Controlling Model-Driven Software Development through Composition Systems

Jendrik Johannes

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
jendrik.johannes@tu-dresden.de

Abstract. In Model-Driven Software Development (MDSD) all artifacts that are created during development can be regarded as parts of the final system. Thus, these artifacts can be seen as components of the system in the sense of Component-Based Software Engineering (CBSE). There, software is split into units such that different concerns are separated. Each functionality of the system can be tracked to the component it is implemented by and therefore fixed, exchanged, extended or individually reused. While in traditional composition systems the components operate independently at run-time, MDSD models are integrated at different points during development time by model transformations. Through this, the advantages mentioned are often lost; and problems occur when one of the components is changed either directly or indirectly during development of a system. This paper proposes an approach that integrates advantages from CBSE into MDSD. In particular the refinements performed during model-driven development are regarded as compositions supported by composition systems. The approach is based on our Reuseware formalism and exemplified on a concrete MDSD process.

1 Introduction

A central goal of Model-Driven Software Development is to regard all artifacts created during the development process as parts of the final system—although not directly but through integrating them into the final system by model transformations. Through these transformations, information is spread over different artifacts residing on different abstraction layers of the system's architecture. This is feasible when the construction of the system is following one straight direction where artifacts are refined towards the system without looking back or making adjustments on a more abstract layer after refinements were performed. This however does not correspond to the software development reality, where system development has to cope with changes and adjustments originating from new requirements or design flaws discovered later in the development process. For this, ideally, any artifact should be opened for modification throughout the whole MDSD process.

We argue that this requires a clear Separation of Concern (SoC) [1, 2] throughout the whole process. That is, it needs to be clear in which model which piece of information was defined. Then, if that information requires change, the change can be performed

on the original information rather than some derivative and all models give a consistent description of the system at any point during development. In this paper we propose an approach for sustainable SoC in MDSD based on concepts from Component-Based Software Engineering (CBSE). More concretely, CBSE concepts as realised in our Reuseware [3, 4] methodology and tooling.

If we look more closely at MDSD processes, each can be broken down to a number of stages. The models in such a process are *refined* from one stage to the next. Different viewpoints can be taken on the *refines* relation since no formal definition of it exists. This leads to a wide range of realisations of this relation in practice: models are *refined* from one stage into another by model transformations which can do anything from only copying the original model until replacing it with something completely new.

In this paper we 1) argue that a clear SoC in MDSD can only be reached by a more precise defining the *refines* relation and 2) propose one such definition that defines *refines* through the *compose* and *component instance-of* relations from the CBSE field. In this solution, we regard each model as a *component* and define different *composition systems* to compose these components.

The paper is organized as follows: Section 2 describes the concepts of Reuseware composition systems and where they can be found in MDSD. We exemplify the idea and show benefits on a concrete MDSD process in Sect. 3. Related work is discussed in Sect. 4; conclusions and future work are discussed in Sect. 5

2 Composition Systems for MDSD

This section shows where composition systems can be found in MDSD processes. For this, we first introduce concepts and terminology. We start by defining the term *model* for this paper. Afterwards, we introduce composition system concepts, how they are realized for software modelling in Reuseware and where they can be found in MDSD processes.

In this paper we use the term *model* as follows: every artifact in a MDSD process is part of a model, including documents and code. This means conceptually that a complete model can be made up of a set of artifacts that assemble a complete description of the system at some level of abstraction. Technically, it means that each artifact has to be written in a language that conforms to a metamodel where all metamodels are defined in a common metamodelling language—which is Ecore/EMOF [5, 6] in our case.

2.1 Concepts of Composition Systems

The composition systems we treat in this paper are defined with Reuseware [3, 4]. Reuseware is based on the concepts of Invasive Software Composition (ISC) [7] and can be used to specify invasive software composition systems for arbitrary Ecore/EMOF-based languages. In [7], a composition system is defined as a triple of composition technique, component model[1] and composition language. We introduce all three con-

[1] The term component model is not to be confused with the term model defined above.

cepts and their realization in Reuseware in the following. For additional details please consult [4] and the Reuseware website[2].

Composition Technique. A composition technique defines how components are physically connected. Each invasive software composition system defined in Reuseware uses the same composition technique based on graph rewriting. It utilises the fact that an Ecore/EMOF metamodel is essentially a typegraph that describes the non-context-free structure of a language such that sentences of that language (i.e., models) can be represented as graphs with a distinct spanning tree (the containment tree). Models are consequently treated as typed graphs. Composition is performed by extracting parts of such graphs—called model fragments—and merging them with other model fragments. The types (i.e., metaclasses) of model elements are considered to evaluate if a composition can be performed.

Component Model. A component model defines what a component is and how it can be accessed. Thus it determines which *composition interfaces* components have. Composition interfaces are required to instantiate components in order to include them into concrete compositions. In Reuseware, model fragments are the components. A model fragment's composition interface consists of a set of *reference points* and *variation points*. Reference points are model elements that are extracted from one model during composition and variation points are model elements that are replaced during composition by reference points. Reference and variation points are grouped into *ports*. This allows for cross-cutting compositions, since elements scattered over a model fragment can be grouped on the fragment's interface.

Composition Language. A composition language can be used to specify *composition programs* that define concrete compositions. Reuseware defines a composition language that allows fragments to be instantiated into *fragment instances*—component instance-of relation—and to be connected through composition links—compose relation. Compose relations link ports on the interface of the instantiated fragments. The variation and reference points behind these ports are then matched (i.e., their metaclasses are checked for compatibility). If the match is successful, the compose relation is valid and the fragments may be merged (i.e., composed) by Reuseware.

2.2 Defining Composition Systems for MDSD Processes

How do we obtain the composition interfaces of the fragments in a MDSD process and how do we obtain composition programs to compose these fragments into models of the application? In fact, this information—which can be seen as architectural information about the system—has to exist in our models already. It defines how different concerns relate to each other. Consequently, SoC can be achieved by making this information explicit.

Reuseware offers the means to make this architectural information explicit through composition systems. As stated, a composition system requires, in addition to the composition technique, a component model and a composition language. For the component model, we need to know how to identify reference and variation points in a fragment. This is done by specifying a set of OCL rules over the fragment's metamodel.

[2] www.reuseware.org

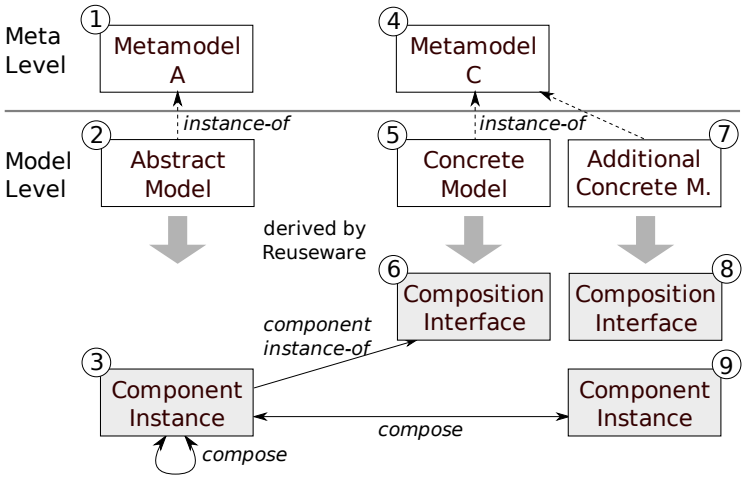

Fig. 1. Reuseware Composition Systems in MDSD

These rules are then applied to derive a fragment's composition interface. Similarly, an arbitrary language—defined through a metamodel—can be treated as a composition language. By giving a set of OCL rules, one can specify how fragment instances and compose relations are identified in a model. These rules are then applied to derive a composition program from (a set of) models.[3] Concrete examples will be shown in the next section.

Figure 1 shows two languages that are used in a MDSD process. Metamodel A (1) defines the *abstract language* and metamodel C (4) the *concrete language*, where the abstract language resides on an higher layer of abstraction than the concrete language. We interpret abstract models[4] (2) as composition programs. This means, a set of OCL rules allows Reuseware to derive fragment instances and compose relations from these models (3). The fragments that are instantiated by the fragment instances through the component instance-of relation are defined in the concrete language (5). That is, a set of OCL rules allows Reuseware to derive a composition interface for these fragments (6) that conforms to the interface expected by the fragment instances. Turning Fig. 1 by 90 degree clockwise would give a picture of a part of a MDSD process, where the component instance-of relation—read in opposite direction—corresponds to a refines relation from the abstract to the concrete model. Such a refinement is also known as *vertical model transformation*; that is, a refinement that leads to a more concrete abstraction layer.

[3] The derivation of composition programs from fragments is a new feature of Reuseware. It is based on the idea concept of *embedded invasive software composition* presented in [8].

[4] Whenever we talk about two models residing on two different abstraction layers, we call the model on the more abstract layer the *abstract model* and the model on the more concrete layer the *concrete model*.

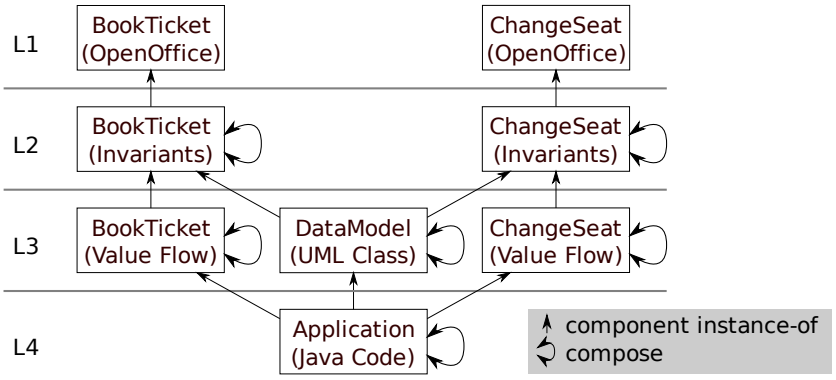

Fig. 2. Overview of the demonstrator MDSD process. In contrast to Fig. 1, there is no distinction made between models and their composition interfaces. Each box depicts a set of models and their composition interfaces.

Note that a fragment instance (derived from the abstract model) can be created without having a fragment (a concrete model) available right away. This can be seen as formulating the *required* interface for the fragment. This is a powerful abstraction, since it only needs to formulate the part of the interface that is really required in the concrete composition in which the instantiation takes place. Thus, a large set of fragments potentially fits the interface.

Another kind of refinement can be performed using the compose relation. We can formulate additional information in the concrete language (7) (8) and connect it through additional compose relations (9). Those additional compose relations could be derived by letting the concrete language act as composition language as well by providing OCL rules for that. This refinement is also known as *horizontal model transformation*; that is, a refinement that takes place within one abstraction layer.

Concrete examples of abstract and concrete languages employing the component instance-of and compose relations for refinement will be shown in the next section.

3 Composition Systems in a Concrete MDSD Process

For demonstration purpose and to show that our approach can indeed be applied in a complete MDSD process, we examine the model-driven development of a concrete demonstrator system. The demonstrator system was mainly inspired by the one used in [9]. It resembles a reservation system in which customers can book tickets and perform other related activities. Figure 2 gives an overview of the development process. Development starts with use case description documents and results in executable Java code. For this demonstrator, we realize two use cases—*BookTicket* and *ChangeSeat*. The system is refined over four abstraction layers. In this paper we can only show ex-

cerpts of the models and the involved composition systems. All models and Reuseware composition system specifications can be found on the web.[5]

System development takes place on four levels of abstraction (L1–L4 in Fig. 2) and employs five different languages, which are:

1. **OpenOffice**. *General purpose document language.* We use OpenOffice[6] documents for use case descriptions. They are models on L1 in our development process. A metamodel for OpenOffice documents was obtained by translating the OASIS OpenDocument RelaxNG schema[7], to which OpenOffice documents conform, into an XSD schema using Trang[8]. The XSD schema was converted into an Ecore metamodel by EMF's XSD-to-Ecore binding[9]. The latter allows us to load OpenOffice documents as models without further conversion. An example is shown in Fig. 4
2. **UseCaseInvariant**. *Textual domain specific modelling language.* In [9], the authors propose a refinement of use cases where they define the *value added invariant* of a use case. The idea is that each actor gives and takes values during the system's lifetime where the total amount of values in the system stays invariant. While [9] uses a graphical notation with actor and note symbols to initially capture value added invariants, we like to use a full *UseCaseInvariant* modelling language to define value added invariants for use cases on L2 in our MDSD process (Fig. 2). Therefore, we defined an Ecore metamodel and a text syntax with EMFText[10] for the *UseCaseInvariant* language.[11] The metamodel contains OCL constrains that check if the invariant for the complete use case is fulfilled (i.e, all values given must be received and the other way around). Figure 5 shows two example models.
3. **UMLClass**. *General purpose modelling language.* [9] also describes how to semi-automatically obtain a data model for the application as a UML class model. We follow this suggestion and use UML class models on L3. We use the Ecore metamodel provided by the Eclipse UML2 project[12] and the graphical TOPCASED editor[13] that uses this metamodel. Figure 7 shows a model defined in this editor.
4. **ValueFlow**. *Graphical domain specific modelling language.* Another refinement proposed by [9] is to specify the actual order in which values are exchanged between actors of a use case in form of statecharts. One statechart for each actor—or *agent* as they are called in this stage—is defined, where each state represents the passing or receiving of a value. Each state is therefore connected with a state of another agent's statechart, where the value is received/passed from. [9] uses UML statechart notation with notes and numbering that informally relate the different statecharts. Since we like to make the relations between the statecharts more explicit, we defined the *ValueFlow* metamodel which exactly defines the concepts

[5] www.reuseware.org/index.php/MDSD
[6] www.openoffice.org
[7] www.oasis-open.org/committees/office/
[8] www.thaiopensource.com/relaxng/trang.html
[9] help.eclipse.org/help33/topic/org.eclipse.emf.doc/tutorials/xlibmod/xlibmod.html
[10] www.emftext.org
[11] www.reuseware.org/index.php/EMFText_Concrete_Syntax_Zoo_Use_Case_Invariant
[12] www.eclipse.org/uml2
[13] topcased.gforge.enseeiht.fr

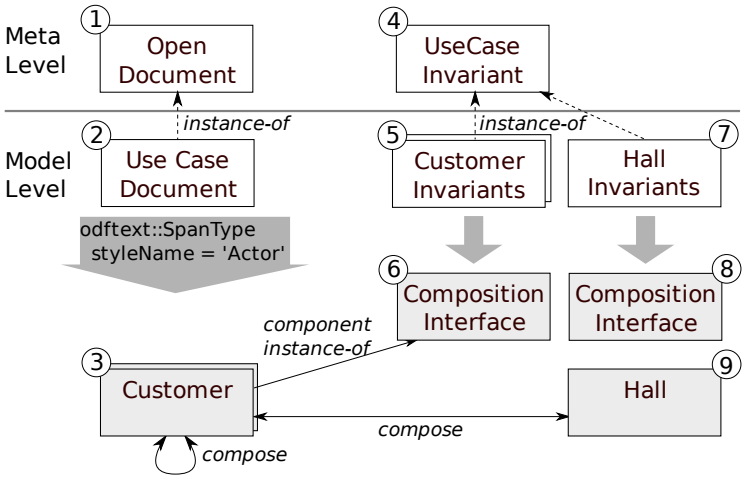

Fig. 3. Selected refinements between L1 and L2 (cf. Figure 1)

required (cf. Fig. 8 for an example).[14] We used EuGENia[15] to specify a graphical syntax and generate an editor for the ValueFlow language.

5. **Java**. *General purpose programming language.* The platform our application runs on is Java that is used on the most concrete layer of abstraction (L4). Thus, all information about the system is contained in this model since it represents the system itself. We treat Java programs as models using JaMoPP[16] which defines an Ecore metamodel for Java and brings the necessary tooling to handle Java programs as models (i.e., a parser and a printer for Java code).

As illustrated in Figure 2, parts of the system can be developed independently up to a certain abstraction level. For instance, each use case can be developed independently up to L3. Ultimately all information is connected in the final system.

The refinement performed in each of these steps is realized through component instance-of and compose relations. The details are shown in the following on the different steps in Sections 3.1 to 3.4.

3.1 Adding Invariants to Use Cases

Figure 4 shows an excerpt of the use case description for *BookTicket*. Although the description contains mostly free text, some structured data can be found as well. We consider the use case documents to be the most abstract models of the system (L1 in Fig. 2). The structured information they contain is which actors exist in the system and in which use cases they participate (e.g., *Customer*, *Clerk* and *Bank* in Fig. 4). To make

[14] www.reuseware.org/index.php/EMFText_Concrete_Syntax_Zoo_Value_Flow
[15] epsilonlabs.wiki.sourceforge.net/EuGENia
[16] jamopp.inf.tu-dresden.de

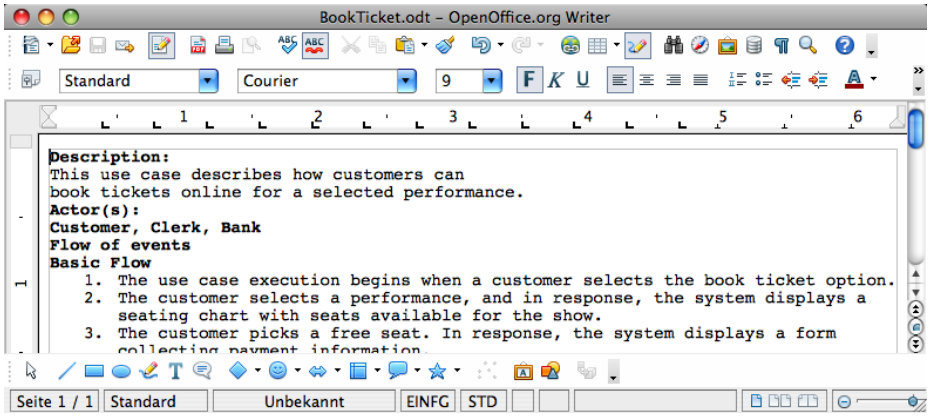

Fig. 4. Book Ticket use case define as OpenOffice document

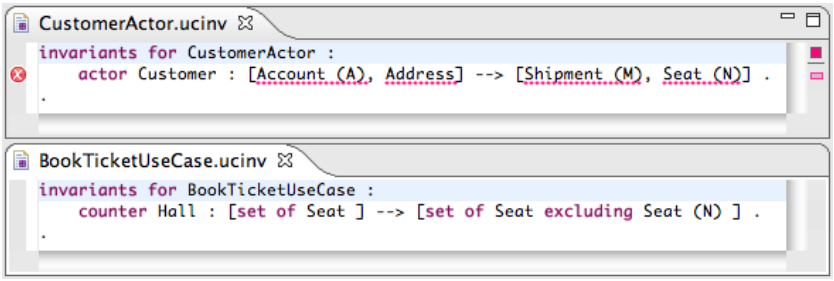

Fig. 5. Value added invariants for actor *Customer* (top) and counter actor *Hall* (bottom)

this information explicit, the actors were marked with a special style called `Actor` in the OpenOffice document.

As illustrated in Fig. 3, for each actor occurrence a fragment instance is derived (1–3). This is expressed by an OCL expression (indicated in Fig. 3) that states that each instance of `SpanType` where the `StyleName` corresponds to `Actor` leads to a fragment instance. Further details, for example where in the repository the fragments for the instances are expected, are defined by additional OCL expressions (not shown).

When we put the OpenOffice documents alone into Reuseware's fragment repository, the system will complain that no fragments for the fragment instances *Customer*, *Clerk* and *Bank* are found on L2. Thus, the developer is directly informed that the invariant model fragments for these three actors need to be defined, which we did (cf. Fig. 3 (4–6)). Figure 5 (top) shows the invariant specification for *Customer*.

With this procedure, we do not only avoid that the parts of the system's composition that are already defined on L1 need to be re-defined on L2, we also ensure that the compositions defined on L1 are honored on L2. This leads to the following first observation.

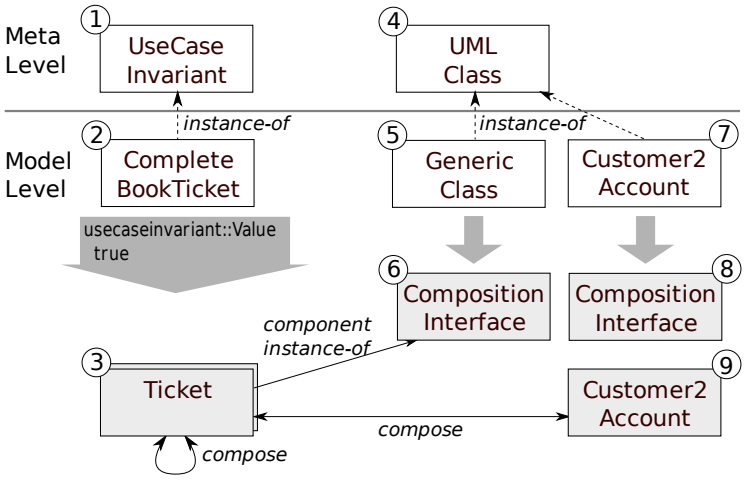

Fig. 6. Selected refinements between L2 and L3 (cf. Figure 1)

Observation 1 (Architecture consistency) *The architecture defined on the abstract layer can be transferred to the concrete layer by extracting it from the abstract layer as a composition that is applied on the concrete layer.*

As stated in the description of the UseCaseInvariant language, the amount of values in the system must be balanced. Sometimes however, values are not directly passed between actors, but are managed by another entity not captured by the use case description. This entity is called `CounterActor` in [9] and in our UseCaseInvariant language. The value *Seat* that a *Customer* receives (cf. Fig. 5 top) for instance is not provided by another actor. Therefore, the counter actor *Hall* (cf. Fig. 5 bottom) is introduced as additional model fragment that is added through an additional compose relation (cf. Fig. 3 (7–9)).

Observation 2 (Architecture extension) *New concerns that arise by lowering the abstraction, and not by changing requirements themselves, can be added through new compose relations. Thus, extending the details about the system's architecture.*

3.2 Creating Class Models from Use Cases

The next step in our development process is to define the data model for the applications (L3 in Fig. 2). [9] informally defines an algorithm to derive a class model for this from use cases with value added invariants. The such derived class model however, requires further refinement in the form of additional associations that can not be derived directly from the use case knowledge.

The first step, deriving an initial class model from use cases, can be performed similar as the refinement in the last section. Now the complete use case invariant model—that is, the complete model produced by executing the compose relations between UseCaseInvariant fragments—is used to derive fragment instances as shown in Fig. 6 (1–3).

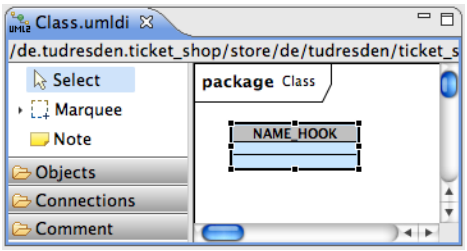

Fig. 7. Generic class model fragment where the name (`NAME_HOOK`) is set during composition

The derivation rules dictate, for example, that for each `Value` a class fragment is instantiated. Since here no additional information for individual classes is required, one generic class fragment (shown in Fig. 7) can be reused—Fig. 6 (4–6).

Observation 3 (Reuse for reoccurring patterns) *Reoccurring patterns on the concrete layer can be created by instantiating one fragment several times in the abstract layer.*

However, when we want to reuse such a fragments several times, it needs to be configured with information from the abstract layer. In our concrete case, the name of each class should be similar to the name of the actor or value it is instantiated from. Reuseware supports the setting of such attributes in a composition through the composition interface.

Observation 4 (Data passing between abstraction layers) *Data is passed from the abstract to the concrete layer by deriving attribute settings for fragment instances that are injected into the corresponding fragments during composition through their composition interfaces.*

In a second step, additional associations are added to the class model. These can be modeled as separate fragments and added through additional compose relations (cf. Fig. 6 (7–9)).

The class model is a place where—until now completely separated—concerns need to be composed. The two use cases, which are developed independently up to now, both influence the data model. With our approach, this is possible, since the set of fragment instances and compose relations can be derived from different models. In this process, reoccurring information (e.g., similar values used in both UseCases) can be joined.

Observation 5 (Composition of concerns) *While SoC is important, concerns eventually have to interact to work together in one system. The places where concerns are joined are explicit in our approach.*

3.3 Adding Data Flow

Another refinement introduced in [9] is the utilisation of statecharts to model the actual flow of values between actors. We realized this concept in the ValueFlow language introduced above.

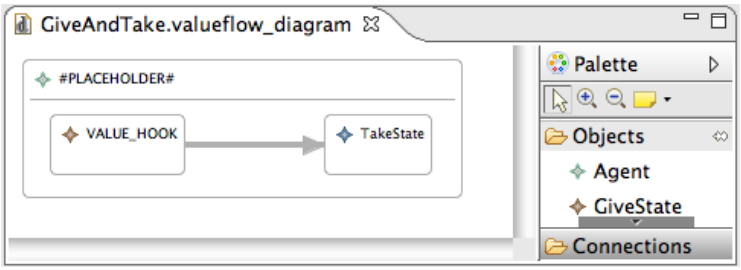

Fig. 8. Generic value flow model fragment containing two states between which a value is passed

The flows are defined in parallel to and independent of the data model on L3. Again, fragment instances and compositions are derived from the use case invariant model on L2. Again, a generic fragment—this time modelling states in the ValueFlow language—can be reused (Fig. 8 shows this generic fragment). The transitions between the states—similar to the additional associations in the UML model—have to be added as additional fragments.

3.4 Refining to Code

The final step in our demonstrator MDSD process is the refinement to code. All information that is still missing (e.g., user interfaces) is added on the code level. Surely, more parts of the system could be modelled (e.g., using an UI modelling language). However, for this demonstrator, we refrain from adding additional languages or abstraction layers.

Since we regard the Java code as another model, this refinement is not different to the others. This time, however, we have to consider several models from which the complete set of fragment instances and compose relations is derived—because concerns need to be composed at this point. These models are all models from L3 (UML class model and both ValueFlow models). This works similar as described in Sect. 3.2, where the data model is derived from both use case models. The fact that the models are defined in different languages now (UML and ValueFlow) is no problem—Reuseware only requires different OCL expressions to handle the different languages.

Observation 6 (Composition of concerns defined in different languages) *Since our approach builds on common language-independent composition concepts, also concerns formulated in different languages can be composed explicitly when refined into a common language.*

The classes from the UML model are transferred to Java in the usual way. For the states in the flow model, handler classes are produced with empty methods, (to provide and consume values), which are later filled by composing in additional Java statement.

In Java, startup code, provide and consume functionalities—partly containing UI code if user interaction is required—are implemented in separate fragments and composed in through additional compose relations.

This section exemplified the use of composition systems to improve SoC, and with it consistency, in MDSD processes. We highlighted the different benefits gained by this approach on examples from a concrete development process.

4 Related Work

Several works tackle the SoC problem in MDSD. One prominent work is the one of Jacobson [10]. It proposes an approach for SoC from use cases to class models. It builds on aspect-oriented programming techniques and concepts. However, parts of that approach involve manual transformations (i.e., following guidelines rather than applying automated model transformations). The approach is limited to a fixed MDSD process heavily involving UML. Thus it does not include other languages and also not the platform code itself. The Hyperspace Approach [11] first introduces Multi-Dimensional Separation of Concerns. Our utilisation and combination of different composition systems very much corresponds to this idea. Each composition system describes one dimension of concern separation. Thus, we took the ideas from [11] to the software modelling domain and provided a language independent implementation in Reuseware—in contrast to [11] which provides the Java-based implementation Hyper/J and is therefore quite code-centric. We in particular emphasized the topic of concern composition—the fact that, even if concerns are separated, they have to be composed at some point to define an integrated system—-which is neglected by many authors discussing SoC but was already highlighted as an imported point in [11] (*It is appealing to think of many concerns as independent or "orthogonal," but they rarely are in practice.* [11]).

Extracting as much information as possible from all artifacts in an MDSD process is an appealing idea. Doing so, however, blurs the borders between problem and solution domain. In our demonstrator, for instance, use case documents, traditionally problem descriptions, were already considered as part of the system design. We agree with Aksit [12] who defines a *concern-oriented design process*, which is, like MDSD processes, a layered process. He takes the viewpoint that the abstract layer defines a *problem* and the *concrete* layer defines the solution. In the next step, the previous solution becomes the new problem and so on. This can be put into relation to our approach: The abstract layer defines compositions of component instances (we formulate the required interface; or the problem). On the concrete layer, we then search for the components conforming to the component instance (we search for the solutions of the problems).

Our demonstrator development process does not use any traditional model-to-model or model-to-text transformations. Many such approaches exist (see [13] for an overview of the most prominent). This does not mean that such transformations can not be combined with our approach. Any model fragment that is input to a composition in our approach could be produced by a transformation. Thus transformations can be integrated where feasible. We were aiming at identifying the structure of the composition (or the architecture) of the system under development. Thus we replace monolithic model transformations that do "something" when refining models by well structured compositions. Finally, our compositions can also be seen as model transformations with specific restrictions embracing the ideas of encapsulation and concern separation from component-based development.

The idea of using CBSE concepts in MDSD and the advantages gained could probably also be realized with other technologies than Reuseware. However, Reuseware's strength is that composition systems can be specified for arbitrary languages and that these composition systems are very flexible with respect to the concern dimension they realize (i.e., compositions can be cross-cutting or homogeneous). Other methodologies would have to bring similar properties. Candidates could be [14] or a combination of approaches presented in [15].

5 Conclusion

In this paper we presented an approach that improves SoC in MDSD processes. This approach is based on Reuseware composition systems. It was presented on a demonstrator process with which a system was developed starting from use case documents and resulting in Java code.

Applying the approach, changes to the system's models can be performed at any abstraction layer and at any point during development. This is because 1) a certain amount of information is hidden behind composition interfaces and can thus safely be changed without breaking the overall system composition 2) changing information that influences the composition interfaces and breaks the overall system composition can still be handled since Reuseware reports to the developer where the composition fails after the change.

A drawback is that all changes have to be made in the original artifact. That is, the model fragment where the information was initially defined. Changing a composed model will have no effect since any change to the system results in a re-composition of those composed model. This can become problematic in large systems, where it might be hard to locate the source of a certain piece of information. To overcome this drawback, tracing techniques can be applied and back-propagation can be utilised to allow changes to composed models which are then propagated to the information source. In fact, we already accomplished this for fragment composition in a single-language setting [16]. In the future, this work can be extended to be applied in full MDSD processes as well. Then one can imagine a system, where the composition systems are almost hidden to the developer, but are used in the background to keep information organized and concerns separated.

In the demonstrator system, we did not utilise traditional model transformations. Instead, composition system specifications for Reuseware have to be provided. Hence, the effort that usually goes into specifying the transformations can be used for composition system specifications. We believe, that these specifications, once defined for a certain process, are highly reuseable in other processes that employ the same languages (e.g., the composition systems that utilise UML and Java in the demonstrator system). Thus, the effort when defining new processes decreases once a library of composition systems is established. Evaluating this is however future work. We plan to realize more MDSD processes, including processes with industrial background, with our approach in the future.

Acknowledgement

This research has been co-funded by the European Commission in the 6th Framework Programme project MODELPLEX contract no. 034081 (cf. `www.modelplex.org`).

References

1. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
2. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12) (1972) 1053–1058
3. Henriksson, J., Heidenreich, F., Johannes, J., Zschaler, S., Aßmann, U.: Extending grammars and metamodels for reuse: the reuseware approach. IET Software, Special Issue on Language Engineering **2**(3) (2008) 165–184
4. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On language-independent model modularisation. In: Transactions on Aspect-Oriented Software Development VI. Volume 5560 of LNCS., Heidelberg, Springer (2009)
5. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd Edition. Pearson Education (2008)
6. Object Management Group: MOF 2.0 core specification. OMG Document (2006) URL http://www.omg.org/spec/MOF/2.0.
7. Aßmann, U.: Invasive Software Composition. Springer, Secaucus, NJ, USA (2003)
8. Henriksson, J.: A Lightweight Framework for Universal Fragment Composition—with an application in the Semantic Web. PhD thesis, Technische Universität Dresden (2009)
9. Roussev, B., Wu, J.: Transforming use case models to class models and ocl-specifications. Int. Journal of Computers and Applications **29**(1) (2007) 59–69
10. Jacobson, I.: Use cases and aspects-working seamlessly together. Journal of Object Technology **2**(4) (2003) 7–28
11. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the hyperspace approach. In: Proceedings of the Symposium on Software Architectures and Component Technology, Kluwer (2000)
12. Aksit, M.: The 7 c's for creating living software: A research perspective for quality-oriented software engineering. Turkish Journal of Electrical Engineering & Computer Sciences **12**(2) (2004) 61–95
13. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. (2003)
14. 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 (2004) 36–45
15. Schauerhuber, A., Retschitzegger, W., Kappel, G., Kapsammer, E., Wimmer, M., Schwinger, W.: A survey on aspect-oriented modeling approaches. Technical report, Johannes Kepler Universität (JKU) Linz, Linz, Austria (2006)
16. Johannes, J., Samlaud, R., Seifert, M.: Round-trip support for invasive software composition systems. In: International Conference on Software Composition 2009. Volume 5634 of LNCS., Heidelberg, Springer (2009) 90–106