# Concern-based (de)composition of Model-Driven Software Development Processes

Jendrik Johannes* and Uwe Aßmann

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

**Abstract.** An MDSD process is often organised as transformation chain. This can threaten the Separation of Concerns (SoC) principle, because information is replicated in, scattered over, and tangled in different models. Aspect-Oriented Software Development (AOSD) supports SoC to avoid such scatterings and tangling of information. Although there are integrations of MDSD and AOSD, there is no approach that uses concern separation for all artifacts (documents, models, code) involved in an MDSD process as the primary (de)composition method for the complete process. In this paper, we propose such an approach called ModelSoC. It extends the hyperspace model for multi-dimensional SoC to deal with information that is replicated in different models. We present a ModelSoC implementation based on our Reuseware framework that organises all information provided in arbitrary models during development in a concern space and composes integrated views as well as the final system from that. This is shown on the development of a demonstrator system.

## 1 Introduction

Model-Driven Software Development (MDSD) aims at decreasing the effort and costs for developing complex software systems. This is achieved by reusing information that is captured in artifacts (documents, diagrams, etc.) that are created during the development of such a system. These artifacts are regarded as *models* of the system and are integrated by means of transformation and composition. By this, the final system is eventually generated—without the need to manually implement the parts of the system that are already defined in models.

MDSD profits from the OMG's metamodelling standards MOF and OCL and their adoption in technologies such as the Eclipse Modelling Framework (EMF) [1]. These technologies include language-independent tools (e.g., model transformation engines) and meta tools for creating Domain-Specific Modelling Languages (DSMLs). The standards and tools allow not only for cost-efficient engineering of new DSMLs but also for the seamless integration of these DSMLs

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

into MDSD processes. Hence, MDSD processes can be tailored with languages suitable for the task at hand and domain experts can participate in development directly. Ultimately, however, the models defined in DSMLs have to be transformed into an implementation to make use of the information they contain. This is often performed stepwise by transforming them into more detailed models and refining these before generating code, which might require further refinement.

In an MDSD process that is organised as transformation chain, replication, scattering and tangling of information can occur when information is repeatedly extracted and distributed over different models. This is the source of traceability and consistency problems which are often tackled by additional techniques on top of model transformations (e.g., publications in [2]). The problem can also be regarded as a Separation of Concerns (SoC) issue as investigated in the area of Aspect-Oriented Software Development (AOSD) [3]. Originally focused on programming languages, AOSD is now concerned with all parts of a software development process and thus there are a number of approaches that integrate ideas and technologies from MDSD and AOSD (e.g., publications in [4]). However, usage of AOSD in MDSD so far mostly focused on either specific parts of the development process (e.g., [5] or [6] for analysis/design; [7] or [8] for implementation) or offered a technology for model aspect weaving (e.g., [9]) without a SoC methodology. Although different approaches can be combined to use AOSD methods in all phases of MDSD processes, there exists, to the best of our knowledge, no approach and no universal technology that organises an MDSD process by concern (de)composition rather than transformation chain (de)composition.

In this paper we present such an approach—the ModelSoC approach as an extension of the hyperspace model for multi-dimensional separation of concern defined by Ossher and Tarr [10]—and a supporting technology. The hyperspace model is well suited as a base for ModelSoC, since it (a) explicitly supports different dimensions for decomposition which is needed because in different DSMLs information is decomposed along different dimensions and (b) is independent of a concrete language (i.e., not limited to e.g. Java) or a concrete language paradigm (i.e., not limited to e.g. object-oriented languages). The approach is implemented in our Reuseware Composition Framework [11].

Our contribution is thus twofold. First, we introduce ModelSoC as an extension of the hyperspace model that can handle replication of information in different formats and usage of DSMLs for composing information. Second, we introduce Reuseware as a framework technology that enables usage of ModelSoC in practice in combination with existing Eclipse-based modelling tools. A comparable technology is not available for the hyperspace model itself that was implemented for Java only in the Hyper/J [10] and CME [12] tools.

The paper is structured as follows: In Sect. 2, we introduce an MDSD process as an example to motivate the need for ModelSoC which we describe in Sect. 3. Next, we discuss how the Reuseware framework is used to put ModelSoC into practice in Sect. 4. The motivating example is revisited in Sect. 5, where we show how it is realised with ModelSoC and discuss advantages of that. In Sect. 6, we discuss related work and conclude the paper in Sect. 7.

## 2 Motivating Example

In the following, we motivate our work using the model-driven development of a reservation system in which customers can book tickets and perform other related activities. This demonstrator system is inspired by an example from [13]. Figure 1a shows the process defined as model transformation chain. Five types of models[1] are used: OpenOffice use case documents (cf., [13] for structure details), UML use case models [14] annotated with invariants (as introduced in [13]), UML class models, Value Flow (a data flow DSML[2]) models and Java [15].

In the chain, information is transported between different models—i.e., different *views* on the system—by model transformations. This is not only necessary to integrate all information into the Java code at the end of the chain, but also to connect new information in different views to existing information. For example, the information about actors participating in use cases, initially defined in OpenOffice documents, is needed in all viewpoints, since it is central information around which other information is defined.

An example of adding information is the refinement of a UML use case model, which we illustrate on the example of the use case *BookTicket* in Fig. 1a. Following [13], UML use cases are annotated with *value added invariants*. This means, that we define values (i.e., business objects) for each actor it holds before and after the execution of the use case. This is done by annotating the actors (not shown in Fig. 1). The total numbers of values in a use case needs to be invariant (i.e., a value that exists before use case execution needs still to be there after execution and a value can not appear out of nowhere). According to [13], violations of invariants may occur if there are actors that are not visible from outside the system (e.g., a passive storage that can not be observed as acting entity from the outside). They propose to add such actors in the UML use case view (but not to the use case documents). In the example of the *BookTicket* use case (cf., [13]), the actor *Customer* has an Account and an Address before execution of the use case and a Shipment and a Seat after execution. The Address is passed to the *Clerk*, while the Account is passed to the *Bank*. The Shipment is received from the *Clerk*. For the Seat however, there is no actor yet owning it before use case execution. Therefore, the new actor *Hall*, which owns the Seat before use case execution, is introduced in the UML use case view (Fig. 1a middle).

Although the transformation chain approach enables us to add information to intermediate models (e.g., adding the *Hall* actor) it has a couple of drawbacks that are often a source of criticism on MDSD. First, once an intermediate model is refined manually, it cannot be recreated automatically, which essentially forbids modifications of models prior in the chain what leads to model inconsistencies that have to be fixed manually. For example, after we added *Hall* in the UML use case view, we cannot change the OpenOffice document and reexecute the transformation to UML use cases without loosing the information about

[1] we treat all artifacts in an MDSD process as models, including documents and code; for each model, a metamodel is available

[2] `http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Value_Flow`

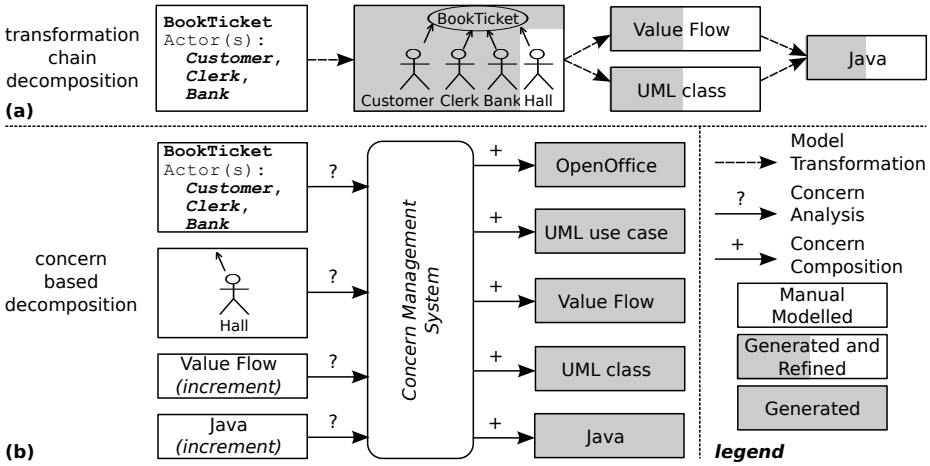


**Fig. 1.** MDSD process organised by: (a) transformation chain (b) concern separation

*Hall*. Second, similar information is extracted several times in the chain, which requires additional effort when defining the transformations. For example, the information about actors is extracted first from the OpenOffice document, then from the UML use case model (by two transformations) and finally from both Value Flow and UML class models to produce the Java code. Third, information is difficult to localise. When a particular information about the system has to be changed, one needs to know, where the information was defined. For example, if one discovers in the Java code that something about *Hall* should be changed, it is difficult to trace if *Hall* was introduced in the OpenOffice or the UML use case view, because the information has been transformed several times.

To overcome these problems, we propose a different way of organising an MDSD process that is illustrated for the example in Fig. 1b. Instead of refining generated models, we propose to define additional information in separate models (Fig. 1b left). Here, the *Hall* actor is defined in a UML model, but separate from the generated UML use case model. All information about the system is organised in a central place—the concern management system—(Fig. 1b middle). Integrated views can be produced by selecting the information required from the pool of all information, transforming it into the correct format and composing it into an integrated model (Fig. 1b right). Instead of decomposing the MDSD process along the transformation chain, the information that is modelled is decomposed and organised w.r.t. the *concerns* that are in the system. For this, the concern management system has to analyse all information from different models to identify concerns and the relationships between them (concern analysis in Fig. 1). Furthermore, the system needs to be able to compose the information to produce different integrated views (concern composition in Fig. 1). In the next section, we present the ModelSoC approach for multi-dimensional concern separation with which concern management systems for MDSD processes can be designed. An implementation of the approach is presented in Sect. 4 that provides the tooling to define and use concern management systems in Eclipse.

## 3 ModelSoC for Multi-Dimensional SoC in MDSD

In this section, we introduce ModelSoC: an approach to design MDSD processes based on concern separation and composition.

The hyperspace model for SoC of Ossher and Tarr [10] supports the decomposition of a system into *concerns* along several *concern dimensions*. These span up an n-dimensional space—a *hyperspace*—where n is the number of utilised concern dimensions. Implementation artifacts are perceived as consisting of *units*. Each unit realises at most one concern in each concern dimension which is defined in a *concern mapping* that maps units to concerns. Units are composed into *hyperslices* and *hypermodules*. A hyperslice composes all units belonging to one concern. Hypermodules compose several hyperslices to a complete software module or program. The hyperspace model leaves it open, what exactly can be treated as unit, how concern mappings and hypermodules are specified and how the actual composition of a system from units is performed. The implementation Ossher and Tarr provide is Hyper/J [10], which supports different types of Java fragments as units (e.g., statements). It contains a dedicated language for concern mapping as well as a language for hyperslice and hypermodule definitions. Such definitions do not only enumerate concerns to compose, but also include calls to operators for composing Java units by bytecode weaving.

Since the conceptual hyperspace model is independent of an implementation language, it can be used to separate concerns in models in an MDSD process, as the one shown in Sect. 2. As concern, we regard a piece of information about the system which does not require further decomposition. Examples of concerns are (a) *Customer participates in Book Ticket* (b) *Bank participates in Book Ticket* or (c) *Account is exchanged between Customer and Bank*. Concerns that follow the same (de)composition concept belong to the same concern dimension. For instance, a new actor is composed into a use case by adding it to a use case model. This works similar for all actors, thus both concerns (a) and (b) belong to the concern dimension *Participation*. On the contrary, a new value exchange is added by stating for a value which actor owns it before and which after use case execution (cf., *value added invariant* in Sect. 2). Thus, (c) follows different (de)composition rules and belongs to another concern dimension (*Exchange*).

We identified three properties of MDSD that are difficult to map to the hyperspace model as it is, because they are either not considered by the model or further refine parts of the model that were *left open* [10]. First (Sect. 3.1), the hyperspace model does not consider that the same information may be present in multiple formats (e.g., same class in UML and Java). Second (Sect. 3.2), automated transformation of information is not covered by the hyperspace model. Third (Sect. 3.3), a refinement of the hyperspace model is that we want to forbid fixed concern mapping or hypermodule specification languages (as it is the case in Hyper/J). This is, because languages that are included in an MDSD process should be chosen based on the needs of the system's domain, which would not be given if a technology enforces the inclusion of a predefined mapping or module language. Therefore, ModelSoC introduces the following three concepts as extensions and refinements of the hyperspace model.

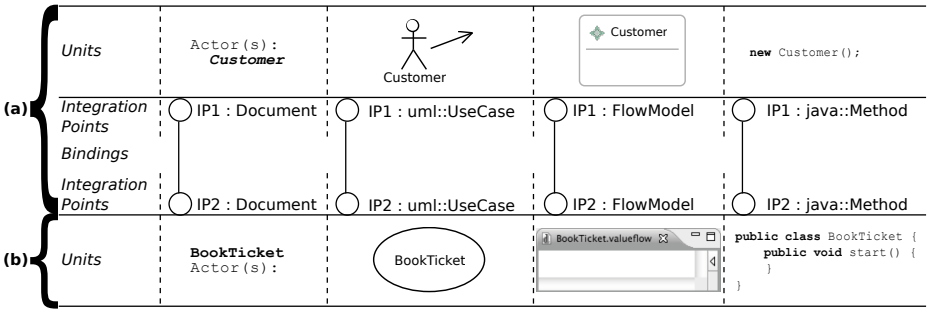


**Fig. 2.** Multi-format units: (a) *Customer participates in Book Ticket* (b) *Book Ticket*

### 3.1 Multi-Format Units: Realisation of concerns in different formats

A fragment of a model that represents a certain concern is a *unit* (in the sense of [10]). Since there may be different viewpoints on the same concern in MDSD (e.g., in Fig. 1, information about actors and use cases is present in multiple views) there can be several units representing the same concern in different formats. We introduce the *multi-format unit* concept that bundles such units. As an example, consider Fig. 2 that shows the multi-format unit that realises the concerns *Book Ticket* and *Customer participates in Book Ticket*. To obtain a view that shows certain concerns, one has to select a viewpoint (e.g., UML use case) and if the format of the viewpoint is supported by all multi-format units that realises the corresponding concerns, the view can be composed. Hence, each multi-format unit supports a set of viewpoints, but not each multi-format unit needs to support all viewpoints used in an MDSD process. The viewpoint that is the final system (Java in the example), is most likely supported by all multi-format units. If support for a new viewpoint is needed for a multi-format unit, it can be added to without altering the existing units (i.e., existing viewpoints) in the multi-format unit (see also Sect. 3.2 below).

A multi-format unit offers *integration points* and *bindings* between them to guide concern composition. Each unit has to support the integration points. This is comparable to the joinpoint concept [16]. Each unit offers its own version of the points typed by metaclasses of the metamodel for the unit (cf., Fig. 2). Thus, integration points can be defined on the metamodel of a unit (cf., Sect. 3.3 below). Since a unit may need integration along several concern dimensions, the set of integration points is not fixed but can be extended, when a new concern dimension needs to be supported. For example, the concern *Book Ticket* in the concern dimension *UseCase* (Fig. 2b) can exist on its own without the need for integration points because the concern dimension *UseCase* is independent of other concern dimensions. The concern *Customer participates in Book Ticket* in the concern dimension *Participation* (Fig. 2a) requires integration with *Book Ticket* since the *Participation* depends on the *UseCase* dimension. Therefore, the multi-format unit for *Customer participates in Book Ticket* defines integration points for its own units and the units of *Book Ticket* and binds them.

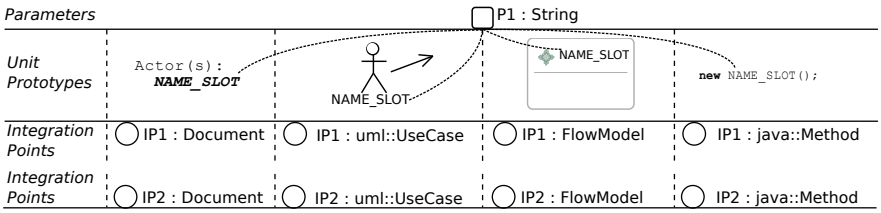


**Fig. 3.** Multi-format unit prototype for *Participation* concern dimension

### 3.2 Multi-Format Unit Prototypes: Automating unit creation

It is not practical to create each unit of a multi-format unit manually. Rather, following the MDSD idea, units holding the same information should be created automatically. We note that different concerns of the same dimension have similar structure. For example, the multi-format unit realising the concern *Bank participates in Book Ticket* looks similar as the multi-format unit of *Customer participates in Book Ticket* shown in Fig. 2a (only the string `Customer` should be exchanged for `Bank` in each unit).

We can use this similarity to abstract a multi-format unit to a *multi-format unit prototype* that defines a common structure for all concerns of one concern dimension. For this, we create one *unit prototype* for each format supported by the multi-format unit prototype. A unit prototype is a small template, that offers parameters for the parts that differ between units of the same concern dimension.

Figure 3 shows the multi-format unit prototype for the *Participation* concern dimension. It has one parameter *P1* for the actor name. A multi-format unit prototype can be instantiated to a multi-format unit by binding each parameter with a value (e.g., *P1* can be bound to `Customer` in Fig. 3 to obtain Fig. 2a) and integration points with each other. Integration points exist in a multi-format unit prototype, but not the concrete bindings, because these integrate individual concerns. Since the parameter *P1* is of the primitive type String, it is similar for all units. A parameter may also have a complex type that can differ for different views. In that case, different versions of a value are needed to bind the parameter. Integration points are always individually bound for each unit.

### 3.3 Meta-level concern mappings and compositions

Figure 4 shows details of a concern management system for MDSD processes (cf., Fig. 1b) using the concepts introduced above. The figure illustrates that each concern dimension (a) has a multi-format unit prototype (b). The instantiation of prototypes (c), which includes binding of parameters and integration points between individual concerns, spans up the concern space (d). Having this space available, a viewpoint can be selected (e) which reduces multi-format units to normal units (f). By interpreting the bindings, these units can be composed (g) to an integrated view in the selected viewpoint (h). Such a view corresponds to a hyper module in the sense of [10].

Once the concern space has been established, steps (d) to (h) can be performed by a universal composition technology for any concern management sys-

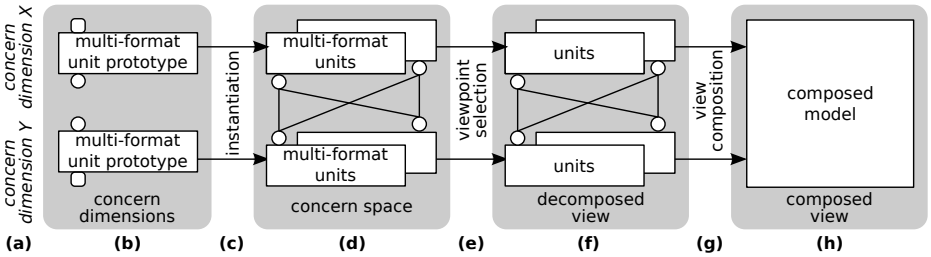


**Fig. 4.** Structure of a concern management system for an MDSD process (cf., Fig. 1b)

tem. Steps (a), (b) and (c) however, which cover the concern analysis phase (cf., Fig. 1b), require configuration for each MDSD process. Concretely, mechanisms are required to (a) define concern dimensions as well as integration points and parameters of the concerns in the dimensions, (b) define multi-format unit prototypes with integration points and parameters, (c) define how parameter and integration point binding information is extracted.

(a) Concern dimensions, concern parameters and concern integration points can be defined independently of models and metamodels for an MDSD process.

(b) Multi-format unit prototypes can be defined by modelling each unit prototype as a model fragment in the corresponding modelling language using an existing model editor (cf., Fig. 3). Parameters and integration points can be specified for each unit prototype based on the prototype's metamodel. For example, the rules for *IP1* must state that in the UML use case format an actor is connected to a use case by adding it to the use case's `uml::Package` and for the Java format by adding the actor instantiation statement to the `java::Method` realising the use case execution. Rules for the parameter *P1* must state that `uml::Actor.name` represents the parameter in the UML format and `java::Variable.name` represents it in the Java format. These rules effectively define the concern mappings. By assigning the model fragments and rules that make up a multi-format unit prototype to the corresponding concern dimension defined in (a), we know to which dimension the instances of the prototype belong.

(c) Concern composition information is available in the user-defined models (Fig. 1b left). In the example, the information that *Customer*, *Bank* and *Clerk* participate in *BookTicket* is given in the BookTicket OpenOffice model and that *Hall* participates in *BookTicket* is given in the Hall UML model. This information can be extracted by rules based on the metamodels of the languages used. For instance, one rule must specify that each OpenOffice document instantiates the multi-format unit of the *UseCase* concern dimension and parameterises it with the name of the document. Furthermore, another rule must state that each mention of an actor in the document instantiates the multi-format unit prototype of the *Participation* dimension (cf., Fig. 3) and composes it with the corresponding use case. This effectively forms the composition script for hypermodules. By assigning the rules for extracting the concern instantiation and composition information to a concern dimension defined in (a), we know which multi-format unit prototype to instantiate, which integration points to address and which parameters to fill with the extracted information.

## 4 Implementation based on Reuseware

In this section we describe the implementation of ModelSoC with Reuseware [11]. Reuseware is an Eclipse-integrated tool that implements Universal Invasive Software Composition for Graph Fragments (U-ISC/Graph). ISC [17] is a program transformation approach that offers basic composition operators which can be combined to support the composition of different types of components. It is suited to realise complex and cross-cutting composition needs that arise when compositions along multiple dimensions are performed. With U-ISC [18], ISC-based composition systems can be created for arbitrary grammar-based languages with tooling that executes composition by merging syntax tree fragments. In U-ISC/Graph, we extended U-ISC to support languages defined by metamodels and work with graph fragments (i.e., model fragments). The first part of this work was presented in [19], where we only considered single modelling languages out of context of an MDSD process. In this paper, U-ISC/Graph is used to its full extent to realise ModelSoC. With U-ISC/Graph, one can define for an MDSD process (cf., Sect. 3.3): (a) concern dimensions with integration points and parameters, (b) multi-format unit prototypes and (c) concern composition rule extraction. These definitions are interpreted by Reuseware which then acts as concern management system for the MDSD process that can be used inside Eclipse in combination with Eclipse-based model editors.

(a) In Reuseware, we can specify *composition systems* that we use to represent concern dimensions. Listing 1 (cf., [11] for notation) shows the composition system for the *Participation* concern dimension. The specification declares `fragment roles`, which we use to model the types of concerns of a dimension. Listing 1 declares the fragment roles `Participant` (which can be an actor) and `Collaboration` (which can be a use case). A fragment role holds a set of `static ports` that we use to declare integration points (`IP1` and `IP2` in the example) and parameters (`P1` in the example). Ports can be connected with `associations`, which define whether two ports are compatible (`IP1` and `IP2` in the example).

(b) To put a composition system into use, it needs to be connected to a modelling language. This is done by specifying a *component model* that connects the concepts of the composition system with the concepts of the language using its metamodel (as shown in Listing 2; cf., [11] for notation). To define multi-format unit prototypes, we define each unit prototype in its languages using a suitable existing model editor and then use component models to identify parameters and integration points and relate them to a composition system and therewith to a concern dimension. For this, each port (i.e., parameter and integration point) is mapped to metaclasses of the metamodel, as shown for UML in Listing 2. The keywords `slot`, `hook`, `anchor` and `prototype` represent different types of addressable points in model fragments that are modified when fragments are invasively composed (cf., [19] for details). When these mapping rules are applied to the multi-format unit prototype of the *Participation* and *UseCase* dimensions (cf., Fig. 3), the parameters and integration points are identified.

(c) Component model specifications define where fragments can be integrated. On the contrary, *composition language* specifications help with defining

which fragments are integrated. A composition language specification (shown in Listing 3; cf., [11] for notation) allows to use a modelling language as composition language. That is, it contains rules to extract information from models on which fragments are parameterised and integrated. For instance, Listing 3 defines that for each part of an OpenOffice document marked with the `SpanType` *Actor*, a new actor fragment is instantiated with the `P1` parameter bound to the name of the actor (extracted from the document with OCL query `self.mixed->at(1).getValue()`; Line 5). Furthermore, we define that the binding of `IP1` and `IP2` is performed between the correct `Participants` and `Collaborations` by extracting the corresponding actor and use case names from the model and its ID (Lines 10/11).

With specifications given for all concern dimensions and viewpoints of a MDSD process, Reuseware acts as the concern management system (cf., Fig. 4) for that process. For this, Reuseware interprets the specifications to derive a *composition program* that represents the complete concern space (d) by showing concerns and relations between them as parameterised and linked unit prototypes. In fact, Reuseware can visualise composition programs graphically with a notation similar to the one used in step (d) in Fig. 4 (boxes with attached circles and lines between them; cf., [19]). To execute a composition program, Reuseware automatically selects suitable composition operators based on the metamodel of the involved units and therefore no further configuration is required for (e) to (h). Reuseware automatically executes the composition for all supported viewpoints by creating a composed model for each. The developer can decide at which view to look by opening a composed model in an editor of his or her choice.

```
1 fragment role Participant {                    fragment role Collaboration {
2     static port IP1; /*an integration point*/      static port IP2; /*an integration point*/
3     static port P1;  /*a parameter*/
4 }                                              }
5 contributing association Participation { Participant.IP1 --> Collaboration.IP2 }
```

**Listing 1.** (a) Composition system for concern dimension *Participation*

```
1 fragment role Participant {                    fragment role Collaboration {
2     port IP1 {                                     port IP2 {
3         uml::Actor is prototype {}                     uml::Package.packagedElement is hook {}
4         uml::Association is prototype {}
5         uml::Association.ownedEnd is slot {}           uml::UseCase is anchor {}
6     }                                              }
7     port P1 {uml::Actor.name is value hook {}}
8 }                                              }
```

**Listing 2.** (b) Parameter and integration point spec. for UML unit of *Participation*

```
1  fragment role Participant {
2      odftext::SpanType if $styleName = 'Actor'$ {
3          fragment = $'Participant:'.concat(self.mixed->at(1).getValue())$
4          ID = $Sequence{'lib','participation','Participant.'.concat(view)}$
5          port P1 { $self.mixed->at(1).getValue()$ }
6      }
7  }
8  association Participation {
9      odftext::SpanType if $styleName = 'Actor'$ {
10         fragment = $'Participant:'.concat(self.mixed->at(1).getValue())$ -->
11         fragment = $'UseCase:'.concat(ID.trimExtension().segment(-1))$
12     }
13 }
```

**Listing 3.** (c) Concern composition (*Participation*) extraction rules for OpenOffice

## 5 Example Realisation and Discussion

To evaluate ModelSoC, we (1) defined the MDSD process introduced in Sect. 2 with Reuseware and (2) used it to develop a first version of the ticket shop system with the features *book ticket* and *change seat*. Afterwards, (3) we extended the MDSD process defined in (1) with a new viewpoint and concern dimension for security and used that to define security properties of the ticket shop system without changing the models defined in (2).

**(1) MDSD process definition** Our MDSD process supports five different viewpoints as introduced in Sect. 2. Four of these viewpoints are used for refinement (i.e., manual modelling), while the UML class viewpoint is only for analysis (i.e., it gives an overview of all classes that appear in Java code, but does not support modification). We identified 11 concern dimensions that we defined as *composition systems* (cf., Listing 1). The average size of these specifications is 16 LOC. Each viewpoint can show concerns of certain concern dimensions as presented on the left side of Fig. 5. To add support for a concern dimension to a viewpoint, a unit prototype (created with a normal model editor) and one component model (cf., Listing 2) was defined (23 in total; average size 26 LOC). Four viewpoints support editing of concerns (i.e., instantiation of unit prototypes) shown on the right side of Fig. 5. To add editing support for a concern dimension to a viewpoint, a composition language specification (cf., Listing 3) was written (15 in total; average size 37 LOC). Note that, as expected in model-driven development, certain concerns are created automatically. For instance, a class is created for an actor as soon as it appears in some use case and therefore the OpenOffice viewpoint influences the class dimension. All marks on the right side of Fig. 5 that have no counterpart on the left side identify such situations. Also, some concerns are shown but are not editable in the corresponding viewpoint (e.g., actors can not be changed in Java). All marks on the left side that have no counterpart on the right side identify these.

If one wants to use ModelSoC for its own MDSD process, one has to write Reuseware specifications as discussed above. This is a metamodelling task which is done instead of writing model transformations. Compared to traditional model transformations, Reuseware specifications are very modular as indicated by the small number of LOCs of the specifications in this example. This is not because the demonstrator system we developed with the process (discussed next) is relatively small—the process itself can be used to develop larger systems.

**(2) MDSD process usage** Once the MDSD process is set up with Reuseware, developers can use existing model editors to edit and view different viewpoints. For composed views that are graphical, Reuseware also performs layout composition [20] in addition to composing the semantic information. Preserving layout between views helps developers to relate views to each other. Because of traceability issues, such layout preservation is often not well supported in transformation chain MDSD processes. Furthermore, developers can make mistakes which lead to inconsistencies that are discovered by Reuseware (e.g., use UML to add an actor to a use case for which no OpenOffice document exists). These errors are annotated to the source models of the error using Eclipse's error

| | usecase | particip. | exchange | flow | trigger | factory | class | dataclass | associate | typebind. | app | security | usecase | particip. | exchange | flow | trigger | factory | class | dataclass | associate | typebind. | app | security |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OpenOffice | x | x | | | | | | | | | | | x | x | | | | x | x | | | x | x | |
| UML use case | x | x | x | | | | | | | | | | | x | x | | | x | x | x | x | x | | |
| Value Flow | x | x | x | x | | | | | | | | | | | | x | x | | | | | | | |
| UML class | | | | | | | x | x | x | | | | | | | | | | | | | | | |
| Java | x | x | x | x | x | x[1] | x | x | x | x | x | x[2] | | | | | | | | | | | | |
| SecProp | x | x | x | | | | | | | | | x[2] | | | | | | | | | | | | x |

**Fig. 5.** Left side: viewpoints (y-axis) supported by concern dimension (x-axis); Right side: modelling languages used to model (y-axis) in concern dimension (x-axis) ([1]*factory concerns* have complex parameters defined for Java format only; [2]*security concerns* have complex parameters defined individually for SecProp and Java formats)

marking mechanism. If the model editor used supports this mechanism well, the developer will most likely understand the error. However, this is not the case for all editors and sometimes external editors (e.g., OpenOffice) are used. Therefore, improving tool support and integration for error reporting and debugging is part of future work. Realising the MDSD process following ModelSoC, we are able to track information that is replicated in, scattered over, and tangled in different integrated views. A drawback of our approach, as presented now, is that the integrated views can not be edited directly. Rather, small models are edited and the integrated views are created immediately for inspection. Since tracing—which is already used for layout preservation—is simple with the explicit concern space representation, we believe that editable views can be realised by using a round-trip mechanism that propagates changes from the integrated views back. Such a mechanism could even allow editing information in a different viewpoint as it was defined in. We presented first successful results in this direction in [21].

While we implemented only two features of a demonstrator system with the MDSD process defined above (1), this process can be used to continue development on this or other (possible much larger) systems. For this, no modification of the process setup from (1) is required. Also for other MDSD processes, parts from (1) can be reused due to high modularity—each concern dimensions can be reused individually. (1) can also be flexibly extended as discussed next.

**(3) MDSD process extension** To support our claim that ModelSoC supports flexible extension of an MDSD process, we extended our process (1) with a new concern dimension for security—after the two functional features were developed. Security is usually a cross-cutting concern that effects several places in a system. For modelling security information, we employed the DSML SecProp[3], to define access rights and encryption needs of the business objects in the system. This DSML was motivated by a DSML that was developed by an industrial partner in a case study in the Modelplex project [22]. There we applied our approach to add a security viewpoint to a system otherwise modelled with UML only. To allow the security modeller to see existing values that need security properties,

[3] `http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_SecProp`

the information from the concern dimensions *Usecase*, *Participation* and *Exchange* needed to be transported to the SecProp viewpoint. This was done by adding new unit prototypes (defined in SecProp) to the corresponding concern dimensions (cf., Fig. 5 bottom). A multi-format unit prototype for the security dimension was introduced supporting the SecProp and Java viewpoints. To allow integration with other dimensions, new integration points were added to the *Exchange* unit prototypes in SecProp and Java (by specifying a new component model; no changes to the existing specifications or Java fragments were required).

Due to the given space limitations, we cannot give more details about the example here. The complete example can be found online[4]. On the website, there are instructions on how to install Reuseware and the modelling tools used in the example. With this, all details of the example can be inspected by the interested reader. The site also contains visualisations of the concern space for the example.

## 6 Related Work

Many traditional SoC approaches are limited in so far that they can only be combined with object-oriented languages (role modelling [23], aspect-oriented programming [16] or Composition Filters [8]). In case of the hyper space model [10], only the implementation Hyper/J is limited to Java, but the model itself can be used with arbitrary languages. Thus, we used it as basis for ModelSoC.

More generic SoC approaches exist in the AOM [4] area. The Theme approach [5] for AO analysis and design works with UML and requirement specifications. However, it is limited to these and enforces the usage of a predefined specification language for *Themes* (i.e., concern mappings). More approaches that are limited to UML and do not consider other DSMLs are discussed in [24].

In the RAM [25] approach, different concerns are modelled in *aspects* where each aspect contains three views (structural, state, message) modelled with UML class, state and sequence models. Thus, an aspect in RAM can be seen as a multi-format unit supporting three different viewpoints, but no new viewpoints can be added which hinders the integration of DSMLs or GPLs such as Java.

RAM makes use of the AOM tools Kompose [9] and GeKo [26] that can be configured by metamodels and thus can be, similar to Reuseware, used with arbitrary DSMLs. They are specialised for each metamodel individually, while in Reuseware the composition system concept allows us to define concern dimensions and relate them to different metamodels which we required for ModelSoC.

AOSD with use cases [6] is related to our example where use case decomposition is one concern dimension. However, we support arbitrary dimensions.

A recent study [27] discusses whether aspect weaving should be performed on models or code. This is motivated by the fact that some approach perform model weaving ([9, 26]), while others offer translations to aspect (e.g., Aspect/J [7]) code ([5, 6]). In our approach, weaving is performed with Reuseware for any viewpoint. While weaving on code level is mandatory to obtain the final system, weaving for other viewpoints can be supported if it aids development.

[4] `http://www.reuseware.org/index.php/Reuseware_ModelSoC`

In MDSD, many approaches support model transformation and manipulation (e.g., QVT [28], ATL [29], Epsilon [30] or SDM [31]). They relate to ModelSoC in two ways: First, all approaches named above give possibilities to declare rules that consist of three parts: (1) a pattern to match, (2) a template-like structure to produce, (3) and a mapping to insert matched data into the template. These three components are also found in ModelSoC. (1) is the concern analysis (2) are the unit-prototypes and (3) is the concern composition. We allow, compared to the other approaches, for independent reuse of (1) and the specification of (3) in concrete syntax. Second, model transformations can be used as basis technology to implement ModelSoC. Reuseware itself is implemented with SDM.

UniTI [32], MCC [33], TraCo [34] and Megamodelling [35] organise MDSD processes by defining relations between models, metamodels and transformations. Compared to ModelSoC, they mainly use transformations as composition methodology. Still, ModelSoC could be used as part of a larger MDSD process and integrated with other model manipulations by one of these approaches. Finally, any MDSD approach and technology can potentially be combined with our approach, since the specifications and composition programs (which represent the concern space) in Reuseware are models with a well defined metamodel.

## 7 Conclusion

In this paper we presented ModelSoC, an approach to organise MDSD processes by concern (de)composition, and an implementation of it. ModelSoC enables universal separation of concerns in MDSD processes that involve arbitrary modelling languages and does not enforce the usage of predefined languages. It thus does not limit the strength of MDSD to utilise DSMLs for different viewpoints in development. Our implementation monitors the consistency of all models of the system and provides a complete view of all concerns and their relationships at each point in time. ModelSoC allows for independent modification and extension of each concern dimension, thus allowing MDSD processes to evolve. We illustrated this on the model-driven development of a demonstrator system.

## References

1. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd Edition. Pearson Education (2009)
2. ECMFA Traceability Workshop Organisers: ECMFA Traceability Workshop Series. `http://www.modelbased.net/ecmda-traceability` (2010)
3. Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: AOSD. Addison-Wesley (2005)
4. Workshop on Aspect-Oriented Modeling Organisers: Workshop on Aspect-Oriented Modeling (AOM) Series. `http://www.aspect-modeling.org` (2010)
5. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley (April 2005)
6. Jacobson, I., Ng, P.W.: AOSD with Use Cases. Addison-Wesley (2004)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Proc. of ECOOP'01. LNCS, Springer (2001)

8. Bergmans, L., Aksit, M.: Composing Crosscutting Concerns Using Composition Filters. ACM **44**(10) (2001) 51–57
9. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A Generic Approach For Automatic Model Composition. In: Proc. of AOM @ MODELS'07. LNCS, Springer (2007)
10. Ossher, H., Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In: Proc. of Symp. on SWArch. and CompTechn., Kluwer (2000)
11. Software Technology Group, Technische Universität Dresden: Reuseware Composition Framework. `http://reuseware.org` (2010)
12. IBM: Concern Manip. Environment. `sourceforge.net/projects/cme` (2006)
13. 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
14. Eclipse: UML2 metamodel. `eclipse.org/modeling/mdt/?project=uml2` (2010)
15. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. of SLE'09. LNCS, Springer (March 2010)
16. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proc. of ECOOP'97, Springer (1997)
17. Aßmann, U.: Invasive Software Composition. Springer (April 2003)
18. Henriksson, J.: A Lightweight Framework for Universal Fragment Composition—with an application in the Semantic Web. PhD thesis, TU Dresden (January 2009)
19. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Language-Independent Model Modularisation. In: TAOSD VI. LNCS, Springer (2009)
20. Johannes, J., Gaul, K.: Towards a Generic Layout Composition Framework for Domain Specific Models. In: Proc. of DSM'09 at OOPSLA. (2009)
21. Johannes, J., Samlaus, R., Seifert, M.: Round-trip Support for Invasive Software Composition Systems. In: Proc. of SC'09. Volume 5634 of LNCS., Springer (2009)
22. Modelplex: D1.1.a (v3): Case Study Scenario Definitions. `modelplex.org` (2008)
23. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data & Knowledge Engineering **35**(1) (2000) 83–106
24. Schauerhuber, A., Retschitzegger, W., Kappel, G., Kapsammer, E., Wimmer, M., Schwinger, W.: A Survey on AOM Approaches. Technical report, JKU Linz (2006)
25. Kienzle, J., Al Abed, W., Klein, J.: Aspect-Oriented Multi-View Modeling. In: Proc. of AOSD'09, ACM (2009) 87–98
26. Morin, B., Klein, J., Barais, O., Jézéquel, J.M.: A Generic Weaver for Supporting Product Lines. In: Proc. of EA'08, ACM (2008) 11–18
27. Hovsepyan, A., Scandariato, R., Van Baelen, S., Berbers, Y., Joosen, W.: From Aspect-Oriented Models to Aspect-Oriented Code? In: AOSD '10, ACM (2010)
28. Object Management Group: Meta Object Facility 2.0 Query/View/Transformation (QVT). `http://www.omg.org/cgi-bin/doc?formal/08-04-03` (2008)
29. Eclipse: ATLAS Transformation Language. `eclipse.org/m2m/atl` (2010)
30. Kolovos, D.S.: An Extensible Platform for Specification of Integrated Languages for Model Management. PhD thesis, University of York (2008)
31. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the UML and Java. In: TAGT'98, Springer (2000)
32. Vanhooff, B., Ayed, D., Baelen, S.V., Joosen, W., Berbers, Y.: UniTI: A Unified Transformation Infrastructure. In: Proc. of MODELS'07. LNCS, Springer (2007)
33. Kleppe, A.: MCC: A Model Transformation Environment . In: Proc. of ECMDA-FA'06. Volume 4066 of LNCS., Springer (2006) 173–187
34. Heidenreich, F., Kopcsek, J., Aßmann, U.: Safe Composition of Transformations. In: Proc. of ICMT'10. LNCS, Springer (June 2010)
35. Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: Proc. of Best Practices for MDSD workshop. (2004)