Virtual Textual Model Composition for Supporting Versioning and Aspect-Orientation[*]

Robert Bill
Institute of Software Technology and Interactive Systems, TU Wien
Vienna
bill@big.tuwien.ac.at

Patrick Neubauer
Department of Computer Science, University of York
York
patrick.neubauer@york.ac.uk

Manuel Wimmer
CDL-MINT, TU Wien
Vienna
wimmer@big.tuwien.ac.at

ABSTRACT

The maintenance of modern systems often requires developers to perform complex and error-prone cognitive tasks, which are caused by the obscurity, redundancy, and irrelevancy of code, distracting from essential maintenance tasks. Typical maintenance scenarios include multiple branches of code in repositories, which involves dealing with branch-interdependent changes, and aspects in aspect-oriented development, which requires in-depth knowledge of behavior-interdependent changes. Thus, merging branched files as well as validating the behavior of statically composed code requires developers to conduct exhaustive individual introspection.

In this work we present VIRTUALEDIT for associative, commutative, and invertible model composition. It allows simultaneous editing of multiple model versions or variants through dynamically derived virtual models. We implemented the approach in terms of an open-source framework that enables multi-version editing and aspect-orientation by selectively focusing on specific parts of code, which are significant for a particular engineering task.

The VirtualEdit framework is evaluated based on its application to the most popular publicly available XTEXT-based languages. Our results indicate that VIRTUALEDIT can be applied to existing languages with reasonably low effort.

CCS CONCEPTS

• **Software and its engineering** → **Model-driven software engineering**; **Domain specific languages**; *Software configuration management and version control systems*;

KEYWORDS

Aspect-oriented modeling, model-driven engineering, model virtualization, aspect weaving, model versioning

ACM Reference Format:
Robert Bill, Patrick Neubauer, and Manuel Wimmer. 2017. Virtual Textual Model Composition for Supporting Versioning and Aspect-Orientation. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3136014.3136037

[*]Updates to the artifact available at http://virtualedit.big.tuwien.ac.at

SLE'17, October 23–24, 2017, Vancouver, Canada
© 2017
ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136037

1 INTRODUCTION

Model composition [19, 24], i.e., also referred to as system composition in a wider engineering perspective, presents a basic model-driven engineering (MDE) process. It involves the combination of multiple models for a variety of operations, such as the identification of conflicts across input models and undesirable emergent properties in composed models [30]. Thus, model composition represents a foundation for essential model management tasks such as model transformation, model comparison, and model merging [3, 29].

State-of-the-art model composition approaches often present limitations in editing composed models, such as the ability to edit arbitrary composed models. Thus, to build a composed model it is necessary to perform a variety of different operations including merging and splitting of multiple input models, which is usually achieved by establishing and maintaining dedicated model transformations. Moreover, to enable users to edit composed models as well as synchronize any changes from the composed model to respective input models and vice versa, it is common to cultivate and sustain yet another set of model transformations or resort to bi-directional transformations, i.e., requiring less, but more complex, individual transformations.

Although model transformations are employed to realize model composition scenarios, they do not possess appropriate means to ease or overcome their manual creation and maintenance. For example, an intrinsic requirement for performing splitting or merging of models includes the fabrication of a result for the union of a given set of model elements. Consequently, current solutions require the developer to manually handle a variety of different model transformations and operations and thus lead to complex, tedious, and time-consuming tasks for construction and maintenance of model composition solutions.

In this paper we present an approach to significantly ease dynamic model composition for multiple models by overcoming manual construction and maintenance of model transformations through the combination of text-based model composition and virtualization[1] in what is subsequently referred to as VIRTUALEDIT. By employing VIRTUALEDIT, several transformations and operations necessary for performing model composition, such as union, are provided by instantiating virtualization concepts and hence fully preserve the ability to edit both, the composed model as well as the input models.

To gauge the prospects of our approach and the validity of its implementation, we evaluate VIRTUALEDIT within the domain of aspect-oriented modeling (AOM) [33] and model versioning [5].

[1]In the most general form, virtualization refers to a concept, which creates the illusion of dealing with a real object, whereas being a proxy mechanism that redirects access and manipulation requests to the virtualized object.

In the next sections we describe (*i*) two use cases within the domain of model versioning and aspect-oriented modeling, (*ii*) VIRTUALEDIT by illustrating a simple demonstration case and presenting design rationales of multi-version models, their augmentation in a virtual editor, and the synchronization of changes, (*iii*) an evaluation based on a set of real-world languages retrieved from Github, (*iv*) a section of related work, and (*v*) the conclusion including a presentation of future work.

2 USE CASES

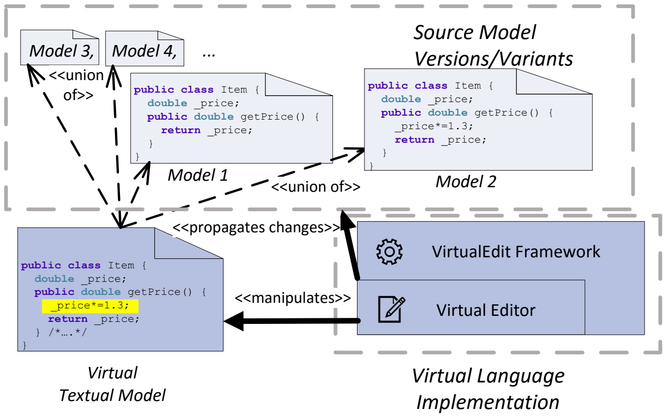
In this section, we describe the basic architectures for employing the VIRTUALEDIT framework in the domain of (*i*) model versioning, i.e., management of concurrently evolving models as well as (*ii*) AOM, i.e., management of woven models, which consist of base models and multiple aspect applications, while dynamically displaying or withholding applied aspects.

In both cases, the user edits a single virtual model that represents a composition of different models. The behavior of the VIRTUALEDIT framework includes the following. First, in case a user dynamically changes the focus, the displayed view is automatically adapted. Secondly, in case a user issues an edit operation, the operation is propagated to suitable base models.

In general, there are two types of merge procedures to be distinguished, which include (*i*) merging multiple models, such as a base model and aspects applied on that, into a single model and (*ii*) merging multiple versions of the same model. In the former, the application of aspects translates to an addition of model differences to a base model of which the union of models is built Changes are only propagated to a single suitable difference. In the latter, changes are propagated to all relevant models.

2.1 Use Case 1: Model Versioning

Fig. 1 presents an overview of VIRTUALEDIT for the use case of editing multiple model versions at the same time. The VIRTUALEDIT framework reads in all model versions in form of a structured tree, i.e., each model may have zero or more successors. Our model representation format uses IDs for identification. If there are no IDs, artificial IDs are calculated by matching each model with it's successor[2]. Then, the virtual edit framework builds the union of all models and displays them in form of a (virtual) textual model - similar to how current version management tools show (text) merges, but being model-aware and supporting an unlimited amount of models. The model elements, which are presented in our use cases, are equivalent to code snippets that only occur in the highlighted subset of all models. A user may select to view only a subset of all model versions considered at a given point in time by which the editor updates its content accordingly. In case changes are performed, these changes are propagated back to all source models of models that are active in the current view.



```
public class Item {
  double _price;
  public double getPrice() {
    return _price;
  }
}
```

```
public class Item {
  double _price;
  public double getPrice() {
    _price*=1.3;
    return _price;
  }
}
```

```
public class Item {
  double _price;
  public double getPrice() {
    _price*=1.3;
    return _price;
  } /*….*/
}
```

Figure 1: Loading multiple models into a merged virtual model.

2.2 Use Case 2: Aspect-Oriented Modeling

Fig. 2 presents an overview of the AOM example in comparison with the conventional procedure of manipulating and debugging a *system model*, which is composed of both *core code*, and *aspect code*. Usually, the *Core Editor*, e.g. the ECLIPSE Java Editor, of the *Core Language Implementation*, e.g., the ASPECTJ framework, which is built on Java, manipulates the system model, which is transformed to *woven code*, i.e., representing aspect code intertwined with core code, or directly to *executable byte code*, i.e., no intermediate *woven code* is produced. Finally, a compiler, such as the Java compiler, transforms woven code to *executable byte code* that can be debugged by employing the *Core Debugger*.

In contrast, in our approach the *Virtual Language Implementation* represents a virtualized version of the *Core Language Implementation*, that enables performing operations which require extensive effort when compared with their operation in terms of the *Core Language Implementation*. For example, usually there are no indicators in the *woven code* that state where and how *core code* has been modified by *aspect code*. Consequently, a user can not differentiate between non-generated code, i.e., *core code*, and generated code, i.e., the *woven code* produced by the weaving process.

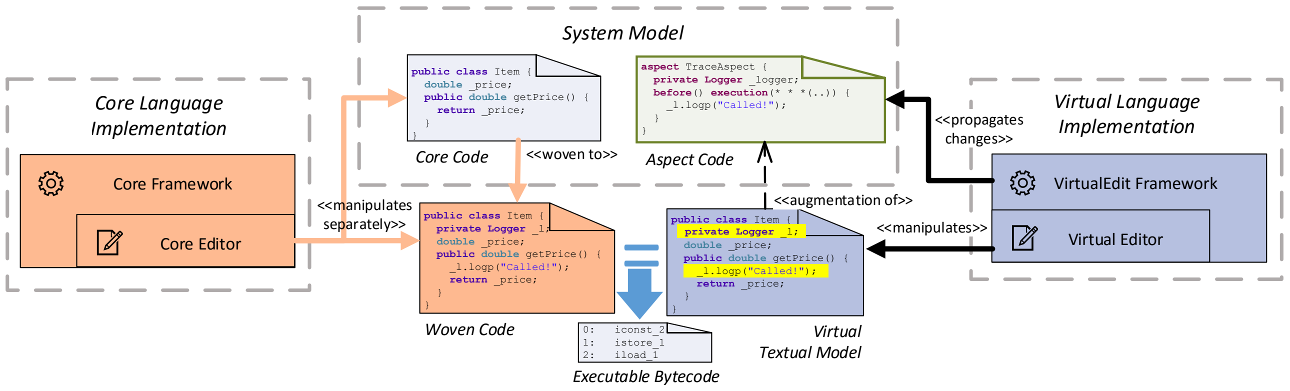
In contrast, our virtual language implementations apply a *Virtual Editor* that produces a *Virtual Textual Model*, i.e., equal to woven code, by applying model transformations, which are enriched with meta-data that allows the differentiation of non-generated and generated parts of a model. In detail, such meta-data, which contains information that associates elements with being part of either source or target of the transformation as well as if they have been modified. In other words, in our approach, the *Virtual Textual Model* essentially represents a particular view to the *system model*. Furthermore, the virtual editor allows disabling and enabling the visualization of particular parts of the *system model* which potentially decrease overall complexity and alleviate speed of versioning tasks due to a reduction of code that previously required a manual investigation by the developer.

3 THE APPROACH

This section describes VIRTUALEDIT, i.e., our virtual text-based model composition approach, which has been implemented in terms of the VIRTUALEDIT framework[3]. We implemented VIRTUALEDIT

[2]The current implementation uses EMF Compare to derive the matches, but is designed to be extensible

[3]A ready-to-use virtual machine image and ECLIPSE instance of the VIRTUALEDIT framework can be retrieved online from http://virtualedit.big.tuwien.ac.at.



```
public class Item {
  double _price;
  public double getPrice() {
    return _price;
  }
}
```

```
aspect TraceAspect {
  private Logger _logger;
  before() execution(* * *(..)) {
    _l.logp("Called!");
  }
}
```

```
public class Item {
  private Logger _l;
  double _price;
  public double getPrice() {
    _l.logp("Called!");
    return _price;
  }
}
```

```
0:    iconst_2
1:    istore_1
2:    iload_1
```

```
public class Item {
  private Logger _l;
  double _price;
  public double getPrice() {
    _l.logp("Called!");
    return _price;
  }
}
```

Figure 2: Conventional AOM approach (left-hand side) compared with our proposed virtualization approach (right-hand side).

based on MDE technologies, in particular the ECLIPSE modelling framework (EMF) [28], the graph transformation framework HENSHIN [4] and the language workbench XTEXT [11].

VIRTUALEDIT provides a base implementation for arbitrary XTEXT-based domain-specific modeling languages (DSMLs), and thus, may be applied to various languages. Its capabilities can be enabled in arbitrary XTEXT-based DSMLs by replacing their binding from the original XTEXT to our virtualized editor. This is achieved by changing four static lines of code and adding a dependency.

In the rest of this section we first demonstrate our virtual textual model composition approach in the context of AOM and then present a detailed report on the implementation of VIRTUALEDIT by focusing on the design rationale of (multi-version) models, their augmentation in the virtual editor, and the synchronization of model changes.

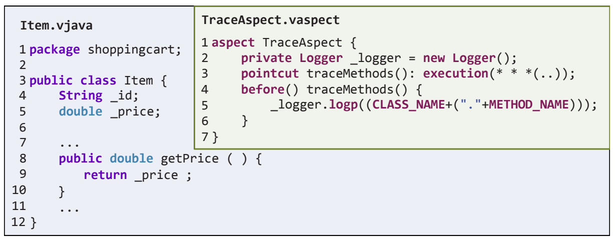
3.1 Demonstration Case

The demonstration case represents the functionality of a shopping-cart, which has been originally provided by Laddad et al. [20]. For sake of brevity, we focus on the `Item` class, which models a shopping item with a price that can be purchased (cf. left part of Fig. 3). In terms of aspects, "`TraceAspect`" (cf. upper-right part of Fig. 3) depicts a typical AOM monitoring technique based on logging method calls. Moreover, the Henshin rule "`freeitems`" (cf. Fig. 4) represents, purely for demonstration purposes, a malicious aspect that has been introduced by a developer to make products free.[4]

3.2 Design Rationale and Realization

In this subsection we present the design rationale used for our approach for aiming towards multi-versions and aspect-oriented model representations, respectively, as well as its realization in the realm of metamodeling frameworks and language workbenches.

3.2.1 Data Structure. To achieve *dynamic model composition* in a generic way, requires establishing a novel data structure for indicating the location of model elements in order to replace them within their conventional structure. In order to support dynamic visualization and inhibition of specific aspects without requiring the re-computation of the effect that such actions have on other aspects, and thus woven code, a model representation that allows adding and removing certain deltas for any occurring delta is required. However, even simple structures, such as sequences with Integer indices give counter-intuitive results when employing conventional delta structures. For example, if such a solution is considered, it leads to the following problems. First, assume the list [a,d], where first b, then c, and finally d is added to get [a,b,c,d]. A typical index-based delta representation could be [add(b,1),add(c,2)]. Applying only the second delta on [a,d] would yield [a,d,c] which does not represent the expected result, i.e., [a,c,d]. Secondly, as a result of the importance



Item.vjava

```
package shoppingcart;

public class Item {
    String _id;
    double _price;

    ...
    public double getPrice ( ) {
        return _price ;
    }
    ...
}
```

TraceAspect.vaspect

```
aspect TraceAspect {
    private Logger _logger = new Logger();
    pointcut traceMethods(): execution(* * *(..));
    before() traceMethods() {
        _logger.logp((CLASS_NAME+("."+METHOD_NAME)));
    }
}
```

Figure 3: The Aspect "`TraceAspect`" (upper-right) an the base code in the ECLIPSE Editor (left).

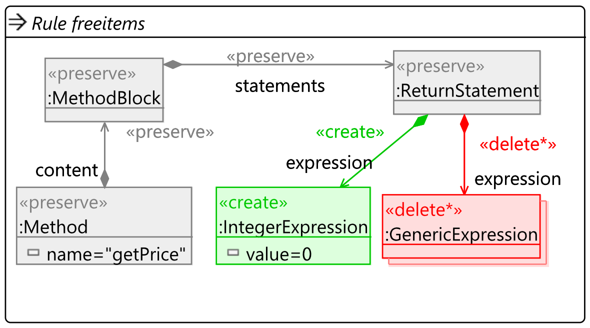


Figure 4: HENSHIN rule "`freeitems`" making items free.

[4]Henshin only deletes matched elements, i.e., not the newly created `IntegerExpression`

of the order in which deltas are being applied, efficiently adding or removing deltas requires the manual specification of resolutions for certain types of conflicts [1].

Thus, we need a model composition that builds a group $(\Delta, \oplus)$ for applying model deltas, i.e., that $\oplus$ is commutative, associative, has a neutral element and has inverse elements for each element. This means that we can unapply a delta by building the inverse and adding it to the result model. A model has the same representation as a model delta. Hence, to solve this, we represent models $\Delta = (I, e, c, A, R)$ as functions with object identifiers I. The metamodel is assumed to be a typical MOF-based metamodel with attributes $\mathcal{A}$, references $\mathcal{R}$, and classes C. The model contains (i) an *existence* set $e : \text{Set}_I$ defining which objects exist, (ii) a classification function $c : I \longrightarrow \text{Set}_C$ associating an object to its class and all super classes and implemented interfaces, (iii) attributes $A \ni a_{i,V} : I \longrightarrow \text{List}_V$ associating a list of values of type V to each object and (iv) references $R \ni r_i : I \longrightarrow \text{List}_I$ associating a list of identifiers to each object. Although unordered, unique attributes and references could map to a set instead of a list from a model representation point of view, we use lists to maintain the specific element order in the edited document. Moreover, instead of representing a containment explicitly, we recursively change the existence value when an object is added to or removed from a containment. As a result, all feature values are kept in case a previously removed object is recreated. There may be additional sets $e_r : \text{Set}_I$ for each resource to define which elements are directly contained in a resource. Finally, models of multiple resources are summed up in order to get the model of the complete resource set.

3.2.2 Groups (Sets and Lists). Additionally, to get a group, we define sets and lists as follows. A set of type V is defined as $\text{Set}_V : V \longrightarrow \mathbb{N}$, where an element is in the set if and only if the function value is greater than zero. A list of type V is defined as $\text{List}_V : P \longrightarrow \text{Set}_V$, where each $p \in P$ is a sequence of integers. Thus, allowing the definition of a lexicographic order for elements of P.

Consequently, insertions can be performed at any location. For example, the indices of an element that is inserted between elements with indices `[1]` and `[2,1]` might be `[1,1]`, `[1,6]` or `[2,-1,0]`. In particular, arbitrary order-preserving suffixes may be generated to ensure that list-position-clashes can not occur. For instance, if the current position is `[1,2]` and the identifier of the aspect is `9`, then the resulting position amounts to `[1,2,9]`. Further, in case unique Integer identifiers can not be assigned to a particular aspect, the aspect's URI is added instead, and thus, re-establishes an aspect's uniqueness.

3.2.3 Functions and Identifiers. All functions are stored as partial functions. Functions of type $I \longrightarrow \mathbb{N}$ return the value 0 for undefined identifiers. Functions of type $V \longrightarrow R$, with R being a function will return a default function, i.e., the function without any value stored, for each parameter. We assume that $\text{sdom} : (A \longrightarrow B) \longrightarrow P(A)$ will return the actual assigned domain for each function. With that, $\text{sdom}(a \oplus b) = \text{sdom}(a \cup b) = \text{sdom}(a) \cup \text{sdom}(b)$ As example, consider the `Item` class depicted in Fig. 3 which currently has two attributes `_id` and `_price`. This model excerpt would look as follows in our representation. We assume the ID of the Item class to be i_{Item}. Then, the existence set contains that object and two objects for the attributes, i.e., $e = \{i_{\text{Item}}, i_{\text{atid}}, i_{\text{atprice}}\}$ which is expressed as $e = f(i) = \{i_{\text{Item}} \mapsto 1, i_{\text{atid}} \mapsto 1, i_{\text{atprice}} \mapsto 1, i \mapsto 0$ else, where the last part $i \mapsto 0$ is the default value, not stored, and omitted in the following. $\text{sdom}(e)$ would yield $\{i_{\text{Item}}, i_{\text{atid}}, i_{\text{atprice}}\}$.

The `Item` class has `class` as only type and the other objects are attributes. Consequently, the classification function c is defined as $c : f(i) = \{i_{\text{Item}} \mapsto \{\text{class}\}, \{i_{\text{atid}}, i_{\text{atprice}}\} \mapsto \{\text{attribute}\}\}$. We have three attributes: two references, one for storing objects in a class and one for storing attribute types and one attribute for storing names.

The class-attribute reference is defined as $r_i(i) = \{i_{\text{Item}} \mapsto [i_{\text{atid}}, i_{\text{atprice}}]\}$, with the list containing two elements, e.g. at position [1,0] and [2,0], which would yield the representation $l(p) = \{[1,0] \mapsto (i_{\text{atid}} \mapsto 1), [2,0] \mapsto (i_{\text{atprice}} \mapsto 1)$, i.e., $l([1,0])$ returns a function which associates 1 to the identifier i_{atid} and 0 for all other identifiers, i.e., only the identifier i_{atid} is stored in the position $[1,0]$.

All model operation functions should allow adding and removing models dynamically. A suitable way of that is to make them build a group. Then a model can be removed by adding the inverse. Thus, we base our model operation functions on the usual addition which is a well-known group.

As a result, our model sum function $\oplus$ is defined as follows:

$$a \oplus b = \begin{cases} I \longrightarrow \mathbb{N}, i \mapsto a(i) + b(i) & a, b \text{ are Sets} \\ V \longrightarrow R, p \mapsto a(p) \oplus b(p) & a, b \text{ Lists or functions} \\ \quad V \longrightarrow R \text{ with } R \text{ List, Set or general function} & \end{cases}$$

3.2.4 Precedence and Conflict Resolution. The model difference function $\ominus$ is defined with $-$ instead of $+$ and presents the inverse of the model sum function $\oplus$. By construction, merge conflicts appear to be resolved implicitly. Deleting objects takes precedence over updating any attribute values and adding new links to these objects[5]. Updating values takes precedence over deleting them. Updating values differently results in both values being added to the feature. However, as the merge is virtual and thus occurs in only in memory, these conflict resolutions are not persisted in the formalization. Custom conflict resolutions could be stored as additional delta model that is able to resurrect deleted objects and thus delete incorrectly updated feature values.

For instance, if we want to add an attribute `_logger` to our model, we can add the original model to a model containing the logger attribute with name and type, i.e., $e = \{i_{\text{logger}}\}$, $a_{\text{name},\text{String}} = (i_{\text{logger}} \mapsto [\text{'\_logger'}]), \ldots$.

The model union function $\cup$ can be defined analogous:

$$a \cup b = \begin{cases} I \longrightarrow \mathbb{N}, i \mapsto \max(a(i), b(i)) & a, b \text{ are Sets} \\ V \longrightarrow R, p \mapsto a(p) \cup b(p) & a, b \text{ Lists or functions} \\ \quad V \longrightarrow R \text{ with } R \text{ List, Set or general function} & \end{cases}$$

The model intersection function $\bigcap$ is defined similarly, with min instead of max. The main practical difference between *sum* and *union* lies in that the neutral element of the sum is the empty model while the neutral element for both union and intersection is the model itself. Thus, we use the sum to compose changes, i.e., model deltas, and the union to compose multiple pre-existing models. Please note the semantic differences between four potential ways of combining multiple model versions based on the same metamodel, i.e., combining the models with (i) the $\oplus$ operator, (ii) the $\cup$ operator, (iii) the $\cap$ operator and (iv) calculating model differences between model

[5]Our implementation deviates from this by currently allowing links to non-existent objects to reduce the number of textual changes.

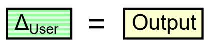
versions and adding these differences to the base model (see Fig. 7). The resulting model of (*i*) and (*ii*) are nearly the same as the sum of positive integers is always greater than zero and the sum of non-negative integers is always zero if all summands are zero. They build a max-model, i.e., a model containing all objects and all feature values of the base models. We use the second variant to ease the implementation of edit operations. Variant (*iii*) builds a min-model, i.e., a model containing only objects and feature-values contained in all base models. Variant (*vi*) uses the previously defined conflict resolution.

3.2.5 Interoperability and Multi-Versions. For interoperability purposes with EMF, we provide an ECore view which provides virtual `EObjects` and virtual `ELists` which are backed by our representational structure. For single-valued features, only the first value in the list is used. For multi-valued features, all values are used. Currently, Strings are considered as atomic values. Thus, two changes to a single String will result in two Strings being stored in the feature slot. If a String would be represented as a list of characters, changes may be included in the more fine-grained character level, and thus, only require the generation of a single String containing both changes.

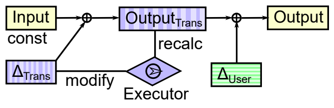
Multi-Versions can be directly described by the model union function above. We change the union model by modifying all base model functions, i.e., functions returning a natural number, so that they return the target value, i.e., we apply the edit model e to the current virtual union u as $u := u \oplus e$ with $(a \cup b) \oplus e = (a \oplus e) \cup (b \oplus e)$. At any time, we can choose to apply an edit operations only to a selection of model versions.

3.2.6 Delta model computation. In the following, we present how the delta model is calculated from aspect applications and how aspect applications can be combined. The data structure, which has been defined beforehand, can also be used to define the structure of aspects, i.e., model transformations, as depicted in our demonstration case. Moreover, the model sum is used to add the base model to the derived changes.

In our approach, all transformation application instances have a single output result and a single model Δ_{User} for user changes. Fig. 5a shows the pseudo-transformation structure of a single base model. The base model is both transformation result and user edit model. Fig. 5b shows the structure of a single aspect instance. An aspect transforms an input to build an output. Hence, instead of directly modifying the input model, all modifications are stored in the delta model Δ_{Trans}. Next, in case the transformation is reapplied, Δ_{Trans} is cleared and $\text{Output}_{\text{Trans}}$ is recalculated. Subsequently, user changes are stored in a separate model that has not been seen

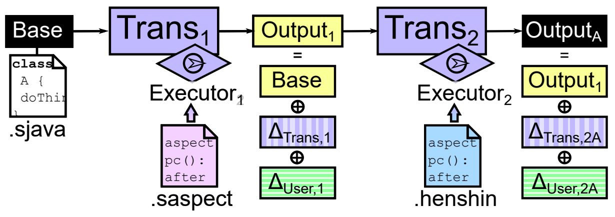


(a) Structure of a single model



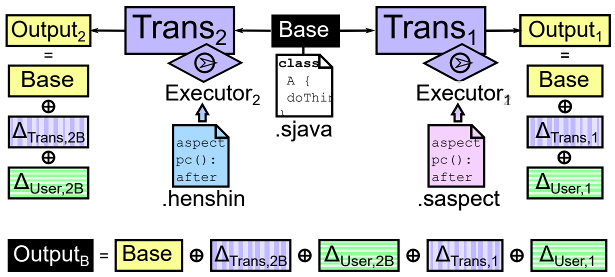
(b) Structure of a single transformation

Figure 5: Structure of transformation providers



(a) chained calculation

Figure 6: Output model generated by a transformation chain.



(b) Independent calculation

Figure 7: Output model generated by parallel transformations.

by the transformation and thus does not affect the results in case the transformation is re-applied. Finally, the final output is both represented by the composition ⊕ of transformation output as well as the user delta.

Our approach makes sure that object identifiers, which have been created by transformations, remain constant for multiple transformation executions so that edit operations remain valid.

Further, to avoid cases in which aspects accidentally create objects that have been removed by other objects, every created object identifier is prefixed with the aspect's id and every created list position is suffixed with the aspect's index or id. Thus, object identifiers are calculated as a function of the executed transformation rule and its parameters. In detail, user edit operations are stored in a suitable delta, which avoids that aspects immediately undo them when they are re-executed, and trigger the following heuristic: If a part is deleted or added, then the edit operation is stored in the delta of the transformation instance of (*i*) the last deletion or addition operation of this part, if any or (*ii*) the addition of the containing object. As a result of storing object removals invisibly to transformations, the view, on which an aspect operates, does not contain a removal operation and hence cannot unapply it.

Moreover, we carry out transformations in sequential order as well as storing model-change operations immediately following their creating transformation, i.e., the transformation that initially created the model that has been modified. In detail, the architecture of our approach implicitly isolates aspects, i.e., represented by transformations in our approach, from model-change operations. Additionally, any preceding transformations are enabled to initialize models without limitations, which otherwise may be imposed by aspects.

Fig. 6 and Fig. 7 show two cases of how our approach employs multiple aspects to produce an output result from a base model that result in distinctively computed delta models. First, the result of the first aspect is used as input for the execution of the second aspect (cf. right part of Fig. 6), i.e., representing a chained-calculation. Secondly, both aspects are executed on the base model (cf. Fig. 7). In both cases, the output is the model sum of base model, transformation deltas, and user deltas. However, in the first case, all aspects are independent from each other and in the second case, the second aspects is able to see all changes that have been performed by the first aspect. Thus, in case the second aspect Trans_2 would add a logging statement to each method, our chained-calculation would also add it to methods that have been generated by the first aspect Trans_1.

Our current implementation supports two kinds of transformation providers. First, the generic HENSHIN [4] transformation provider allows to define aspects by executing HENSHIN model transformations. Secondly, the AOM-specific transformation provider for our VASPECT language creates transformations from aspect definitions. Finally, transformation provider instances, which have distinct transformation and user deltas, are created for each base model of the VJAVA language.

3.3 Editor Augmentation

In terms of editor-augmentation, we employ a customized XTEXT editor to display our virtual model. In detail, we synchronize the XTEXT model with our virtual model as a consequence of the XTEXT framework not offering direct model manipulation. As a result, the editor is augmented with information from the virtual model, which is synchronized with the XTEXT model.

Moreover, for each structural feature value, we determine all sources, i.e., all deltas and possibly the base model for the AOM use case and all source models for the versioning use case, which have contributed to a particular feature value. In the AOM case, we distinguish between different types of features values: (*i*) *nonderived feature values*, i.e., all such sources are user deltas or the base model, (*ii*) *derived feature values*, i.e., all such sources are transformation deltas, (*iii*) *partly derived feature* values, in all other cases.

In the versioning case, we distinguish between (*i*) *base feature values*, i.e., feature values occurring in all source models and (*ii*) *nonbase feature values*, i.e., feature values not occurring in at least one single source model.

Additionally, at least partially derived feature values and nonbase feature values are highlighted in different colors. Also, our implementation provides an aspect/model selection view that enables users to select specific aspects/models, which they want to see in the editor. In detail, the model that is shown in the editor, is calculated as sum of the base model, all transformation user deltas, and (only) transformation deltas of selected aspects (cf. Fig. 8) or the union of all selected source models. Additionally, any time a user selects aspects to be displayed, the editor view is updated accordingly.

Hence, employing our virtual editor on the running example (cf. Section **??**) enables viewing and hiding particular aspects (cf. left-hand side of Fig. 8) as well as ease the identification and isolation of undesired behavior that has been woven into the final system due to an error in the aspect definition (cf. right-hand side of Fig. 8). Similarly, we can easily see changes done in specific models.

3.4 Synchronization of VIRTUALEDIT model and XTEXT model

The general synchronization workflow of our approach entails that modifications, which are performed on the *Virtual Textual Model* by employing the virtual editor, trigger the execution of alterations that carry out the necessary adaptations of the system model as well as eventual changes in the content visualized by the virtual editor.

To ensure the correct matching of elements, we store the target virtual object for each object in the editor as text annotation which we can use to build a correspondence map between XTEXT-`EObjects` and VIRTUALEDIT-`EObjects` based on the position of the elements in the text. As a result of the reparse operations performed by the XTEXT framework, annotating model elements does not present a viable solution. Hence, the synchronization has to be performed in both directions as well as in a recursive fashion on the root elements of a resource by synchronizing all feature values and their contained model elements. We synchronize feature values by applying only patches to each feature value to avoid unnecessary changes that eventually lead to a loss of formatting.

Xtext *model to* VirtualEdit *model.* In this case, if matching elements do not share the same type any more, the type of the VIRTUALEDIT model element is directly changed by changing (only) the object-to-class function. As a result, feature values of repeated type changes are preserved. Elements are created by choosing the correct user edit delta to place the elements in and generating a new URI in that user edit delta.

VirtualEdit *model to* Xtext *model.* In this case, if matching elements do not share the same type any more, a new element of the correct type is created and all features values of features which exist in both types are copied. Next, elements are created using standard Ecore facilities for element creation. After the system model has been synchronized, all textual annotations regarding mapping and derivation the status of are updated.

3.5 Current Limitations

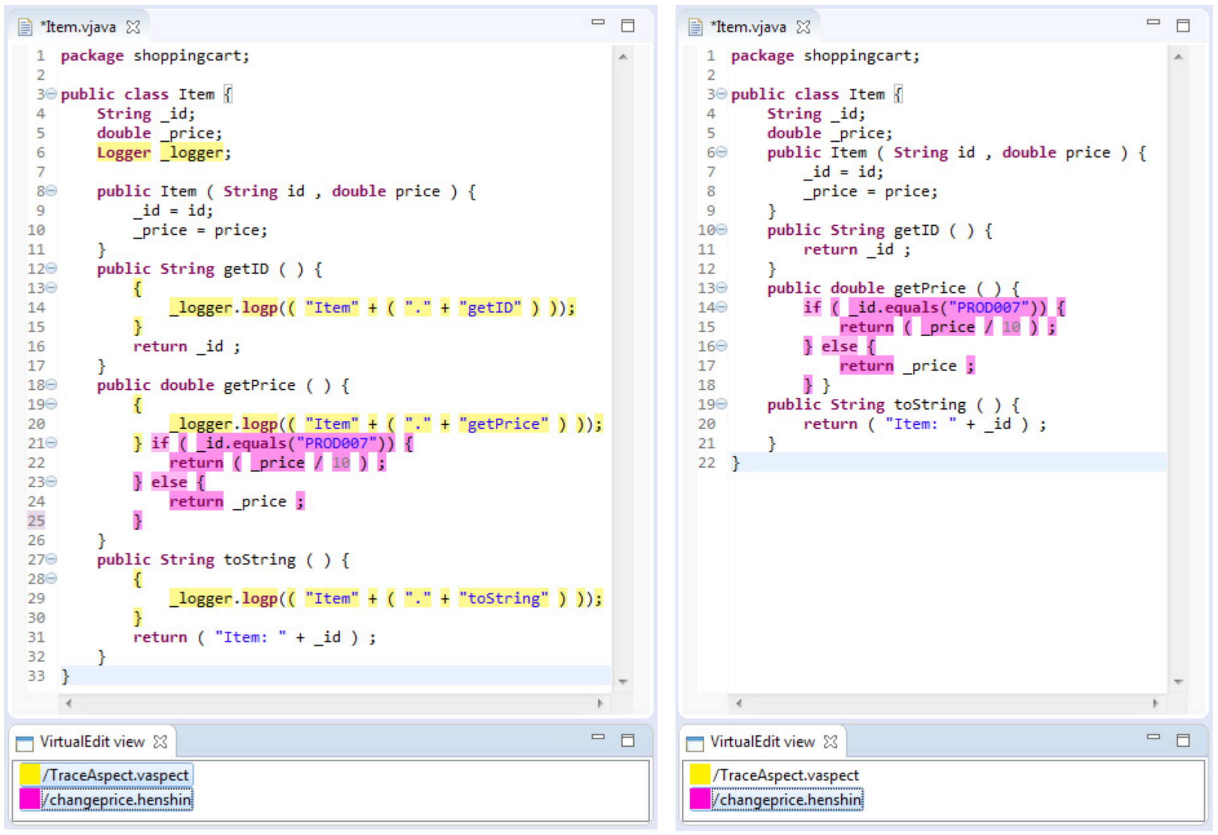
Our approach and the VIRTUALEDIT framework currently have the following limitations.

First, the VIRTUALEDIT model composition does not retain Core Model editability for all types of aspects, i.e., it cannot propagate all changes back to the Core Model where it would be possible in principle. In detail, source-level modifiers are represented in terms of primitive operations, i.e., *ADD* (+) and *REMOVE* (-). Therefore, merging, reordering, or interleaving of model elements can not be explicitly represented.

The tooling implementation currently does not make use of all potential features of the approach. For example, (meta-)information contained in the VIRTUALEDIT model like multi-to-single-feature conflicts and exact source locations are not visualized in the editor window.

4 EVALUATION

In general, the evaluation of the VIRTUALEDIT framework follows the guidelines for case study research in software engineering [25] and is based on a set of demonstration cases involving the most popular real-world languages of different domains. The objects of study



```
package shoppingcart;

public class Item {
    String _id;
    double _price;
    Logger _logger;

    public Item ( String id , double price ) {
        _id = id;
        _price = price;
    }
    public String getID ( ) {
        {
            _logger.logp(( "Item" + ( "." + "getID" ) ));
        }
        return _id ;
    }
    public double getPrice ( ) {
        {
            _logger.logp(( "Item" + ( "." + "getPrice" ) ));
        } if ( _id.equals("PROD007")) {
            return ( _price / 10 ) ;
        } else {
            return _price ;
        }
    }
    public String toString ( ) {
        {
            _logger.logp(( "Item" + ( "." + "toString" ) ));
        }
        return ( "Item: " + _id ) ;
    }
}
```

🟨 /TraceAspect.vaspect
🟪 /changeprice.henshin

```
package shoppingcart;

public class Item {
    String _id;
    double _price;
    public Item ( String id , double price ) {
        _id = id;
        _price = price;
    }
    public String getID ( ) {
        return _id ;
    }
    public double getPrice ( ) {
        if ( _id.equals("PROD007")) {
            return ( _price / 10 ) ;
        } else {
            return _price ;
        } }
    public String toString ( ) {
        return ( "Item: " + _id ) ;
    }
}
```

🟨 /TraceAspect.vaspect
🟪 /changeprice.henshin

Figure 8: Applying the VIRTUALEDIT virtual editor for the visualization of all available aspects and HENSHIN rules (left) and the sole visualization of the "`freeitems`" HENSHIN rule (right).

and evaluation results are publicly available on the VIRTUALEDIT project website http://virtualedit.big.tuwien.ac.at.

4.1 Setup

Objective. The objective of the evaluation of our framework is to assess its capability to enable multi-versioning and aspect-orientation applied to real-world languages of different domains.

The cases. Representative cases include a set of publicly available XTEXT-based DSMLs and the application of our framework in the context of scenarios that involve multi-versioning and aspect-orientation.

Theory. We hypothesize that the VIRTUALEDIT framework can be applied for enabling multi-versioning and aspect-orientation in real-world XTEXT-based DSMLs by selectively focusing on specific parts of their models.

Research questions. RQ1: Is the VIRTUALEDIT framework capable to handle typical multi-versioning and aspect orientation scenarios? ***RQ2***: How integrable is VIRTUALEDIT to real-world XTEXT-based language implementations, and thus, offers typical multi-versioning and aspect orientation capabilities?

Selection strategy. First, our selection strategy involves issuing queries to Github for retrieving all projects containing XTEXT grammar files. Next, resulting projects are sorted based on their number of stargazers, i.e., amount of users that have added a particular project to their list of starred projects. Finally, the set of our study objects is formed by selecting the five most popular, i.e., highest-ranked according to their number of stargazers, real-world XTEXT-based projects that have been retrieved from Github.

Method. The methodology of our evaluation follows the subsequently mentioned steps that are repeated for each object of study. We (*i*) create a new XTEXT project by employing the XTEXT creation wizard, which creates an exemplary DSML skeleton, (*ii*) replace its skeleton-grammar by a grammar that we retrieved from a real-world XTEXT project, (*iii*) execute the project creation workflow to generate an executable DSML implementation, (*iv*) configure the generated implementation to employ VIRTUALEDIT editors for versioning and aspect-orientation, (*v*) select an available real-world model as well as a previous version of the same model available in the history of the real-world XTEXT project repository, (*vi*) apply a model transformation that conducts changes to the model, and (*vii*) load both the historical as well as the current model with our VIRTUALEDIT model editor, respectively.

4.2 Results

In the endeavour to answer RQ1, we first examined the extend of the VIRTUALEDIT framework to support typical concepts that appear in the AOM world [33]. Thus, we considered several tutorials and books on ASPECTJ, e.g., by Laddad [20] and DZone [17], during the design and implementation of the VASPECT language, which covers recurring AOM concepts, such as, pointcuts, i.e., matching conditions, and advices, i.e., code modifications. Further, we found that many Java-based aspect languages instrument byte code instead of producing woven code.

To answer RQ2, we validated the applicability of our framework to our set of study objects. First, we found that the three projects `wesnoth/wesnoth`, `eclipse/smarthome`, and `ufoai/ufoai` could successfully be employed by VIRTUALEDIT and two projects, i.e., `antlr4ide` and `Jnario`, could not be employed as a result of their dependence on additional Java files or other grammars, which require the functionality of imports that is not yet supported by our current implementation. Moreover, we found that replacing the application of the default XTEXT editor with our virtual editor can be performed with an acceptable amount of effort and thus enable the use of VJAVA-files as described earlier. However, we found that VIRTUALEDIT requires valid input models with references that are available within non-imported models. Thus, models that contain such references require manual investigation before they can be displayed by the VIRTUALEDIT editor.

Furthermore, the implementation of the code required to execute VASPECT aspect definitions, which produce woven code, is sufficiently easy. On one hand, we found out that ASPECTJ pointcuts and advices can be seen as transformation context and actions, which has been shown for generic aspects in modeling [23]. For example, *execution* together with *before*, *after* or *around* only add the advice code at a specific point and do not even require dynamic conditions. On the other hand, some advices, such as *cflow* may require a static and thread-local variable that has to be checked at runtime. Thus, we hypothesize that most remaining advices, may be implemented with a similar effort.

During the evaluation, we found numerous bugs in our implementation and a potential limitation. Files in the wesnoth language lose their formatting, possibly due to their use of hidden tokens. However, we think that these bugs are not a result of the approach itself, but rather of implementing the approach without testing enough. In fact, several of the most important bugs were detected and fixed as a result of the evaluation.

To summarize, we conclude that, VIRTUALEDIT is capable to handle multi-versioning and aspect orientation scenarios found in selected literature and our current implementation is integrable to a subset of investigated real-world XTEXT-based language implementations.

4.3 Discussion and Validity

Although the current implementation of VJAVA does not provide dedicated support for debugging, which would allow the developer to set conditional breakpoints on a generated advice and hence focus on a particular aspect during the debugging process, we hypothesize that such a debugger eases the detection and reasoning behind erroneously applied advices, which are based on dynamic values. In other words, the causes of violated requirements may be found with less effort when selectively enabling advice-postconditions. Moreover, our composition allows developers to easily edit code that has been generated by advices as well as preserve such edit-operations during the re-application of aspects. Thus, existing challenges such as code location and data values as well as limitations of existing fault models, which do not claim to be complete, i.e., able to represent any possible kind of fault [10] are tackled by our approach through the concept of virtualization.

Internal Validity. The internal validity of our evaluation is limited to a subset of Java and ASPECTJ, which have been implemented in terms of VJAVA and VASPECT, respectively. Hence, the compatibility of our approach with the complete set of concepts available in Java and ASPECTJ, which have not been applied in any of the investigated examples found in books and tutorials but may be depicted in different AOM applications, has still to be evaluated. In other words, during the construction of VJAVA and VASPECT we did not evaluate the impact and possible limitations of using aspect applications as transformations.

External Validity. Although our evaluation is based on real-world languages of different domains, our findings are limited to a set of investigated demonstration cases. To provide a good level of representativeness of the employed cases, we investigated the most popular publicly-available DSMLs. However, we cannot state any results going beyond the selected cases before making a larger study with a statistically significant amount of DSMLs.

5 RELATED WORK

This section discusses work related to our VIRTUALEDIT model composition approach applied by clustering it in (*i*) View-based Modeling, (*ii*) Model Composition, and (*iii*) AOM.

View-based Modeling. Goldschmidt et al. [13] present a survey that analyzes and organizes existing approaches for view(point)-based aspects in DSMLs, which are scattered across publications, and contributes a taxonomy on view-based modeling from a tool-oriented perspective. For example, their taxonomy includes means to describe editor capabilities such as “bidirectionality”, i.e., ability to synchronize models and their views, and “update strategy”, i.e., when to execute synchronization transformations. ModelJoin [7] and EMF Views [6] (previously VirtualEMF [9]), enable the creation of model views, which combine models of different metamodels with an SQL-like syntax. Further, they also use EMF but do not provide a virtual textual editor, which enables dynamic visualization and hiding of aspects.

Generally, view(point)-based approaches typically focus on providing multiple different views on the same model as opposed to one complete or filtered view on multiple models. On the contrary, the VIRTUALEDIT approach enables dynamic views as well as direct, multi-language, and language-independent manipulation to which a view may be associated. As a result, dynamic views and direct multi-language model manipulation, which is achieved by our approach, may ease and speed-up the process of debugging due to decreased complexity and accelerated falsification and localization of errors.

Model Composition. Although, model composition has been investigated in literature from various angles, such as (*i*) specific application on model families [26], (*ii*) formal semantics and potential composition operators [15, 31], as well as (*iii*) methods for automating the identification and composition of relationships among elements [12, 18], several challenges have been addressed by EMF

Views [6] (previously VirtualEMF [9]), which combines heterogeneous and interrelated models in terms of on-demand computed views defined by an SQL-like and XTEXT-based query language, including increased efficiency in memory consumption and formation time caused by data duplication. Similarly to Goldschmidt et al. [13], "View Scopes" represent the visualization of a specific selection of elements. Kolovos et al. [18] introduce the "Epsilon Merging Language (EML)" and an approach to merge multiple models, which are based of different metamodels, into one model. However, users have to manually create EML-based matching rules instead of being automatically provided with a virtualized view that presents the merge-result.

AOM.. Hovsepyan et al. [16] show that modeling in terms of *all-aspectual processes*, in which concerns are kept separated, increases modeling performance to up to 20% and results in smaller, less complex, and more modular implementations when compared with hybrid processes, in which concerns are composed. Thus, all-aspectual processes also shorten the versioning cycle.

Schoettle et al. [27] present "TouchCORE" (previously "TouchRAM"), i.e., a modeling tool to support Concern-Driven Software Development (CDSD). TouchCORE enhances tracability in CDSD by visualizing feature models, e.g., in terms of class diagrams, and thus exemplifies one use case of our more generic approach. In detail, our approach may be applied in CDSD to derive such visualizations but also in other code weaving or code virtualization scenarios.

Mehmood et al. [22] present a systematic mapping study, which identifies two ongoing distinct lines of research: (*i*) model weaving as special case of a model-to-model transformation and (*ii*) approaches that transform aspect models into a target AO language, such as ASPECTJ, and thus rely on target language weavers to deal with crosscutting aspects. They state that approaches following the first line of research, are (*i*) rare and limited in the sense that they disregard advanced pointcut specification and (*ii*) predominantly static and therefore unable to weave and un-weave aspects during model execution [14, 32]. Our approach follows the first line of research on AO presented by Mehmood et al. However, instead of composing base and aspect model separately, both core and crosscutting concerns are edited in one place, i.e., in our VIRTUALEDIT editor. As a result, the modeler stays within the all-aspectual process, which has been found to lead to better performance [16], and simultaneously compose core and crosscutting concerns. Moreover, our approach is dynamic and hence prepared for weaving and un-weaving during model execution. Further, the DSMLs in the AOM use case, on which our approach has been applied, may be able to be extended to support advanced pointcut specifications.

Eaddy et al. [10] highlight challenges associated with source-level debugging, in which debuggers strive to maintain the illusion of a source-level view of program execution by maintaining a correspondence between source and compiled code. They emphasize that the consequence of surrendering correspondence, which is a result of applying various transformations, leads to the inability to perform source-level debugging, which makes matching expected and actual behavior difficult for the human debugger. Thus, giving rise to *code location* problems, i.e., displaying the wrong call stack of source line, or depicting byte code instead of source code, and *data value* problems, i.e., incorrect displaying of new fields or variables, that occur when correspondences between source code variables and memory locations have been obscured. Moreover, Eaddy et al. define "full source-level debugging" as a set of six AOM-specific activities that represent an extension of an AOM fault model by Ceccato et al. [8], i.e., itself an extension of Alexander et al. [2]. Additionally, AOM-specific fault models presented in literature do not claim to be complete and thus their application is limited to particular parts of source code that represents one of those AOM-specific activities. Consequently, faults introduced by (*i*) activities that are not covered by the fault model and (*ii*) faults that arise from base code, are neglected by existing AOM-specific fault models. When compared to our approach, we neither impose limitations on AOM-specific activities and thus specific fault types but provide a framework that is capable to deal with complete source code debugging. Furthermore, as a consequence of preserving the correspondence between source and target (woven) code, the problems associated with *code location* and *data value* are implicitly omitted in our approach.

6 CONCLUSION AND FUTURE WORK

In this work, we presented a virtual model composition approach to support versioning and AOM by enabling developers to (*i*) dynamically include or exclude source models from the merged model view, (*ii*) dynamically show and hide individual aspect applications without affecting the actual application of aspects by highlighting elements with their different origins and at the same time (*iii*) preserve editing capabilities by eventually redirecting model operations to the base model or the source models or store them in delta models. Moreover, our model representation enables a commutative and associative addition and a subtraction of models as well as change conflicts to be resolved implicitly.

The results of our initial experience report, i.e., evaluating our approach as well as its implementation in the VIRTUALEDIT framework, indicate that both advices and multiple model versions can be suitably represented in virtual code. Hence, we hypothesize that source-level software versioning significantly benefit from virtual views that are created by our VIRTUALEDIT model composition approach.

However, to evaluate our hypothesis for the aspect orientation showcase, the implementation of a debugger, which require considerable effort but has been done several times for different editors, has to be considered.

Therefore, future work involves the extension of our approach as well as its implementation. Regarding the approach, (*i*) the derivation of identifiers will be made customizable [21], such that developers can specify certain model elements as equal, and (*ii*) the transformation execution and model composition, which are currently separated, will be merged by extending transformation providers to directly derive output model from input model and offer means for asynchronous execution of performance-intensive transformations.

Regarding the implementation, (*i*) the performance will be improved by caching complete models and employ incremental transformations, (*ii*) the implementation of features, which address identified limitations (cf. Section 3.5), (*iii*) the implementation of a debugger and other assistive features in order to fulfill the means for conducting a user study to evaluate the impact of our approach on the productivity in the development of dynamically composed systems and (v) make more meta-information about the VIRTUALEDIT model accessible in a user-friendly way, a.o. to support the merge process by the visualization and configuration of tentative merges..

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

This work is supported by the Austrian agency for international mobility and cooperation in education, science and research (OeAD), financed by funds from the Austrian Federal Ministry of Science, Research and Economy (BMWFW). under Grant No.: ICM-2016-04969 and by Austrian Federal Ministry of Science, Research and Economy (BMWFW) and the National Foundation for Research, Technology and Development (CDG).

REFERENCES

[1] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In *Proceedings of the 6th International Conference on UML*. Springer.

[2] Roger T Alexander, James M Bieman, and Anneliese A Andrews. 2004. Towards the systematic testing of aspect-oriented programs. *Technical Report, Colorado State University* (2004).

[3] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios S. Kolovos, Ivan Kurtev, and Richard F. Paige. 2006. A Canonical Scheme for Model Composition. In *Proceedings of the 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*. 346–360.

[4] Enrico Biermann. 2010. EMF Model Transformation Based on Graph Transformation: Formal Foundation and Tool Environment. In *Proceedings of the 5th International Conference on Graph Transformations*. Springer.

[5] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An Introduction to Model Versioning. In *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*. Springer.

[6] Hugo Brunelière, Jokin García Perez, Manuel Wimmer, and Jordi Cabot. 2015. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In *Proceedings of the 34th International Conference on Conceptual Modeling (ER)*.

[7] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. 2016. View-based model-driven software development with ModelJoin. *Software and System Modeling* 15, 2 (2016), 473–496.

[8] Mariano Ceccato, Paolo Tonella, and Filippo Ricca. 2005. Is AOP code easier or harder to test than OOP code. In *Proceedings of the 1st Workshop on Testing Aspect-Oriented Program (WTAOP)*.

[9] Cauê Clasen, Frédéric Jouault, and Jordi Cabot. 2011. VirtualEMF: A Model Virtualization Tool. In *Proceedings of Advances in Conceptual Modeling. Recent Developments and New Directions (ER)*.

[10] Marc Eaddy, Alfred V. Aho, Weiping Hu, Paddy McDonald, and Julian Burger. 2007. Debugging Aspect-Enabled Programs. In *Proceedings of the 6th International Symposium on Software Composition (SC)*.

[11] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA'10)*.

[12] Franck Fleurey, Benoit Baudry, Robert B. France, and Sudipto Ghosh. 2007. A Generic Approach for Automatic Model Composition. In *Proceedings of MODELS*.

[13] Thomas Goldschmidt, Steffen Becker, and Erik Burger. 2012. Towards a Tool-Oriented Taxonomy of View-Based Modelling. In *Proceedings of Modellierung*.

[14] Iris Groher and Markus Voelter. 2007. XWeave: models and aspects in concert. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling*.

[15] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2007. An Algebraic View on the Semantics of Model Composition. In *Proceedings of the 3rd European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA)*.

[16] Aram Hovsepyan, Riccardo Scandariato, Stefan Van Baelen, Yolande Berbers, and Wouter Joosen. 2010. From aspect-oriented models to aspect-oriented code?: the maintenance perspective. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD)*.

[17] Kabir Khan. 2008. An Introduction to Aspect-Oriented programming with JBoss AOP. https://dzone.com/articles/an-introduction-aspect-oriente. (2008).

[18] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. Merging Models with the Epsilon Merging Language (EML). In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*.

[19] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. 2006. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proceedings of the International Workshop on Global Integrated Model Management (GaMMa'06)*.

[20] Ramnivas Laddad. 2003. *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press.

[21] Philip Langer, Manuel Wimmer, Jeff Gray, Gerti Kappel, and Antonio Vallecillo. 2012. Language-Specific Model Versioning Based on Signifiers. *Journal of Object Technology* 11, 3 (2012), 4:1–34.

[22] Abid Mehmood and Dayang N. A. Jawawi. 2013. Aspect-oriented model-driven code generation: A systematic mapping study. *Information & Software Technology* 55, 2 (2013), 395–411.

[23] Katharina Mehner and Gabriele Taentzer. 2005. Supporting Aspect-Oriented Modeling with Graph Transformations. In *Proceedings of the Workshop on Early Aspects*.

[24] Dirk Ohst, Michael Welle, and Udo Kelter. 2003. Differences between versions of UML diagrams. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE)*.

[25] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. 2012. *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley.

[26] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. 2010. Developing next generation ADLs through MDE techniques. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*.

[27] Matthias Schöttle, Nishanth Thimmegowda, Omar Alam, Jörg Kienzle, and Gunter Mussbacher. 2015. Feature modelling and traceability for concern-driven software development with TouchCORE. In *Companion Proceedings of the 14th International Conference on Modularity*.

[28] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[29] Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert B. France, and James M. Bieman. 2004. Model Composition Directives. In *Proceedings of the 7th International Conference on UML*.

[30] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. 2014. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software and System Modeling* 13, 1 (2014), 239–272.

[31] Antonio Vallecillo. 2010. On the Combination of Domain Specific Modeling Languages. In *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA)*.

[32] Jon Whittle, Praveen K. Jayaraman, Ahmed M. Elkhodary, Ana Moreira, and João Araújo. 2009. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. *Trans. Aspect-Oriented Software Development* 6 (2009), 191–237.

[33] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elizabeth Kapsammer. 2011. A Survey on UML-based Aspect-oriented Design Modeling. *ACM Comput. Surv.* 43, 4 (2011), 28:1–28:33.