical editor that is built on top of the Eclipse Platform [21] and the Graphical Modeling Framework (GMF) [16]. An example of a fragment composition program written in our editor is shown in Fig. 14. The pallette on the right offers means to create composition links, composition steps, participations of composition links in composition steps and so-called fragment queries, which will be examined later.

In our editor, a fragment is represented by a rectangle that has its composition ports attached as circles, where contributing ports are depicted as black circles, receiving ports as white circles and configuring ports as white circles with a dashed line. The composition ports are automatically extracted from the fragment when the fragment is dropped onto the editor canvas. Composition links are represented as lines between composition ports. A configuring link is shown as a dashed line and a contributing link, which defines a direction of composition from the contributing to the receiving port, as an arrow visualising this direction. Within the fragment, a composition port group—represented by a small rectangle—is connected to its participating composition ports. To allow for grouping of composition links to steps, syntax for defining composition steps is necessary. A composition step is represented by an ellipse that references all associated composition links by dashed lines.

Sometimes, multiple fragments need to be composed in essentially similar ways. In our activity-diagram example, there may be more than one check activity fragment

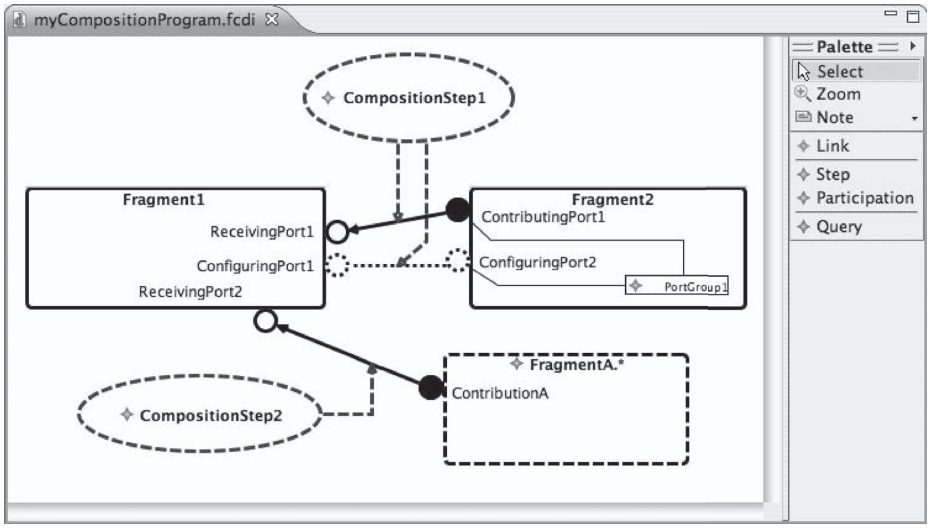

Fig. 14. Fragment Composition Editor.

to be woven into the core activity. Although we could express each such composition individually, this would require a lot of duplication in the composition code. To avoid such duplication, we introduce the syntactical concept of *fragment queries.* An example of a fragment query is the dashed box named `FragmentA.*` in Fig. 14. Note that this is purely a concept of the composition language as every composition including a fragment query can be transformed into a set of compositions without queries.

Fragment queries group a set of fragments and treat the complete group like a single fragment. Figure 15 shows an overview of the essential syntactical concepts involved in expressing fragment queries. It can be seen that fragment queries can be nested hierarchically inside each other. An elemental fragment as defined in the component model is represented by the `PhysicalFragment` class in the figure.

Fragment queries can be defined in essentially two ways: a) by enumerating the fragments to be encompassed by the query, and b) by providing an expression describing the set of fragments to be included. Both approaches are supported by our composition language using regular expressions as query expressions. Fragment queries add an additional level of quantification to the composition. It is interesting to see that this enables us to distinguish quantification introduced by fragment developers (using groupings between variation and reference points) and fragment users (using fragment queries). This adds a new level of control not supported by typical AOP/AOM realisations in the literature.

To be able to view a fragment query as a fragment again, we need to define how the fragment query's composition interface is determined. Basically, the composition interface of a fragment query reflects the interfaces of its element fragments. However, variation and reference points of the same name and type that occur in different element fragments are merged into one variation or reference point for the fragment query and

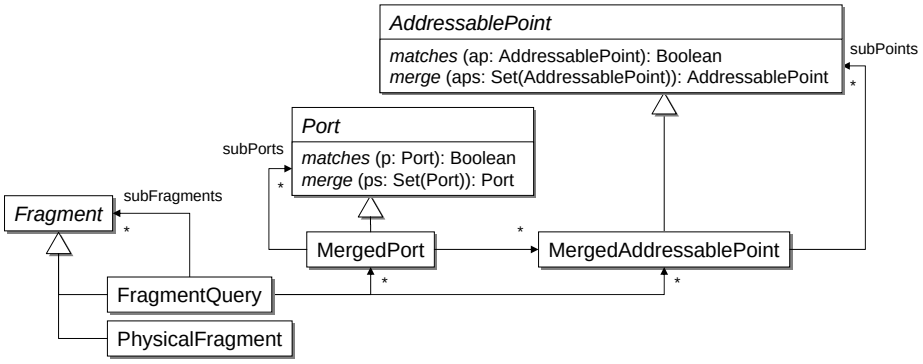

Fig. 15. Essential concepts of fragment queries.

```
1  context AddressablePoint::matches (ap: AddressablePoint) : Boolean
2  post: result = (typeMatch (ap)) and
3                 (pointName = ap.pointName) and
4                 (apElements.type->forAll (t1 |
5                       ap.apElements.type->forAll (c2 | c2 = c1)
6                    ))
7
8  context AddressablePoint::typeMatch (ap: AddressablePoint) : Boolean
9  post: result = (self.oclIsKindOf (Prototype) and ap.oclIsKindOf (Prototype)) or
10                (self.oclIsKindOf (Anchor)    and ap.oclIsKindOf (Anchor)) or
11                (self.oclIsKindOf (Slot)      and ap.oclIsKindOf (Slot)) or
12                (self.oclIsKindOf (Hook)      and ap.oclIsKindOf (Hook)) or
13                (self.oclIsKindOf (MergedAddressablePoint) and
14                 subPoints->forAll(ape | ap.typeMatch(ape))) or
15                (ap.oclIsKindOf (MergedAddressablePoint) and
16                 ap.elements->forAll(ape | self.typeMatch(ape)))
```

Listing 1. Definition of matching between addressable points. Two addressable points should be merged if they have the same name, are of the same type, and the type of the elements they are associated with is the same

similarly for ports and port groups. To define precisely how the composition interface of a fragment query is derived from the composition interfaces of its element fragments, we need to introduce a few helper concepts. To do so, in the following, we use the Object Constraint Language (OCL) [22] to formally express additional concepts for our metamodel classes. The OCL constraints we will show in Listings 1 to 5, are hence an integral part of the metamodel of our composition description language.

First, we need to define which variation or reference points should be merged. To this end, in Fig. 15, we have introduced the operation `matches()` on addressable points, that returns `true` if two addressable points are sufficiently equal to be merged into one. Listing 1 shows the definition of `matches()`. Note that these functions make use of a—previously unshown—association from addressable points to elements of a fragment. This association is accessed through its association end `apElements`.

```
context AddressablePoint::merge (aps: Set(AddressablePoint)): AddressablePoint
pre:  aps->forAll (ap | self.matches (ap))
post: (typeMatch (result)) and
      (result.oclIsKindOf (MergedAddressablePoint)) and
      (result.pointName = pointName) and
      (result.subPoints = aps->collect (ap |
           if (ap.oclIsKindOf(MergedAddressablePoint)) then
             ap.subPoints
           else
             ap
           endif
         )->union (
           if (self.oclIsKindOf (MergedAddressablePoint)) then
             self.subPoints
           else
             self
           endif
         )) and
      (result.apElements = result.subPoints.apElements)
```

Listing 2. Merge operation defined for addressable points

```
context Port::matches (p: Port) : Boolean
post: result = (portName = p.portName) and
               (addressablePoint->size() = p.addressablePoint->size()) and
               (addressablePoint->forAll(ap | p.addressablePoint
                                 ->exists (ap2 | ap.match(ap2)))) and
               (p.addressablePoint->forAll(ap | addressablePoint
                                   ->exists (ap2 | ap.match(ap2))))
```

Listing 3. Definition of matching between ports. Two ports should be merged if they have the same name and group matching addressable points

The merging of matching addressable points is represented by another operation: `merge()`. Listing 2 shows its specification. This operation always creates a `MergedAddressablePoint` collecting all the merged addressable points. Note that anchors can only be merged with anchors, slots with slots, hooks with hooks, and prototypes with prototypes. The introduction of a `MergedAddressablePoint` allows composition interfaces of fragment queries to be viewed in two ways:

1. From the outside, the addressable points in the composition interface of a fragment query look just like any other addressable point. In particular, the elements they refer to can be accessed through the `apElements` association end.
2. The composition system can further inspect merged addressable points and identify for each sub-point the fragment it came from and the elements it refers to. This will be used in describing the composition technique later in this section.

Based on these definitions of how to merge addressable points, we can now define how ports are to be merged: We begin, again, by defining which ports may be merged. We do so in the `matches()` operation of `Port`. Its definition can be seen in Listing 3.

Merging ports is done by merging all matching addressable points in the ports and creating a new `MergedPort` of the same name and with all the merged addressable points inside it. We refrain from expressing this in OCL as it is quite straight forward.

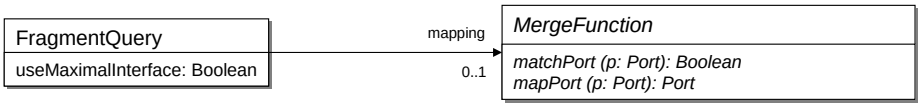

Fig. 16. Metamodel of merge functions

There are various ways of defining a fragment query's composition interface from the composition interfaces of its element fragments:

1. *Maximal interface.* Intuitively, the maximal interface is the union of all composition interfaces of all element fragments where matching ports have been merged as defined by the `merge()` operation.
2. *Minimal interface.* The minimal interface contains only those ports that exist in *all* element fragments. Merging applies as for the maximal interface.
3. *Merging function.* Such a merging function allows to merge ports fulfilling some condition into one port, possibly with a new name. This approach can be combined freely with the two above.

Figure 16 shows the additional metamodel elements required for supporting merge functions and minimal vs. maximal fragment-query interfaces. For each fragment query, we can provide one additional merge function, which will handle all port merges for this query. The merge function provides two query operations: `matchPort` is used to determine if a given port is to be subject to treatment by the merge function. `mapPort` is used to identify the port that is the result of applying the merge function.

Listing 4 shows how the composition interface of a fragment query is derived from the composition interfaces of its element fragments. Lines 4–19 produce the maximal interface, with Lines 5–10 taking into account an optional merge function. The remainder from Line 20 restricts the interface to the minimal interface if required. Finally, we need to ensure that merged ports in a fragment query respect all the rules imposed by any port groups in the participating fragments. This is shown formally in Listing 5.

Fragment queries expressed in our prototypical graphical composition description language will always use the minimal composition interface. Additionally, a merging function can be provided, using regular expressions over port names to express the `matchPort` operation. `mapPort` is implemented implicitly by creating a port with a generated name. Fragment queries are represented by a rectangle with dashed lines (cf. Fig. 14). Accompanying query expressions can be edited via a properties view in the editor.

4.3 A Language-Independent Composition Technique for Model Composition

Once a composition program is defined over a set of fragments, it can be executed, merging the involved fragments into bigger fragments or complete models. In this process, each composition step is executed individually, transforming the fragments involved. We will first look at the overall processing of composition programs and steps

```
context FragmentQuery
inv compositionInterface =
    elementFragments.compositionInterface
      ->iterate (p: Port; cmpIntf: Set(Port) = Set{} |
        if (mapping->notEmpty() and mapping.matchPort (p)) then
          -- merge using merging function
          cmpIntf->reject (p1 | p1.matches (mapping.mapPort (p)))
                 ->including (p.merge (
                      cmpIntf->select (p1 | p1.matches (mapping.mapPort (p))))
                      .rename(mapping.mapPort (p).name))
        else
        if (cmpIntf->exists (p1 | p.matches (p1))) then
          -- merge implicitly matching ports
          cmpIntf->excluding (p1 | p.matches (p1))
                 ->including (p.merge (cmpIntf->select (p1 | p.matches (p1))))
        else
          cmpIntf->including (p)
        endif
       )
      ->reject (p: Port |
          (not useMaximalInterface) and
          elementFragments->exists (f | not f.compositionInterface->exists (
                  p1 | p.matches (p1) or
                       (mapping->notEmpty() and mapping.matchPort (p1) and
                        p.matches (mapping.mapPort (p1)))
                )
          )
        )
```

Listing 4. Composition interface of a fragment query.

```
context FragmentQuery
inv: subFragments.portGroup->forAll (pg |
         self.portGroup->exists (pg2 |
             pg2.port.oclAsType (MergedPort).subPorts->containsAll (pg.port)
           )
       )
```

Listing 5. Port merges must additionally maintain any port groups set up on any contained fragments

and then look into the details of how fragments are merged. The processing of a composition is sketched in Figure 17 to support the explanations below. At the end we will shortly discuss the interpretation of fragment queries.

Executing composition steps and programs A composition program consists of composition steps; each of them is executed individually. Before this can be done, the execution order of the steps has to be determined. This order is controlled by the contribution links. Such a link gives a certain role to the two fragments involved: one fragment (defining the receiving port of the link) is the *receiving fragment*, the other (defining the contributing port of the link) is the *contributing fragment*. Receiving fragments can be compared to *cores* and contributing fragments to *aspects* in AOP. This means that we can weave in a contributing fragment into several receiving fragments. In each composition step the role of each fragment (involved through one or several links) has to

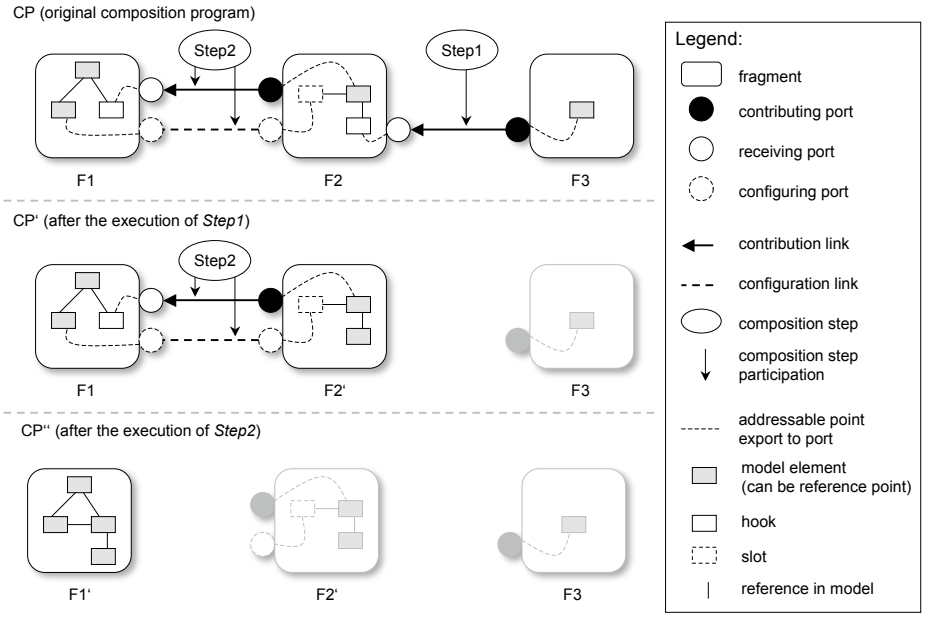

Fig. 17. Stepwise processing of a composition program.

be clear. Otherwise, the composition step is invalid. The following restrictions for a composition step can be derived from this:

1. The involvement of a fragment in a composition step has to be defined through at least one contribution link. The other way around: A fragment cannot be involved in a composition step through configuration links only. If it would be, it would neither be receiving nor contributing.
2. If a fragment is involved in a composition step through more than one composition link, all these links have to have the same direction, because a fragment cannot be contributing and receiving within one composition step.

The composition program CP in Figure 17 defines a composition of the three fragments $F1$, $F2$ and $F3$ by declaring two composition steps, $Step1$ and $Step2$. $F2$ is involved in both steps, because both have links to ports of $F2$. In the context of $Step1$, $F2$ is a receiving fragment while $F3$ is a contributing fragment (and $F1$ is not involved). In the context of $Step2$, however, $F2$ is a contributing fragment while $F1$ is a receiving fragment (and $F3$ is not involved).

Executing a composition step means that all contributing fragments are integrated into the receiving fragments—resulting in a new set of *composed fragments* of similar size as the set of receiving fragments. During this process, the contributing fragments are not directly integrated, but a *fragment copy* of them. Thus, the original contributing fragments remain available as contributions for other steps executed at a later point.

When a composition step was successfully executed it is removed from the program and the receiving fragments of the step are replaced by the composed fragments. Thus, the processing of the whole composition program is an iterative process in which eventually all steps have been executed and the set of composed fragments of the last executed step is the result set.

When $Step1$ of CP (cf. Figure 17) is executed the result can be visualised as a modified composition program CP': The content of $F3$ (the only contributing fragment) is copied and integrated into $F2$ (the only receiving fragment) leading to $F2'$. $F2$ is replaced by $F2'$ and $Step1$ is removed.

As we have seen, only receiving fragments are modified and contributing ones remain unchanged by the execution of a step. This property determines the execution order of steps. The next step to execute is always one where all contributing fragments are not receiving fragments of any other remaining step in the composition program. Thus, they can be safely copied because further modification cannot occur. Note that—because the execution of a step modifies its receiving fragments and removes the step from the program—fragments that were receiving at the beginning of the composition process lose this property at some point.

In the composition program CP of Figure 17, $Step1$ can be executed, because its only contributing fragment $F3$ is never a receiving fragment in the context of any step of CP. $Step2$ on the other hand cannot be executed, because its only contributing fragment $F2$ is a receiving fragment in the context of another step ($Step1$). In CP' however, where $Step1$ has been removed, $Step2$ can be executed resulting in CP''. $F1'$ in CP'' is the resulting model of the complete composition.

It is obvious that invalid composition programs can be defined where no step fulfils the required property—which basically means that they define cyclic dependencies between the fragments. It might also be that such a situation is reached during the iterative execution process if the program is not analysed beforehand. We believe that such invalid programs are counter intuitive and will seldom occur in practice. Should they occur, however, invalid programs can be detected by our tool.

Matching addressable points and merging fragments Until now we have described the general process of executing composition steps. What we have not discussed yet is how receiving and contributing fragment are merged concretely when a step is executed.

A merging is done per composition link and involves only the addressable points behind the two ports the link connects. The first thing that has to be done, before the actual merge, is to determine which variation point of a port can be replaced by which reference point of the other port. The first requirement is that anchors can only replace slots and prototypes can only replace hooks (this is the language-independent typing). The second necessary requirement for such a replacement is that the language-dependent types (i.e., the metaclasses for which the addressable points were introduced) of both points *match*. This means the language-dependent type of a reference point has to be the same as, or a subclass of, the language-dependent type of variation point. This ensures that the composed model, where the reference points replaced the variation points, is still a syntactically valid instance of the metamodel of the used component description language.

If for each prototype involved in a composition step a hook with a matching type can be found, the composition is executable. This is necessary to physically integrate the fragments. If some hooks, anchors, or slots are not addressed this is in general no problem. It is also possible to address the same hook or slot multiple times if the multiplicity of the affected references allow it. Sometimes also different matches are possible. In such non-deterministic situations the composition step is not valid.

To increase determinism, further matching strategies for addressable points are needed. We use the naming of addressable points for this purpose: if a reference point could be bound to different variation points, the names of the points are taken into account. This was sufficient for the examples we inspected so far. However, with respect to reuse, the names of the points might not match directly, additional matching strategies—for example based on regular expressions—could be used.

For some compositions it might well be required that not only the binding of prototypes but also of other kinds of addressable points should be enforced when two ports are linked. These questions, however, very much depend on the concrete language and modularisation approach at hand. Therefore, further experiences are needed to learn about the problems and requirements.

Once all pairs of hook–prototype and slot–anchor bindings are determined, the merging of fragments is simple. For a hook–prototype pair, the *containing reference*[5] to the hook is re-routed to the prototype or, in cases where the prototype concept was introduced by a syntactic metaclass extension, to the element referenced in the *content* reference of the prototypes (cf. Sect. 4.2). This effectively adds content to the fragment that contains the hook. A slot–anchor pair is resolved by taking all *non-containment references* to the slot and re-routing them to the anchor (or to the element referenced in its *content* reference). It should be noted that re-routing references means that a hook or slot can stay referenced for further addressing, if the multiplicities of all affected references allow that.

Note that in the composition illustrated in Figure 17, each port has only one addressable point. Thus the matching is straightforward. It is assumed in the figure that the language-dependent types match in all three cases. When $Step1$ and $Step2$ are executed, variation points are replaced with reference points.

Our implementation of the composition technique also enforces that all ports addressed on a contributing fragment in one composition step must be grouped into one port group. While this is redundant for the execution of the composition—the grouping can be assumed by the fact that the ports are addressed together in one step—it is essential information for the developer. If a fragment comes with a large composition interface where different ports should be addressed in different composition steps, the developer would have little chance to know which configuring ports are to be addressed together with which contributing ports.

Executing fragment queries As mentioned, fragment queries are solely constructs of the composition definition language to ease composition program development. In the

[5] A containing reference is a reference that holds the actual definition of an element. Each element in a model (with the exception of one root node) has exactly one. Which references are containing is defined in the metamodel.

general composition program execution process, fragment queries are treated as usual fragments (i.e., they have a receiving or contributing character).

If a composition step is to be executed that involves a fragment query the query is expanded. One can think of this process as drawing each fragment that is grouped by the query individually into the composition program and then defining composition steps for all possible compositions of the single fragments. If we define, for instance, a composition step that involves one fragment query, which groups k fragments, as receiving "fragment" and another fragment query, which groups l fragments, as contributing "fragment" the transformation would produce $l * k$ composition steps. Each of these steps composes one fragment from the first query with one from the second query. Then all the steps are executed.

If receiving fragments belonged to a fragment query, the corresponding composed fragments are again grouped into a fragment query that replaces the original query in the composition program.

In this section we have demonstrated our language-independent model composition approach in detail. In the next section we explain how the approach and its concepts have been implemented in a tool that can be used to solve the problems shown in the introductory examples: the Reuseware Composition Framework.

5 Tooling: The Reuseware Composition Framework

The last section demonstrated our language-independent model composition approach and its concepts. These concepts were implemented in the Reuseware Composition Framework, available from [12]. The implementation is based on the Eclipse Modelling Framework (EMF) [23]. In this section we briefly describe the architecture of the tool, which we use in the next section to realise concrete solutions for the problems shown in the introductory examples.

The tooling is split into a developer kit, which is used to instantiate the framework for concrete languages, and runtime tooling, which is used by the end-users of composition systems. Section 5.1 describes the developer tooling and Sect. 5.2 the runtime functionality.

5.1 CoMoGen: The Reuseware Development Kit

The development kit is named *CoMoGen*, which stands for *Component Model Generator*. The name was given by the central functionality provided by the tool—generating a new component model for a given language based on its metamodel. CoMoGen offers the *reuse extension* metalanguage in which a developer can express syntactic as well as semantic reuse extensions (cf. Sect. 4.2) .

Figure 18 shows the architecture of CoMoGen. On the lowest layer, it uses the functionality of EMF: EMF's resource management is used to load and store metamodels; EMF's code generation is applied to generate metamodel code; EMF's metamodeling facilities—arranged around the metalanguage *Ecore*[6]—are utilised to construct, modify and annotate metamodels.

[6] Ecore implements the OMG's EMOF standard [24]

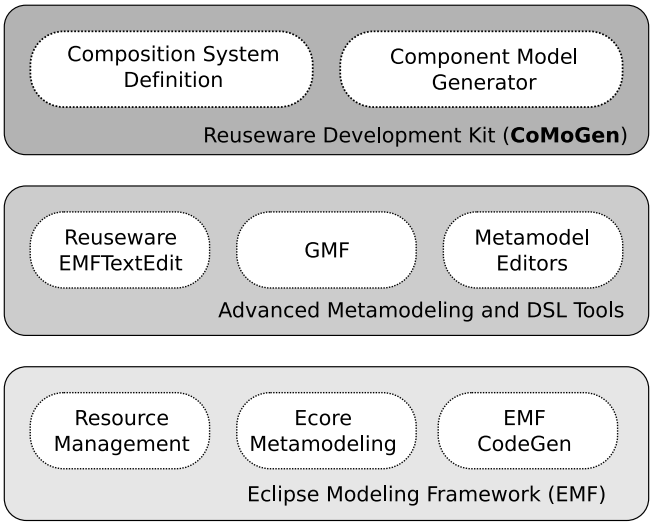

Fig. 18. Overview of the Reuseware Development Kit (CoMoGen) architecture

Next, CoMoGen interacts with other EMF-based metamodeling tools that provide metamodels and other specifications, for instance about a language's syntax. Examples of such tools are: Ecore metamodel editors (like the one contained in [25]), the mentioned GMF [16] or tools for defining textual syntax (like *EMFTextEdit* [26]).

Our implementation then offers facilities to define composition systems (e.g., an editor for the reuse extension language) and the component model generator itself, which modifies Ecore metamodels following the algorithm from Sect. 4.2 and adds annotations with OCL expressions to the metamodels using Ecore's annotation mechanism.

5.2 CoCoNut: The Reuseware Runtime

CoCoNut, the *Composition Core Runtime*, implements the composition algorithm and provides tooling to define concrete model compositions. Due to the language independence of our approach, the tooling, which is based on the language-independent concepts only, can be reused in any composition system defined with Reuseware.

The CoCoNut architecture is shown in Figure 19. Again we use EMF's model resource management and Ecore-based model manipulation facilities to load, save and compose model fragments. On top of this, CoCoNut implements an extended resource management which explicitly knows about model fragments. This means that it can identify fragments by unique identifiers and can present them—for instance in a fragment browser—by showing only their composition interfaces. Furthermore, CoCoNut includes composition program development tools like the editor presented in Sect. 4.2. The composition engine implements the composition algorithm described in Sect. 4.3 by using EMF facilities to copy and compose fragments in-memory.

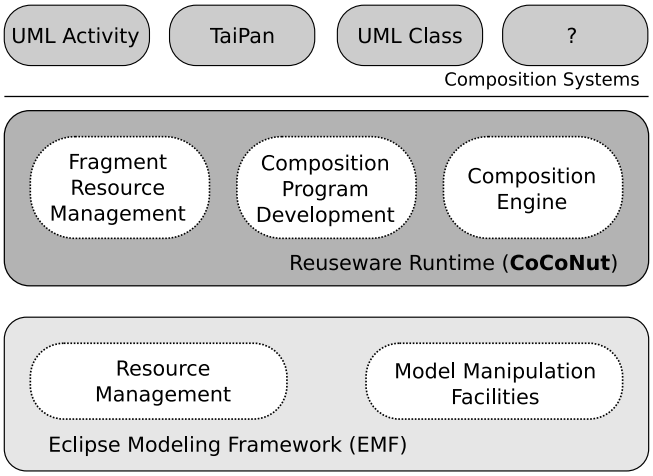

Fig. 19. Overview of the Reuseware Runtime (CoCoNut) architecture

All components of CoCoNut are aware of the concepts of our composition approach. They know that a certain element of a model fragment belongs to the fragment's composition interface by inspecting its metaclass and evaluating OCL expressions annotated to the fragment's metamodel. Thus, a new composition system can be plugged into CoCoNut by providing a metamodel that has been extended and annotated by CoMoGen. No additional implementation effort is required.

Latest information about the Reuseware Composition Framework and the available tooling can be found on the Reuseware website [12].

6 Examples

In this section we take up the examples introduced in Sect. 2. For each example, we will demonstrate how the language extension (cf. Sect. 4.2) is performed to make the applied modelling language a suitable component description language. Next, we show how the composition interfaces for the example components are defined and discuss the composition programs producing the desired results. Then, we briefly look at possible variations of the composition programs to highlight the advantages of using fragment model components rather than monolithic models.

6.1 Implementation of a Simple Business Process Extension System

To extend the UML activity diagram language we perform only semantic extensions to maintain tool support. However, to give the fragment developer control over defining addressable points, we apply a small UML profile and use semantic extensions to map

(1) hook extension: ActivityNode

isAddressablePoint	self = self.activity.node–>any (true)
portNameExpression	self.activity.name.concat('ExtensionPoint')

(2) prototype extension: ActivityNode

isAddressablePoint	self.getAppliedStereotype('reuseuml::Slot').oclIsUndefined()
portNameExpression	self.activity.name.concat('Definition')
groupNameExpression	self.activity.name

(3) hook extension: ActivityEdge

isAddressablePoint	self = self.activity.edge–>any (true)
portNameExpression	self.activity.name.concat('ExtensionPoint')

(4) prototype extension: ActivityEdge

isAddressablePoint	true
portNameExpression	self.activity.nam.concat('Definition')
groupNameExpression	self.activity.name

(5) slot extension: ActivityNode

isAddressablePoint	not self.getAppliedStereotype('reuseuml::Slot').oclIsUndefined()
portNameExpression	self.getValue(self.getAppliedStereotype('reuseuml::Slot'), 'portName')
groupNameExpression	self.getValue(self.getAppliedStereotype('reuseuml::Slot'), 'groupName')
pointNameExpression	self.getValue(self.getAppliedStereotype('reuseuml::Slot'), 'pointName')

(6) anchor extension: ActivityNode

isAddressablePoint	not self.getAppliedStereotype('reuseuml::Anchor').oclIsUndefined()
portNameExpression	self.getValue(self.getAppliedStereotype('reuseuml::Anchor'),'portName')
groupNameExpression	self.getValue(self.getAppliedStereotype('reuseuml::Anchor'),'groupName')
pointNameExpression	self.getValue(self.getAppliedStereotype('reuseuml::Anchor'),'pointName')

Table 1. Semantic UML language extension in the following format: (*ref. number*) (*type of variation point*) extension: (*metaclass*)

applied stereotypes to addressable points. The profile (*reuseuml*) is tailored for the activity diagram scenario and simplistic: it defines two stereotypes *Anchor* and *Slot* each with the tagged values *portName*, *groupName*, and *pointName*.

Table 1 enumerates all semantic extensions we defined. In (1) and (3) we define that each activity offers an implicit extension point: one of the nodes (edges) is, in addition to its native semantics, also a hook. We group them together into a port named after the activity's name such that they appear as a single extension point for the activity (a receiving port). In (2) and (4) all elements contained in an activity (i.e., its nodes and edges) are identified as prototypes and associated with a port named after the activity's name. Only nodes that have the *Slot* stereotype applied are ignored here. At last, (5) and (6) define that nodes with a *Slot* (or *Anchor)* stereotype are treated as slots (or anchors respectively). The properties of the addressable point are derived from the tagged values of the stereotype application.[7]

[7] The operation *self.getValue(stereotypeApplication, 'taggedValue')* can be used in the Eclipse UML2 [27] implementation, which we use in our tool, to obtain a tagged value of an applied stereotype.

<<reuseuml::Anchor>> checkFork : ForkNode

portName	CheckActivity
pointName	IN
groupName	

<<reuseuml::Anchor>> checkJoin : JoinNode

portName	CheckActivity
pointName	OUT_YES
groupName	

<<reuseuml::Anchor>> checkMerge : MergeNode

portName	CheckActivity
pointName	OUT_NO
groupName	

Table 2. Anchor stereotype applications on the process order model (cf. Fig. 1) in the following format: << (*stereotype*) >> (*targeted model element*) : (*metaclass*)

<<reuseuml::Slot>> InitialNodeCREDIT : InitalNode

portName	CheckActivity
pointName	IN
groupName	CreditCardCheck

<<reuseuml::Slot>> FINISH : FinalNode

portName	CheckActivity
pointName	OUT_YES
groupName	CreditCardCheck

<<reuseuml::Slot>> CANCEL : FinalNode

portName	CheckActivity
pointName	OUT_NO
groupName	CreditCardCheck

Table 3. Slot stereotype applications on the credit card check model (cf. Fig. 2)

We are now ready to prepare the order processing model from Fig. 1 and the credit card check model from Fig. 2 for composition. The order processing model now implicitly offers a receiving port *OrderProcessingExtensionPoint* (cf. (1) and (3) in Table 1). Additionally, the *checkFork*, *checkJoin*, and *checkMerge* nodes should be addressable to connect additional check activities to them. We do that by applying stereotypes to these nodes and setting the tagged values *portName* and *pointName* as shown in Table 2.

The credit card check model implicitly exports its content—that is its two actions and all control flows—to a contributing port *CreditCardCheckDefinition* (cf. (2) and (4) in Table 1). To connect the edges correctly to the nodes of the order processing model later, we apply the slot stereotype on the initial and final nodes of the credit card check model. We add them to the *CreditCardCheck* group and give them the same point names (cf. Table 3) as used for the anchors (cf. Table 2) in the order processing model. This enables the composition engine to match the anchors and slots as desired.

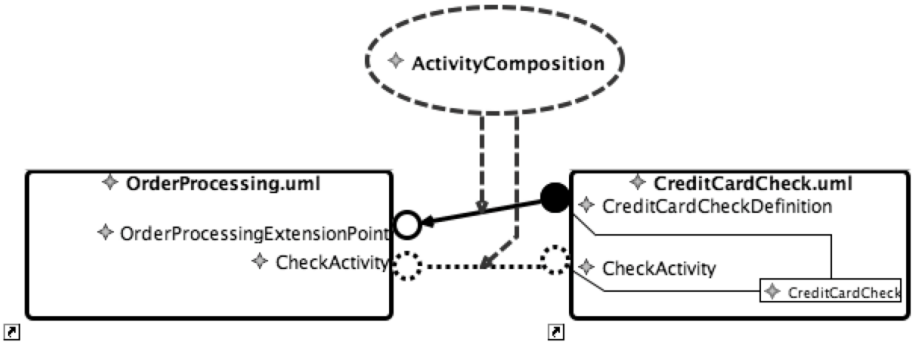

Fig. 20. The composition program to compose the order process and the credit card check

We now load the fragments into the composition program editor that displays their composition interfaces only. In the composition program, we link the contributing port *CreditCardCheckDefinition* with the receiving port *OrderProcessExtensionPoint* and the two configuring ports *CreditCardCheck* and *CheckActivitiesExtension*. The two links are assigned to the step *ActivityComposition*. When we execute the composition program using our implementation of the composition technique from Sect. 4.3, we obtain a composed model as shown in Fig. 3.

This example has demonstrated how a system for the desired activity diagram composition can be defined. With this system similar activity extensions and variations can be defined and executed. For instance, imagine a scenario where a large set of check activities that all declare a contributing port with an *IN*, an *OUT_YES*, and an *OUT_NO* slot (i.e., offer a similar composition interface) are available in a library. There might be checks related to software products (*CheckSW*.uml*) and checks related to hardware products (*CheckHW*.uml*). Companies want to incorporate checks according to the products they sell into their ordering process. We can use a fragment query to tailor the process by selecting a set of activities and composing them into the ordering process. By varying the query, we can adjust the defined system to the customer's needs: *CheckSW*.uml* for a software selling company, *CheckHW*.uml* for a hardware selling company, and *Check*.uml* for a company selling both.

6.2 Implementation of a Modular Ship and Cargo Distribution System

To add modularisation to the TaiPan language, we also use *syntactic* extensions. This is possible because the tooling is generated with GMF and can be regenerated and extended. Nevertheless, we also define some semantic metaclass extensions to introduce a default composition interface for certain model components.

Table 4 enumerates all extensions performed. Extensions (1), (4), and (6) are defined to make an aquatory model component extensible. Note that these are all semantic extensions that define hooks (to put ships into the aquatory) and anchors (to enable ships to address ports and routes). Thus, they define a default (or implicit) interface for aqua-

(1) hook extension: Ship

isAddressablePoint	self = self.aquatory.ships–>any (true)
portNameExpression	'shipExtension'

(2) prototype extension: Ship

isAddressablePoint	true
groupNameExpression	flotilla
portNameExpression	ships

(3) slot extension: Port

syntactic extension

(4) anchor extension: Port

isAddressablePoint	true
portNameExpression	self.location.concat('Port')

(5) slot extension: Route

syntactic extension

(6) anchor extension: Route

isAddressablePoint	true
portNameExpression	self.description

(7) hook extension: LargeItem

syntactic extension

(8) prototype extension: LargeItem

isAddressablePoint	true
portNameExpression	self.article

Table 4. Taipan language extension

tory models and save the developer of such models the effort of defining each hook and anchor explicitly. Still, the developer should be aware of the existence of the default interface which he can always inspect in our composition program editor. All the syntactic extensions—(3), (5), and (7)—are intended for the developers of *flotilla* model components. They can use slots for the ports and routes of ships. Inside the cargo bays of ships they can define hooks for large items. Additionally, extension (2) will automatically export each individual ship to the composition interface. Extension (8) defines all large items in *cargo* model components to be prototypes such that they can be addressed for composition with flotilla models.

The port and cargo models (cf. Fig. 5 and Fig. 7) do not require any further editing, because they only offer a default interface. The flotilla is extended with defined slots and hooks. We extended the TaiPan graphical editor to support the declaration of these elements and used them in Fig. 21.[8]

Figure 22 displays the composition program that composes the example aquatory, flotilla, and cargo model components into a single TaiPan model as shown in Fig. 4. The composition is separated into three independent composition steps that are executed

[8] This was a surprisingly easy task, because the editor was generated from a very abstract model (called *sketch model* in GMF) that assigns graphical representations to metaclasses. So we only had to select graphics for slot and hook representations and assign them to the slot and hook metaclasses of the reuse metamodel and then regenerate the editor.

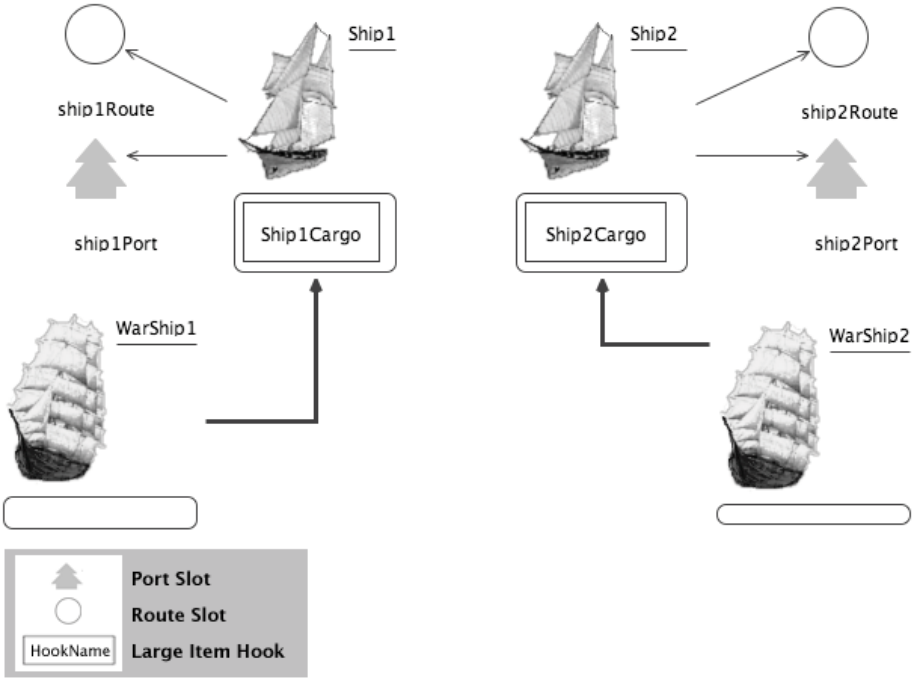

Fig. 21. The flotilla model with slots and hooks

one after another. It demonstrates how the flotilla model first receives—through the two composition steps *CargoIntoShip1* and *CargoIntoShip2*—and then contributes—through the composition step *FlotillaIntoAquatory*. Interpreting the composition as an aspect-weaving, the flotilla model first plays the role of a core and then of an aspect.

We can further use aspect-oriented concepts, when we replace the flotilla model in the example by a fragment query grouping many such models as shown in Fig. 23. Assuming that we have a second flotilla model *MyFlottillaB.taipan* in our repository, the fragment query *MyFlottilla.*.taipan* groups the two flotilla models *MyFlottilla.taipan* and *MyFlottillaB.taipan*. *MyFlottillaB.taipan* defines an additional ship with the ports *Ship3Port*, *Ship3Route* and *Ship3Cargo*. Using regular expressions, we merge the ports from the different flotilla models as follows:

- *Ship(2|3)Port* merges *Ship2Port* and *Ship3Port*.
- *Ship(2|3)Route* merges *Ship2Route* and *Ship3Route*.
- *ships* merges *ships* from *MyFlottilla.taipan* and *ships* from *MyFlottillaB.taipan*.
- *Ship(2|3)Cargo* merges *Ship2Cargo* and *Ship3Cargo*.

Similar to distributing an aspect over a core, we load the same cargo (*tobacco*) into two ships in the *CargoIntoShip2+3* composition step. Through the merged ports

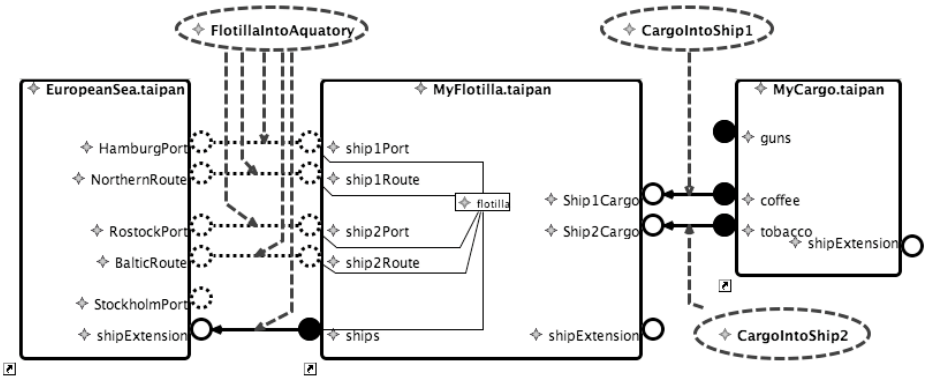

Fig. 22. The composition program to compose the aquatory, flotilla, and cargo model components

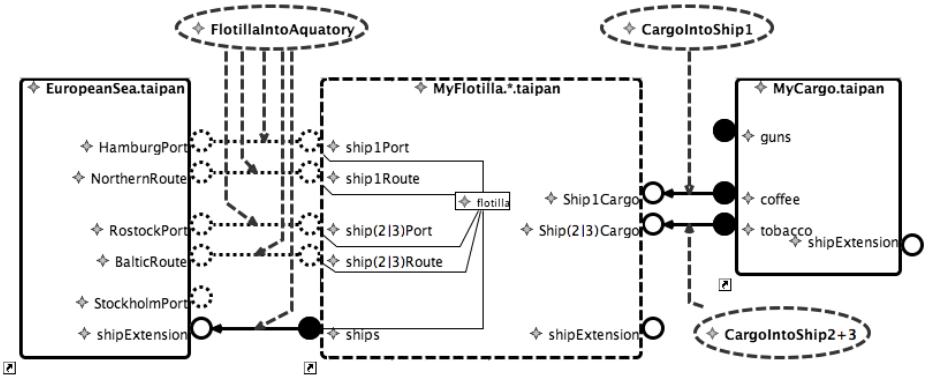

Fig. 23. The composition program to compose the aquatory, two flotilla, and cargo model components by applying a fragment query

Ship(2|3)Port and *Ship(2|3)Route*, we ensure that the two ships get the same route and destination assigned in the *FlotillaIntoAquatory* composition step.

The composition step *CargoIntoShip2+3* resembles an aspect-weaving: The fragment query *MyFlottilla.*.taipan* and the port merge *Ship(2|3)Cargo* quantify over a set of *core* flotilla models and the tobacco cargo *aspect* is distributed over it in a cross-cutting manner.

6.3 Other Examples

In this section we briefly discuss additional applications, we were and are still working on. More details and examples can be found on the Reuseware website [12].

Class Diagram Weaving In this application we introduced aspect-oriented concepts into *Ecore* (which could similarly be done for UML or other languages with a class concept). The idea is to distinguish between core classes—which are complete classes that offer an interface for extension—and advice classes—which define operations and features (possibly referring other advice classes) to be reused as extensions for core classes. Advice classes can be woven into core classes, which means that all their operations and features are injected into the core class. If an advice class has references to other advice class, all these classes have to be bound in one composition step.

The composition system is defined in terms of semantic extensions only. To distinguish between core and advice classes, a name convention on the package that contains the classes is used: a package with *advice* in its name contains advice, others contain core classes. A core class has hooks for both its lists of operations and features. Additionally the class itself is an anchor. The two hooks and the anchor of each core class are exported to a receiving port that is named like the class itself. An advice class, on the contrary, defines two lists with prototypes—its operations and features—and itself as a slot. The two prototype lists and the anchor of each advice class are exported to a contributing port that is named like the class itself.

As an example, one can consider the, rather technical, aspect of a subject-observer relationship that can be modelled in two related advice classes: *Observer* and *Subject*. The two classes appear as contributing ports on the composition interface of the advice fragment and can be linked individually to two classes of a core fragment in a composition step. A more detailed example and the complete definition of the composition system can be found on the web.[9]

Weaving Java Classes As mentioned in the above application, the composition system for class weaving can be ported to other languages that have a class concept. This is not limited to graphical languages, but can also be done for textual languages, as long as a proper metamodel exists (and a tool that can parse textual model definitions). We defined such a metamodel for (a subset of) Java and then realised the class weaving composition system for Java. We have already used this as an example in [14]. There, we applied syntactic extension.

We modified the composition system to work with semantic extensions; similar to the Ecore weaving system. Interestingly, we can reuse the composition programs defined for Ecore fragments above for similar examples based on Java fragments, without having to change the composition programs at all. In MDD, one can benefit from this, for instance, in code generation: model fragments can be translated to code individually—reducing complexity of generation and keeping the separation of concerns from the models down to the code, which is important when the code is manually modified after generation. To integrate the code fragments one can reuse the composition program already defined on the modelling level. This is one direction of future work, were we investigate how our approach can help in broader MDD settings, were different languages and composition systems are involved. More information about our experiments with Java can be found on the web.[10]

[9] http://reuseware.org/index.php/Ecore_Aspect_Weaving

[10] http://reuseware.org/index.php/Java

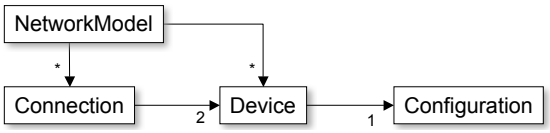

Fig. 24. Excerpt from the metamodel of a network configuration DSL

Including Modularisation and Aspect-Orientation in a DSL under development When a DSL is developed, our approach can be used to add modularisation capabilities to the language and profit from our existing tools for composition definition and execution. This example demonstrates this on a DSL for network configuration. Figure 24 shows an excerpt from the metamodel: A *network model* consist of *connections* and *devices* that have a *configuration* attached. Now, independent definition and reuse of *configurations* should be supported. Hence, we add hooks and prototypes for *configurations*; and we are done with the definition.

Networks with hooks instead of concrete configurations and configuration prototype fragments can now be modelled. Configurations can be bound to hooks using the composition editor. Thanks to the support for quantification through fragment queries and merged ports, we can also distribute one configuration over a large network model. More details about this example can be found on the web.[11]

Query Modularisation A complex composition system we built based on our earlier, grammar-based, work ([13]) is a module system for the XML query language Xcerpt [28]. Details about this systems are published in [13] and [29]. The system extends a language that previously included no notion of modularisation with a module system that performs encapsulation and enables the developer to import modules and to control the data flow between modules.

The system was not defined through concepts introduced in this paper, but uses our previous grammar-based approach. Information about the Xcerpt module system can also be found on the web.[12]

Managing Variability in Software Product Line Engineering In Software Product Line Engineering (SPLE), one of the main challenges is expressing and managing the variable parts in product lines. Although there already exist different variability concepts and patterns, all of them are tied to a specific level of abstraction in the software development process (e.g., models or code).

Due to its language independence and its built-in concept for expressing variability, ISC is an interesting technique to express and manage variability on each stage of a multi-staged software development process which we examined in a case study where we developed a simple product line of time-sheet applications.

[11] http://reuseware.org/index.php/CIM_DSL_Extension
[12] http://xcerpt.reuseware.org/

In this case study we used Reuseware's modularisation concepts to decompose the variable parts of the product line on both the modelling level and code level. We created fragments for each feature realisation and used Reuseware's graphical composition language to specify the composition of those variable parts of the product line with the core. Since we aim at an automated product-instantiation process, we created a mapping between conceptual variability models and composition steps of the composition programs using our tool *FeatureMapper*[13] [30]. This mapping is interpreted by a dedicated product-instantiation workbench we developed in the context of the case study, which only executes the actual composition if the corresponding feature from the variability model should be included in the concrete product variant.

7 Related Work

In [31], Klint et al. identify the need for an engineering approach to language development. Our Reuseware tool, presented in this paper, can be viewed as a form of meta-grammarware in their sense; that is, as a "software that supports concrete grammar use cases by some means of meta-programming, generative programming or domain-specific language implementation" [31, p. 342]. Our approach uses metamodels (which are grammars in the sense of Klint et al., who use the term grammar in a slightly more general sense essentially as 'anything describing a language') to generate composition systems and enable the execution of compositions through model or program transformation for languages that originally did not support composition.

As Reuseware and invasive software composition have originally been developed for textual languages, in the following, we first discuss a number of related work in the area of textual languages before also discussing some related work in modelling. For space reasons, neither discussion is meant to be complete, but rather to give an insight into some of the manifold research approaches in the expanding field of grammarware engineering. We will particularly focus on approaches in the aspect-oriented community.

Several approaches exist that provide aspect orientation for the .NET platform and claim that this makes their approaches language agnostic or independent. For example, Aspect.NET [32] uses static weaving based on binary assemblies to provide AspectJ-like AOP for .NET, Compose* [33] is an implementation of Composition Filters for .NET. These approaches work on the level of the Common Language Infrastructure (CLI) and hence are indeed independent of specific programming languages. At the same time, however, working at the CLI level also means that these approaches cannot provide language-specific modularisation concepts. Our approach works at the level of each individual language itself. While this makes it more complicated to mix modules of different languages, it enables us to build custom modularisation techniques for each language.

Fractal Aspect Components (FAC) [34] is an extension of the Fractal Component Model [35] to support Aspect-Oriented Programming. It aims at bridging the gap between Component-Based Software Engineering and Aspect-Oriented Programming.

[13] http://featuremapper.org

FAC introduces several additional concepts to the Fractal Component Model to compose Aspectisable Components and Aspect Components. FAC is similar to the Reuseware approach, because the component model is designed in a language-independent way. This allows for conceptual reuse within different implementations. It is, however, also different in many ways. For example, aspect binding and composition programs do not abstract from the implementation of the component model in FAC. With the graphical fragment composition editor, the Reuseware approach provides a general-purpose way to express compositions independently of the fragment component's core language.

In [36], Gray and Roychoudhury present a technique for constructing aspect weavers for arbitrary languages. They define an aspect weaving language (called *Aspect Domain*) which can be used to define weavings for different languages. They argue that a common superset of weaving operations can be applied to arbitrary languages, while certain languages require specific extensions. The weaving language is comparable to our composition language. One important difference is that Gray and Roychoudhury do not extend languages, because their components (that is, *core* and *advice* artefacts) only have implicit composition interfaces—which is reasonable, since they focus on legacy systems written in existing languages—while we focus on programs and models under development. Furthermore, our approach can also deal with non-textual languages described by a metamodel.

The *Mjølner System* and the *Beta language* [37] were the first to introduce the concept of slots. In Beta, any programming construct can be replaced by a slot typed with the non-terminal corresponding to that construct. Beta also supports a notion of inheritance of grammar types. Binding of slots happens when the name of a fragment and the name of a slot in the same project match. Our approach extends the Beta approach in two ways:

1. We introduce additional types of variation points, like anchors, hooks, and prototypes. Additionally, we introduce the new (language-independent) abstraction of ports that gives more control to the fragment developer when defining an interface. The linking of ports is also an explicit operation allowing the definition and variation of composition programs, while Beta uses implicit matching of names only.
2. We extend the concept to any language that can be described by a metamodel. Different from Beta, our tool allows arbitrary languages to be extended with a composition system.

The *Software COMPOsition SysTem* (COMPOST) [38], the demonstrator system of [11], is a predecessor of our current system, which introduced many of the concepts available in our approach, but was limited to Java and XML. For each new language that should be supported by COMPOST a large amount of implementation work is required. In [13] we introduced the first version of the Reuseware system which was capable of extending grammar-based textual languages and performing compositions of syntax trees, without the requirement for manual implementations. We took first steps in extending these concepts towards metamodel-based (possibly graphical) languages in [14]. There, we introduced the concept of fragment queries but did not elaborate on the details of metamodel extension or the composition algorithm. Novel in the current work are also the concepts of ports, composition links, and composition steps, which were not

required in the syntax-tree composition approach ([13]) and were not yet applied in the compositions presented in [14].

Our notion of fragment components is comparable to the notion of *syntactic units* presented in [39]. Syntactic units are arranged in *syntactic unit trees* that can be likened to composition programs. In this approach, so-called extension spots can be defined as alternatives for any fragment of code derivable from a non-terminal. Compared to our approach, there is no formalisation of language extensions which allows for tailored extension of a language (to only allow the desired amount of variability) and generation of language-specific tooling.

In the area of model-based approaches, Model Weaving is strongly related to the work presented in this paper. It allows for combining two or more models to form a composed or woven model. AMW, the Atlas Model Weaver [40] is a tool that allows generating model transformations based on a so-called *Weaving Model*. The Weaving Model consists of links between two or more models that are used to generate model transformations and model weavings.

Another approach to model weaving presented in [9] by Heidenreich and Lochmann stems from Product-Line Engineering and provides means to express *Aspectual Features* in separate models which are woven into a core model according to the feature selection of the product line. The authors are using graph-rewrite systems to weave the Aspectual Features to the core model. This idea was adopted in the design of the XWeave [41] tool by Groher and Völter. XWeave is integrated in the openArchitectureWare tool chain and uses name correspondence and regular expressions for model weaving as our composition language does.

However, the work presented in this paper goes beyond existing model weaving. It unifies weaving and composition operations on both model and text artefacts through the general concepts of addressable points and fragment queries.

The Generic Modeling Environment (GME) [42, 43] offers generic means to build UML-based DSMLs and also allows for defining concrete syntax for those languages. It supports partitioning of models according to *aspects* that are defined on the metamodel level. While this increases understandability and maintainability of complex models, it does not address the issue of reusability of language modules, the goal of Reuseware. In [44], the authors introduce the concept of metamodel composition to GME where existing language modules and newly developed languages can be composed by dedicated composition operators. This fosters reuse of modularisation techniques which is the driving force behind our work. Compared with the Reuseware approach, metamodel composition as presented within GME does not allow for language-agnostic interpretation of the reused language modules.

Many aspect-oriented approaches to modelling have been developed, most of which are specific to one particular modelling language. A large number of these approaches is inspired by aspect technology as introduced in the area of AOP—for example, [7] presents an approach to aspect-orientation for state machines that is closely inspired by AspectJ technology. At the same time, approaches are beginning to appear that show composition techniques differing from aspect-oriented ideas. For example, in [10], Whittle et al. present an approach that uses pattern matching on state-machine concrete syntax and graph transformation to describe aspects on state machines.

[45, and references therein] presents a generic framework for composing different views on a model. The approach distinguished a *matching* and a *merging* phase. The matching phase determines which model elements in two models should be merged together, while the merging phase performs the actual merging. Merging is implemented in a completely language-independent fashion. Matching is language dependent and the match rules must, therefore, be provided in a specialisation of the framework. However, the framework defines an interface for the match rules, which is, to our understanding, based on matching metaclasses and signatures. Our approach can also be seen to distinguish a matching and a merging phase. However, both phases are expressed language-independently by composition diagrams in our composition description language. Specialisation to specific languages is only necessary to identify how addressable points etc. can be expressed for model components. In our approach, matching must be done for each composition individually. In contrast, [45] use matching rules that are defined once for a specific language and then applied to multiple combinations of models. We are planning, however, to extend the approach presented in this paper to support concepts similar to such matching rules. For textual languages we have already presented such an approach in [46] under the name of a *light-weight dedicated composition system* (LWDCS).

C-SAW [47] is a general model transformation tool that also supports some form of aspect-oriented modelling independently of the specific modelling language. Developers write so-called *aspects* or *strategies*, model transformations expressed in the Embedded Constraint Language (ECL), querying for a number of model elements and then modifying these. Reuseware also is based on model transformation. However, the collection of model elements to be transformed is encapsulated in an explicit construct—the model fragment—rather than implicitly represented in a query inside the composition program.

8 Conclusions and Outlook

Modularising models is becoming increasingly important, especially due to the fact that model-driven development approaches are requiring richer and more complex models to be constructed. Not only are models growing in complexity and becoming harder to overview, but many different modelling languages—domain-specific modelling languages—are being developed alongside general-purpose ones such as UML. As we have demonstrated with use-cases for both kinds of languages, it is important to be able to construct larger models from smaller and better understood ones. The first use-case concerned the modularisation of UML activity diagrams, while the second use-case described how models of a domain-specific language (called TaiPan) can be split into different concerns. We have in this paper presented a language-independent approach to enable component-oriented thinking and development for modelling languages.

We proposed two ways of extending modelling languages with component capabilities. The first involves an extension of the underlying modelling language's metamodel in order to define components' interfaces, while the second can extract such interfaces implicitly. Avoiding metamodel extension has the benefit that already developed editors and tools will not break. However, for certain domain-specific modelling languages an

extension of the language metamodel can make sense and be an easier approach, as we have demonstrated on the TaiPan modelling language. Hence, both approaches can be useful depending on the particulars of the addressed language and the desired modularisation.

We would not have been able to reach our solution without implementing the ideas and applying them on examples. Our current implementation [12] is based on the Eclipse Modeling Framework and offers GUI tooling as plug-ins for the Eclipse platform. The main components of our tool are the graphical composition program editor presented and a fragment management system that extends the general resource management of the Eclipse Modeling Framework [23]. Because of the integrated Eclipse platform [21], on which many modelling tools are based, our tool can directly interact with tooling of the used component description languages. These tools are used to define fragments and view composition results. In the examples, for instance, we used the TOPCASED UML Editor [25] and the TaiPan editor. The importance of providing such a tool should not be underestimated for future research: It enables us to do case-studies more quickly and the good integration with existing modelling tools may improve acceptance in the community.

For the future we plan to do further case-studies to clarify the open questions of what additional matching concepts are needed in composition program definitions to match the ports of composition links and in fragment queries. This issue is also related to the concepts of *complex composition operators*, which we introduced as means to define composition systems for grammar-based languages [46, 13]. Such operators allow for the grouping of several composition operations that work together on a set of fragments and variation points. This grouping is similar to the grouping of addressable points into ports but defines the binding between variation and reference points explicitly. We believe that both concepts can be unified and that complex composition operators can be translated into composition programs of the approach presented in this paper. Doing this would unite our grammar-based and our metamodel-based approaches.

In the future we will formalise our composition technique which we described in this paper and implemented in the tool. This will give a formal definition of what a *valid* and what an *invalid* composition program is and will enable an analysis of the limits of our approach.

We also see potential in applying our approach in a larger model-driven development process, where different languages are utilised. We believe our approach will show its advantages in such a scenario, that is, where modularisation issues of all involved languages can be solved with a common base component model and a language-independent composition description language. In general it becomes easier to relate artefacts even when they are written in different languages, because they share certain parts of their component models. Composition programs can, for instance, be reused at different abstraction levels of an MDD process, where only details, but not the architecture, of a system change. We took a first step in this direction in [14] where we used the same composition program to compose UML and Java fragments.

Acknowledgement

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net) as well as the 6th Framework Programme project MODELPLEX contract number 034081 (cf. http://www.modelplex.org) and by the German Ministry of Education and Research (BMBF) within the project feasiPLe (cf. http://www.feasiple.de).

References

1. Ritsko, J.J., Seidman, D.I.: Preface. IBM Systems Journal – Special Issue on Model-Driven Software Development **45**(3) (2006)
2. Object Management Group: UML 2.0 infrastructure specification. OMG Document (October 2004) URL http://www.omg.org/cgi-bin/doc?ptc/04-10-14.
3. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, IEEE Computer Society (May 2007) 54–63
4. Peterson, J.L.: Petri nets. ACM Computing Surveys **9**(3) (September 1977) 223–252
5. Aldawud, O., Cazzola, W., Elrad, T., Gray, J., Kienzle, J., Stein, D., eds.: 10th Workshop on Aspect-Oriented Modeling (AOM at AOSD'07) co-located with the 6th International Conference on Aspect-Oriented Software Development (AOSD'07), Online Proc. (March 2007) URL: http://www.aspect-modeling.org/aosd07/.
6. Aldawud, O., Cazzola, W., Elrad, T., Gray, J., Kienzle, J., Stein, D., eds.: 11th International Workshop on Aspect-Oriented Modeling (AOM at MoDELS'07) co-located with ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems MODELS 2007, Online Proc. (September 2007) URL http://www.aspect-modeling.org/models07/.
7. Zhang, G., Hölzl, M., Knapp, A.: Enhancing UML state machines with aspects. [48] 529–543
8. Colyer, A., Clement, A., Harley, G., Webster, M.: Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools (The Eclipse Series). Addison-Wesley, Reading, MA, USA (2004)
9. Heidenreich, F., Lochmann, H.: Using graph-rewriting for model weaving in the context of aspect-oriented product line engineering. In: 1st Workshop on Aspect-Oriented Product Line Engineering (AOPLE'06) co-located with the International Conference on Generative Programming and Component Engineering (GPCE'06), Portland, Oregon, Online Proc. (October 2006) URL http://www.softeng.ox.ac.uk/aople/aople1/.
10. Whittle, J., Moreira, A., Araújo, J., Jayaraman, P., Elkhodary, A., Rabbi, R.: An expressive aspect composition language for UML state diagrams. [48] 514–528
11. Aßmann, U.: Invasive Software Composition. Springer, Secaucus, NJ, USA (2003)
12. Software Technology Group, Technische Universität Dresden: Reuseware Composition Framework (April 2008) URL http://www.reuseware.org.
13. Henriksson, J., Heidenreich, F., Johannes, J., Zschaler, S., Aßmann, U.: Extending grammars and metamodels for reuse: the reuseware approach. IET Software **2**(3) (2008) 165–184
14. Heidenreich, F., Johannes, J., Zschaler, S.: Aspect orientation for your language of choice. [6] URL http://www.aspect-modeling.org/models07/.

15. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns co-located with OOPSLA 2000, Minneapolis, MN, USA (October 2000)
16. The Eclipse Foundation: Graphical Modeling Framework (April 2008) URL http://www.eclipse.org/gmf/.
17. Sun Microsystems: Enterprise JavaBeans Specification, version 2.0. Final Release (August 2001)
18. Object Management Group: CORBA 3.0 new component chapters. OMG Document (October 1999) URL http:// www.omg.org/ cgi-bin/ doc?ptc/ 99-10-04.
19. Fiala, Z.: Design and Development of Component-based Adaptive Web Applications. PhD thesis, Technische Universität Dresden, Dresden, Germany (February 2007)
20. Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, ACM Press (September 2005) 166–175
21. The Eclipse Foundation: The Eclipse Platform (April 2008) URL http://www.eclipse.org.
22. Object Management Group: UML 2.0 OCL specification. OMG Document (October 2003) URL http://www.omg.org/cgi-bin/doc?ptc/03-10-14.
23. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)
24. Object Management Group: MOF 2.0 core specification. OMG Document (January 2006) URL http://www.omg.org/spec/MOF/2.0.
25. The Topcased Project Team: TOPCASED (April 2008) URL http://www.topcased.org.
26. Software Technology Group, Technische Universität Dresden: EMFTextEdit Tool (January 2008) URL http://www.emftextedit.reuseware.org.
27. The Eclipse Foundation: UML2 Project (April 2008) URL http://www.eclipse.org/modeling/mdt/?project=uml2tools.
28. Bry, F., Schaffert, S.: The XML query language Xcerpt: Design principles, examples, and semantics. In Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R., eds.: Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems. Volume 2593 of LNCS. Springer, London, UK (2003) 295–310
29. Aßmann, U., Berger, S., Bry, F., Furche, T., Henriksson, J., Johannes, J.: Modular web queries – from rules to stores. In Meersman, R., Tari, Z., Herrero, P., eds.: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops (Part II). Volume 4806 of LNCS., Vilamoura, Portugal, Springer (November 2007) 1165–1175
30. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping Features to Models. (Informal Research Demonstration). In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany (May 2008)
31. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering **3**(14) (July 2005) 331–380
32. Safonov, V., Gratchev, M., Grigoryev, D., Maslennikov, A.: Aspect.NET – aspect-oriented toolkit for Microsoft.NET based on Phoenix and Whidbey. In Knoop, J., Skala, V., eds.: 4th International Conference .NET Technologies, Plzen, Czech Republic, University of West Bohemia (May 2006) 19–30
33. García, C.F.N.: Compose* – a runtime for the .Net platform. Master's thesis, Vrije Universiteit Brussel, Belgium (August 2003) More information also at http://composestar.sf.net/.
34. Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L.: A model for developing component-based and aspect-oriented systems. In Löwe, W., Südholt, M., eds.: 5th International Symposium on Software Composition (SC'06). Volume 4089 of LNCS., Vienna, Austria, Springer (March 2006)

35. The Fractal Project Team: The Fractal Project (April 2008) URL http://fractal.objectweb.org/.
36. Gray, J., Roychoudhury, S.: A technique for constructing aspect weavers using a program transformation engine. In Murphy, G.C., Lieberherr, K.J., eds.: 3rd International Conference on Aspect-Oriented Software Development (AOSD'04), Lancaster, UK, ACM Press (March 2004) 36–45
37. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, Reading, MA, USA (June 1993)
38. The COMPOST Consortium: The COMPOST system (April 2008) URL http://www.the-compost-system.org.
39. Majkut, M., Franczyk, B.: Generation of implementations for the model driven architecture with syntactic unit trees. In Crocker, R., Jr., G.L.S., eds.: 2nd Workshop Generative Techniques in the context of MDA co-located with OOPSLA 2003, Anaheim, CA, USA, Online Proc. (October 2003)
40. The AMW Project Team: Atlas Model Weaver (April 2008) URL http://eclipse.org/gmt/amw/.
41. Groher, I., Völter, M.: XWeave: Models and aspects in concert. [5] URL: http://www.aspect-modeling.org/aosd07/.
42. Vanderbilt University, Institute for Software Integrated Systems: GME: The Generic Modeling Environment. http://www.isis.vanderbilt.edu/Projects/gme/ (2008)
43. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. Technical report, Vanderbilt University, Institute for Software Integrated Systems, Nashville, TN, USA (2000)
44. Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., Maroti, M.: On metamodel composition. In: IEEE International Conference on Control Applications 2001 (CCA'01), Mexico City, Mexico (September 2001) 756–760
45. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A generic approach for automatic model composition. [6] URL http://www.aspect-modeling.org/models07/.
46. Henriksson, J., Aßmann, U., Heidenreich, F., Johannes, J., Zschaler, S.: How dark should a component black box be? The Reuseware Answer. In Weck, W., Reussner, R., Szyperski, C., eds.: 12th International Workshop on Component-Oriented Programming (WCOP) co-located with 21st European Conference on Object-Oriented Programming (ECOOP'07). Volume 4906 of LNCS., Berlin, Germany (July 2007)
47. Gray, J., Lin, Y., Zhang, J.: Automating change evolution in model-driven engineering. IEEE Computer **39**(2) (February 2006) 51–58
48. Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07). Volume 4735 of LNCS., Springer (October 2007)