A Generative Approach to Change-Driven Consistency in Multi-View Modeling

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

ABSTRACT

When software architectures are modeled from different viewpoints using different notations, it is necessary to keep information that appears in several models consistent. To achieve consistency for a specific system, developers need two competences: First, they have to be able to express the conceptual relationships between the elements of the involved modeling languages and domains. Second, they have to be able to enforce these relationships by implementing model transformations that keep specific model instances consistent. Current transformation approaches, however, do not separate this conceptual challenge of specifying consistency from the technical complexity of implementing it.

To ease multi-view modeling, we present a generative approach, in which change-driven in-place transformations are generated from abstract specifications in order to sustain consistency between several models. These specifications can be expressed by domain experts using a consistency language that supports declarative mappings, normative invariants, and imperative response actions. The transformations generated from consistency specifications make it possible to reuse and customize technical solutions to typical consistency preservation problems. We will evaluate our approach with two case studies of component-based engineering.

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 Engineering, Transformations, Co-Evolution, View-Based Modeling, Model Synchronization

1. INTRODUCTION AND MOTIVATION

Many software systems are modeled using several modeling languages in order describe the systems appropriately for every specific task. These different models may conform to different metamodels, and they represent views in which identical parts of a system can be described in different and redundant ways. The architecture of a component-based software system, for example, may be modeled in an abstract way using an Architectural Description Language (ADL). The internal structure and behavior of individual components is, however, often better described with class and sequence diagrams of the Unified Modeling Language (UML) or with modeling languages tailored to a specific domain. In such a multi-view setting, elements may appear in several models and may be changed in isolation. It is therefore necessary to keep these models consistent when changes occur during the development and maintenance of the system.

Current approaches for consistency preservation in multi-view modeling, however, mix the conceptual challenge of *expressing* consistency relations between domain elements with technical challenges of *realizing* them. They use general-purpose transformation languages, such as Query/View/Transformation (QVT), or Triple Graph Grammars (TGGs), which also support completely different transformation scenarios, for example, empty target models or several input models. These languages cannot take advantage of the specific conditions of consistency preservation transformations. Therefore, developers have to repeatedly implement solutions to technical transformation problems that are almost identical in all consistency preservation scenarios.

In this paper, we refine our proposal [13] for a language for consistency specifications, which can be used to automatically generate change-driven consistency preservation transformations. It supports declarative mappings between metaclasses similar to QVT relations, as well as normative invariants defined in the Object Constraint Language (OCL), and imperative transformation code to customize default reactions to model changes. From a specification, incremental consistency preservation transformations, which propagate incoming changes, are generated using an intermediate language for mapping expressions. Our approach is part of the Vitruvius framework for view-centric engineering [16, 15], in which all model changes are recorded and in which custom views that integrate information from multiple metamodels can be defined. Altogether, we try to reduce the accidental complexity of keeping models consistent by generating change-driven transformations from consistency specifications that are defined in a language that we have created specifically for this purpose. The resulting tool prototype will show how developers can be supported in specifying and enforcing consistency with a specifications editor, transforma-

QoSA'15, May 4–8, 2015, Montréal, QC, Canada.

ACM 978-1-4503-3470-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2737182.2737194.

tion generator, and propagation engine that executes these transformations. Our approach can directly be used for any models that conform to an Ecore-based metamodel and could be adapted for other EMOF-based modeling languages.

We will evaluate our approach using two case studies from the domains of software development and automotive engineering. In both cases, the models overlap in terms of components, interfaces, and connectors. Whether our consistency language and transformation generator are also beneficial if modelling languages do not share a common paradigm, such as components, has to be evaluated in future work.

The remainder of this paper is structured as follows: In section 2, we present the involved modeling languages. In section 3, we formulate goals and research questions. In section 4, we describe our approach and discuss essential challenges. In section 5, we propose a conceptual mapping between elements of our case studies. In section 6, we present our plans for evaluation. In section 7, we discuss related work, and in section 8 we conclude and present future work.

2. BACKGROUND AND FOUNDATIONS

In this section, we provide information on the modeling languages of the case studies that we use to evaluate our consistency preservation approach and tool prototype.

2.1 Model-Driven Engineering

In Model-Driven Engineering (MDE), all design and development artefacts are derived from models with a fixed syntax: they have to conform to a metamodel in order to be processed in automated transformations. The syntactic metamodel constraints make it possible to generate source code or other artefacts from such models. The concrete representation of a model is separated from its abstract syntax in order to ease the independent development of editors and transformations. The model semantics are usually indirectly defined by the target models produced by model transformations, or the source code obtained from code generators.

2.2 Component-Based Software Development

Palladio Component Model (PCM)

The PCM is an Architectural Description Language (ADL) for component-based software systems. It provides all concepts that are necessary to model reusable components and interfaces in a system-independent repository. At this high level of abstraction, a component is only defined by its provided and required interfaces, which list signatures of services. Concrete systems are expressed with assembly contexts that instantiate components. Assembly contexts are linked at provided and required roles of interfaces using assembly connectors. They are also used in composite components, which delegate service calls internally using delegation connectors.

Java Model Printer and Parser (JaMoPP)

We use the Java Model Printer and Parser (JaMoPP) [10] to treat Java source code as ordinary instances of a Java metamodel. This makes it possible to analyze and transform Java code the same way other models are processed and relieves us from parsing and printing source code.

Java Modeling Language

Contracts for Java source code can be defined with the Java Modeling Language (JML) [18]. It supports the definition of contracts for all elements of Java interfaces and classes. They can contain statements, such as pre- and post-conditions, as well as invariants, and modifiers, for example to mark methods that have no side-effects.

Eclipse Plug-Ins

To implement reusable components that can easily be replaced or reconnected in Java source, the extension mechanism of the Eclipse IDE, which is based on plug-ins, can be used. These Eclipse plug-ins are OSGi bundles that can define explicit dependencies to other plug-ins. They may define extension points and provide extensions for them.

2.3 Automotive Engineering

EAST-ADL

EAST-ADL [3] can be used to model automotive electrical and electronic systems on five abstraction layers: A *vehicle* level describing a software product line, an *analysis* level defining abstract functions of the system, a *design* level decomposing these functions, a *hardware architecture*, and an *implementation* level referring to an AUTOSAR model.

The elements that are relevant in our case study are part of the functional model defined on the analysis level: A prototype conforms to a type in a similar way as component instances conform to component types in other ADLs. Types are linked with connectors using ports, which are first-class entities in EAST-ADL, i.e. they can exist independent of types.

Systems Modeling Language (SysML)

The Systems Modeling Language (SysML)[1] is a general-purpose modeling language of the OMG for the specification, analysis, design, verification, and validation of systems. It is aligned with the ISO 42010 standard and reuses and modifies UML diagram types and provides new ones.

For our case studies, mainly blocks, which are nested in viewpoints, which are packaged in views, are relevant. These blocks are connected using flow ports.

MATLAB Simulink

Simulink[1] is a tool for graphical modeling and programming of system simulations and analyses, which is embedded into the MATLAB environment for numerical computing. It supports blocks that pass data via ports, which are connected using lines that convey signals.

3. GOALS AND QUESTIONS

Our approach for multi-view consistency is led by the following two research goals: *G1: Classify the consistency relations that are possible between different modeling languages. G2: Ease the bidirectional preservation of consistency for models of different languages.*

We pursue these goals by answering the following two main research questions: *Q1: What kinds of consistency relations can be present between elements of modeling languages and how can they be expressed precisely? Q2: Can we keep models automatically consistent using transformations that are generated from abstract consistency specifications and that are triggered by individual changes?*

Each of these main research questions can be divided into sub questions: *Q1.1: Can we partition all possible consistency*

[1] omgsysml.org and mathworks.com/products/simulink

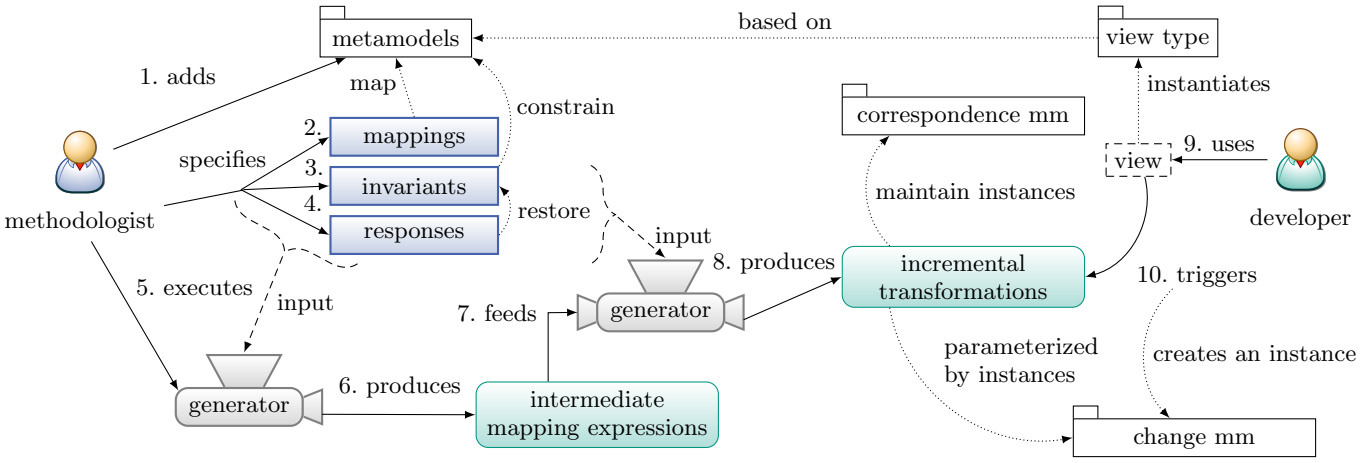

Figure 1: Defining, generating, and executing change-driven consistency preservation transformations

relations between elements of modeling languages? Q1.2: Can we formally define each relation subset of the partition so that every possible consistency relation can be formally expressed? Q1.3: Can we prove for each subset that consistency is achieved (semi-)automatically?

Q2.1: Can we specify consistency according to at least one partition subset with a language based on declarative mappings, normative invariants, and imperative responses? Q2.2: Can we remove syntactic variances in an intermediate mapping language to ease static analyses and code generation? Q2.3: Can we generate incremental consistency preservation transformations from these mappings and invariants while embedding the response snippets? Q2.4: Can we keep models consistent according to the specification by triggering these transformations after changes?

4. APPROACH AND CHALLENGES

In this section, we first explain how we monitor changes with the Vitruvius framework to keep its models consistent using our approach. Then, we present our language for consistency specifications and explain how transformations that propagate monitored changes can be generated from these specifications. Last, we discuss challenges of such a language and propagation approach and sketch how we address them.

4.1 Change-Driven Consistency

The Vitruvius framework [16] is based on the core idea of Orthographic Software Modeling (OSM) [1]: all information of a software system is represented in a Single Underlying Model (SUM) and can be accessed solely by specific views. In OSM, a special developer role called *methodologist* is responsible for building a SUM metamodel and the necessary views before a system is developed. Vitruvius, however, uses a Virtual Single Underlying Model (VSUM), which can be used like a single model, but internally consists of instances of several metamodels. It is built automatically from a so-called *meta repository*, which is created by a methodologist to manage all view types, metamodels, and their relations. To ensure compatibility with existing modeling and analysis tools, it is possible to add metamodels and views without modifications. This possibility to add project-specific metamodels is important, because no fixed meta repository (or SUM metamodel) would be suited for all software engineering scenarios. Because the inner modular structure of the VSUM is decoupled from its representation in the views, the methodologist cans reuse views, metamodels, and their relations across several projects. This decoupling makes it also easier to react to evolving languages or tools because it limits the impact of changes in a single metamodel.

All views have to monitor sequences of atomic changes and report them to the framework so that they can be propagated inside the VSUM. In general, such change information cannot be derived unambiguously from model differences and therefore has to be recorded [17]. Internally, our consistency approach uses a correspondence model to propagate every change individually to elements of other models in the VSUM. The code that is triggered by a change is obtained from an abstract consistency specification that is expressed using the language that we present in the next section.

4.2 Mappings, Invariants and Responses

To answer our research questions, we are currently developing and evaluating a domain-specific language for the specification and enforcement of consistency specifications in multi-view modeling projects. This *MIR* language contains concepts for declaratively *mapping* elements of different metamodels, normatively specifying consistency checks (*invariants*), and imperatively preserving consistency (*responses*). For every pair of metamodels with a semantic overlap that should be kept consistent, such MIR expressions can be specified by a methodologist or domain expert.

In the language and editor, the three parts for *specifying*, *checking* and *preserving* consistency are strictly separated. In the first part, declarative *mappings* specify semantic correspondences between metaclasses, their attributes, and references using conditions that restrict and compute attribute and reference values of model instances. This language part is similar to the QVT-R language, but due to the special context of change-driven propagation transformations, we do not have to support problematic features, such as pattern matching, check-only semantics, rule dependencies, direction reversal, or explicit keys. In the second part, *invariants* involving elements of a single or several metamodels can be formulated using OCL. These invariants can use the full OCL language and may expose parameters, which are bound in case of violations to be used in the third language part. In the

third part, *responses* to specific types of changes or invariant violations can be defined using the imperative transformation language Xtend. Thus, the power and expressivity of this general-purpose language can be used if the declarative and normative language constructs for synchronization mappings are not sufficient and clean-up actions or conflict resolutions shall be implemented. The three language parts also play a different role when we generate change-driven propagation transformation from specifications in a two-step process.

4.3 Generating Propagation Transformations

The process for defining, generating, and executing consistency preservation transformations using our MIR language is shown in Figure 1. The methodologist, who is responsible for the meta repository, view types, metamodels, and consistency specifications, first adds metamodels to the meta repository (1). Then, he or she specifies mappings between metamodels (2), defines invariants that constrain these metamodels (3), and optional responses that may restore these invariants (4). Afterwards, the methodologist executes a generator (5), which produces intermediate mapping expressions (6). These intermediate expressions are fed into a second generator (7) together with the invariants and responses to produce change-driven consistency preservation transformations (8). For every metaclass and feature involved in a mapping and every modification type a separate transformation is produced. If the methodologist has defined response transformation snippets, they replace or refine these generated transformations. If a developer changes an instance of a metaclasses in a view conforming to a view-type (9), these final transformations are triggered using an instance of a generic change metamodel (10). If the change has unclear effects on corresponding models, no transformation is executed and the developer is asked to manually clarify his intent. This semi-automatic step and possible manual rollbacks after invariant violations are not shown in Figure 1.

To simplify static analyses of mappings and ease the generation of executable transformations, we are developing a language for intermediate mapping expressions. These expressions are used to eliminate syntactic variants for semantically equivalent mappings. In contrast to the mappings defined by domain-experts, the generated intermediate mapping expressions have no textual syntax, contain references to pre- and post-conditions instead of nested mappings, and explicitly list metaclass mappings that result indirectly from reference mappings. We generate separate mapping expressions for different change origins and propagation directions, but we do not distinguish change types (create, update, or delete). This distinction is only made when the final, executable transformation code is generated together with a dispatcher that reflects the effects of metaclass inheritance on mappings.

4.4 Language and Propagation Challenges

To illustrate the fundamental problems addressed by our approach, we discuss three central challenges that we address in our work. Other challenges, such as the semi-automated resolution of inconsistencies that are not removed by our generated transformations, will be addressed by adapting existing solutions that are used in different contexts.

Language Challenge: Bidirection without Bijection

In order to be applicable in many scenarios, it is essential for our MIR language to keep elements consistent in a bidirectional way in cases where no bijective value mapping was provided. This can be achieved, for example, by defining two opposing injective functions that are inverse for the intersection of domain and codomain. Another way to accomplish bidirectional consistency without a bijective mapping is to require the definition of consistency predicates so that consistency can be sustained manually when a predicate violation was detected automatically. We are exploring such possibilities with suitable constructs in our MIR language.

Language Challenge: State-Based Propagation

For some metamodels concepts and relations, it can be more convenient to define consistency preservation in a state-based manner instead of delta-based response actions. For method contract reevaluations after a change of an arbitrary method body element, for example, detailed change information is not needed because the method body has also to be verified completely. Therefore, we are evaluating how we can integrate mechanisms for state-based propagation into our change-driven language in such a way that the correct propagation mechanism can always be selected unambiguously.

Propagation Challenge: Rollback / Internal History

If a change leads to unknown or undesired effects during consistency preservation, we have to revert all primary effects of the user and all secondary effects of our transformations. In this context propagations of change propagations are particularly challenging: A change in one model indirectly leads to a change in another model, which is carried out by a consistency preservation transformation, but is in turn propagated to a third model just like an ordinary change by a user. To address this problem, we have to store nested rollback information or separate versions of all models.

5. MAPPING CASE STUDIES

In the following, we sketch the conceptual mapping between the languages of a component-based software development case study and an automotive engineering case study. Background information on the mapped languages and domains is provided in subsection 2.2 and 2.3.

5.1 Mapping Software Components

The mapping for software components that are developed with the Palladio Component Model, Java, and Eclipse plugins is shown in Table 1 and based on [14]. It does not support other component implementation strategies, such as the broker pattern, but it can be expressed very briefly with our MIR language so that we expect to be able to generate most transformation code. The mapping for Java and JML contracts is fixed by the language specification and not shown.

5.2 Mapping Automotive Models

For our automotive case study, we define mappings for SysML and Simulink as well as for Simulink and EAST-ADL. In Listing 1 and 2, we present some exemplary MIR listings for the central metaclasses of these metamodels: view, block, model, and port. These mappings are not sufficient to keep SysML, Simulink and EAST-ADL models consistent, but they cover the most important metaclasses, features, and relations to illustrate our mapping language.

6. EVALUATION

PCM	Java	Eclipse
repository	main, contracts, and datatypes package	main plug-in
component	subpackage of main package with public facade class	plug-in
interface	interface in main package	extension point
service signature	method signature	–
datatype	class with getters and setters for inner types in datatypes package	–
required role	member and setter for required interface in component facade class	plug-in dependency
provided role	component facade class implements provided interface	extension

Table 1: Mapping between architectural model elements (PCM) and source code elements (Java)

We will answer our research questions using the two case studies presented above. The models of both case studies were created independently, but as there are no predefined change sequences, we have to define them to evaluate the consistency preservation transformations. Similarly, we have to define the mappings, invariants, and responses, from which these transformations will be generated, based on existing, manually implemented transformations [15]. Using the first case study, we will show whether it is possible to keep component models and their implementation consistent using our approach. With the second case study, we will demonstrate whether the overlapping elements of different architectural models of the automotive domain can also be kept consistent this way. Both case studies help us in reaching *G1*.

In order to assess some of the benefits of our approach with respect to *G2*, we will conduct a quasi-experiment with graduate students and professional engineers. They will design, implement, and evolve component-based software and a) use a general-purpose transformation language, or b) the languages and generators of our approach to keep development artefacts consistent. We will measure the time spent on consistency keeping and count the number of inconsistencies, but we will not be able to control all influencing variables.

7. RELATED WORK

To support different views and modeling languages, various approaches for multi-view modeling have been presented. In recent multi-viewpoint modeling approaches, correspondences are specified between viewpoints to sustain consistency. Some implementations use declarative logical languages [7] or UML in conjunction with QVT-R [20]. Others use coupled transformations that exploit explicit correspondence links [23] to propagate changes to dependent viewpoints. In contrast to our approach, the transformations are defined directly between the view types with *viewpoint languages* not between metamodels. For such synthetic approaches, the link complexity grows exponentially with the number of views. For Vitruvius, however, the complexity only grows exponentially in the number of metamodels if every metamodel has an overlap with every other metamodel.

```
map sysML.View and smlnk.Model as model {
 with name[String] and name[String]
 with packagedElements[Viewpoint] and model {
  when { name = name }
  with nestedClassifier[Block] and subBlocks[Block];
} }
map sysML.Block and smlnk.Block {
 with name[String] and name[String]
 with ownedAttributes[FlowPort] as flowPort
  and inPorts[InPort] {
  when { flowPort.direction = FlowDirection.in }
  with name[String] to name[String]
 }
 with ownedAttributes[FlowPort] as flowPort
  and outPorts[OutPort] {
  when { flowPort.direction = FlowDirection.out }
  with name[String] to name[String]
} }
```

Listing 1: Extract of an exemplary consistency mapping for concepts of SysML and Simulink

Projective approaches [1, 2, 7] try to avoid this complexity by projecting all views from a central model. Such a central model limits the expressive power that is available in views and it has to be designed upfront. This can make it difficult to support and evolve existing metamodels and views or to add new ones.

Many multi-view modeling tools, e.g., ModelBus [11] or Sphinx [5], provide advanced notification and merging features but lack concepts for expressing semantic consistency.

Several transformation approaches based on Triple-Graph Grammars (TGGs) support incremental consistency preservation [19]. Some of them have been applied to SysML and AUTOSAR models in the automotive domain [9]. 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. This is similar to the mappings of our approach, but in many cases TGGs are restricted to monotonic productions that cannot delete elements [21]. Furthermore, many TGG-based approaches are limited to equivalence relations for attributes and do not support bidirectional calculations of attribute values.

Various other approaches use mapping languages [22], ensure consistency based on change observation [6], demand properties for consistency transformations [12], or describe

```
map eastadl.FunctionType and smlnk.Block {
 with port[FunctionFlowPort] and inPorts[InPortBlock] {
  when { port.direction = EADirectionKind.in
   || port.direction = EADirectionKind.inout }
 }
 with port[FunctionFlowPort] and outPorts[OutPortBlock] {
  when { port.direction = EADirectionKind.out
   || port.direction = EADirectionKind.inout }
} }
```

Listing 2: Extract of an exemplary consistency mapping for concepts of Simulink and EAST-ADL

theoretical foundations for consistency [4] based on lenses [8] and category theory. We are, however, unaware of any related work in which views and consistency preserving transformations are generated from abstract mapping specifications in an automated manner.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a refined research proposal for generating change-driven consistency preservation transformations from mappings, invariants, and responses written with a consistency specification language. We formulated research goals and questions centered on a classification of consistency relations and sketched our method for enforcing those relations that can be expressed with our language. We illustrated case studies of component-based engineering in the software development and automotive domain and we presented our evaluation plans. In future work, we will finish the development of a prototype featuring an editor and a customizable generator for our consistency language.

Acknowledgments

This work was partially funded in the KASTEL project by the German Federal Ministry of Education and Research.

I am grateful to R. Reussner, E. Burger, M. Langhammer, and D. Werle for their fruitful comments and great ideas.

References

[1] C. Atkinson et al. “Orthographic Software Modeling: A Practical Approach to View-Based Development.” In: *Evaluation of Novel Approaches to Software Engineering*. Vol. 69. Communications in Computer and Information Science. Springer, 2010, pp. 206–219.

[2] A. Cicchetti et al. “A hybrid approach for multi-view modeling.” In: *Electronic Communications of the EASST* 50 (2011).

[3] P. Cuenot et al. “The EAST-ADL Architecture Description Language for Automotive Embedded Software.” In: *Model-Based Engineering of Embedded Real-Time Systems*. Vol. 6100. LNCS. Springer Berlin Heidelberg, 2010, pp. 297–307.

[4] Z. Diskin. “Model Synchronization: Mappings, Tiles, and Categories.” In: *Generative and Transformational Techniques in Software Engineering III*. Vol. 6491. LNCS. Springer Berlin / Heidelberg, 2011, pp. 92–165.

[5] S. Eberle. “Using Sphinx to create multi-language multi-view DSL tool environments.” talk. 2012.

[6] A. Egyed. “Fixing Inconsistencies in UML Design Models.” In: *Software Engineering, 29th International Conference on*. 2007, pp. 292–301.

[7] R. 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.

[8] J. N. Foster et al. “Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.3 (2007).

[9] H. Giese et al. “Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent.” In: *Graph Transformations and Model-Driven Engineering*. Vol. 5765. LNCS. Springer Berlin / Heidelberg, 2010, pp. 555–579.

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

[11] C. Hein et al. “Model-Driven Tool Integration with ModelBus.” In: *Workshop Future Trends of Model-Driven Development*. 2009.

[12] I. Ivkovic and K. Kontogiannis. “Tracing evolution changes of software artifacts through model synchronization.” In: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. 2004, pp. 252–261.

[13] M. E. Kramer. “Synchronizing Heterogeneous Models in a View-Centric Engineering Approach.” In: vol. 227. GI Lecture Notes in Informatics. Doctoral Symposium. GI e.V., 2014, pp. 233–236.

[14] M. E. Kramer and M. Langhammer. “Proposal for a Multi-View Modelling Case Study: Component-Based Software Engineering with UML, Plug-ins, and Java.” In: VAO ’14. ACM, 2014, 7:7–7:10.

[15] M. E. Kramer et al. *Realizing Change-Driven Consistency for Component Code, Architectural Models, and Contracts in Vitruvius*. Tech. rep. 2015.

[16] M. E. Kramer et al. “View-centric engineering with synchronized heterogeneous models.” In: VAO ’13. ACM, 2013, 5:1–5:6.

[17] M. Langhammer and M. E. Kramer. “Determining the Intent of Code Changes to Sustain Attached Model Information During Code Evolution.” In: vol. 34 (2). Softwaretechnik-Trends. GI e.V., 2014.

[18] G. T. Leavens et al. “JML: A Notation for Detailed Design.” In: *Behavioral Specifications of Businesses and Systems*. Vol. 523. The Springer International Series in Engineering and Computer Science. Springer US, 1999, pp. 175–188.

[19] E. Leblebici et al. “A Comparison of Incremental Triple Graph Grammar Tools.” In: *Electronic Communications of the EASST* 67 (2014).

[20] J. R. Romero et al. “Realizing Correspondences in Multi-viewpoint Specifications.” In: *Enterprise Distributed Object Computing Conference, 2009. EDOC ’09. IEEE International*. 2009, pp. 163–172.

[21] A. Schürr. “Specification of graph translators with triple graph grammars.” In: *Graph-Theoretic Concepts in Computer Science*. Vol. 903. LNCS. Springer Berlin Heidelberg, 1995, pp. 151–163.

[22] M. Verhoef et al. “A multi-paradigm mapping method survey.” In: *Workshop on Modeling of Buildings through their Life-cycle*. 1995, pp. 233–247.

[23] M. Wimmer et al. “Viewpoint Co-evolution through Coarse-Grained Changes and Coupled Transformations.” In: *Objects, Models, Components, Patterns*. Vol. 7304. LNCS. Springer Berlin Heidelberg, 2012, pp. 336–352.