Flexible Views for View-Based Model-Driven Development

Erik Burger
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
burger@kit.edu

ABSTRACT

Model-driven development processes suffer from growing complexity, which leads to information spread across heterogeneous metamodels as well as drift and erosion between architecture and implementation. In this paper, we present a view-based modeling approach based on Orthographic Software Modeling (OSM), and introduce *flexible views* as a concept for the creation of custom, user-specific views. The envisioned benefit of the approach is to improve software quality, to increase consistency between the various modeling artifacts 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

General Terms

Keywords

Model-driven software development; View-based modeling; Orthographic Software Modeling

1. INTRODUCTION

Most of today's software delopment processes make use of models and model-based technologies to cope with the complexity of larger systems by expressing features of the system on a higher level of abstraction. With the extensive use of models in complex systems, these models can themselves grow large and become too complex to be understood by a single developer. These models are often instances of several heterogeneous metamodels, across which information is spread. Thus, inconsistencies can occur, leading to drift and erosion between the models and the implementation of the system. To lower the complexity for the developer, partial views on the system restrict the amount of information that is presented to the developer and structure the way of displaying and manipulating information. Pre-defined views are however often limited to the information contained in one specific metamodel, and cannot be defined by the developers themselves.

In this paper, model-driven development techniques are used to define a view-based modeling approach based on the Orthographic Software Modeling concept by Atkinson et al. [2]. Dynamically created, so called *flexible views* are introduced to help the software developer to focus on the parts of the system which are relevant for his or her current role, and offer an abstraction for the rest of the system. Flexible views can be defined by a lightweight domain-specific language (DSL), and are part of a novel construction method for the single underlying model (SUM), a central concept on which the OSM approach is built.

The goal of our approach is to improve software quality by giving 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 are customized to the needs of a single developer and can be defined by the developers themselves. They reduce the complexity by displaying only the type of information which is relevant at the time of modeling. Flexible views may also be used to hide information and thus to implement access control to parts of a software system.

2. FOUNDATIONS

2.1 Model-Driven Development

Model-driven development (MDD) [28, 26] puts models into the centre of the development process and provides a clear separation of views and models. 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 essentially represent the same system. Developers have to take manual efforts to achieve synchronisation of information across these models. This is why developers usually work in only one kind of model at a time.

In *view-based modeling* [16], all access to the models is organised in views, which in general are partial, i.e., do not show the complete model, thus reducing the complexity for the developer by presenting him or her only the relevant parts of a system.

The terms *view* and *view point* have been defined for software architectures and architectural description languages in the IEEE 1471/ISO 42010 standard [18, 19], which only give a broad definition of the terms and do not distinguish between view types and actual view instances. We will use the more precise definition of Goldschmidt et al. [15, 16], given in Definition 1, instead.

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.*

In UML, for example, diagram types such as sequence or class diagrams would be view types. A view is the actual diagram containing classes A, B and C. The static architecture or dynamic aspects of a system would be view points in UML.

The main difference of this definition to the IEEE/ISO standard is the introduction of the term *view type*, which can be interpreted as a metamodel for actual views. Goldschmidt also defines defines different notions for completeness and partiality of view types and views, such as containment-completeness, selectional completeness, and a definition for overlaps of view types [15, sect. 4.4].

A developer will rarely use a complete view on a system because it is often too complex and tends to be confusing. This is why today's software modeling tools offer possibilities to restrict a view to certain elements; the possibilities for creating custom views are however limited to the selection mechanisms implemented in the tools, so that often only a manual selection of elements is possible. The idea of restricted views for developers goes back to the 1990s [13, 20, 24], but has not been accepted as a development paradigm yet. We are conviced that today's model-driven techniques like model transformations and the support for textual syntaxes offers greater possibilities for view-based approaches.

2.2 Orthographic Software Modeling

With the *Orthographic Software Modeling (OSM)* [2] approach, Atkinson et al. aim to establish views as first-class entities of the software engineering process. In the envisioned view-centric development process, all information about a system is represented in a single underlying model (SUM); thus, the SUM even transcends the function of being a model, but becomes the system itself. This makes the approach radical in the sense that even source code is treated as only a special textual view. The SUM itself has to contain all execution semantics and could theoretically also be executed without the use of source code.

The authors of OSM suggest that user-specific custom views be generated dynamically, based on model transformations from and to the SUM. These views are organized in dimensions that are ideally independent from each other (orthogonal). Technically, a view is a model of its own which also has a metamodel (view and view type in terms of Definition 1). Model-to-model transformations create the views dynamically from the SUM. This also – theoretically – solves the issue of keeping views consistent, since every edit operation in a view is immediately represented in the SUM and thus available to all other views. 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. Although writing bi-directional transformations is a difficult task, the complexity of a hub-and-spoke architecture like OSM is linear in terms of the number of transformations that have to be written and maintained, in contrast to the exponential number of transformations in a peer-to-peer synchronisation scenario for views, which must also be bi-directional (see Figure 1).

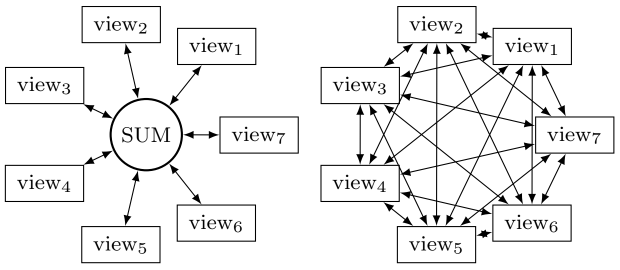


Figure 1: Hub-and-spoke vs. peer-to-peer

The KobrA Method [3] is a prototypical implementation of the OSM approach, based on UML and OCL. KobrA is however lacking a method for the construction of the single underlying model (and the metamodel it is based on), which is supposed to carry all information of the software development lifecycle.

2.3 Bidirectional Transformations

Depending on the transformation language used, bidirectionality is difficult to maintain or to prove. Algebraic frameworks and methods like the lenses approach by the Harmony group [14] offer a sound theoretical foundation, but are not yet mature enough to be used in practice. Triple graph grammars [21] can be used to define bidirectional transformation rules, which are in practice often transformed into pairs of unidirectional transformations.

Perdita Stevens distinguishes bidirectional transformations from bijective transformations [29], and identifies several requirements that bidirectional transformations should satisfy: In the context of QVT-R, the basic requirements are *correctness* (the transformation engine produces models that satisfy the relations), *hippocraticness* (models that satisfy a relation are not modified by the executions of the transformation engine), and additionally *undoability.* Diskin et al. [11] give a weaker definition for *undoability* and *invertibility* that is more suitable for practical applications of bidirectional transformation and which is based on lenses. They also argue that *delta-based* approaches are superior to *state-based* approaches: While state-based transformations have the complete models in their current status as input and output, delta-based approaches use the differences between models. These deltas may carry semantic information that cannot be computed by comparing model versions, such as refactorings like renaming and changes in containment. This is especially important in scenarios where UUIDs/primary keys are not always available, as it is the case in EMF, where they are only optional for model elements.

Bidirectional transformations can be classified into *symmetric* and *assymetric* cases: In the symmetric case, both source and target model of a transformation can contain new information that is not yet present in the respective other model and has to be added there by the transformation. In the asymmetric case, one model does not contain new information, which is the case for (non-editable) views. In the context of view-based modeling, the views in general may be editable, and thus, the framework must support changes in both the underlying model and in the view. For read-only views, it is sufficient to define only a unidirectional transformation, thus reducing the complexity.

3. EXAMPLE

To demonstrate our idea of view-based development, we have chosen a component-based development process based on the Palladio Component Model (PCM) [6], UML class diagrams, and Java code, as displayed in Figure 2. Information about the same software system is represented on different levels of abstraction in these three formalisms: software architecture and performance properties are represented in PCM; the class structure is represented in the UML class diagrams; the implementation and full runtime semantics are represented in Java, either as a plain textual representation of code or using a model-based representation like MoDisco [7].

Different developer roles (not displayed in the figure) may use (legacy) view types like component diagrams or class diagrams (VT_1) to access the information in the metamodels, but would also like to have integrated views like the "component-class implementation view" (VT_2) displayed in the example.

4. FLEXIBLE VIEWS FOR VIEW-BASED MODEL-DRIVEN DEVELOPMENT

The approach presented in this paper is based on the concept of Orthographic Software Modeling. Current OSM publications are however lacking a definition of the single underlying model or a way of constructing it. In the prototypical implementation [3], the authors built a metamodel for the SUM manually and from scratch. Existing MDD projects will however contain several legacy metamodels which have to be migrated to be used in a view-based approach. We therefore propose the construction of a *modular SUM* which contains several (legacy) metamodels, but can be accessed as a single entity: View types act as interfaces and are the only way of accessing and manipulating information. The sub-metamodels themselves do not have to be modified to work in the modular SUM, which makes the approach non-invasive and backwards-compatible. In the example of Figure 2, the SUM metamodel (indicated as the large circle in the middle) consists of three sub-metamodels (PCM, UML, Java) and also stores additional information, e.g., the class-component mapping, which is not part of any of the single models. Developers can continue to use their legacy view types like class diagrams and Java source code, but can also use integrated views that gather information from heterogeneous metamodels. Consistency constraints (displayed as double arrow $\leftrightarrow$) guarantee that the SUM is always in a consistent state if changes are made to one of the sub-models. These constraints are not necessarily defined on every pair of sub-models, as in the example, where there is no constraint between PCM and Java directly. The definition of these constraints and the update mechanisms for the restoration of a valid model are part of the view-based approach, but not in the focus of this paper.

For the creation of custom, user-specific views on a modular SUM, we present the concept of *flexible views.* A flexible view can be pre-defined or defined at runtime (of the development framework). It is dependent from the instance level of the underlying models; i.e., following Definition 1, it defines a view type as well as a view. Different flexible views can re-use an existing view type. This can be compared to a view in relational databases, which defines the schema of the result (i.e. the view type), and the result set itself (the view). A flexible view is defined by rules that determine its contents and behaviour; these rules can be altered for specific modeling purposes by developers themselves or by an additional developer role. In the OSM approach, a "methodologist" role is suggested, who defines the rule set for each software project.

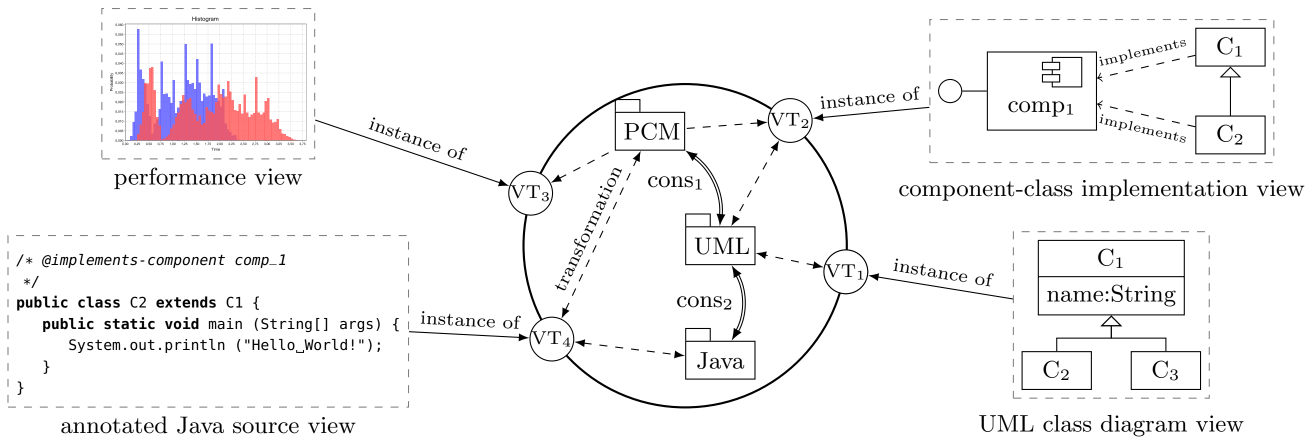
DEFINITION 2. Flexible views *are partial views on a software system. They*

- *can be defined at runtime of the development framework*
- *are defined on modeling instances*
- *may introduce new elements to subsume elements of the underlying model*
- *support an online modeling workflow*
- *contain rules describing*
  - *the selection of elements which are displayed (based on instance properties)*
  - *which edit operations are possible*
  - *how edit operations are reflected in the underlying model*

A view type can be understood as a metamodel, with the single views as instances of this metamodel respectively (indicated by the solid arrow $\longrightarrow$). To create a view, a model transformation has to be executed from the SUM to the view type. In the example of Figure 2, this is indicated as a dashed arrow ($\leftarrow - - \rightarrow$). In case of the legacy view type VT_1 (class diagram), the view type corresponds to only one sub-metamodel of the modular SUM metamodel; hence, the view type is fully editable, and the transformation is the identity relation. (Additional operations between the sub-models of the SUM may be necessary so that all consistency constraints are satisfied after an editing step.) The legacy view types enable the usage of existing modeling tools and can also be used for im- and export. View type VT_2 represents a more complex example: A developer would like to have a "class-component implementation view" which shows instances of two heterogeneous metamodels (PCM and UML) and an *implements*-relation between them. The purpose of this view type is to edit the mapping information between classes and components, but not to edit the class or component model. Thus, the developer creates a flexible view with the following parameters:

- PCM components and UML classes can only be displayed, but not be deleted or edited, e.g., renamed
- details of classes and components are omitted, e.g. attributes are not shown for classes, only provided interfaces are shown for components
- the *implements*-relation is editable

The synchronisation between the view and the SUM depends on the design decision of where to store the *implements*-relation; since it is neither present in the PCM nor in the UML metamodel, it could be stored as an annotation



```
/* @implements-component comp_1
 */
public class C2 extends C1 {
   public static void main (String[] args) {
      System.out.println ("Hello_World!");
   }
}
```

Figure 2: Example: Component-based development

on either side, or in an additional artifact, e.g., a mapping model which would have to be added to the SUM then. Let us assume that a methodologist has decided to store the information as an annotation in the UML sub-model of the SUM. Since components cannot be edited by a view of this view type, the transformation from PCM to VT_2 will be unidirectional; although classes themselves are not modified by the view either, but the mapping information is added as an annotation, the transformation from UML to VT_2 is bi-directional. Neither of the metamodels has to be modified, since the existing UML annotation mechanism is used. The user of the view needs not know how the implements-relation is technically represented in the SUM; it is even possible to change the representation strategy without modifying the view type.

In general, readability and editability of views can also be used to implement access control to the system. A software engineer who is responsible for the object-oriented design of a system may not be authorized to change the software architecture, since this is the responsibility of a software architect. In the example, the component-class implementation view type VT_2 guarantees that the software engineer does not change the architecture by making the components read-only, but gives the ability to update the class-component mapping.

The synchronisation by transformations in our approach should be based on the propagation of deltas, as proposed in [11]. For the propagation of changes from the view to the SUM, editor traces can be used to determine the deltas; for the other direction, a diffing mechanism such as EMF-compare [12] computes the deltas, so delta-based synchronisation is used for editing, but state-based synchronisation for updates in views after changes to the SUM. The deltas should be classified according to their impact on the consistency of the SUM, so that the framework can already suggest strategies for restoring consistency if an edit operation violates constraints. In [8], the author presented a classification scheme for metamodel evolution that can be adapted for the use in SUM-based modeling; the effect of a delta could then be estimated based on its impact on the consistency.

As mentioned in subsection 2.3, it is difficult and in many cases impossible to define bi-directional transformations. It is however necessary to have bi-directionality for the synchronisation of views with the SUM. To adress this problem, we will provide a catalogue of possible operations and rules for the definition of view types and consistency constraints, which include a description of their invertability and the limitations, e.g., that non-invertable operations will lead to a read-only view. The goal is that the methodologist who adds a metamodel to the modular SUM or the developer who defines a custom view is provided with information on how the choice of consistency constraints or selection mechanisms affects the editability of views.

5. RELATED WORK

5.1 Aspect-Oriented Software Development

The concept of *Aspect-Oriented Programming* has also been extended to model-driven development, yielding *Aspect-Oriented Modeling* and, more generally *Aspect-Oriented Software Development (AOSD)* [9, 30]. One main focus of AOSD is "breaking the hegemony of the dominant decomposition" [5] in software development: software is re-structured along system-wide, so called *cross-cutting* concerns, which have to be distinguished from non-cross-cutting concerns like component or object structures. In contrast to view-based modeling, where all information is contained in the model and is organised and structured in views, the information in AOM contained in seperate concerns. Changes to such a concern are not reflected in the underlying system immediately, but unified afterwards in a *weaving* or *composition* process. In our proposed view-based approach, aspects are expressed in partial views that always in sync with the underlying model, so there is no weaving or composing; the model is designed in a way that the information needed in the views can be extracted and manipulated by model-to-model transformations.

5.2 Distributed Modeling Approaches

The *ModelBus* approach [17, 1] aims to provide sharing mechanisms for models in a distributed and heterogenous model-driven process. A central repository serves as storage for models, which are then transformed into tool-specific formats using service-based invocation techniques. ModelBus also supports collaborative work on a software model using different tools, e.g., IBM Enterprise Architect and Papyrus. ModelBus is a tool-centric approach that is suitable for the the interchange of EMF/MOF-based metamodels, describing the same domain, between heterogeneous modeling tools. It is not suitable for the integration of information from heterogeneous metamodels.

DuALLy [23] is a framework that aims to create interoperability between architecture description languages (ADL). It uses higher-order transformations to translate existing languages via a hard-coded core set of architectural models and weaving models (the so-called A0 profile). This is similar to the idea of the SUM in the OSM approach, but limited to the domain of architectural engineering, since the concepts in the A0 model are fixed and cannot be extended to other domains. An implementation of DuALLy based on Eclipse and ATL exists.

5.3 Databases

Many of the problems that are encountered in view-based modeling have counterparts in database research. Relational databases also offer the possibility to create views which may be editable. Also, 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 the partial view is manipulated, the *view update problem* [4, 10] arises, which is a central issue in relational databases which is well understood, but mainly unsolved [22]. The process of re-integrating 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 in a certain view can also be reverted in that same view without losing information in the underlying database. We will chose the same approach for the definition of views in the OSM approach; 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 [14], which can be applied to model transformations using graph structures (see subsection 2.3).

Integrating heterogeneous metamodels and instances bears similarities the well-known problem of schema integration of heterogeneous databases [25, 27]: 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, and is why we refrain from defining automatisms for the identification of similarities in heterogeneous metamodels automatically, e.g., by graph-based comparison using heuristics. 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. 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 for the rapid creation of specialized views, which is not described here due to limited space. An engine that uses metamodel generation and transformation generation creates view types the views from textual queries. This enables the developer to create and modify custom views without having to design metamodels and transformations manually. Currently, the prototypical implementation supports read-only views.

We plan to validate our approach by applying it to the component-based development process of the Palladio metamodel [6]. As sketched in Figure 2, 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 system, e.g., the MediaStore example from [6] 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 our orthographic 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 valide the following theses: Firstly, the feasibility of the OSM approach in general; secondly, that the view-based approach reduces the complexity for the developer; thirdly, that the approach improves the quality of software, i.e., improves the consistency and reduces drift and erosion.

7. CONCLUSION/NEXT STEPS

We have presented a view-based development process based on the OSM approach and introduced a novel modeling paradigm called *flexible views*. We motivate this approach by a component-based usage scenario 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. Furthermore, editability restrictions serve as an access control mechanism to support a software development process with structured permissions for the different developer roles.

As next steps, we plan to extend our textual DSL for the definition of 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 bidirectionality. 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.

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.

References

[1] Eric Armengaud et al. “Model-based Toolchain for the Efficient Development of Safety-Relevant Automotive Embedded Systems”. In: *SAE 2011 World Congress & Exhibition*. 2011.

[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] F. Bancilhon and N. Spyratos. “Update semantics of relational views”. In: *ACM Trans. Database Syst.* 6.4 (Dec. 1981), pp. 557–575.

[5] Elisa Baniassad and Siobhan Clarke. “Theme: An Approach for Aspect-Oriented Analysis and Design”. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 158–167.

[6] Steffen Becker, Heiko Koziolek and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: 82 (2009), pp. 3–22.

[7] Hugo Bruneliere et al. “MoDisco: a generic and extensible framework for model driven reverse engineering”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ASE ’10. Antwerp, Belgium: ACM, 2010, pp. 173–174.

[8] Erik Burger and Boris Gruschko. “A Change Metamodel for the Evolution of MOF-Based Metamodels”. In: *Modellierung 2010, Klagenfurt, Austria, March 24-26, 2010*. Ed. by Gregor Engels, Dimitris Karagiannis and Heinrich C. Mayr. Vol. P-161. GI-LNI. 2010.

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

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

[11] Zinovy Diskin et al. “From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case”. In: *Model Driven Engineering Languages and Systems*. Ed. by Jon Whittle, Tony Clark and Thomas Kühne. Vol. 6981. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 304–318.

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

[13] Anthony Finkelstein et al. “Viewpoints: A Framework for Integrating Multiple Perspectives in System Development”. In: *International Journal of Software Engineering and Knowledge Engineering* 2.1 (1992), pp. 31–57.

[14] 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.

[15] Thomas Goldschmidt. “View-based textual modelling”. PhD thesis. Karlsruhe, 2011.

[16] Thomas Goldschmidt, Steffen Becker and Erik Burger. “View-based Modelling - A Tool Oriented Analysis”. In: *Proceedings of the Modellierung 2012, Bamberg*. Mar. 2012.

[17] Christian Hein, Tom Ritter and Michael Wagner. “Model-Driven Tool Integration with ModelBus”. In: *Workshop Future Trends of Model-Driven Development*. 2009.

[18] “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.

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

[20] P.B. Kruchten. “The 4+1 View Model of architecture”. In: *Software, IEEE* 12.6 (Nov. 1995), pp. 42–50.

[21] Marius Lauder et al. “Bidirectional Model Transformation with Precedence Triple Graph Grammars”. In: *Modelling Foundations and Applications*. Ed. by Antonio Vallecillo et al. Vol. 7349. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 287–302.

[22] 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.

[23] I. Malavolta et al. *Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies*. Tech. rep. 1. Jan. 2010, pp. 119–140.

[24] Janis Putman. *Architecting with RM-ODP*. Upper Saddle River, NJ: Prentice Hall, 2001.

[25] 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.

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

[27] 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.

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

[29] Perdita Stevens. “Bidirectional model transformations in QVT: semantic issues and open questions”. In: *Software and Systems Modeling* 9 (1 2010), pp. 7–20.

[30] 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.