Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2016,7
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling

Proceedings

2 March 2016, Karlsruhe, Germany

Colin Atkinson, Erik Burger, Thomas Goldschmidt, Ralf Reussner
(Editors)

2016

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

Preface

Modern software engineering paradigms, such as model-driven development, multi-view modelling, or role-based software development, use different types and combinations of abstraction techniques to decompose systems into human-tractable pieces. This leads to an increasing number of models and views that have to be considered, which presents fundamental challenges for engineers of complex software-intensive systems. Software developers need technologies for operationally managing views of systems in a consistent way, and software architects require concepts that indicate in which way views and models should be developed, evolved, and navigated as projects evolve.

The goal of this workshop is to distil a common understanding of existing approaches and current research directions in treating heterogeneous models of software and systems.

Workshop Co-organizers

Colin Atkinson
Chair of Software Engineering
University of Mannheim
B6, 26
68159 Mannheim, Germany
E-mail: atkinson@informatik.uni-mannheim.de
Web: http://swt.informatik.uni-mannheim.de/

Erik Burger, Ralf Reussner
Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76131 Karlsruhe, Germany
E-mail: burger@kit.edu, reussner@kit.edu
Web: http://sdq.ipd.kit.edu/

Thomas Goldschmidt
ABB AG
Forschungszentrum
Wallstadter Straße 59
68526 Ladenburg, Germany
E-mail: thomas.goldschmidt@de.abb.com

Programme Committee

Contents

Towards a Configuration Framework for Orthographic-Software-Modeling Environments

Colin Atkinson
Chair of Software Engineering
University of Mannheim
Mannheim, Germany
atkinson@informatik.uni-mannheim.de

Christian Tunjic
Chair of Software Engineering
University of Mannheim
Mannheim, Germany
tunjic@informatik.uni-mannheim.de

ABSTRACT

View-based modeling approaches are today widely used to specify software systems because they allow the associated complexity to be distributed over multiple, separately manageable perspectives of a system. However, virtually all existing view-based modeling approaches have "hardwired" viewpoint frameworks that cannot be easily changed by users. Customizing a view-based modeling environment for a specific methodology is consequently a complex and error prone task since it involves changes to the core implementation code. Therefore, a simple and systematic process allowing methodologists to configure a view-based modeling environment is required which ensures that all requirements for view-based modeling are fulfilled, i.e. that the complete system can be specified using mutually consistent view-types. In this paper we outline such an approach in the context of Orthographic Software Modeling.

CCS Concepts

•**Software and its engineering → Model-driven software engineering;** Abstraction, modeling and modularity;
•**Information systems →** *Information integration;*

Keywords

Model Transformations, Single Underlying Model, Views

1. INTRODUCTION

As the size and complexity of software systems grow so do the size and complexity of the models needed to specify them. View-based modeling approaches, which go back to the early '90s [10], attempt to address this problem by allowing modelers to describe a system using many separately manageable perspectives, thereby separating concerns. As long as views are available for all relevant concerns, and can be used to completely and consistently specify any required information, complete specifications of the system under development can be created.

Over the years numerous view-based specification methods have been proposed such as RM-ODP [13], Archimate [12], View-based textual modelling [11] or the Zachman Framework [16], but these differ widely in the way they conceptualize and support views [5]. Arguably one of the most systematic and complete proposals is the Orthographic Software Modeling (OSM) approach [2] which supports subject-oriented views and is completely prescriptive about what views should be generated in a project, what content they should contain and how they should be related.

The underlying principles of OSM are not tied to any specific set of views or view-types, however. To date, prototype OSM environments have focused on the KobrA [1] method as the underlying motivation for the views to be supported, since this is based on precisely the principles of strictly prescribed views and view content (in fact KobrA was the inspiration for OSM). However in principle, any suitably adapted view-based method could be supported using an OSM environment as long as the views are not hardwired into the implementation as in previous prototypes ([6]).

To make this possible in practice, an OSM environment must be implemented in a generic way so that it can be configured using relatively straightforward steps to support different view-based methods. However, this is easier said than done. First, since OSM uses a hyper-cube based approach for identifying views, view-types and navigation routes, the appropriate dimensions and dimension elements need to be defined. Second, since OSM is based on a strictly-projective approach to view generation, the appropriate Single Underlying Model (SUM) and view-type definitions need to be defined. Third, since OSM supports subject-based views, the appropriate dimension elements and view-type parameterizations need to be defined. Finally, this configuration information needs to be packaged into concise methodology descriptions which can be easily understood by human users and efficiently put into effect by the OSM environment.

In this paper we describe our preliminary work in realizing such a configuration approach for a prototype version of a generic OSM environment at the University of Mannheim. In the next section we first describe the idea of OSM method configurations and the elements from which they are constructed. In Section 3 we then describe how view-types are defined and parameterized to support views focused on different subjects. Section 4 then describes how view-type definitions and views are assigned to cells of the hyper-cube to support dimension based navigation, and Section 5 provides an overview of the overall process used to construct OSM method configurations. Section 6 concludes the paper.

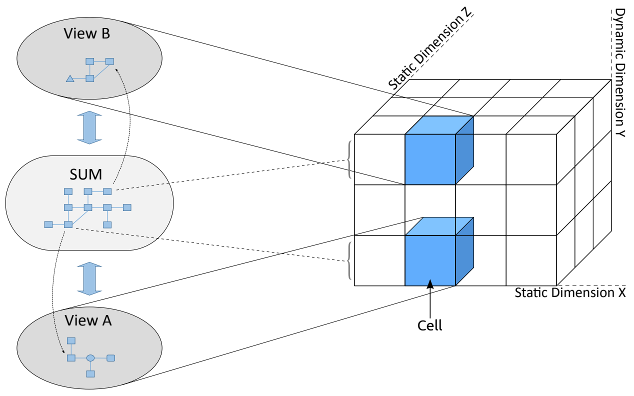


Figure 1: OSM Method Configuration Overview

2. OSM METHOD CONFIGURATIONS

A generic OSM environment has to be customized to a specific view-based methodology before it can be used to model a particular software system. In our approach this is achieved by means of a so called **OSM method configuration** which defines what views need to be created to specify a system, what these views should contain and how these should relate to one another. The process of configuring an OSM method configuration is performed by a so called **methodologist** who is an expert in the chosen methodology, but is not involved in the specification process. This is perform by an **architect** or modeler once the OSM method configuration has been deployed and the generic OSM environment has been instantiated for the specific methodology to be used.

An OSM method configuration consists of three parts: (a) the definition of the concepts used in the single-underlying-model to capture user data, (b) the definition of views projected from the SUM which are used to visualize and add data to the SUM and (c) the definition of a hyper-cube defining the navigation routes for accessing the available views.

Formally, the ingredients of an OSM method definition (M) can be defined as a set containing the SUM (S), view definitions (VD) and a hyper-cube (HC), where VD is the set containing all defined views (V_x).

$$M := \{S, VD, HC\}$$
$$VD := \{V_1, ..., V_n\}$$

Figure 1 gives an overview of an OSM method configuration. The left-hand side presents the SUM and two views ($ViewA, ViewB$) which are projections of particular parts of the SUM. The right-hand side presents the multi-dimensional hyper-cube which in this example is spanned by two static dimensions (X, Z) and one dynamic dimension (Y). Each cell of the hyper-cube is identified by a unique set of choices for each dimension and represents a viewpoint that can have at most one associated view.

Static dimensions have elements (i.e. choices) that are predefined by the method (e.g. the PIM or PSM levels in a platform-independence dimension), while dynamic dimensions evolve as part of the specified model. The dynamic dimension *Component* for example might have elements (i.e. values or choices) corresponding to the components that are available in the SUM. This is shown in Figure 1 by the dashed lines connecting a concept from the SUM with a value in the dynamic dimension Y. The assignment of views to cells of the hyper-cube is part of the OSM method configuration and is performed by a methodologist.

In general, of course an OSM method configuration contains more than the two views shown in Figure 1 and the hyper-cube can be spanned by more than three dimensions.

3. PARAMETERIZED VIEW-TYPE DEFINITIONS

The views in view-based modeling approaches show a particular part of the SUM describing the system under development. How they do this is determined by three things: (a) what kind of information is shown by the view, (b) what notation is used to show the information and (c) which specific part of the SUM is shown in the view. The first two of these (a and b) can be defined a priori by the methodologist (i.e. before the start of the system modeling process) and thus can be thought of as representing the type of the view, while the latter (c) can only be determined as the model evolves and thus represents the focus or subject of the view.

At environment configuration time, therefore, the only ingredient of views that can be fixed is their view-type. In order to re-use the view-type definitions the content of a view – i.e. the information from the SUM to be presented by an "instance" of the view-type – can be selected via a parameter of the model-to-model transformations that "project" SUM information into a view. This allows the subject of the view, which is the only dynamic value controlling the view generation, to be separated from the static view-type definition using model-to-model transformations and domain specific language definitions.

The possibility to use parameters in view generation leads to views which have a "focus", i.e. they show the system from a particular perspective focused on a particular set of concepts rather than showing an arbitrary part of the system using a predefined, generic view of the entire system. In other words, this ability supports so called *subject*-oriented views in which views can differ by their subject as well as their type. As far as we know OSM [2] is the only approach which supports this capability. Most view-based approaches, such as Archimate [12] only allow users to "focus" views by defining ad-hoc content or to populate predefined views with ad-hoc content, or both [3, 4]. The flexible views approach of Burger [8] allows view content to be defined at projection time, but does not tie its selection to navigation dimensions.

Figure 1 shows two views of the same type projected from the SUM. Although their view-type is the same, however, their content is different since they have different subjects, as defined by the parameter to the project transformations – in this case the parameters are values of the dynamic dimension Y. The parameter(s) of view-types that define the focus of specific views (i.e determine their *subject*) are matched to dynamic dimension elements from the SUM. Moreover, these dynamic dimension elements are usually represented in the views. This is shown in Figure 1 by the dashed arrows pointing from the concepts in the SUM to their corresponding projection in the views.

4. HYPER-CUBE CONFIGURATION

In order to provide an intuitive and efficient navigation metaphor over all available views in an OSM environment

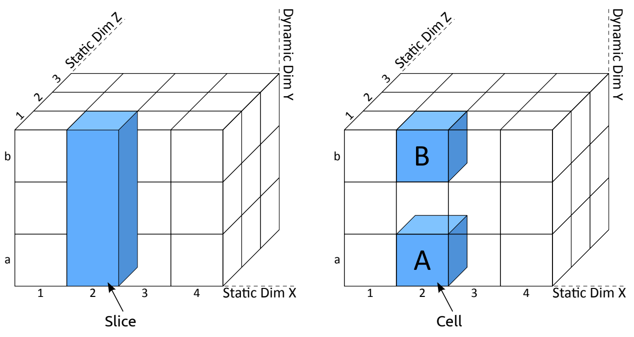


Figure 2: Slice and Cell in Hyper-Cube

we use a multi-dimensional hyper-cube. This is analogous to the conceptual cube used in data warehousing [14]. The online analytical processing (OLAP) approach is used to define "views" which present business reports for business intelligence by answering multi-dimensional analytical queries [9]. In contrast to our navigation approach the OLAP approach is driven exclusively by dynamic dimensions. The combination of both kinds of dimensions (i.e. static and dynamic dimensions) is unique to OSM.

The static dimensions which span the hyper-cube are defined by the methodologist and contain static values which cannot be changed by architects. Once the dimensions and dimension elements of the hyper-cube have been defined, a key step is to assign views to its cells. In the first step, the view-type definition is assigned to a slice of the hyper-cube defined by static dimensions by choosing one value for each static dimension (note that both the view-type and the slice are static). Such a slice is shown in the left-hand side of Figure 2 where the static dimension X is set to value 2 and static dimension Z to value 1. The dynamic dimension Y is not set to a specific value. The right-hand side of Figure 2 shows two cells (A, B) which are part of the slice in the left-hand side hyper-cube. The cells are distinguished by concrete values from the dynamic dimension Y – value a for cell A and value b for cell B. The slices defined by choosing concrete values for all static dimensions contain all views of one particular view-type.

In the case of Figure 1, the slice spanned by values of the static dimensions contains both views (blue cells). In other words, both views in the example $(ViewA, ViewB)$ have the same view-type since their static dimension values are the same. In the second step a specific view is "instantiated" from the view-type by choosing values from the dynamic dimensions for the parameters for the model-to-model transformations of the view-type. In order for the model-to-model transformation to be able to generate the view the values of the dynamic dimensions assigned to view-types must be compatible with the parameters. Once a valid assignment has been made the appropriate view can be generated and associated with the corresponding cell which is uniquely identified by the choice of dimension elements.

In implementation terms, the static dimensions corresponding to a view-type are "hardwired" into the transformation definitions while the assignment of dynamic dimensions to parameters is achieved using wildcards since these values are not available at the time the configuration is performed. The wildcards are evaluated at runtime when a concrete system specification (i.e. SUM content) is available. Using wildcards it is possible to specify that a view exists only if a (specific) value in a particular dynamic dimension exists. If, for example, one dynamic dimension contains all *components* which are available in the SUM, then the views which require the existence of a value of this dynamic dimension would only be available if at least one *component* exists in the SUM. The chosen example is very simple in order to convey the core ideas. In order to support more complex values in dynamic dimensions, decision trees [7] supporting decisions over multiple levels can be used.

Ideally all cells of the hyper-cube should have views assigned to them since sparse cells are undesirable in an OSM environment in order to enhance the efficiency of the navigation approach. The views which are assigned to cells form the VD part of the OSM environment definition for a given methodology (M).

5. CONFIGURATION PROCESS

The concepts used to visualize information in a view are usually similar, if not identical, to those used to represent the corresponding information in the SUM, and often the two are designed together. The subset of concepts in the SUM from which the concepts in a view-type are projected is referred to as the "scope‘" of the view. As explained above, the task of the methodologist is to define how instances of concepts in the scope of a view-type are mapped to instances of concepts in the view language based on the specified subject of the view at projection-time.

Figure 3 shows the process supporting the configuration of an OSM environment for a particular methodology. The rectangle notation is used for activities while the parallelogram notation is used for artifacts. Solid lines show the flow of activities while the dashed lines show the flow of artifacts denoting where the artifacts are produced and used.

In order to adapt an existing methodology for use in an OSM environment the methodology's existing support for OSM concepts must first be determined, e.g. what view-types are needed, what dimensions are recognizable and what is the subject of existing views. These questions are answered in the **analyse method** step and are stored in the artifact *method description*. Using the method description the SUM, the views and the hyper-cube are defined in the step **configure OSM environment**. This contains three sub-steps for the needed actions. In the first step, **define SUM/types**, the methodologist defines the types needed to capture the domain information that needs to be stored in the SUM. The resulting artifact of this step is the SUM (S) metamodel. The step, **define view-types**, is used for defining the views which are needed according to the *method description*. Defining the views means to identify the view-types and write the projection rules describing how views are projected from the SUM. In order to ensure the propagation of information in both directions synchronizable, projective views are needed [15]. The resulting artifact here is the set of all view-type definitions for the methodology (VD). Finally, the step, **define hyper-cube**, allows the methodologist to define the hyper-cube for view navigation using the information from the *method description*. Defining the hyper-cube means to define the static and dynamic dimensions that span the hyper-cube. Furthermore, this step is used to assign views to the cells of the hyper-cube. The artifact which results from this step is the hyper-cube (HC) description. All the sub-steps of the **configure OSM environment** step use the *method description* and contribute

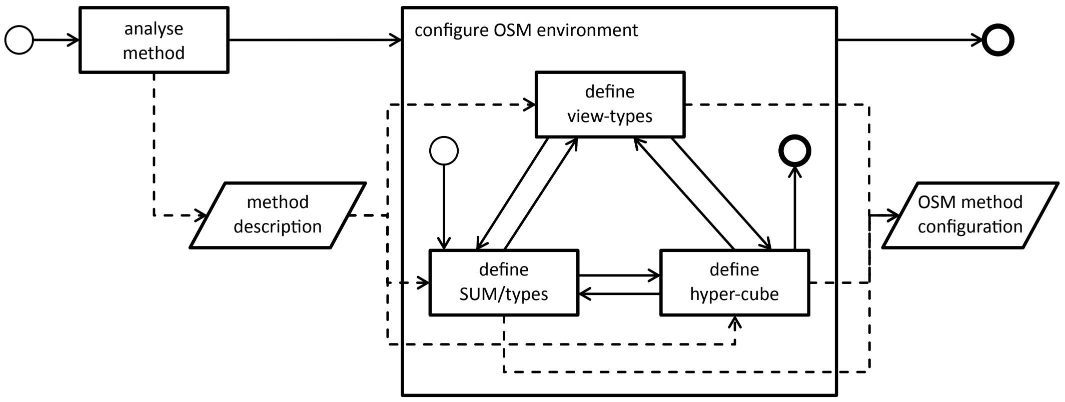


Figure 3: Process for Configuration of OSM Environment (role = methodologist)

a specific piece of information to the common resulting artifact – the *OSM method configuration*. Since the definition of the SUM, the views and the hyper-cube go hand in hand the three sub-steps can be applied in an arbitrary order and as often as needed.

6. CONCLUSION

In this paper we presented an approach for defining OSM method configurations for customizing a generic OSM environment to support particular view-based methods. We described the ingredients for an OSM configuration description and the concept of parameterized view-type definitions which use values from a conceptual navigation hyper-cube to specify the subject of a view. Furthermore we showed how view-types and view (instances) can be assigned to the cells of this hyper-cube to provide a simple, intuitive and systematic way of navigating over all views. The presented process simplifies the task of configuring an OSM environment to support and enforce the rules of systematic view-based modeling environments such as RM-ODP [13].

7. REFERENCES

[1] C. Atkinson. *Component-based Product Line Engineering with UML*. Addison-Wesley object technology series. Addison-Wesley, 2002.

[2] C. Atkinson, D. Stoll, and P. Bostan. Orthographic software modeling: A practical approach to view-based development. In *Communications in Computer and Information Science*, pages 206–219. Springer Science + Business Media, 2010.

[3] C. Atkinson and C. Tunjic. Criteria for orthographic viewpoints. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '14*. Association for Computing Machinery (ACM), 2014.

[4] C. Atkinson and C. Tunjic. Towards orthographic viewpoints for enterprise architecture modeling. In *18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOC Workshops 2014, Ulm, Germany, September 1-2, 2014*, pages 347–355, 2014.

[5] C. Atkinson, C. Tunjic, and T. Moller. Fundamental realization strategies for multi-view specification environments. In *Enterprise Distributed Object Computing Conference (EDOC), 2015 IEEE 19th International*, pages 40–49, Sept 2015.

[6] C. Atkinson, C. Tunjic, D. Stoll, and J. Robin. A prototype implementation of an orthographic software modeling environment. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '13*, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[7] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[8] E. Burger. *Flexible Views for View-based Model-driven Development*. PhD thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, July 2014.

[9] E. Codd. Providing OLAP to User-Analysts: An IT Mandate. Codd & Date. 1993.

[10] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 02(01):31–57, mar 1992.

[11] T. Goldschmidt. *View-based textual modelling*. PhD thesis, Karlsruhe, 2011.

[12] M. Iacob, D. H. Jonkers, M. Lankhorst, E. Proper, and D. D. Quartel. Archimate 2.0 specification: The open group. Van Haren Publishing, 2012.

[13] ISO/IEC. RM-ODP. Reference Model for Open Distributed Processing. *ISO/IEC 10746, ITU-T Rec. X.901-X.904*, 1997.

[14] R. Kimball and M. Ross. The Data Warehouse Toolkit. *Data Wareh. Toolkit - Complet. Guid. to Dimens. Model.*, 2002.

[15] C. Tunjic and C. Atkinson. Synchronization of Projective Views on a Single-Underlying-Model. In *Proc. 2015 Jt. MORSE/VAO Work. Model. Robot Softw. Eng. View-based Software-Engineering - MORSE/VAO '15*, pages 55–58, New York, New York, USA, 2015. ACM Press.

[16] J. A. Zachman. A framework for information systems architecture. *IBM Syst. J.*, 26(3):276–292, 1987.

Projecting UML Class Diagrams from Java Code Models

Heiko Klare, Michael Langhammer, and Max E. Kramer
Chair for Software Design and Quality (SDQ)
Institute for Program Structures and Data Organization (IPD)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{heiko.klare@student.kit.edu, michael.langhammer@kit.edu, max.e.kramer@kit.edu

ABSTRACT

In model-driven software development, source code and other artifacts are used to describe and develop a software system. UML class diagrams are one of the most common models that are used. A UML class diagram models classes and interfaces of a software system as well as their relations.

The usage of UML class diagrams in addition to source code can lead to drift and erosion if the models are not kept consistent with code changes and vice versa: Existing solutions solve this problem using consistency mechanisms that update the source code and UML class diagram accordingly. The development and maintenance of such consistency mechanisms can result in considerable effort and costs.

In this paper, we present a prototype for a new UML class diagram editor that is realized as a projection of a Java source code model. The editor does not use an explicit UML model. It provides another concrete syntax for a subset of the source code elements and their relations. A model represenation of the source code is used as a single underlying model (SUM) for the projective UML class diagram view. As a result, code and diagrams are updated automatically without the need for a consistency mechanism. The current minimal UML class diagram editor uses state-of-the-art model-driven software technologies and adds less than 2000 LLOC to the Eclipse IDE, Sirius and JaMoPP.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods; D.2.11 [**Software Architectures**]: Languages

Keywords

UML class diagram, projective view, round-trip engineering

1. INTRODUCTION

In modern model-driven software development, not only source code is used to develop a software system, but also other artifacts, such as component or class diagrams. For object-oriented software development, UML class diagrams [Obj11] are a common language to model software systems [Lan+14]. Within a UML class diagram, elements such as classes, interfaces and methods as well as the relations between them are represented. Certain details, such as the implementation of method bodies, are not shown. Hence, an UML class diagram can provide a quick overview of a software system. To have up-to-date UML class diagrams during the evolution of a software system, the UML class diagrams have to be kept consistent with the source code. If the diagrams are not kept consistent with the code, the well-known problems architecture drift and architecture erosion [PW92] arise. To solve these problems, a lot of tools, especially for UML class diagrams, already exist.

We subdivide these tools into three categories. The first category of tools generate UML class diagrams dynamically from the source code. This generated diagrams help to get an overview about the software system but can not be edited. The second category comprises forward engineering tools, which can be used to generate a blueprint of the system's source code from UML class diagrams. The third category contains tools that combine the first two categories. Hence, tools in the third category support round-trip engineering between source code and UML class diagrams. This means as soon as developers change the code the architecture is updated immediately and visa versa. A lot of tools that fall into the third category, e.g. UML Lab[1], have been developed. To our knowledge, all of these tools use an explicit consistency mechanism to keep the source code consistent with another model, which is used for the UML class diagram.

In this paper, we present a round-trip engineering approach for the third category that does not need to use an explicit consistency mechanism. Instead, we create a projective view onto the source code, which omits the problem of keeping different artifacts consistent during the development process. By creating a projective view, we achieve consistency between UML class diagrams and source code by design: all changes made in the UML class diagram editor are changes to a different representation of a subset of the source code so that there is no need for change propagation or something similar. Since we do not use a consistency mechanism, our editor should be easier to maintain in cases where either the UML metamodel or the source code metamodel changes. In such cases, we only have to update the view definition instead of changing the consistency mechanisms. Our approach can be seen as an implementation of the Orthographic Software

[1]http://www.uml-lab.com/

VAO 2016, 2 March, 2016, Karlsruhe, Germany
© 2016 Copyright held by the owner/author(s).
ISSN 2190-4782.

Modelling (OSM) approach, which was introduced by Atkinson et al. [ASB10]. OSM introduces the idea of a Single Underlying Model (SUM) that contains all artifacts, which are used to develop a specific software system. The access to the SUM is only possible via defined views. In our approach, we use the source code as the SUM. As views, we use the standard source code view and the UML class diagram view. Figure 1 exemplifies the functionality of our editor. The figure shows a class diagram with two classes, two interfaces and three methods. An edit operation is performed in step (1): A new method called *download* is added within the source code. The UML editor is updated (2) after saving the source code files containing the new information.

If one wants to show information in the UML class diagram that can not be generated from the source code solely, we propose, similar to other tools, an embedding mechanism into the source code. One important concept, for example, that can be used within the UML diagram but does not have a source code equivalent are the associations. To embed associations in the source code, we have developed annotations that allow developers to define the source and target multiplicity as well as the kind of the association. Hence, the creation of a projective view is possible because our UML class diagram does not contain information that is not contained within the source code already.

Our prototypical implementation of the Projective UML class diagram editor for Java (ProjUMLed4J) can be downloaded online[2] and uses state-of-the-art model-driven software development technologies: ProjUMLed4J is based on Eclipse Sirius [VMP14] and the Java Model Parser and Printer (JaMoPP) [Hei+10]. Both tools are realized using the Eclipse Modeling Framework (EMF). While our approach for a projective UML editor could be used for any object-oriented language, the implementation is currently limited to Java.

The remainder of the paper is structured as follows. In Section 2 we introduce the necessary foundations. Section 3 gives an overview about the related work. In Section 4 we explain the realization and features of our new UML editor. In Section 5 we discuss the advantages and disadvantages of our approach. Section 6 concludes the paper and gives a brief outlook on the future work.

2. FOUNDATIONS

2.1 Model-Driven Software Development

In MDSD (Model-Driven Software Development) models are the primary artifacts of the development process. This means that in contrast to model-based software development, models are not only used for some tasks such as documentation. Instead, every artifact is either a model conforming to an explicit metamodel, which describes the set of allowed instances, or it is derived from a model. Therefore, even code is either generated from models or treated as a model that is used to obtain other models. Source code can be treated as a textual representation of a code model with special printing and parsing capabilities. This makes it possible to apply techniques that were developed for arbitrary models and to abstract away from the textual nature of code. Programs that consume or produce models are called model transformations and can also be described as model instances of a transformation metamodel.

[2]https://sdqweb.ipd.kit.edu/wiki/Vitruvius/ProjUMLed4J

2.2 Class Diagrams of the Unified Modeling Language (UML)

A well-known diagram type of the UML ISO/IEC 19505-2:2012(E) standard are class diagrams. They represent classes and interfaces of object-oriented software that may be grouped into packages. Many concepts of class diagrams, such as inheritance, attributes, or operations, have direct equivalents in object-oriented programming languages, such as Java. The initial purpose of the UML was, however, not code generation or co-evolution but support for the documentation and design of software. Therefore, there is not a universal mapping to object-oriented code for all concepts that are available in UML class diagrams. Special associations between classes, such as aggregations, which represent whole-part relationships, can, for example, be realized in different ways in object-oriented code. Round-trip engineering tools for UML class diagrams and object-oriented code rely on such mappings although they do not always make them explicit.

2.3 Eclipse Modeling Framework (EMF) and Sirius

EMF is a set of tools and plug-ins for the Eclipse IDE that provides all infrastructure necessary for model-driven development. It provides the metamodelling language Ecore, which is closely aligned to the Essential Meta-Object Facility standard ISO/IEC 19508:2014(E), and is used to define metamodels and implement their instances in a variety of tools and domains. Code generation facilities of EMF provide a very convenient way to obtain code for instantiating Ecore metamodels and editing in a customizable tree-based editor.

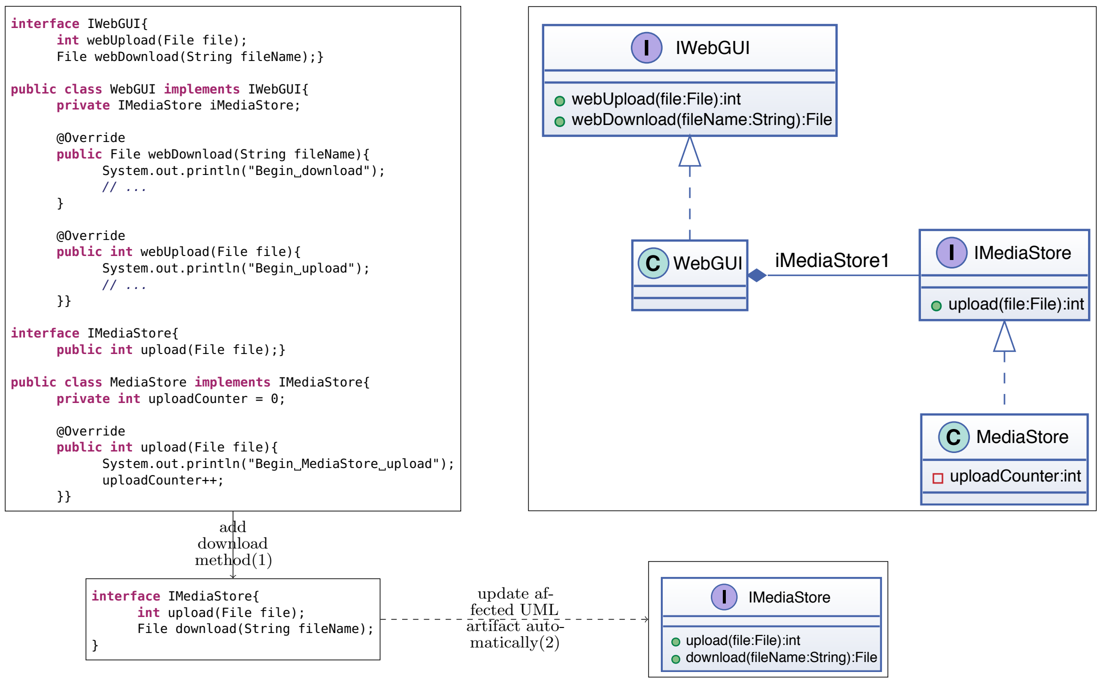
To obtain grapical editors for Ecore-based models the Sirius [VMP14] framework can be used. With Sirius, so called representation descriptions can be defined. These descriptions specify how elements of the represented models shall be displayed and how they can be edited. A description can be instantiated for an Ecore-based model and results in a diagram for that model. The descriptions are interpreted dynamically, which means that an existing diagram is opened with the actual version of the representation description and not with the one it was created with. Each description must be assigned to a viewpoint, which summarizes descriptions that belong together and can be simultaneously activated or deactivated for a project that uses Sirius diagrams. To create diagrams with Sirius, a Sirius session must be created. A Sirius session exists for each project that uses Sirius diagrams. The session contains references to the used models and the created representations with their layout information. It is persisted in a special file in the root folder of the project.

2.4 Java Model Printer and Parser (JaMoPP)

JaMoPP [Hei+10] parses Java source code and represents it as an instance of an Ecore metamodel so that model-driven tools can be applied to it. It also supports printing generated or modified code models without any loss of information. Printing and parsing of source code is performed in a fully transparent way so that transformations and analyses of Java models do not have to consider the special serialization as Java source files. Cross-references between model elements are established after parsing so that tools that use JaMoPP can easily navigate the models and perform analyses.

2.5 Software Architecture Views

The ISO/IEC/IEEE 42010:2011(E) standard mentions two



```
interface IWebGUI{
      int webUpload(File file);
      File webDownload(String fileName);}

public class WebGUI implements IWebGUI{
      private IMediaStore iMediaStore;

      @Override
      public File webDownload(String fileName){
            System.out.println("Begin_download");
            // ...
      }

      @Override
      public int webUpload(File file){
            System.out.println("Begin_upload");
            // ...
      }}

interface IMediaStore{
      public int upload(File file);}

public class MediaStore implements IMediaStore{
      private int uploadCounter = 0;

      @Override
      public int upload(File file){
            System.out.println("Begin_MediaStore_upload");
            uploadCounter++;
      }}
```

```
interface IMediaStore{
      int upload(File file);
      File download(String fileName);
}
```

Figure 1: Example of the UML editor's functionality: After adding the *download* method to the interface *IMediaStore* the UML editor is automatically updated.

different approaches for architectural views that can be applied to any kind of view: *Synthetic* approaches integrate views with each other and have to manage correspondences and updates. *Projective* approaches use a central repository from which all views are derived respectively projected.

3. RELATED WORK

UML class diagrams are a common and established representation of software fragments. In the last years, many tools for generating, editing and viewing class diagrams have been developed. As we already introduced in Section 1, we subdivide these tools into three categories, which are *read-only views* generated directly from the source code, *forward engineering tools* that generate code stubs from the UML class diagram, and *round-trip engineering tools* providing a synchronization mechanism for code and the diagram. Since our tool falls into the third category, we compare our approach with others in the same category.

UML class diagram editors providing round-trip engineering with the source code can again be subdivided into three different categories based on the way they represent the additional semantic information, which a class diagram contains as opposed to source code. Most of the existing tools fall into the first category. They either use a separate UML model that stores the whole information that is needed for the diagram or an artifact containing the additional information compared to the source code. Tools of the second category use a central model that stores the whole information of all related artifacts. Thus, the model contains the UML information as well as implementation and further details. The third and last category aggregates tools which use no additional artifact to store semantic information but extract it from the code or, if necessary, store it directly in it.

The list of approaches that fall in the first category is long. Popular tools are ArgoUML [Ram+03], IBM Rational Software Architect [Cla10], MagicDraw [No 12] and UML Lab, which arose from FuJaBa [Nic+00]. The first three approaches use explicit synchronization mechanisms that must be triggered by the user. While an explicit update in one direction, from diagram to code or vice versa, mostly tries to integrate the changes into the other artifact, some tools simply generate the other artifact again. In the case of ArgoUML, this means that changes in a UML class diagram can only be transformed into a new and empty code template but not be integrated into existing code. MagicDraw also provides an explicit synchronization mechanism of diagram and code. Changes are integrated into the other artifact. Nevertheless, even simple modifications such as the renaming of a field are not detected. Instead, the field exists twice after the synchronization of the renaming operation. On the contrary, UML Lab uses an approach combining implicit and explicit synchronization. If an artifact that is affected by a change is currently opened, it is automatically updated. For instance, if a class is opened in a Java source code editor and modified in a UML class diagram, this modification is automatically synchronized with the code. To ensure consistency with closed artifacts, an explicit synchronization can also be triggered. UML Lab also stores the additional semantic information of UML class diagrams inside the code using formalized code comments. This allows the sharing of these information without sharing the diagram itself. The Rational Software Architect also provides a primarily implicit synchronization mechanism. Modifications in one view are transferred to the other. However, modifications of associations inside the code are not represented in the UML diagram.

A popular tool that falls into the second category is the Enterprise Architect [Spa14]. It provides a toolset for the design of software architectures containing several UML diagram types. A central model contains the information that is necessary for all diagrams and for the generation of code for different programming language from the architectural design. Modifications in a UML class diagram are persisted in this central model. However, the synchronization with the source code has to be called explicitly. This also means that concurrent modifications in the diagram and the code are not possible because one overwrites the other.

The last category covers tools that use only the source code as the model for the UML class diagram. Because these tools do not use an additional model for storing further information, there is no mechanism for the synchronization of semantic elements. A popular tool that falls into this category is Together from Borland [Bor05]. The central feature of Together is the so called LiveSource mechanism, which synchronizes the class diagram with the source code. The semantic information that is represented by the UML class diagram is either already stored in the source code or persisted in formalized code comments, for example, for multiplicities of associations. Together is a quite unintrusive tool since it does not influence an existing Java project apart from the mentioned code comments. The layout information of the class diagrams is stored in a separate folder of the project.

A tool that follows an approach that is similar to the one we present here was published by the developers of JaMoPP. Their GMF-based editor [Hei+09] uses JaMoPP to present the classes inside a package and their relationships graphically. The editor is not UML compliant and is not compatible with the latest version of Eclipse but is conceptually also a graphical editor that uses the source code as the underlying model. The UML Aid Explorer[3] is another tools that can be assigned to the third category of UML class diagram tools. It provides the generation of read-only class diagrams based on the source code as the underlying model. Due to this mechanism, the diagram is always in sync with the source code. Additional information of UML class diagrams is extracted from the model, if possible, but cannot be modified due to the missing editability of the diagram contents.

The language workbench mbeddr [Voe+13] is built on top of the Meta Programming System (MPS) [Voe13] and allows projective editing of Java source code [PSV13] and many other languages. It uses an Abstract Syntax Tree (AST) instead of a textual serialization from which all views are projected. All manipulations in the views are translated to manipulations of the central AST. Java code is edited in a projective editor that offers similar functionality to editors that have to parse the code. Currently, it is however only possible to visualize and navigate UML class diagrams in mbeddr but no edit operations are supported.

4. UML CLASS DIAGRAM EDITOR

The editor that we introduce in this paper provides functionality for the viewing and editing of classes in a UML class diagram that builds on Java source code as the underlying model. A single diagram represents one package of an existing Java project since a package represents a semantically isolated set of classes. However, the concepts we present can be easily extended to arbitrary sets of classes.

[3]http://www.objectaid.com

The implementation of our approach uses the EMF tools Sirius and JaMoPP. JaMoPP allows us to treat Java source code as an Ecore-based model. For the graphical representation of the model as a UML class diagram we chose Sirius. Certainly, any other tools providing these functionalities can be used for these tasks. The use of state-of-the-art model-driven software development technologies allows us to implement our minimal prototype in less than 2000 LLOC. The remainder of this section explains how our diagram generation and diagram editor work, and what the limitations of the implementation currently are.

4.1 Persisting semantic UML information

UML class diagrams are highly abstract representations of software fragments. Thus, most of their elements can also be found in more precise artifacts such as source code. Only few UML elements do not have a clear equivalent in the code. Especially association properties, such as multiplicities, belong to these elements that can be implemented in code in many different ways, if at all, and thus cannot be unambiguously extracted from code. This information has to be stored separately.

In Section 3, we summarized several tools that synchronize program code and UML class diagrams and therefore use some mechanism to store the additional semantic information of class diagrams. While most of them use a second artifact to store some kind of code-independent model that is supplemented with this information, some of them, such as UML Lab, embed the information in the program code.

Since we rely on the capabilities of Sirius and JaMoPP, the easiest way to provide the UML specific information is to integrate it into the source code. This decision has several further advantages regarding consistency, which we will discuss later. Since the information increases the code size, we have decided to use Java annotations for storing it. Annotations are compact and cannot be easily corrupted like comments, for instance, through accidental modifications in the Java code view, because they are a feature of the programming language that is syntactically and semantically checked by the compiler.

An association is stored by writing the annotation *@Association* next to a field that represents the association in one direction. Default values for all necessary association properties are specified in the annotation definition and can be overwritten in each case. Currently, supported values are the multiplicities and the type of an association, which can be an aggregation or a composition.

4.2 Diagram generation process

Our process of generating a UML class diagram for a specific Java package can be generally separated into two steps: Initially, an optional preprocessing step extracts semantic UML information out of the program code and embeds this information in the code to generate a more expressive diagram. The second step is the diagram generation itself, that again consists of several steps that complete in a UML class diagram that is opened in an editor.

Source code preprocessing

Some of the information in a UML class diagram that cannot be obviously extracted from the program code can be approximated conservatively. We assume that a field with a type that is defined in the same package shall be presented as an association. Therefore, an association annotation is created for all the affected elements.

An interesting property of associations are the multiplicities. The common multiplicities *0..1* and *0..** can be reliably reproduced from code. Consider the following listing. During the generation of a UML diagram we generate the annotation *@Association* for the attribute *myStringList* if *MyString* is in the same package as *MyClass*. The multiplicity values are represented by annotation attributes and are set to *1* by default. Since the *myStringList* references an arbitrary number of *MyString* objects, the target multiplicity is overwritten with *0..**, whereas the upper bound is represented by *-1*.

```
public MyClass {
     @Association(targetLowerMultiplicity=0,
      targetUpperMultiplicity=-1)
     private MyString[] myStringList; }
```

More precise multiplicities, especially limited ranges, are hard to assure in code and even more hard to extract from code. Generally, there are two types of associations. If they are single-valued, the code contains a field with the type of the association target. Furthermore, if a field is declared as *final*, its value cannot be changed and thus the multiplicity can be set to *1*. By contrast, if the association is multi-valued, the field is a collection of the association target type, which can be any class implementing the *Collection* interface, an array, or a user-defined container class. While the first two can be easily determined by investigating the inheritance hierarchy respectively check whether the type is used within an array, the latter one cannot be recognized as a multi-valued association generally.

We generate this information for each field of classes in the considered package investigating and modifying the JaMoPP model of the code. This process could also be performed based on the Abstract Syntax Tree to avoid the complexity of JaMoPP. To improve the approximation results, the process could also be realized semi-automatically. In reasonable cases, the user could be asked for more specific property values. An example could be the check of a collection size for a limited value inside a method, which might be an indicator for the upper bound of the association multiplicity. To reduce the number of annotations, our approach could be easily adapted to only persist annotations for non-default values and to implicitly handle default values in the view.

In contrast, an association cannot be automatically identified as an aggregation or composition since the information can even not be extracted using static Java code analysis.

Diagram generation

The generation of a UML class diagram consists of three major steps. At first, a Sirius session has to be created and prepared. After that, the needed resources have to be added to the session and finally the diagram has to be instantiated. The diagram generation process is shown in Figure 2.

Sirius sessions are persisted in special files saving the resources and diagrams the session contains. The implementation of Sirius assumes such a file for every project that diagrams shall be created for. Thus, the diagram generation initially creates a session for the affected project if it does not already exist. Otherwise, the existing session is opened, which means that the persisted state, that contains the resources and the already created diagrams with their layout information, is read and resources of the session are loaded.

A Sirius session may contain different types of diagrams. These types are organized in viewpoints, which can be enabled or disabled. To enable the creation of UML class diagrams, its containing viewpoint is activated in the Sirius session.

For each diagram, Sirius needs a single root element whose content is displayed. Therefore, we cannot simply add the Java source files we want to display but need an element containing them. Since we decided to present a single Java package in each diagram, we use the package as the root element. Packages are not persisted but implicitly defined through the package declaration of Java classes. Thus, we have implemented a virtual package EMF resource that generates a package element instance of the JaMoPP metamodel. Afterwards, the Java classes inside the package are added as compilation units to this package element. This package resource is added to the Sirius session to be used as the root element for the diagram that is about to be created.

Sirius has an integrated mechanism for synchronizing the used resources with the diagram contents. It observes the persisted resources for modifications and reloads them in case of a change. Since the added package is a virtual element and has no persisted equivalent that can be monitored for changes, in this state no synchronization between code and the diagram would happen. To achieve this, the resources of the compilation units inside the virtual package element must be added to the Sirius session as well. Using the resources that are contained in the package element, Sirius reloads the Java classes on changes and implicitly updates the classes inside the virtual package that is displayed as well.

The mechanics of JaMoPP require a further adjustment of the previous process. When loading a class with JaMoPP, a proxy resolution, loading all classes which the currently loaded one depends on, is performed by default. These dependencies are treated as special proxy objects to avoid that their dependencies are resolved as well. If, in our scenario, a class of the package that is currently loaded depends on a class of the same package that will be loaded later, a proxy object gets created first. Although the class is completely loaded afterwards, the proxy object stays in the resource set. This way the Sirius session could potentially load classes twice, which will cause synchronization problems. If a class gets modified in the diagram, this change gets written back to the code. The proxy objects of the same class recognize that change of their resource and, therefore, this change is tried to be written back to the model. This violates the read-only state of the transaction, which is founded in the fact that the model should only be read to write the change to the Java file. Because of that, we need to ensure that the Sirius session contains each class only once. We achieve this by disabling the proxy resolution mechanism of JaMoPP while adding the classes to the Sirius session and reactivate it afterwards.

Finally, we can create an instance of our Sirius UML class diagram definition for the virtual package element. The diagram is automatically generated for the contents of the given package element with a default layout determined by Sirius and is opened in a Sirius diagram editor.

4.3 UML class diagram usage

After creating and opening the UML class diagram, a state-of-the-art Eclipse Sirius editor is shown. This editor shows the classes, interfaces and methods with parameters and return types as well as the visibility of the elements. Furthermore, it shows the relations of the elements in terms of inheritance and associations.

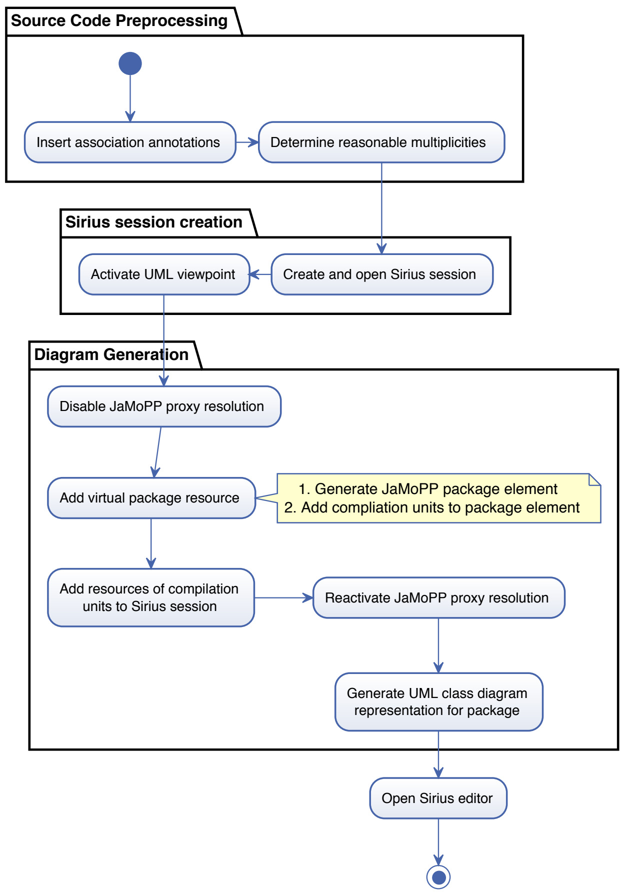

Source Code Preprocessing
Insert association annotations
Determine reasonable multiplicities
Sirius session creation
Activate UML viewpoint
Create and open Sirius session
Diagram Generation
Disable JaMoPP proxy resolution
Add virtual package resource
1. Generate JaMoPP package element
2. Add compliation units to package element
Add resources of compilation units to Sirius session
Reactivate JaMoPP proxy resolution
Generate UML class diagram representation for package
Open Sirius editor

Figure 2: UML class diagram generation process for a specific package

As mentioned above, the editor allows users to change the model elements. For example, a name of the method can be changed by clicking on the method and typing the new name. Since the editor is just another view onto the source code, the source code is updated automatically after saving the editor. For rename operations, the editor automatically supports refactoring for the classes that are currently displayed. The reason for that is that each object of the model instance is created and instantiated once by JaMoPP. Other occurrences of the objects in the same model are only references to the original model. Moreover, the toolbox of the editor allows users the creation of fields, methods, classes and interfaces.

4.4 Features

We already introduced the functionality of the editor in Section 1. Further features are presented in this section. Within our editor, the navigation from the UML class diagram to the source code is possible by double clicking on the class.

Figure 3 shows the steps that are necessary to create a method or a class. The creation of a method is rather simple since we only have to create a new method element in the JaMoPP model. It is automatically synchronized to the code because Sirius calls the save operation of the JaMoPP model. The creation of a class is far more complicated: First, we have to create a file for the compilation unit. Within this

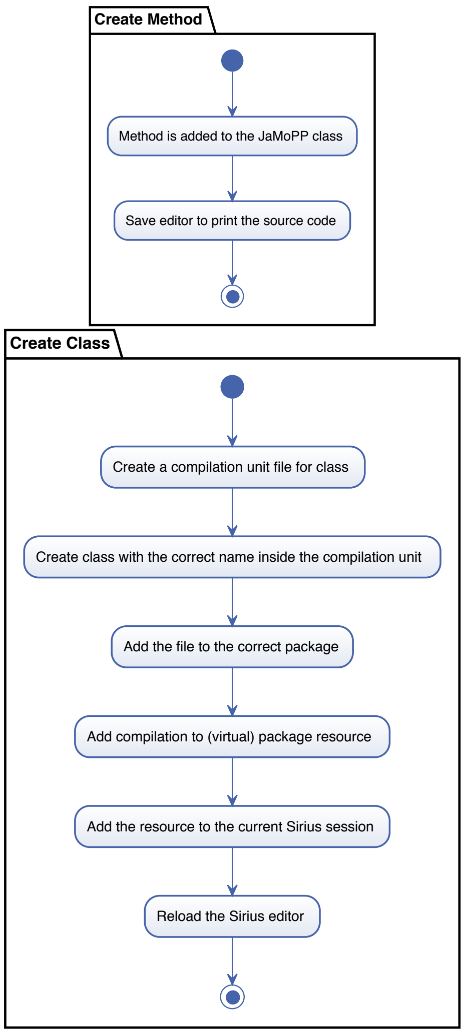

Create Method
Method is added to the JaMoPP class
Save editor to print the source code
Create Class
Create a compilation unit file for class
Create class with the correct name inside the compilation unit
Add the file to the correct package
Add compilation to (virtual) package resource
Add the resource to the current Sirius session
Reload the Sirius editor

Figure 3: Steps that are necessary to create a method in comparison to steps that are necessary to create a class. To create a class first we have to create a resource and a new compilation unit.

compilation unit, we save the contents of the new class as text. After that, the file has to be added to the correct package. To get the new compilation unit into the Sirius session, we have to add it to the current Sirius session and to the virtual package resource. As the last step, we reload the Sirius editor to show the newly created class within the editor. This approach is complicated since Sirius, like other editor frameworks for EMF models, are optimized for only one resource containing one root element. In our case, we have many resources (the Java source code files), with each of them containing one root element.

In the future, a support of existing UML tools is planned. Therefore, we need an importer and an exporter from and to the UML Diagram Interchange format. This can be done using model-to-model transformations between the UML class diagram metamodel and JaMoPP.

4.5 Limitations

The current realization of our UML class diagram editor proves the feasibility of the concept of a projective and editable class editor based on Java code. Nevertheless, the implementation is limited in the elements it shows and the modifications it allows. Currently, no stereotypes and generics are shown and associations are always unidirectional. The persistence of bidirectional associations requires an extension of the current annotation mechanism for associations.

The editability of elements inside the editor currently covers a set of elementary modifications, that are essential for the usability of the editor. Some features which we plan to support in the future are the modification of multiplicities, switching between the presentation of a field as an attribute or as an association, the modification of class names and the modification of attribute, parameter and return types.

The modification of values, such as types, multiplicities and names, that are presented in the diagram can usually be achieved with the capabilities of Sirius. An actual limitation of Sirius is the editability of attributes of a relationship between elements, such as the multiplicities of an association. Thus, it is currently not possible to realize the modification of multiplicities using Sirius. Realizing the modification of types with Sirius features would also be limited since Sirius only allows the selection of types that are available in the Sirius session. Although it is theoretically possible to add all potentially used types as resources to the Sirius session, this is impossible in practice. The set of potentially used types is very large, for instance, the whole Java API would have to be available. The memory consumption of JaMoPP would be too high due to the large models and the loading process would take too long for an acceptable usability. One option for the realization of these modification is the use of the Extensible Editing Framework (EEF) to realize a two-step import mechanism that does not need to load all classes prior to their usage.

Although our editor provides a limited set of displayed elements and possible modifications at the moment, it can be easily extended by all the missing features to provide a full UML class diagram. Merely the integration of advanced UML elements, such as annotation classes or qualifiers, would require major modifications. Indeed, even established tools with UML editors ignore these elements. Finally, it is just more difficult to implement such elements with our approach than the already existing features but not impossible.

5. DISCUSSION

In this section we discuss our approach in comparison to other approaches and give an outlook to what can be done with projective views on source code. Until now, we executed some tests of our prototypic editor on some example projects (e.g. the simple project that can be seen in Figure 1). Part of our future work is to provide a case study where we apply our approach to open source projects. Since we use JaMoPP to parse and print Java source code into an EMF model representation and Sirius for creating the view, our approach is embedded into the Eclipse IDE.

Compared to most other tools, our editor only depends on the source code. This makes it easy to integrate our UML class diagram editor into existing development tool chains. In fact, there is no integration effort needed to use our tool with other tools. This means that developers can stick with tools they use to change the source code. The UML class representation is updated automatically if any changes are made to the source code. Since we embed the associations in the source code and thus do not need any other artifacts, our approach can be easily used in collaborative software development processes. Simultaneous editing of model diagrams, however, is not supported. If another artifact stores the additional semantic UML information, it has to be shared as well. In the case of a conflict due to concurrent changes by different developers, both artifacts have to be merged ensuring their consistency. Using our approach, the risk of a faulty merge is minimized due to the fact that the UML information is stuck to the element it describes in the source code. The layout information of the diagrams is not saved in the source code, but in the Sirius session file that exists for each project. Thus, if diagrams and their layouts shall be shared between developers, this file has to be shared between them as well.

Compared to other tools, we do not need a consistency mechanism. Ensuring consistency is one of the most difficult parts of the approaches using an additional artifact to store the UML diagram information. Each tool shows a faulty behavior in at least one situation. Just to give some example, UML Lab does not handle the moving of a class into another package correctly; MagicDraw does not recognize a renaming of a field, which leads to a duplication of the field after a synchronization. Enterprise Architect displays a field as an attribute as well as an association without ensuring consistency between them. Renaming the association leads to an additional attribute with the old name, and both elements are synchronized to the code that afterwards contains two fields instead of one. Since our approach uses the source code as a SUM and UML diagrams are only projected from it, it does not need an explicit synchronization mechanism and it cannot exhibit a faulty behavior in these situations as long as the generic synchronization logic of Sirius works well.

The missing necessity of a consistency mechanism also brings an advantage if one of the involved metamodels changes. If the Java metamodel changes, e.g. if a new Java version is released, only JaMoPP has to be updated. If the changes are not affecting the UML class diagram, which was the case from Java 1.5 until the current Java Version 1.8, the class editor and the view still work. In this case, our approach does not differ much from other approaches. They also have to adapt their source code parser and printer or generator if the Java language changes.

Since we are not using a dedicated UML metamodel, our projective view still works if the UML metamodel changes. However, in this case, our UML class diagram does show the new features that were added in the UML metamodel changes. To get an adapted UML class diagram, the view definition has to be updated. After doing that, the view shows the new UML model and the source code is kept consistent automatically. Tools that use a consistency mechanism need to update the view and the consistency mechanism, e.g., the transformations that are used to keep the UML class diagram consistent with the code.

To exemplify this, we consider a change that could be introduced into the UML metamodel: A new *EventInterface* class is introduced into the UML metamodel, which should be displayed in UML class diagrams. In the source code, an *EventInterface* is represented as a normal interface. To figure out whether a normal interface or a new event interface should be generated for an event interface, there are two possibilities: Either an event interface has to provide a specific method, or we can annotate standard code interfaces with a new *EventInterface* annotation. To adapt our editor, we have to specify the representation of the new *EventInterface* in the editor. To enable the creation of an event interface within our editor, we have to add a new tool in the tool section and specify the meta class of JaMoPP (in this case *Interface*) that should be created. Tools that use a consistency mechanism would have to adapt at least the consistency mechanism as well as the view to reach the same functionality.

Our approach of having a projective view and using the source code as SUM could be extended to further UML views. For example, it is possible to create a UML package diagram view. A UML package diagram shows all packages of a software system and its dependencies. The necessary information to create such a model is already contained in the source code, and there is no need to extend the source code with additional annotations or comments. If one wants to use the approach of creating a projective view with the source code as SUM to create, for example, an activity diagram, the challenge arises that additional information is required that is not contained in the source code already. This information could be added by creating additional annotations or comments for the affected source code element, e.g., classes, methods or even statements within the method bodies. However, if too much additional information is included in the source code, it gets bloated and confusing for developers. To circumvent that, a new source code view could be created that shows the code without the annotations. However, this view is not easy to create since Sirius can not be used for the view creation. Hence, the approach of having projective views seems promising if only information that is contained in the source code must be displayed. To investigate this claim, the creation of additional views will be part of our future work.

6. CONCLUSION

In this paper we presented a projective UML class diagram view using the source code as SUM. The view is created using Sirius and JaMoPP. It gives an overview about the classes within a package and supports editing of the class diagram. Furthermore, we support associations and their multiplicities using Java annotations. Since we use JaMoPP, changes made in the view are automatically saved within the source code. Compared to existing tools we do not need a consistency mechanism to keep source code and the UML class diagram consistent during the software evolution.

In future work, we will extend our preprocessing to infer more association properties, such as *ordered* and *unique* for lists and set, as well as qualifiers for maps. Furthermore, technology-specific annotations, such as "OneToMany" of the Java Persistence API (JPA), could be supported. We will investigate whether it is possible to use our approach to create further projective views using the source code as SUM. We also plan to evaluate the maintainability of our approach in comparison to the maintainability of other tools.

References

[ASB10] C. Atkinson, D. Stoll, and P. Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *Evaluation of Novel Approaches to Software Engineering.* Vol. 69. Communications in Computer and Information Science. Springer, 2010, pp. 206–219.

[Bor05] Borland Software Corporation. *Borland Together UML 2.1 Guide Version 2008 R3*. 2005.

[Cla10] Claire Liu. *Round Trip Engineering Scenario using Rational Software Architect and ClearCase Remote Client*. 2010.

[Hei+09] F. Heidenreich et al. *Jamopp: The java model parser and printer.* Tech. rep. 2009.

[Hei+10] F. Heidenreich et al. "Closing the Gap between Modelling and Java". In: *Software Language Engineering.* Vol. 5969. LNCS. Springer Berlin Heidelberg, 2010, pp. 374–383.

[ISO11] ISO/IEC/IEEE 42010:2011(E). *Systems and software engineering – Architecture description*. International Organization for Standardization, Geneva, Switzerland, 2011, pp. 1–46.

[ISO12] ISO/IEC 19505-2:2012(E). *Information technology – Object Management Group Unified Modeling Language (OMG UML), Superstructure.* International Organization for Standardization, Geneva, Switzerland, 2012, pp. 1–758.

[ISO14] ISO/IEC 19508:2014(E). *Information technology – Object Management Group Meta Object Facility (MOF) Core.* International Organization for Standardization, Geneva, Switzerland, 2014.

[Lan+14] P. Langer et al. "On the Usage of UML: Initial Results of Analyzing Open UML Models." In: *Modellierung.* Vol. 19. 2014, p. 21.

[Nic+00] U. A. Nickel et al. "Roundtrip engineering with FUJABA". In: *Proceedings of the 2nd Workshop on Software-Reengineering (WSR), August.* 2000.

[No 12] No Magic, Inc. *MagicDraw Technical Overview*. 2012.

[Obj11] Object Management Group (OMG). *Unified Modeling Language (UML), Infrastructure Specification – Version 2.4.1*. 2011.

[PSV13] V. Pech, A. Shatalin, and M. Voelter. "JetBrains MPS As a Tool for Extending Java". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools.* PPPJ '13. ACM, 2013, pp. 165–168.

[PW92] D. E. Perry and A. L. Wolf. "Foundations for the Study of Software Architecture". In: *ACM SIGSOFT Software Engineering Notes* 17.4 (1992), pp. 40–52.

[Ram+03] A. Ramirez et al. *ArgoUML User Manual A tutorial and reference description.* 2003.

[Spa14] Sparx Systems. *Enterprise Architect User Guide*. 2014.

[VMP14] V. Viyovic, M. Maksimovic, and B. Perisic. "Sirius: A rapid development of DSM graphical editor". In: *Intelligent Engineering Systems (INES), 2014 18th International Conference on.* IEEE. 2014, pp. 233–238.

[Voe+13] M. Voelter et al. "mbeddr: instantiating a language workbench in the embedded software domain". In: *Automated Software Engineering* 20.3 (2013), pp. 339–390.

[Voe13] M. Voelter. "Language and IDE Modularization and Composition with MPS". In: *Generative and Transformational Techniques in Software Engineering IV.* Vol. 7680. LNCS. Springer Berlin Heidelberg, 2013, pp. 383–430.

Towards Metamodel Integration Using Reference Metamodels

Johannes Meier
Software Engineering Group
University of Oldenburg, Germany
meier@se.uni-oldenburg.de

Andreas Winter
Software Engineering Group
University of Oldenburg, Germany
winter@se.uni-oldenburg.de

ABSTRACT

The complexity of modern software engineering projects increases with growing numbers of artefacts, domain specific languages, and stakeholders with their concerns. To overcome these demands, different viewpoints are used to describe different languages specifying different artefacts, specific concerns of stakeholders, and domain specific languages. Therefore, the use of different viewpoints together in one software engineering project increases and requires technical support for automatic synchronization of the used viewpoints. This paper gives an overview about use cases for viewpoint synchronization and compares their fulfillment by existing approaches. As result, this vision paper proposes a new approach for synchronization of viewpoints to overcome the presented use cases with focus on reduction in synchronization and integration effort, on reuse of integration knowledge, and on viewpoint evolution.

1. MOTIVATION

Viewpoint-oriented software engineering is becoming more and more an essential paradigm in modern software engineering. It allows different viewpoints on the current system under development, and is motivated by an increasing number of different artefacts and languages, which are involved in modern software systems. Working with one artefact written in one language out of several means using one viewpoint out of several viewpoints pointing on the same information of the system.

As an *example* to describe and develop software, viewpoints for representing requirements in textual form, designing required static data in form of UML class diagrams, object-oriented implementation using Java, and for testing with JUnit testcases could be used. They all are working on information, which form together the domain. The term *domain* describes all relevant information of the current project. This example stems from the domain of object-oriented software development (OOSD) and is an ongoing one through the complete paper.

Different viewpoints support different concerns of different stakeholder with *tailored viewpoints*. Following the definitions of ISO 42010:2011 [8], a *viewpoint* determines selected concepts of the domain addressing specific concerns. A *view* contains a subset of the concrete information of the domain on instance level and is conform to one viewpoint.

An important property of viewpoints on technical level is the possibility for *overlapping viewpoints* [7]. This allows using same concepts in different viewpoints. As an example, the tester needs at least reading access to the source code, and to the requirements which are giving specifications what to test. This concept allows several stakeholders to work together at the same software project using different viewpoints. Overlapping viewpoints raise the problem of duplicated data which are changed through different viewpoints which is a source for inconsistencies, which leads to the need for *synchronization* of data for being consistent [6].

Working with several viewpoints requires also, that further information on content level "between" the viewpoints can be expressed. As an example, relations between the viewpoints for requirements and Java source code specify, which part of the source code fulfills which requirement. To support this kind of traceability between viewpoints on content level, new viewpoints could be defined for this purpose.

These aspects have to be realized on technical level, before user are able to work in viewpoint-oriented way. While working, the user needs support to synchronize viewpoints with other ones. Because of this different use cases and their actors, this position paper prepares several use cases in viewpoint-oriented projects as first step (Section 2). After comparison with existing approaches, which shows different fields for improvements, this position paper introduces a new approach for the synchronization of viewpoints for user, and for reduced integration effort for methodologists (Section 3). The approach will also support existing viewpoints and corresponding existing data, the evolution of that integration by the methodologists, and the reuse of integration effort in future projects by methodologists. This position paper ends with a conclusion and an outlook in Section 4.

2. USE CASES

The two most important stakeholder in viewpoint-oriented projects are the *user* who uses one or more viewpoints for working with different artefacts, and the *methodologist* [1, 15] who creates and manages the viewpoints and their relationships with each other. In the following, several use cases for user and methodologists are described and their fulfillment by existing approaches discussed.

2.1 Integrate existing Viewpoints

Initially, the methodologist integrates existing viewpoints somehow. Suitable existing approaches from literature are divided into *synthetic* and *projectional* approaches, following the ISO for Architecture Description 42010:2011 [8]. These two approaches are visualized in Fig. 1 using the ongoing example. Synthetic approaches keep the views of the different viewpoints unchanged and independent from each other, and require some kind of synchronization between all viewpoints for consistency (Fig. 1). Main characteristic is the *pair-wise synchronization* between the viewpoints to propagate changes from one viewpoint into all other viewpoints. To synchronize overlapping concepts like classes of viewpoints for Java and class diagrams, synchronizations directly between these two viewpoints are introduced.

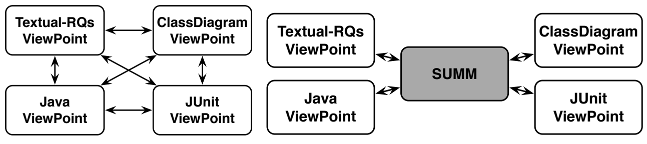


Figure 1: Synthetic vs. Projectional Approaches

The several *synthetic* approaches differ from each other by using different techniques for synchronizations: *Triple Graph Grammars* (TGGs) are used for specifying bidirectional relations between different graph languages [13]. In opposite to that, [12] uses *QVT-R* for synchronization.

Projectional approaches integrate the existing viewpoints into one singe *integrated metamodel* (later called SUMM) which contains the concepts off all viewpoints in an integrated manner, whereby synchronization is done only between viewpoint and SUMM. Each viewpoint works with a subset of the concepts of the SUMM, and propagates changes in the view to the central model. In Fig. 1, the central metamodel called SUMM contains all concepts of the viewpoints for textual requirements, UML class diagrams, Java source code, and JUnit test cases. Duplicated concepts like classes in the viewpoints for Java source code and UML class diagrams are only once in the SUMM which saves pair-wise synchronization effort. Instead, the viewpoints containing a subset of the concepts of the SUMM propagate changes in their views into the SUM, which forwards the changes to the other views which contain also these changed concepts. There are also several *projectional* approaches which differ from each other mostly in the underlying techniques.

Orthographic Software Modeling (OSM) [1] uses a so called Single Underlying Model (SUM) which contain all information about the current domain. Therefore, a metamodel for the SUM is needed, the Single Underlying MetaModel (SUMM). The SUM will be changed only through changes in views on it, respectively viewpoints on the SUMM. The views will be synchronized with the SUM by transformations. An environment for OSM is realized prototypically [2]. The issue, how to get a SUMM, is marked as future work [2].

The *Vitruvius* approach [10] follows the projective OSM ideas of a SUM with views on it, but implements this idea internally by a so-called modular SUM (MSUM). For that, the existing models are kept independent from each other like in synthetic approaches (without an explicit SUMM), and are combined with relations between each pair of viewpoints expressed through a new DSL called MIR, from which synchronization transformations are derived. In the end, on technical level, Vitruvius can be seen as synthetic approach.

2.2 Import and integrate existing Data

If in viewpoint-oriented projects data already exist which are conform to the integrated viewpoints, then these data have to be imported and reused. In particular, if data are coming from different overlapping viewpoints, not only the viewpoints have to be integrated, but also the instance data. Additionally, existing tools use models which conform to fixed metamodels of viewpoints. As result, the approach has to keep the existing viewpoints as first viewpoints to allow import and export for existing data and tools. This use case is motivated also in Vitruvius [10].

An advantage of synthetic approaches is, that existing viewpoints are directly usable, because only additional mechanisms for synchronization between the existing viewpoints will be created. Therefore, no additional effort is required to keep existing viewpoints and views usable for existing tools. Keeping existing viewpoints is significant easier in synthetic approaches, because projectional approaches take the existing viewpoints and integrate them into a SUMM. The presented projectional approaches in the modeling area do not give hints, how to deal with existing viewpoints.

2.3 Create new Viewpoints

In approaches for viewpoint-oriented engineering, methodologists have to create new viewpoints to support new external tools with a fixed viewpoint and new concerns of stakeholders. Important is, that new viewpoints have to use all aspects of the current domain [6]. In particular, concepts of several existing overlapping viewpoints together with their interrelations have to be reusable for new viewpoints. This allows easy creation of new viewpoints while reusing the elaborate work for integration.

Reusing concepts of different viewpoints is hard in synthetic approaches, wherefore Vitruvius introduces a declarative DSL called ModelJoin [3]. With ModelJoin, new viewpoints can be defined declaratively using concepts of several viewpoints, while the required metamodel of the new viewpoint and the synchronization transformations will be generated from the declarations. For the projectional approaches, [4] has similar ideas like the OSM approach and focuses on how to create new viewpoints on basis of a SUMM. For that, [4] developed an editor which helps to create new viewpoints as subsets of the SUMM. The needed synchronizations between new viewpoints and the SUMM will be generated, which use model differences for propagating changes between SUM and all views.

2.4 Synchronize Views

After integrating viewpoints and importing existing data, user read and write all data through initial and new viewpoints. Changed concepts in one viewpoint have to be synchronized into all viewpoints containing these concepts to keep overlapping viewpoints consistent.

General problem of synthetic approaches is the *square number of relations between the viewpoints*, which synchronize overlapping views of overlapping viewpoints to avoid inconsistencies. This results in heavily increasing initially creation effort. By contrast, in projectional approaches the number of required synchronizations is linear in the number of viewpoints. This is achieved by deleting duplicated concepts in the SUMM. At runtime, changes in a view are synchronized into the SUM, and from there forwarded into other views which also contain the changed elements.

2.5 Create a similar Integration again

Another problem is the initial effort for methodologists to integrate all viewpoints. While this initial effort is in general not avoidable, the reuse of previous done integration in future projects simplifies viewpoint integration. As an example, in a future project, requirements, Java code, and testcases should be used like before, but now Extended Entity Relationship (EER) diagrams should be used for describing static data. It would be nice to reuse the integration of requirements, source code, and testcases, and exchange class diagrams through EER diagrams easily. This results in the problem, how to reuse integration knowledge in future projects in the same domain. Ideas for *reusing integration knowledge* inside domains are not mentioned in the presented related work, which shows field for optimization.

2.6 Evolve Viewpoints

After finishing the integration of viewpoints, the initial viewpoints and their integration evolve because of, as examples, new versions of the tool specifying the viewpoint, or new version of Java like in the ongoing example. It is important, that evolution is possible, all unrelated metamodels remain the same, and that the co-evolution of existing models will be handled [6]. Neither the presented synthetic nor projectional approaches support currently this use case.

3. NEW APPROACH

This section proposes a new approach for viewpoint integration. Summarizing the related work for the different use cases in viewpoint-oriented projects, the integration and synchronization of projectional approaches has linear effort compared with square effort for synthetic approaches, while projectional approaches lack in handling existing data. Because this limitation is removable with manageable effort (compare Section 3.2), and the creation of new viewpoints is possible in both approaches but easier in projectional ones, the new approach of this position paper extends the projectional OSM approach. The following text show, how this new approach look like and will fulfill all the use cases.

3.1 Integrate existing Viewpoints

To create the SUMM which have to contain the concepts of all viewpoints exactly once, the metamodels of the viewpoints will be taken and integrated into one big metamodel, the SUMM. This integration will be done by the methodologist who has to define on semantic level, which concepts are duplicated in several viewpoints, and which additional relations between concepts have to be added. On technical level, several operators will by applied on the metamodels in a step-wise way. Besides typical operators like `add`, `change`, and `delete` [11], some more specialized refactorings like merging two classes representing the same concept into one single class are required. The selection of appropriate operators is part of future work. These operators allow the integration of the metamodels of the different viewpoints into one single underlying metamodel which is required by the OSM approach as input.

3.2 Import and integrate existing Data

An open issue in the existing OSM approach is, how to handle existing data which are conform to the integrated metamodels. This is important, because existing tools, for example, for modeling UML class diagrams, should be used together with the new approach. As an example, existing Java source code should be used further on. This means in both cases, that existing viewpoints and their data (here for UML class diagrams and for Java source code) have to be usable together within the new approach. On technical level, the concrete metamodels have to be kept as viewpoints on the final SUMM through the complete integration process.

This should be ensured by the in Section 3.1 proposed step-wise execution of operators on the metamodels with parallel creation of transformations for the model-co-evolution [5]. After integrating the viewpoints into the SUMM, the corresponding co-evolution-transformations allow the integration and import of the existing data into the SUM. If each step is combined with a complementary operator and a complementary co-evolution-step for the other direction, the data export from the SUMM into the viewpoints will be executable.

3.3 Create new Viewpoints

To create new viewpoints onto an existing SUMM, the existing projectional approach of [4] will be reused: After selecting a subset of the SUMM which forms the viewpoint, the required synchronizations between new viewpoints and the SUMM will be generated.

3.4 Synchronize Viewpoints

Extending the OSM approach allows linear effort for synchronization transformations. For future implementations, approaches for the propagation of model differences between views and the SUM like [15] which is used in the OSM approach or like [4] can be used also in this approach.

3.5 Create a similar Integration again

Looking at the ongoing example, concrete metamodels for Java code and class diagrams are integrated into a SUMM following the OSM approach. If a new project uses C++ instead of Java, the complete integration has to be performed again, while for integration purposes only general concepts like classes and methods are required. To allow such reuse of integration knowledge in the new approach, these general concepts will be moved into reference metamodels (RMMs).

Following [16], reference models are modeling the main and general characteristics of sets of systems of the same kind, and hiding specialized aspects of individual systems [9]. Reference models serve as reference point for specialized models [16] and support the construction of specialized models reusing the concepts of the reference model [14].

Because the specialized models are metamodels for Java and C++, a *reference metamodel* will be created. Java and C++ have both generalizable characteristics like classes and methods, and specialized aspects like different handling of pointers. The generalizable characteristics are part of the reference metamodel for object-oriented programming languages, while the specialized aspects remain in the concrete metamodels for Java and C++ (Fig. 2). The RMM for

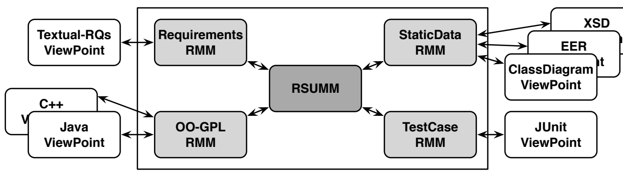


Figure 2: Concepts of the new approach

the subdomain of object-oriented programming languages would contain at least classes containing methods. Concrete metamodels for Java and C++ will be mapped onto their corresponding RMM, whereby as example the details of the Java and C++ metamodels are ignored on RMM level. The RMMs of all subdomains will be integrated into the RSUMM. The RSUMM contains the integration on conceptual level, like here the integration of classes and methods from object-oriented programming languages, with classes and attributes of data description languages.

In concrete projects, for each subdomain one concrete metamodel will be selected, like Java as programming language and Extended Entity Relationship (EER) diagrams as data description language. Based on the RSUMM, the RMM parts will be replaced by concrete metamodels to get the SUMM, whereby the integration knowledge will be derived from the integration done in the RSUMM. The derived SUMM is the same like in the OSM approach.

This approach allows to integrate the subdomains as representatives for the main concepts only once in form of the RSUMM for each domain, instead of each time in form of SUMMs for each new project. The integration in form of the RSUMM can be reused for each project by "instantiating" each subdomain by the currently needed concrete metamodel to get a SUMM. This allows arbitrary combinations of concrete metamodels and their viewpoints.

Another advantage of the new approach is, that for new concrete viewpoints like XML Schema Definition (XSD), its mapping to the StaticDataRMM is enough, and no complex integration with other subdomains has to be performed, because the integration is done before using the RMM. After creating the mapping, XSD can be used directly.

The technique for mappings between concrete metamodels and their reference metamodels is part of future work, which could apply step-wise metamodel changes (Sec. 3.1).

3.6 Evolve Viewpoints

After finishing the metamodel integration by the methodologist and while user are working with the integrated viewpoints, evolution can occur to the integrated viewpoints: The evolution of new viewpoints (CMMs) basing on the SUMM can be expressed again as metamodel changes together with the creation of model-co-evolution transformations to handle the instance level. The evolution of the CMMs can be simplified in this approach by distinguishing changes into changes which affect only unimportant concepts which are not part of the RMM, and into important changes which affect both the CMM and the RMM. While the former case can be realized as refactoring of the CMM, the latter case requires also changes in the RMM and therefore also in the RSUMM, which will be complex and requires further investigations. But after handling the evolution task in the RSUMM, all derived SUMMs benefit and apply them.

4. CONCLUSION

For technical support of viewpoint-oriented software engineering, the viewpoints have to be integrated by the methodologist to offer the user consistent information across several viewpoints. Therefore, this position paper presented different use cases in viewpoint-oriented projects, and compared their fulfillment by several synthetic and projectional approaches. As result, synthetic approaches have the problem of square synchronization effort, projectional approaches lack in supporting existing viewpoints, and all approaches do not support the evolution of viewpoints and the reuse of integration results in future projects.

To overcome these limitations, this paper proposes a new approach following the OSM approach to have only linear synchronization effort. The new approach extends the OSM approach by support for existing viewpoints, which allows using existing data and reusing existing tools. This is reached by rigorously creating model-co-evolution mechanisms for all metamodel changing steps, which are needed to map concrete metamodels to their reference metamodels and to integrate reference metamodels into the RSUMM.

Main contribution and benefit of the new approach compared to existing approaches is the reuse of integration effort for future projects within the same domain, which is not in the focus of other approaches. This is reached by shifting the integration of concrete metamodels to the integration of concepts expressed in reference metamodels (RMM) containing the main concepts of subdomains. This allows the methodologist to select one of several possible metamodels like for Java or C++ for the current project. As result, the complex integration will be done once on reference level, and will be reused while deriving SUMMs for current projects.

The proposed ideas of this position paper will be concretised and implemented in a framework for viewpoint-oriented software engineering. As domain for validation, the ongoing example of this paper of object-oriented software development will be used, which could be extended by further subdomains like project management, or documentation.

5. REFERENCES

[1] C. Atkinson, D. Stoll, and P. Bostan. Supporting View-Based Development through Orthographic Software Modeling. *Enase*, pages 71–86, 2009.

[2] C. Atkinson, D. Stoll, C. Tunjic, and J. Robin. A Prototype Implementation of an Orthographic Software Modeling Environment. VAO 2013.

[3] E. Burger, J. Henss, M. Küster, S. Kruse, and L. Happe. View-based model-driven software development with ModelJoin. *Software & Systems Modeling*, 2014.

[4] A. Cicchetti, F. Ciccozzi, and T. Leveque. A hybrid approach for multi-view modeling. *Recent Advances in Multi-paradigm Modeling*, 50, 2011.

[5] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. *12th EDOC'08*, 2008.

[6] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *FOSE 2007*.

[7] T. Goldschmidt, S. Becker, and E. Burger. Towards a Tool-Oriented Taxonomy of View-Based Modelling. 2012.

[8] IEEE. ISO/IEC/IEEE 42010:2011 - Systems and software engineering - Architecture description. 1–46, 2011.

[9] H. Krallmann and B. Scholz-Reiter. Cim-KSA - Eine Rechnergestützte Methode für die Planung von Cim-Informations- und Kommunikationssystemen. *Informatik-Fachberichte*, 258:57–66, 1990.

[10] M. Kramer, E. Burger, M. Langhammer. View-centric engineering with synchronized heterogeneous models. *VAO'13*.

[11] D. Kuryazov and A. Winter. Representing Model Differences by Delta Operations. *EDOCW 2014*.

[12] J. R. Romero, J. I. Jaén, and A. Vallecillo. Realizing correspondences in multi-viewpoint specifications. *EDOC 2009*.

[13] A. Schürr and F. Klar. 15 Years of triple graph grammars: Research challenges, new contributions, open problems. *Lecture Notes in Computer Science*, 5214:411–425, 2008.

[14] O. Thomas. Understanding the Term Reference Model in Information Systems Research: History, Literature Analysis and Explanation. *LNCS*, 3812 (Chapter 45):484–496, 2006.

[15] C. Tunjic and C. Atkinson. Synchronization of Projective Views on a Single-Underlying-Model. *VAO 2015*.

[16] A. Winter. *Referenz-Metaschema für visuelle Modellierungssprachen.* Deutscher Universitäts-Verlag, 2000.