View-Centric Engineering with Synchronized Heterogeneous Models

Max E. Kramer
Karlsruhe Institute of Technology, Karlsruhe, Germany
max.e.kramer@kit.edu

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

Michael Langhammer
Karlsruhe Institute of Technology, Karlsruhe, Germany
michael.langhammer@kit.edu

ABSTRACT

Model-Driven Engineering provides an abstract representation of systems through the use of models and views. For complex systems, however, finding a single model and a single view to represent all relevant information of the system is infeasible. Specialized models for specific subsystems, domains or abstractions are more concise and thus more efficient than monolithic models. Furthermore, different tasks and concerns often require different views on the same model. Sustaining the consistency between different views and models is hard, especially if new models and views are dynamically added.

In this paper, we present an approach that supports flexible views that may involve multiple models conforming to different metamodels. The approach is based on Orthographic Software Modeling and synchronizes individual instances using model transformations. These transformations are generated from view type definitions, metamodel correspondence rules and invariants, which are defined in a domain-specific language. We illustrate our approach with an application that combines component-based architectures with object-oriented source code and class diagrams.

Categories and Subject Descriptors

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

Keywords

Model-Driven Engineering, View-Based Modeling, Component-Based Software Architectures, Synchronization

1. INTRODUCTION

Model-driven development processes suffer from fragmentation of information across instances of different metamodels, which are used to address the view points of the underlying domain [4]. This can lead to redundancies if the metamodels share a semantic overlap, or even to inconsistencies if models are contradicting each other, but describe the same system. The synchronisation of these artefacts is a complicated task which requires manual effort.

View-based modelling techniques [13, 1] try to address this problem by treating views as first-class entities of the modelling process: Information is stored in a central model, while users operate on custom views that represent only the model parts and the kind of information that is relevant to them. These views are partial, i.e. they do not show the complete model, thus reducing the complexity for developers. From a model-driven perspective, a view is a special model that conforms to a view type metamodel. To create a view, a software developer has to choose an existing view type or define a new one, and implement model transformations from the central metamodel to the view. Although this concept solves the problem theoretically, existing view-based approaches fail to define a metamodel for the central model, since such a universal metamodel would have to cover all possible view points of software development. Furthermore, each definition of a new view type requires the creation of a new metamodel and new model transformations, which is a time-consuming and error-prone process.

We therefore propose the VITRUVIUS[1] approach for view-centric model-driven software development. VITRUVIUS implements the concept of dynamically created *flexible views*. The approach is based on the *Orthographic Software Modeling (OSM)* concept by Atkinson et al. [1], but does not use a monolithic single underlying metamodel (SUM). Instead, legacy metamodels are combined in a non-invasive way to serve as a modular underlying model.

The contribution of this paper is the presentation of a view-based development process based on a modular single underlying model. We describe the construction of the modular SUM, regarding the handling of legacy metamodels, coupling of consistency rules and synchronisation mechanisms with view types, and the definition of user-specific flexible views. The remainder of this paper is structured as follows: In Section 2, we explain the foundations of VITRUVIUS and flexible views. Section 3 provides an overview over the individual artefacts, processes, and characteristics of our approach. Details concerning the synchronization between models are discussed in Section 4. An application of our approach to component-based software architectures exemplifies the concepts in Section 5. Section 6 discusses related work and Section 7 draws some final conclusions.

[1] VITRUVIUS (VIew-cenTRic engineering Using a VIrtual Underlying Single model)

VAO '13, July 2, 2013, Montpellier, France
Copyright 2013 ACM 978-1-4503-2041-2
http://dx.doi.org/10.1145/2489861.2489864 ...$15.00.

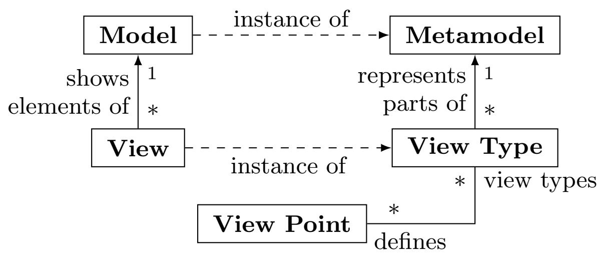


Figure 1: Overview of view type, view and view point. A view point relates to a metamodel and is represented by one or multiple view types.

2. FOUNDATIONS

In Model-Driven Engineering (MDE), the central artefacts of software systems are models which conform to metamodels. These models are used to derive all other artefacts of the software system using transformations or generations [19].

In view-based modelling, views are the primary entities which are used to access and modify models [13]. A view typically shows only a part of a system. The goal of view separation is to reduce the complexity for developers. Every view needs to conform to a view type and pertains to a view point. Fig. 1 shows the relation between view type, view, and view point. We use the definitions for these concepts that are given in [5]:

> 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, e.g., the static architecture or dynamic aspects of a system.

The Orthographic Software Modelling (OSM) approach by Atkinson et al. [1] introduces the idea of storing all data of a software system in a Single Underlying Model (SUM). Accessing and modifying the software system is only possible via predefined view types. The views are dynamically generated from the SUM. In the approach presented in this paper, we follow the idea of generating views to edit and view parts of a system. We also maintain the requirement that all access has to occur via views and that all information is centrally maintained. The differences between our approach and OSM, e.g. the omission of the requirement for an explicit SUM, are discussed in Section 6.

The *Flexible views* [5] concept offers the rapid definition of custom, user-specific views for view-based development. A flexible view and the view type it conforms to are generated from a view specification in a light-weight Domain-Specific Language (DSL). The concept gives users the possibility to add new view types and views to the pre-defined views in a rapid manner, since the users need to define neither the metamodel behind the view type nor the necessary transformation rules between the source models and the view explicitly; these artefacts are generated automatically from the definition of flexible views.

3. OVERVIEW

In this section, we present the VITRUVIUS approach for developing and analysing systems that are represented using multiple modelling languages by models that conform to multiple metamodels.

3.1 Mandatory Views

Our approach provides flexible views conforming to well-defined view types that may combine elements of different metamodels as shown in Fig. 2. The flexible view definition FV_1 specifies which elements (*selection*) of which metamodel types (*structure*) are displayed. It may also restrict the set of *operations* that are permitted on the displayed elements. For the individual views conforming to a specific view type, the origin of the elements is transparent. View types that combine or omit elements and trivial view types that display every element unchanged cannot be distinguished. This is possible, because all access to system elements has to occur through views conforming to view types.

3.2 Controlled Complexity

All views have to report atomic modifications of the elements that they are displaying. Based on this information, internal model instances of the involved metamodels are synchronized. These internal models cannot be directly accessed from outside. This allows us to control the complexity of synchronizations through the provided view types. For example, effects of operations that are not permitted do not need to be considered for synchronization.

3.3 Abstract Consistency

It is possible to define declarative *correspondence* and *consistency* rules for all pairs of metamodels with elements representing the same entities. If a view reports modifications, synchronization transformations that are generated from these definitions and modification *response* actions are executed. This separation of an abstract mapping between metamodels and the resulting synchronization operations decouples metamodels to allow for evolution according to their individual evolution pressure. If a metamodel shall be updated or added, it is sufficient to update or add its correspondence rules and the view types that display its elements.

3.4 Ease of Use

If elements of different metamodels shall be displayed as a single element in a view type, then this information can be used to propose correspondence relations between these metamodels. In return, correspondence relations that link equivalent elements in different metamodels can be used to suggest view types that combine these elements. This should reduce the effort needed to develop new view types and ease the addition of new metamodels. Furthermore, the generation of views and synchronization transformations from declarative definitions relieves the user from implementing these views and transformations.

4. SYNCHRONIZATION

A central idea of our approach is to derive synchronization transformations from declarative metamodel correspondence rules, from consistency invariants and from imperative actions that are formulated as responses to modifications. A benefit of this generative approach is that users do not need

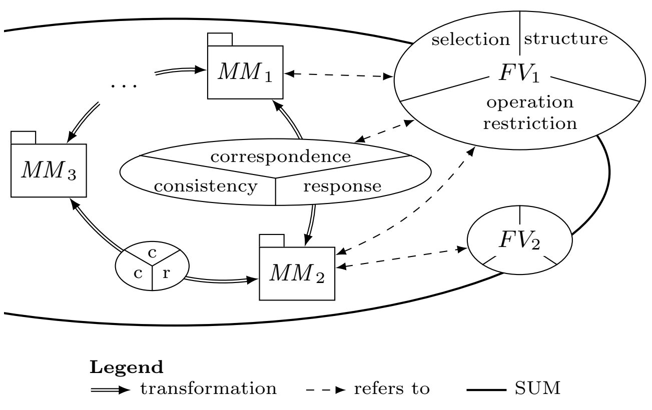


Figure 2: Structure and relations of view types, metamodels and their synchronization

to deal with technical details of synchronization transformations because they are separated from abstract synchronization logic. Furthermore, the division into individual rules and actions eases the reuse within projects with identical or similar metamodels.

We briefly describe the overall flow of information and detail the individual steps of synchronization afterwards. In the first step, the user specifies a structural mapping between metamodel elements with a DSL for correspondence. Then, he defines structural constraints as invariants using the Object Constraint Language (OCL). Last, the user may specify response actions that should be triggered after specific modifications or invariant violations. These response actions are expressed using a general-purpose transformation language and may be used to restore violated invariants. We generate synchronization transformations for individual metaclasses and modification types from the mapping, invariants and response actions. The generated transformations are triggered on behalf of an individual view through the corresponding view type, which also establishes and persists instance-level correspondences between model elements.

4.1 View Changes Trigger Transformations

Because all access to models occurs using views conforming to view types, we do not need to compute differences to cope with model modifications. For every element of a view, the view type definition specifies whether the element stems from one or multiple metamodels. Whenever a modification occurs in a view, the corresponding view type triggers a transformation that is chosen based on the element type and modification type. The decision cascade exhibits three important characteristics: First, for view type elements resulting from multiple metamodels, a precedence relation for these metamodels is used. This precedence is either explicitly specified or implicitly derived from the order in which the elements where mentioned in the definition. Second, the transformation direction is only determined by the precedence of the metamodels and the bidirectionality of references. Third, conflicts are only possible if an update occurs for a bidirectional reference for which the metaclasses at both ends of the reference have a contradictory metamodel precedence.

4.2 Synchronization Rules, Invariants and Response Actions

Synchronization logic is defined using a DSL that consists of three individual parts for declarative correspondence rules, consistency invariants and imperative modification response actions. Metamodel correspondences can only be defined for exactly two metamodels, but the other two parts of the DSL can be used for specifications for a single metamodel as well as for multiple metamodels. Correspondence rules specify which metaclasses of a metamodel correspond to which metaclasses of another metamodel. They also specify which features, i.e. attributes and references, of a mapped metaclass shall be mapped to features of the other metaclass. The DSL allows for nested in-line mappings of metaclasses within feature mappings. Consistency is specified using OCL invariants that have to be named and may expose parameters for response actions. Fine-grained synchronization behaviour may be specified as actions in response to violated invariants or after ordinary modifications. All actions may restrict the type of the modification after which the action shall be executed and the type of the element that was affected by the modification. If an action restores a violated invariant, it has to use all or none of the parameters that are exposed by the invariant. Corresponding elements may be retrieved with helper functions that use the instance-level correspondence information, which is maintained according to modification operations. A response action for UML class diagrams and Java, for example, may respond to the creation of a class without an explicit package by retrieving the default package and adding the class to it.

4.3 Generating Sync Transformations

The synchronization rules, invariants and actions that are specified using the DSL are used to generate transformations for every mapped feature of every mapped metaclass. These transformations are expressed with a general-purpose transformation language so that they can be debugged using existing debuggers. At the moment, it is unclear whether all rules, invariants and actions for a set of metamodels can directly be reused in multiple projects. If project-specific synchronization is necessary it should be separated from project-independent synchronization in order to ease reuse.

If not all necessary synchronization tasks can be performed automatically the user has to be supported in triggering the right synchronizations manually. For the prototype implementation of our approach, we plan to inform users about automatic resolution of inconsistencies and plan to design semi-automated conflict resolution dialogues according to the best practices reported by Grundy et al. [14].

5. APPLICATION SCENARIO

5.1 Scenario

We evaluate our approach and a prototypical implementation with a first application scenario, which combines models of a component-based architecture, UML class diagrams, and Java source code. We use the Palladio Component Model (PCM) [3], an Architectural Description Language (ADL) for component-based systems, which was initially developed for performance predictions at design time. In the first application of our approach, however, we only consider common architectural elements and view types of the PCM. The

PCM repository view type displays components and the interfaces they are providing and requiring; the PCM system view type shows the composition and connectors of component instances. The goal for this application scenario is to synchronize a component-based architecture model, UML class diagrams and Java source code (see Fig. 3). We plan to provide the following four view types (VT):

VT_1: A UML class diagram view to edit and display the classes of the software system.

VT_2: The component-class implementation view uses information from the PCM and the UML class diagrams, and displays the classes which realizes a component.

VT_3: The standard PCM component repository and system composition views.

VT_4: The annotated Java source code view uses information from the PCM and Java source code. The view allows the editing of source code and displays the corresponding PCM element of the current code fragment.

As described in Section 4, we have to define correspondence rules between the metamodels we use. Since object-oriented code does not provide constructs for architectural elements, the definition of synchronization rules between PCM, UML class diagrams, and source code, poses the challenge of mapping components etc. to UML and source code elements like classes and packages. Because these mappings cannot be assigned to a single metamodel the resulting instance-level correspondence information is stored in hidden separate models that conform to the correspondence rules. This correspondence information can be represented in view types as shown for the component-class implementation view in Fig. 3.

5.2 Synchronization Example

In the following we present a simplified example that uses our DSL to synchronize *OperationInterfaces* of the PCM *Repository* with UML *Interfaces*. To illustrate the three different DSL parts for correspondence, consistency and response actions we provide three small code snippets.

The first part of the DSL is used to define a correspondence mapping between the metaclasses of the two involved metamodels and between the features of these metaclasses.

```
import http://sdq.ipd.uka.de/PCM/Repository/5.0 as repo
import http://www.eclipse.org/uml2/2.1.0/UML as umlcd

// correspondence rules
map repo:OperationInterface to umlcd:Interface
  with signatures::OperationSignature
      to ownedOperation::Operation
    with returnType::DataType
      to ownedParameter::Parameter.type::Type
      when ownedParameter.direction = return
    and with parameters::Parameter
      to ownedParameter::Parameter
      when ownParam.direction <> return
```

Each *OperationInterface* of a PCM *Repository*, which contains all components and interfaces, is mapped to an UML interface. The metaclasses *OperationSignature* and *Operation* are mapped to each other in a nested in-line mapping that also maps the references *signatures* and *ownedOperation* which are typed with these metaclasses. The parameter metaclasses and references of *OperationSignature* and *Operation* are mapped in two more in-line mappings that are bound to conditions formulated in when-clauses: If the *direction* of a UML *Parameter* is *return*, the type of the owned parameter is mapped to the return type of the PCM *OperationSignature*. In all other cases, a UML *Parameter* is mapped to a *Parameter* of the PCM.

The second part of the synchronization DSL can be used to define consistency constraints as OCL invariants. In our example, this is used to specify that all interfaces in PCM repositories have to exhibit pairwise distinct names.

```
// consistency invariant
context repo:Repository
inv uniqueInterfaceNames(i::repo:Interface , j :: repo: Interface ):
 self . interfaces −>forAll(i,j | i .entityName <> j.entityName)
```

The invariant is named *uniqueInterfaceNames* and exposes two PCM *Interfaces* that can be bound in response actions.

The third and last part of the DSL is used to define a response action that restores the *uniqueInterfaceNames* invariant if it is violated after the creation of a PCM *Interface*.

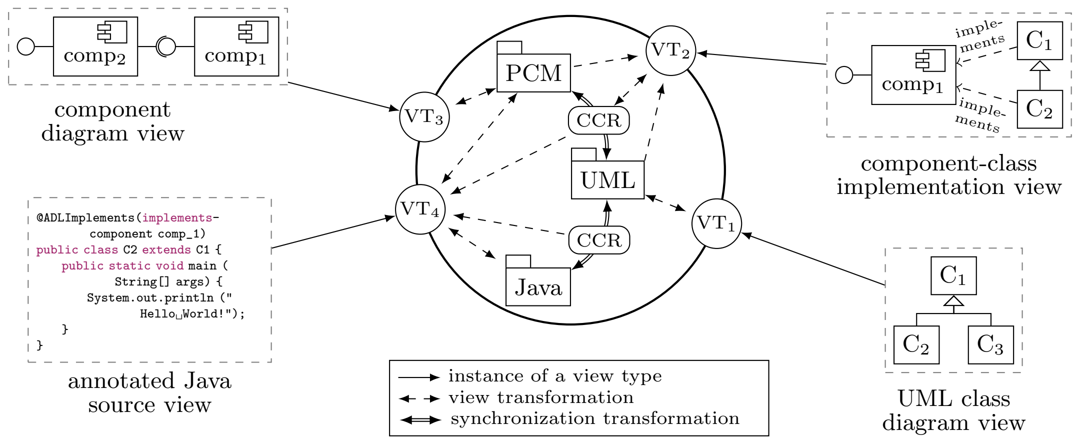
```
// response action
var interfaceNameCount : Map<String,Integer>
on creation of interface:repo: Interface
restore inv interfaceNamesUnique(i::repo:Interface,
  j :: repo: Interface )
by {
  var occurrences = interfaceNameCount.get(i.entityName)
  occurrences = (occurrences == null) ? 2 : occurrences++
  interfaceNameCount.put(i.entityName, occurrences)
  interface .entityName += occurrences
}
```

First, a global variable that maps names of interfaces to their number of occurrences is declared. Then the trigger conditions for the response action are specified using the keywords *on creation of* and *restore inv*. They ensure that the response action is only triggered if an instance of the *Interface* metaclass is created and the specified invariant is violated. In this case, the response action determines how many *Interfaces* were already created with the name that caused the collision. If it is the first collision, the number of occurrences is set to *2*. Otherwise the number of occurrences is incremented by one. Last, the calculated number of occurrences is appended to the name of the newly created *Interface* and stored for future name collisions.

The three snippets for our synchronization DSL do not represent a complete use case but are shown to exemplify some of the possibilities of our approach and the DSL. For the full application scenario one has to specify similar mappings, invariants and response actions for all metaclasses and structural constraints that should be sustained automatically.

6. RELATED WORK

ModelBus [15] is a framework for model-driven tool integration and service orchestration for Service Oriented Architectures (SOA). In order to use ModelBus, existing web services and tools have to be extended to trigger the creation of models. By means of specific adapters a communication bus can be used to access models and functional services. ModelBus internally merges model versions and provides a change notification system. A major difference to our approach is that all tools and transformations have to be adapted to communicate with a bus and in order to trigger the creation of models. Furthermore, no concepts for managing the semantic overlap between different languages are provided.



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

Figure 3: The component-based application scenario with a modular SUM that consists of PCM models, UML class diagrams, and Java source code.

Giese et al. [12] presented a model synchronization approach based on Triple-Graph Grammars (TGGs). TGGs are triples of directed, typed graphs that describe transformations using a left hand side, an interface and a right hand side together with morphisms that map the interface to the left hand side and to the right hand side. This is similar to the correspondence mappings of our approach but requires conversions for models and graphs because the mapping is not directly described in terms of the modelling language. Furthermore, TGGs can only be used for monotonic productions and therefore graph rules cannot delete elements [18]. Giese et al. initially used TGGs for incremental transformations based on a dependency analysis between transformation rules. Later on, they applied their approach to synchronize SysML and AUTOSAR models in a fully bidirectional way [11]. In this application they support two modes: a transformation mode that applies rules as long as matches are located and a synchronization mode that transforms only nodes that were flagged by a change listener. This is different to our approach where changes trigger transformations that were generated for a specific operation on a specific type and which are applied only on the elements that are the subjects of change.

In multi-viewpoint modeling [10, 17], correspondences are specified between viewpoints to achieve synchronisation. The approach has been implemented prototypically with declarative logical languages [10] and for UML with QVT-R and MediniQVT [17]. The correspondences in multi-viewpoint modeling are defined directly between the view types, which are called *viewpoint languages* by the authors. Therefore, the complexity of correspondence links grows exponentially with the number of view types, in contrast to approaches that use a central underlying model in a hub-and-spoke manner.

Diskin et al. [8] presented an approach for consistency checking of heterogeneous models. It is based on category theory and therefore applicable to all graph-like structures. They transform heterogeneous models into homogeneous models with a merge based on explicit instance-level mappings that have to be defined manually. This reduction makes it possible to apply existing techniques for correspondence spans, which are similar to the mapping rules of our approach. The approach of Diskin et al. can also calculate values for intermediate types using OCL, treats indirect model overlap and checks constraints that reside in none of the involved metamodels. Nevertheless, this approach to global consistency checking was developed in order to check consistency and not for sustaining it by synchronization means. Furthermore, the authors of this graph-based approach state that “the main question is how effectively a multimodelling tool based on the framework could be implemented” [8].

DUALLy [16] is an approach for synchronizing different ADLs. Similar to the original concept of OSM, it uses a central ADL to synchronize all other ADLs in a hub-and-spoke manner. If multiple notations are used, kernel extensions [7] have to prevent the loss of information that cannot be represented using the central ADL. Non-bijective transformations between instances of ADLs can be handled using a unification-based matching process proposing alternative models [9]. In contrast to our approach, however, DUALLy does not support partial or joined views and is restricted to architectural modelling languages.

Chichetti et al. [6] presented an approach for user-defined views that display a subset of an overall model. Transformations produce difference models that are used in a higher-order transformation to generate transformations that propagate changes by passing through the overall model. Difference computation and change propagation profit from the subset relationship between view metamodels and the overall metamodel. In contrast to our approach, differences have to be calculated by matching elements based on identifiers that are marked as unique by the user. In addition, an overall metamodel, which can only omit but not join elements, has to be synthesized from individual views.

As mentioned in Section 2 Atkinson et. al. [1] presented the OSM approach, which introduced the idea of storing all data of a software system in a SUM. Our approach has two main improvements compared to OSM: Firstly, we do not need a predefined metamodel that includes all information of the software system. Instead we are using multiple metamodels that are synchronized via consistency rules. Secondly, our approach is easier to extend, e.g. we can include new metamodels into our modular SUM without adapting existing metamodels. OSM theoretically is not limited to software engineering. Kobra [2], the prototypical implementation of OSM, is, however, limited to software engineering.

Our approach is also not limited to software engineering as it can be used whenever two or more metamodels have to provide joined and synchronized views. We plan to apply our approach, for example, in the energy domain to synchronize metamodels which are used by different participants of the energy market.

7. CONCLUSIONS

In this paper, we have presented a view-centric engineering approach which synchronizes instances of different metamodels using generated transformations. Flexible views serve as the only point of access for tools and transformations. Declaratively defined view types and correspondence mappings between metamodels are used to generate synchronization transformations that can be extended by modification response actions. This reduces the complexity of synchronizations and decouples the involved models and metamodels for better evolution support. We exemplified our approach using an application to component-based software development where architectural models are synchronized with UML Class Diagrams and Java source code. This application illustrates how existing views can be complemented with new views that display synchronized information.

References

[1] 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.* Vol. 69. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2010, pp. 206–219.

[2] Colin Atkinson et al. "Modeling Components and Component-Based Systems in KobrA". In: *The Common Component Modeling Example.* Springer, 2008, pp. 54–84.

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (Jan. 2009), pp. 3–22.

[4] Lars Bendix and Pär Emanuelsson. "Requirements for Practical Model Merge – An Industrial Perspective". In: *Model Driven Engineering Languages and Systems.* Vol. 5795. LNCS. Springer Berlin / Heidelberg, 2009, pp. 167–180.

[5] Erik Burger. "Flexible Views for View-Based Model-Driven Development". In: *WCOP '13: Proceedings of the 18th international doctoral symposium on Components and Architecture.* to appear. Vancouver, Canada: ACM, 2013.

[6] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. "Supporting Incremental Synchronization in Hybrid Multi-view Modelling". In: *Models in Software Engineering.* Vol. 7167. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 89–103.

[7] Davide Di Ruscio et al. "Model-Driven Techniques to Enhance Architectural Languages Interoperability". In: *Fundamental Approaches to Software Engineering.* Vol. 7212. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 26–42.

[8] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. "Specifying overlaps of heterogeneous models for global consistency checking". In: *Proceedings of the First International Workshop on Model-Driven Interoperability.* MDI '10. Oslo, Norway: ACM, 2010, pp. 42–51.

[9] Romina Eramo et al. "A model-driven approach to automate the propagation of changes among Architecture Description Languages". In: *Software and Systems Modeling* 11 (1 2012), pp. 29–53.

[10] 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.

[11] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. "Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent". In: *Graph Transformations and Model-Driven Engineering.* Vol. 5765. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 555–579.

[12] Holger Giese and Robert Wagner. "From model transformation to incremental bidirectional model synchronization". In: *Software and Systems Modeling* 8 (1 2009), pp. 21–43.

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

[14] John Grundy, J. Hosking, and W.B. Mugridge. "Inconsistency management for multiple-view software development environments". In: *Software Engineering, IEEE Transactions on* 24.11 (1998), pp. 960–981.

[15] Christian Hein, Tom Ritter, and Michael Wagner. "Model-driven tool integration with modelbus". In: *Workshop Future Trends of Model-Driven Development.* 2009, pp. 50–52.

[16] I. Malavolta et al. "Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies". In: *Software Engineering, IEEE Transactions on* 36.1 (Jan. 2010), pp. 119–140.

[17] 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.

[18] Andy Schürr. "Specification of graph translators with triple graph grammars". In: *Graph-Theoretic Concepts in Computer Science.* Vol. 903. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 151–163.

[19] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.