Flexible Views for Rapid Model-Driven Development

Erik Burger
Karlsruhe Institute of Technology, Karlsruhe, Germany
erik.burger@kit.edu

ABSTRACT

Model-driven development processes that use several metamodels suffer from information spread across heterogeneous instances. Models can be managed with specialized partial views that aggregate the information from heterogeneous instances and offer the approprate kind of abstraction for developers. The creation and maintenance of such views requires, however, high manual effort. In this paper, we present the usage of *flexible views* in the VITRUVIUS approach for view-centric engineering. Flexible views serve as custom, user-specific views which can be defined at development time using a textual domain-specific language. Developers can rapidly create specialized views instead of manually creating the appropriate metamodels and transformations, which is a time-consuming and error-prone task. The envisioned benefit of this approach is to improve software quality, to increase consistency between the various modeling artefacts in model-driven software development, and to reduce the complexity for software developers.

Categories and Subject Descriptors

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

Keywords

Model-Driven Software Development, view-based modeling

1. INTRODUCTION

Software projects that make use of metamodelling techniques and models face the problem that information about the entities of interest is spread across instances of different metamodels. We will call these instances *heterogeneous models* in the following. Although these artefacts describe the same system, several metamodels are necessary to represent various levels of abstraction, to describe different view points on the system, or to guarantee compatibility to legacy software, which may require the usage of specific metamodels. In extensive systems, these models can become numerous, large and too complex to be understood by a single developer. Thus, inconsistencies and redundancies can occur, leading to drift and erosion between the models and the implementation of the system. To lower the complexity for developers, view-based approaches offer partial views on the system which restrict the amount of information that is presented to the developer and fulfill the specific information need of the developer. Pre-defined views are, however, often limited to the information contained in one specific metamodel or offered by a certain development tool, and cannot be defined by the developers themselves according to their information need.

In this paper, we present the usage of *flexible views* [6] for rapid Model-Driven Development in the context of the novel view-based modeling approach VITRUVIUS, which is based on *Orthographic Software Modeling (OSM)* by Atkinson et al. [2]. VITRUVIUS enables the software developer to focus on the parts of the system which are relevant for his or her current role by offering view types that aggregate information from heterogenous models. In addition to pre-defined view types, developers can use a lightweight domain-specific language (DSL) to define flexible views, which are custom and user-specific, as suggested in the original OSM concept. A protoype for the creation of flexible views has been developed using the popular Eclipse framework and the model-driven standards Xtext, Xtend2 and QVT-O.

The goal of VITRUVIUS is to give developers permanent access to consistent, up-to-date and complete information about the system under development, tailored to the information needs of different developer roles, e.g., domain experts, system architects, or software developers. Flexible views can be customized to the needs of a single developer and can be defined by the developers in textual form, speeding up the development process since manual efforts like the creation of metamodels and transformations are avoided. Flexible views also reduce the complexity of development by displaying only the type of information which is relevant at the time of modeling.

The rest of this paper is structured as follows: In section 2, we will shortly present view-based MDD and the OSM-based VITRUVIUS approach. They serve as the foundation for the application of flexible views, which we will describe in section 3. Our prototypical implementation will be presented in section 4, together with plans for validation in the context of VITRUVIUS. The paper concludes with related work section 5 and an outlook on future research directions section 6.

VAO '13, July 2, 2013, Montpellier, France

http://dx.doi.org/10.1145/2489861.2489863 ...$15.00.

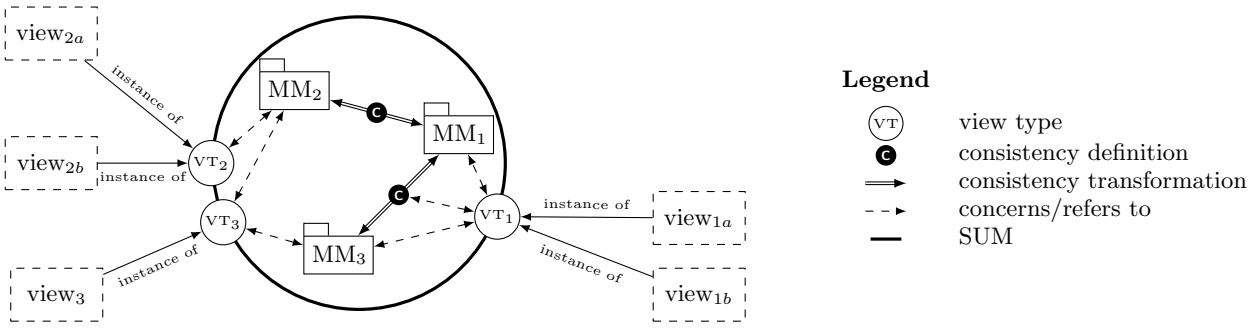

Figure 1: Architecture of Vitruvius using a modular single underlying model (SUM)

2. FOUNDATIONS

Model-driven development (MDD) [23, 21] puts models into the centre of the development process and provides a clear separation of views and models. A key benefit of the MDD approach is the possibility to define domain-specific languages and generate appropriate tooling easily. Model transformations and generators can be used to translate instances between these modeling languages and create textual representations. Developers can use several modeling languages in the different phases of a software development process, like models for requirements engineering, architecture and object-oriented design, specialized models depending on the domain for which the software is being developed, and code (which can also be seen as a model) for the implementation and runtime behaviour of system. During the MDD process, several models can be used, which may differ in the level of abstraction and the type of modeled information, but should represent the same system. Developers have to take manual efforts, however, to aggregate information information across these models. This is why developers usually work in only one kind of model at a time, and often neglect inconsistency problems, since it is difficult to gain information that is spread across several models from different languages and to keep information consistent across them. In *view-centric modeling* approaches, all access to these models is organised in views. We will use the definition of the terms *view, view type* and *view point* from previous work [14, 6], which is slightly different from the the IEEE 1471/ISO 42010 standard [15, 16].

Definition 1. *A* view type *defines the set of metaclasses whose instances a view can display. It comprises a definition of a concrete syntax plus a mapping to the abstract metamodel syntax. The actual* view *is an instance of a view type showing an actual set of objects and their relations using a certain representation. A* view point *defines a concern.*

The views in view-centric tools are partial, i.e., do not show the complete model, which reduces the complexity for modelers; the view types are, however, usually pre-defined and can not be varied by the developers themselves.

The *Orthographic Software Modeling (OSM)* [2] approach realizes a view-centric software development process. In OSM, all information about a software system is represented in a single underlying model (SUM); even source code is treated as only a special textual view on this model. User-specific custom views, which are organized in independent (orthogonal) dimensions, are generated dynamically based on model transformations from and to the SUM. This concept requires however that bi-directional transformations exist for every view type (i.e., metamodel); they provide the synchronization of views with the SUM, and edit operations are propagated back to the SUM likewise. OSM also encompasses a development process with a *developer* role, who uses the generated views, and a role called *methodologist*, who creates the different view types along the orthogonal dimensions. The SUM concept is theoretically elegant and solves many problems of view-based approaches that synchronise views directly [20, 11], such as the exponential increase of bi-directional synchronisation rules. The OSM approach does however neither provide a universal metamodel for the SUM nor a construction principle for such a metamodel.

The novel Vitruvius approach is based on the concept of Orthographic Software Modeling, but, in contrast to the monolithic approach in the existing implementation KobrA [3], is built on top of a *modular SUM*, which contains several (legacy) metamodels, but can be accessed as a single entity (See Figure 1). *View types* act as interfaces and are the only way of accessing and manipulating information. The contained metamodels themselves do not have to be modified to work in the modular SUM, which makes the approach non-invasive and backwards-compatible. The sub-models of the SUM are synchronised via bi-directional consistency transformations, which are coupled with the view types.

3. RAPID MODEL-DRIVEN DEVELOPMENT USING FLEXIBLE VIEWS

View-based modeling approaches give developers a standardised way of accessing information of a system by structuring the access through the view types. It is, however, difficult to create views that span multiple metamodels; in the definition of [14], a view type is even adjacent to only a single metamodel. In the Vitruvius approach, *flexible views* [6] are used for the the rapid construction of custom, user-specific views, which aggregate information from heterogeneous models.

Definition 2 (Flexible View). *A* flexible view *on a set of models contains:*

- *the definition of a* view type*, i.e., the type of model elements which can be contained in the flexible view. These types need not be the same than those of the source models.*
- *the definition of a* view*, i.e., the selection of elements*

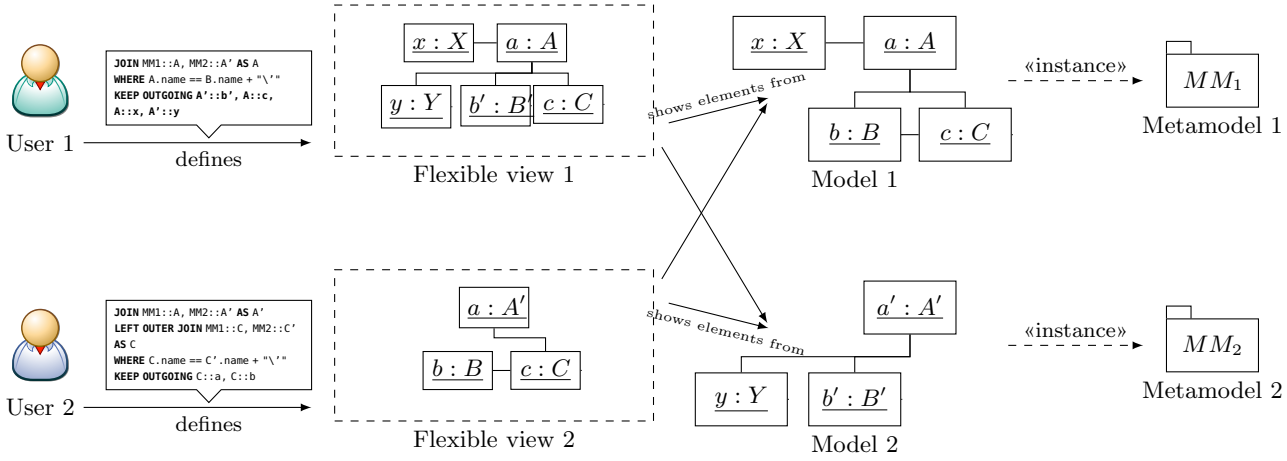

Figure 2: Flexible views usage example (view types not displayed)

that is contained in the flexible view. This selection is based on instance properties.

- *a set of rules for editability of the view and for the back propagation to the source models.*

The concept of flexible views is depicted in Figure 2: In the hypothetical example, two metamodels MM_1 and MM_2 and their respective instances *Model 1* and *Model 2* share a semantic overlap, i.e., different elements of MM_1 and MM_2 represent the same entitity type in the system of interest. In the example, this is indicated by similar naming of the elements on metamodel level (A corresponds to A', etc.) and on model level (a corresponds to a', etc.). Each of the models carry additional information which is not available in the respective other model, i.e., the elements x, y and the references to these elements. *User 1* wants to create a view that aggregates all information from *Model 1* and *Model 2* while at the same displaying overlapping elements as one element. The resulting *Flexible view 1* integrates information from both models and identifies the overlapping elements by a naming convention. *User 2* creates a view which shows only the overlapping elements, but always the elements of type C. The views in this example are read-only.

The rules for the selection of element types and elements are defined by a textual query expression here in the callout blocks on top of the "defines" arrow, which uses a textual syntax similar to SQL. Thus, the user has to identify the semantic overlap and write a query accordingly. The rules for view creation do however not include concepts for an automatic matching of elements, so there is no heuristic or matching algorithm that identifies similarities in the metamodels. The user is expected to understand the semantics of the metamodels on which the flexible view operates. The textual expression defines the classes and features which are part of the view (i.e., the *view type*), similar to a database query defining the columns of a query result; it also defines the elements in the actual view (i.e., the *view*), similar to the rows of a database query result.

In this example, flexible views offer developers the possibility to create new view types and views rapidly having to define neither an intermediate metamodel on which the view type is based, nor the transformation rules which are neccessary for the mapping of the source models and the view. These artefacts are generated automatically by a framework implementing the Vitruvius approach. Flexible views have a transient nature and can change rapidly as the developer modifies the textual definitions and experiments with different variants. If the textual definition of a flexible view evolves, the resulting metamodel and transformations co-evolve automatically, since they are generated from the textual definition. Therefore, users can develop different versions of flexible views without having to keep metamodel and transformation consistent manually.

Once a proper abstraction is found, a flexible view can be persisted and can then be re-used in the same way as other, pre-defined non-flexible views. The artefacts generated from the flexible view definition (metamodels, traces, transformations) can also serve as a starting point for new non-flexible view types, which has the advantage that the metamodels and transformations do not have to be written from scratch. If more sophisticiated view types are required, developers can adapt the target metamodel and the transformations manually, and can generate and customize editors for the target metamodel.

4. IMPLEMENTATION PROTOTYPE AND PLANS FOR VALIDATION

As a prototype for the definition of user-specific views at runtime, we have created a DSL with a textual syntax similar to SQL, which is not described here due to limited space. We have implemented an Eclipse-based engine that uses metamodel generation and transformation generation based on Xtext, Xtend2, and QVT-O to create the view types and views from textual queries. The architecture of our prototpye is depicted in Figure 3. This enables the developer to create and modify custom views without having to design metamodels and transformations manually. Currently, the DSL and its prototypical implementation only supports read-only views.

We plan to validate our approach by applying it to the component-based development process of the Palladio metamodel [5]. We plan to combine the component-based modeling formalisms of PCM with a description of the class structure in UML and a code representation in Java. A larger example

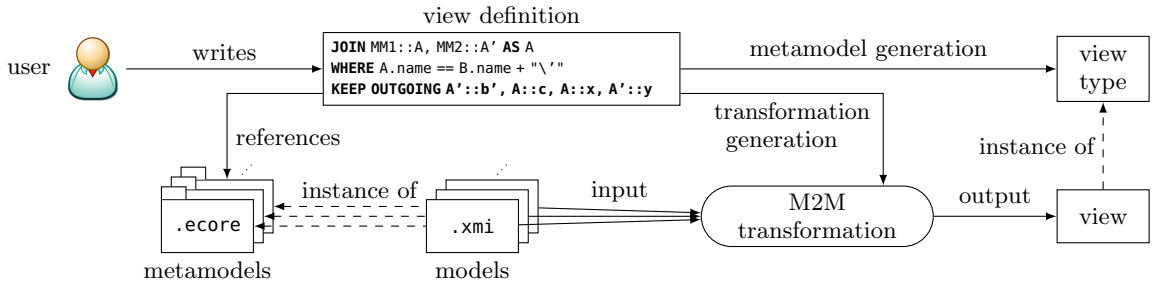

Figure 3: Implementation prototype architecture

system, e.g., the MediaStore example from [5] or the common component modeling example (CoCoME) [3] will be used since the software architecture description and an implementation already exist.

Based on these systems and a SUM-metamodel that combines PCM, UML and Java, we will test the migration of legacy systems to the orthographic Vitruvius approach. With the resulting model, a comparison of a SUM approach and a non-SUM-approach can be conducted. We plan to evaluate different evolution scenarios for software in a qualitative study that involves computer science students with education in model-driven engineering. The goals of this study are to validate the following theses: Firstly, the feasibility of the Vitruvius approach in general; secondly, that flexible views reduce the complexity for the developer; thirdly, that the approach improves the quality of software, i.e., improves the consistency and reduces drift and erosion.

5. RELATED WORK

In the context of the Eclipse Modeling Framework (EMF), several projects deal with the merging of heterogeneous models. They usually require the a priori definition of a fixed target metamodel [17, 8] or a connecting "glue" model [12, 1]. For this reason, they cannot be used for the rapid creation of user-specific views. Aspect-Oriented Modeling (AOM) approaches [7, 24] also structure large systems along specific concerns, but require an explicit weaving process, which is unnecessary in our view-based approach.

Many of the problems that are encountered in view-based modeling have counterparts in database research. Relational databases also offer the possibility to create (editable) views. A relational view defines a schema of its own, just like a view in MDD has a metamodel (i.e., the view type). The query mechanism in relational databases serves the function of a model-to-model transformation in MDD. If data in a partial view is manipulated, the *view update problem* [4, 9] arises. This central issue in relational databases is well understood, but mainly unsolved [18]. The process of reintegrating changes on a partial view into the underlying database is called *translation*; it has been shown that such a translation does not always exist for any kind of view update, and that it is undecidable if a unique translation exists. The problem can be alleviated by carefully designing the views, so that every edit operation of a user can also be reverted in the same view without losing information in the underlying database. We will chose the same approach for the definition of views in Vitruvius; by defining view types that limit the possible operations, we can guarantee that consistency constraints in the underlying SUM are not violated. In recent database research, the view-update problem has also been investigated for tree-like structures [13], which can be applied to model transformations using graph structures.

Integrating heterogeneous metamodels and instances bears similarities to the well-known problem of schema integration of heterogeneous databases [19, 22]: A semantic understanding of both domains is necessary to define the mapping of elements; hence, it cannot be fully automated. Furthermore, a global database schema is used to express data from various sources. This is also true for model-driven development, so we refrain from defining automatisms for the identification of similarities in heterogeneous metamodels automatically, e.g., by graph-based comparison using heuristics as in [1, 10]. We suggest that the methodologist must understand the semantics of the metamodels, which act as the modular parts of the SUM, between which he or she will define the consistency constraints and evolution policies.

6. CONCLUSION AND NEXT STEPS

We have presented a view-based development process based on the Vitruvius approach and *flexible views*. We motivate this approach by a model-driven development process in which information is spread across several metamodels and implementation code, which together form a modular single underlying model of the software system. Flexible views structure the access to the information in the system and reduce complexity for the developer by displaying only relevant information, while at the same time giving the developer the possibility to rapidly design new view and view types with the help of a textual DSL. The metamodels and transformations behind the view type do not have to be creted manually. The benefits of the proposed approach are increased consistency of systems in model-driven software development with reduced complexity for the single developer, leading to higher software quality.

As next steps, we plan to extend our textual DSL for the definition of editable flexible views, since it currently only supports read-only views. For this goal, we will create a catalogue of operations that can be used to create flexible views and categorize them with respect to invertability and bi-directionality. The synchronisation of views and the SUM can then be defined with the use of bi-directional model transformations, in combination with the consistency constraints between the sub-models of the SUM. Furthermore, editability restrictions can serve as an access control mechanism to support a software development process with structured permissions for the different developer roles.

References

[1] *Atlas Model Weaver*. URL: http://www.eclipse.org/gmt/amw/.

[2] Colin Atkinson, Dietmar Stoll and Philipp Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek, César González-Pérez and Stefan Jablonski. Vol. 69. Communications in Computer and Information Science. Berlin/Heidelberg: Springer, 2010, pp. 206–219.

[3] Colin Atkinson et al. "Modeling Components and Component-Based Systems in KobrA". In: *The Common Component Modeling Example*. Ed. by Andreas Rausch et al. Vol. 5153. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, 2008, pp. 54–84.

[4] François Bancilhon and Nicolas Spyratos. "Update semantics of relational views". In: *ACM Trans. Database Syst.* 6.4 (Dec. 1981), pp. 557–575.

[5] Steffen Becker, Heiko Koziolek and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82 (2009), pp. 3–22.

[6] Erik Burger. "Flexible Views for View-Based Model-Driven Development". In: *Proceedings of the 18th international doctoral symposium on Components and architecture*. WCOP '13. Vancouver, British Columbia, Canada: ACM, 2013, pp. 25–30.

[7] Ruzanna Chitchyan et al. *Survey of Analysis and Design Approaches*. Tech. rep. AOSD-Europe, May 2005.

[8] Cauê Clasen, Frédéric Jouault and Jordi Cabot. "VirtualEMF: A Model Virtualization Tool". In: *Advances in Conceptual Modeling. Recent Developments and New Directions*. Ed. by Olga De Troyer et al. Vol. 6999. LNCS. Springer Berlin / Heidelberg, 2011, pp. 332–335.

[9] Edgar Frank Codd. *The relational model for database management: version 2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[10] *EMF Compare*. Dec. 2012. URL: http://wiki.eclipse.org/EMF_Compare/FAQ.

[11] Romina Eramo et al. "Change Management in Multi-Viewpoint System Using ASP". In: *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*. 2008, pp. 433–440.

[12] Franck Fleurey et al. "A Generic Approach for Automatic Model Composition". In: *Models in Software Engineering*. Ed. by Holger Giese. Vol. 5002. LNCS. Springer Berlin / Heidelberg, 2008, pp. 7–15.

[13] J. Nathan Foster et al. "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem". In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 233–246.

[14] Thomas Goldschmidt, Steffen Becker and Erik Burger. "View-Based Modelling – A Tool-Oriented Analysis". In: *Modellierung 2012*. Vol. P-201. GI-Edition – Lecture Notes in Informatics (LNI). Elmar J. Sinz, Andy Schürr, 2012.

[15] "ISO/IEC Standard for Systems and Software Engineering – Recommended Practice for Architectural Description of Software-Intensive Systems". In: *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15* (July 2007), pp. c1–24.

[16] *ISO/IEC/IEEE Std 42010:2011 – Systems and software engineering – Architecture description*. Los Alamitos,CA: IEEE, 2011.

[17] Dimitrios Kolovos, Richard Paige and Fiona Polack. "Merging Models with the Epsilon Merging Language (EML)". In: *Model Driven Engineering Languages and Systems*. Ed. by Oscar Nierstrasz et al. Vol. 4199. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 215–229.

[18] Jens Lechtenbörger. "The impact of the constant complement approach towards view updating". In: *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '03. New York, NY, USA: ACM, 2003, pp. 49–55.

[19] M.P. Reddy et al. "A methodology for integration of heterogeneous databases". In: *IEEE Transactions on Knowledge and Data Engineering* 6.6 (Dec. 1994), pp. 920–933.

[20] José Raúl Romero, Juan Ignacio Jaén and Antonio Vallecillo. "Realizing Correspondences in Multi-viewpoint Specifications". In: *Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*. 2009, pp. 163–172.

[21] D.C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2 (Feb. 2006), pp. 25–31.

[22] Amit P. Sheth and James A. Larson. "Federated database systems for managing distributed, heterogeneous, and autonomous databases". In: *ACM Comput. Surv.* 22 (3 Sept. 1990), pp. 183–236.

[23] Thomas Stahl et al. *Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management*. 1. Aufl. Heidelberg: dpunkt-Verl., 2005.

[24] Manuel Wimmer et al. "A survey on UML-based aspect-oriented design modeling". In: *ACM Comput. Surv.* 43 (4 Oct. 2011), 28:1–28:33.