Traceability enabled by Metamodel Integration

Johannes Meier, Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{meier,winter}@se.uni-oldenburg.de

Abstract

Traceability supports software development by connecting separated software artifacts explicitly with each other using traceability links. While intra traceability within single artifacts is solvable by a single metamodel, for inter traceability between different separated artifacts their metamodels have to be related to each other. Therefore, this paper presents an approach to realize inter traceability by integrating the metamodels of the artifacts into one metamodel together with their traceability information. The approach is demonstrated in a small software project using requirements, class diagrams, and source code.

1 Motivation

Traceability is "the ability to describe and follow the life of a software artifact" [1, 2, 3]. After "modeling the relations between software artifacts in an explicit way" [3], these relations between software artifacts are called *traceability links*. Traceability links can be exploited among others for discussing with stakeholders, for documentation, and for propagating changes from on to another related artifact to ensure consistency. Therefore, traceability is important in software projects.

As example, in a simplified software development project, there are requirements, class diagrams, and source code as artifacts. The requirements are documented as list of textual requirements, class diagrams model the data structures, and Java source code forms the final software product. In this project, there are two main scenarios for traceability:

1. Each requirement should be fulfilled by a method in the source code. Since this relation often exists only implicitly, traceability links should explicitly link each requirement with its fulfilling method. As benefit, validation checks are possible, e.g. whether each requirement is linked with at least one method.
2. The required data structures are modeled with class diagrams and implemented with Java at the same time in two separated artifacts. Here, traceability has to ensure the consistency between classes within class diagrams and the corresponding classes within the source code.

This simplified example will be used for motivation and application throughout this paper.

[4] distinguishes between intra (called horizontal in [4]) and inter (vertical) traceability. *Intra traceability* describes traceability of software artifacts within the same level of abstraction or development phase. An example for intra traceability are methods called by other methods, which are all defined within Java source code. *Inter traceability* describes traceability between software artifacts of different type, abstraction level, or development phase. The introduced traceability scenarios are examples for inter traceability, like source code which fulfills requirements.

Intra traceability can be realized by describing the affected artifact by a model which conforms to a metamodel. If the metamodel provides corresponding references between the meta-classes, traceability links can be added on model level. As simplified example in Figure 1, a Java metamodel like [5] (an alternative is [6]) represents declarations of methods by the meta-class `MethodDeclaration` and the use of methods by `MethodInvocation`. The traceability between

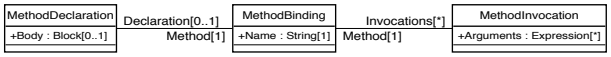

Figure 1: Intra Traceability within in a Java Metamodel

declaration and use is realized by `MethodBinding` and the connecting references. Navigation and querying for declarations and invocations of given methods is enabled by such metamodels and intra traceability. As another example, for renaming a method (`MethodBinding.Name`), the new name has to be used by related declaration and invocations. Therefore, traceability is needed for `MethodDeclaration` and `MethodInvocation` to retrieve its name from the linked `MethodBinding`.

In the case of *inter traceability* between two different artifacts like requirements and source code, this solution is not possible with only two separated metamodels for each artifact. With only two separate metamodels available, no additional meta-class or relation can be added between them. Therefore, the independent metamodels have to be related to each other. Section 3 presents the idea of metamodel integration [7], which takes existing separated metamodels and constructs one single, integrated metamodel containing the content of all separated metamodels. As goal, this paper shows, that metamodel integration enables traceability between different artifacts.

2 Related Work

This Section classifies existing traceability approaches, and points to important challenges for traceability approaches. The fulfillment of these challenges by the new approach is discussed along its description in Section 3 and its application in Section 4.

The related work regarding traceability can be devided into two main groups [1]: The first group describes traceability of elements which are passed and transformed through *model transformation chains*, which are often used in Model-Driven Development (MDD). Traceability links can be computed by analyzing the applied transformation rules, or by recording them during the transformation execution.

Since step-wise transformations of artifacts into other artifacts are not used in software development projects like the ongoing example, this paper focuses on the second group, which describes *traceability between different existing and evolving artifacts*. Since these artifacts are isolated by physically separated files, or different tools, traceability between these heterogeneous artifacts is required [8].

For *storing identified traceability links*, there exist two main approaches [9]. The first approach saves traceability links *within the existing models* as additional model elements. Because this embedding and in-place-storage also changes the corresponding metamodel, existing tools using the metamodel might not work after this embedding. This would require effort for the adaptation of existing tools.

Challenge 1 requires further use of the existing and unchanged, separated models and metamodels. As comparable approach with an integrated metamodel, [10] ignores the initial artifacts.

On the other hand, in-place-storing supports readability for humans [11], and querying of model elements with their intra traceability. As benefit for the user, also inter traceability links have to be presented together with their traced elements.

Challenge 2 requires the easy visualization and query of intra and inter traceability and its traced content. Because in-place-storing can save only intra traceability links, it does not fit here.

The second approach stores all traceability links *externally within a new traceability model* conforming to a *traceability metamodel*. This additional traceability model stores only the inter traceability links which refer to the traced elements, which are stored in the existing concrete models. To realize connections between traceability links and their traced elements over model boundaries, unique and stable identifiers are required for the resolution of the traced model elements [9]. Because of this technical disadvantage,

Challenge 3 avoids stable and unique identifiers for each model element. Approaches creating an integrated traceability metamodel on-demand like [9], and [11] require stable and unique identifiers.

Two general kinds exist how to represent the required traceability links [11]: The first kind allows traceability links *unlimited between all types of elements* of all models. Project specific limitations, e.g. requirements are traceable to methods, but not to single statements, can only be realized by manually added constraints limiting the connected types, or multiplicities. Examples for those approaches can be found in [12], and on reference level in [10].

The second kind comprises traceability metamodels which are manually created for a specific project and purpose. Advantage is the possibility of strongly typed traceability links with project specific definitions, while the creation of these specific traceability metamodels takes more effort [11]. Possible traceability links of existing approaches differ regarding type (untyped vs. only a small number of pre-defined types like `Dependency` or `Refinement` vs. arbitrary user-defined types), hierarchy of allowed types, multiplicities, allowed kinds of constraints, and some more [1]. To minimize required constraints and to maximize the degree of adaptation to project specific needs,

Challenge 4 requires traceability links with user-defined types and multiplicities. In this way, the approach of this paper can be used for the traceability of arbitrary artifacts and project settings.

3 Metamodel Integration

Central idea of this approach is to keep the existing artifacts in form of their models, conforming to metamodels, unchanged and independent to overcome Challenge 1. These existing metamodels (models) will be called *concrete metamodels* (concrete models) within this paper. The traceability links will be stored externally within a new metamodel.

In contrast to most existing approaches which use dedicated traceability metamodels only for traceability content, this new metamodel contains the content of all concrete metamodels together with meta-classes and references describing traceability. Therefore, this new metamodel is one Single Underlying MetaModel (SUMM) [13], which integrates all artifacts and their traceability by their metamodels. The same counts for the model level, which forms one Single Underlying Model (SUM) [13] containing the concrete models of each artifact together with traceability links. All existing artifacts are connected with each other by their traceability. In contrast to [14], SUMM and SUM are permanently persisted. Visualizations and queries for the traceability together with the traced entities are straightforward (Challenge 2), because they work on one integrated metamodel and one integrated model.

How to create such SUMM out of existing concrete metamodels is drafted in [7]: Starting with one concrete metamodels, special operators are applied in step-wise way to the current metamodel. Each operator improves the existing metamodel by a small change, like adding a new relation (AddRelation) between two classes to represent their traceability.

3.1 Traceability Scenarios

Before defining these operators in Section 3.2 and listing available operators in Section 3.3, this Section shows how to apply these operators regarding traceability of the ongoing example project.

Initially, the concrete metamodels for requirements, class diagrams, and source code are independent. Therefore, these metamodels are integrated into the SUMM using the operator IntegrateMetamodel, which copies all three metamodels into the SUMM and all three models into the SUM. Now, the different artifacts are connected on technical level, but still without traceability connections to each other.

To add traceability links between methods in the source code and its fulfilling requirements on model level (Scenario 1 in Section 1), on metamodel level, a new reference between the meta-classes `MethodBinding` and `Requirement` is required. Because this reference is currently missing in the SUMM, it has to be added. For that, the operator AddRelation adds a new relation between the two classes in the SUMM (metamodel change). Now on content level, traceability links can be added and stored. To compute some initial traceability links for existing methods and requirements (model change), project-specific decision logic is required to determine these traceability links (additional decisions). Because this new relation is named and has a target type together with multiplicities, Challenge 4 is fulfilled. Because the complete content of all concrete models is handled together with the traceability links within one SUM, no unique and stable identifiers are required, because links between objects are directly connected and stored (Challenge 3). Compared with the related work (Section 2), *inter traceability issues are solved by intra traceability within the SUM.*

Scenario 2 in Section 1 is related to the traceability of classes in class diagrams and source code. Traceability is required, because the same class can be available as `KClass` in class diagrams and as `TypeBinding` in source code, which leads to inconsistency. As example, renaming of a class leads to changing two names. Therefore, on model level, instances with the same name are the same class, and should be related to each other. Instead of adding traceability links between the two meta-classes, this approach allows to remove this duplicated information by unification. Therefore, an operator called MergeTwoClasses will be applied to these two meta-classes which leads to one meta-class representing both classes from source code and classes from class diagrams. On model level, this leads to removed duplicated objects. Hence, *traceability of duplicates is realized by elimination instead of linking.*

3.2 Properties of Operators

As indicated in Section 3.1, each operator has the following properties:

Metamodel change The operator changes some (often small) part of the current metamodel, e.g. to add a new reference for traceability.

Model change Because of the change in the metamodel, corresponding changes in the models are required to keep models and metamodel consistency. Therefore, each operator is a coupled operator introduced by [15], coupling the metamodel change with changes in the model to keep it consistent to the changed metamodel.

Additional decisions Some operators require additional decisions, how changes on the model level should be handled. Adding a new relation on metamodel level (AddRelation) for traceability leads to the question, which traceability links on model level should be added. Because this decision depends on the specific needs for each project, the operator can be parameterized by individual code for this decision.

Bi-directionality of operators is achieved by combining each operator with an inverse operator, so that applying the forward operator and then the backward operator will lead to the initial metamodel. The inverse operator for AddRelation is RemoveRelation, which removes an existing relation on metamodel level together with all corresponding links on model level. This property is required to support the transformation of concrete models into a SUM (forward direction), and to divide a SUM into the concrete models (backward direction) for Challenge 1, which will be discussed in more detail in Section 4.4.

3.3 Operators

The following operators are usable for the integration of metamodels regarding traceability scenarios:

IntegrateMetamodel adds an existing metamodel to the current metamodel by copying all classes, relations, and generalizations. On model level, the external model is copied into the existing model. Therefore, SUMM and SUM are created by adding external (meta)models to the existing (meta)model. Additional decisions are not required. The bidirectionality is reached by deleting all copied classes on metamodel level and all their instances on model level.

AddRelation adds a new relation between two existing classes on metamodel level. On model level, new links will be added between existing objects, corresponding to the deposited decision. The inverse operator is RemoveRelation, which removes an existing relation on metamodel level together with all corresponding links on model level (skipped here).

MergeTwoClasses into one class is required to unify information which is available twice. On metamodel level, two classes are replaced by one class which gets all attributes and references of the two source classes. A decision is required to determine the pairs of objects which should be re-

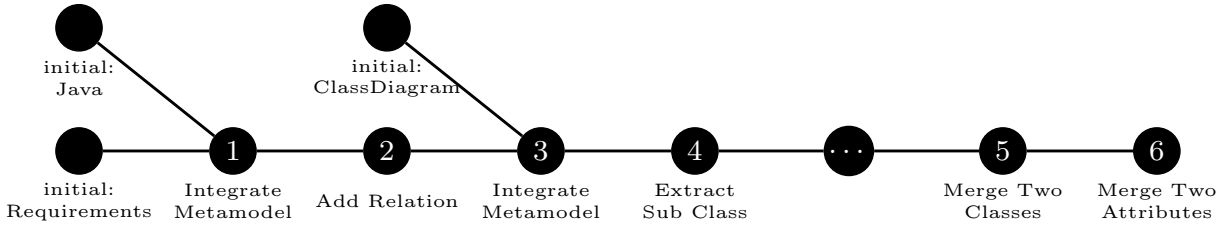

Figure 2: Applied Operators for the Traceability scenarios of the example Software development project

placed by one object on model level. The inverse operator is SplitClass (skipped here).

MergeTwoAttributes into one attribute replaces two attributes within the same class by one attribute. On model level, a decision determines the value of the new attribute given the values of the old attributes. The inverse operator is SplitAttribute (skipped here).

There are operators supporting the metamodel integration, but not specific for traceability issues:

ExtractSubClass creates a new sub class in the metamodel. Depending on a decision, the type of existing instances of the old super class will be changed to the new sub class. The inverse operator is InlineSubClass (skipped here).

4 Application

This Section applies the presented approach for metamodel integration of Section 3 to the ongoing example. The application is implemented using Java, reusing some coupled operators [15], and parts of the model migration infrastructure from the EDapt project (`https://www.eclipse.org/edapt/`).

ECore [16] was chosen as language to describe all metamodels. As precondition, each artifact has to be available as model conforming to a metamodel: The textual *requirements* are managed in CSV tables (Figure 3, top), from which the metamodel in Figure 3 (bottom) was derived. For modeling class diagrams,

ID	Author	Text
rq-1	Johannes Meier	The system has to support universities, students, and lectures.
rq-2	Andreas Winter	The student must be able to register for an event.
rq-4	Johannes Meier	A lecture must be able to be published.

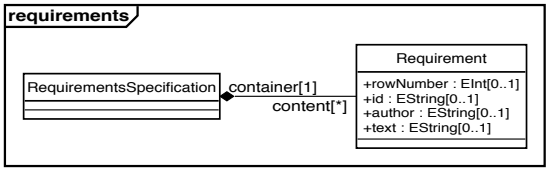

Figure 3: Requirements: View (top), Metamodel (bottom)

a collaborative editor for UML class diagrams [17] was used. Java source code is represented using a metamodel, which bases on transformations from the Eclipse JDT to ECore [5].

For showing how to deal with traceability in this example, four main activities in traceability approaches, slightly adapted from [1], will be shown: Representation (4.1), Identification (4.2), Use (4.3), and Maintenance (4.4) of traceability links.

4.1 Representation

During this activity, the project setup regarding traceability issues defines, in particular, how to represent and store traceability links. Following the metamodel integration approach of Section 3, the operators depicted in Figure 2 will be applied to get a SUMM supporting the required traceability scenarios: The integration is started with the metamodel for requirements as initial metamodel. After loading the metamodel for Java, the operator IntegrateMetamodel ❶ integrates it on technical level into the Java metamodel by copy. The model containing the Java source code is copied into the model already containing all requirements. The current SUMM contains now elements to describe Java and requirements, while the current SUM contains all available requirements next to the complete Java source code.

The operator AddRelation ❷ solves the traceability Scenario 1 by adding a new reference between the meta-classes `Requirement` and `MethodBinding` (Challenge 3+4), shown in Figure 5. On model level, traceability links can be added between requirements and methods. The new traceability links, roughly depicted in Figure 4, are created by decision logic, which links requirements to those `MethodBinding`s, whose `Name` is contained in the `Requirement`'s `text`. Requirements and methods are integrated and traceable now.

Using the operator IntegrateMetamodel ❸, the metamodel to describe class diagrams and the model containing a class diagram are integrated. As result, the current SUMM contains concepts for Java, requirements, and class diagrams, while the current SUM contains the available Java source code, the existing requirements, and a class diagram. All three artifacts are now part of one artifact. While Java and requirements are already integrated, the class diagram is still without traceability links.

As preparation for its integration, the operator ExtractSubClass ❹ specializes the class `TypeBinding` into `ClassTypeBinding`. Because `TypeBindings` represent also enums, primitive types, `null` and so on, instances which represent classes are migrated to the new sub class. Parts of the resulting SUMM are depicted in Figure 6. The new class with its generalization is marked in gray. For brevity, some more used operators are skipped in this paper.

Now the traceability Scenario 2 regarding the consistency issues of `KClasse`s and `ClassTypeBinding`s is addressed by the operator MergeTwoClasses ❺,

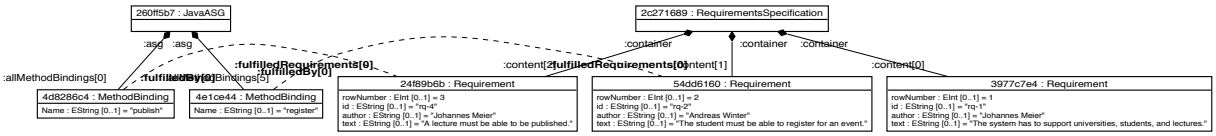

Figure 4: Model after applying Operator AddRelation ❷ with new traceability links (dashed links)

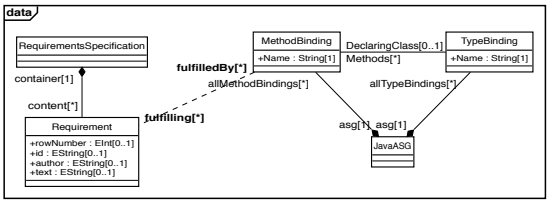

Figure 5: Integrated Metamodel after Operator ❷

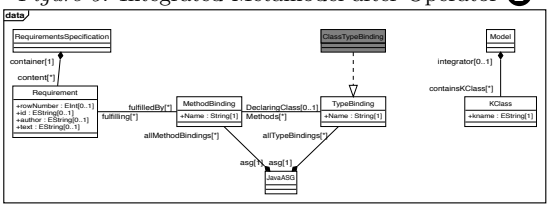

Figure 6: Integrated Metamodel after Operator ❹

which moves all features and generalizations of `KClass` to `ClassTypeBinding`, and removes `KClass` afterwards. As result in Figure 7, `ClassTypeBinding`s

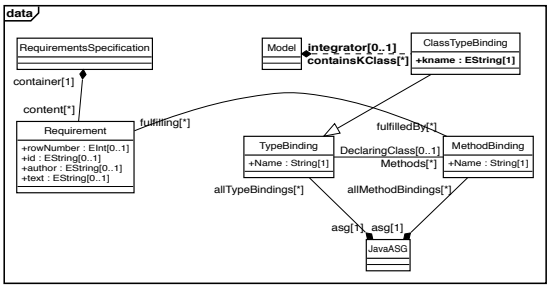

Figure 7: Integrated Metamodel after Operator ❺

can now be contained by `Model`s, too. The changes on model level are depicted in Figure 9, in which the top line represents the model before executing the operator, and the bottom line shows the result afterwards. Each instance of `KClass` became an instance of `ClassTypeBinding` and get slots for the new features. The decision logic to determine which `KClass` instance should be merged into which `ClassTypeBinding` checks, whether the values of the attributes `kname` and `Name` are the same. As final result, each instance of `ClassTypeBinding` contains all information of both initial instances. The traceability is solved by merging two related instances into one instance which saves one traceability link. The problem of inconsistency because of duplicated content is solved by elimination of the duplication and ensuring a single point-of-truth. In the final step (shortened for this paper), the two attributes `Name` and `kname` representing the same content are merged into one attribute using the operator MergeTwoAttributes ❻. Now, renaming of a class is done by changing only its `Name` value without inconsistencies.

As result of the representation activity, a chain of operators is defined. The execution of this operator chain leads to the SUM conforming to the SUMM, while traceability information is part of the SUM itself. Comparing with the related work (Section 2), the traceability metamodel is part of the SUMM.

4.2 Identification

The identification of traceability can be done during the execution of operators as well as afterwards using the SUM. During the execution of operators, decisions can be used to handle the traceability on model level. As example, the AddRelation operator allows to specify, which traceability links should be added (Section 4.1). This case is important to create traceability links for already existing artifacts.

After creating the SUM, traceability links, e.g. between requirements and methods, can be added directly to the SUM, because all existing methods and requirements are together available in the SUM. To

ID	Text	Fulfilling Methods
rq-1	The system has to support universities, students, and lectures.	
rq-2	The student must be able to register for an event.	Student.register()
rq-4	A lecture must be able to be published.	Lecture.publish()

Figure 8: New Viewpoint for Traceability Management

ease the manipulation of the huge SUM, new viewpoints can be defined on top of the SUMM, showing the user views like in Figure 8. This case should be used since SUMM and SUM were set up.

In both cases, arbitrary methods and techniques for the identification of traceability links can be used. Because all available information are together in the SUM, this identification is eased.

4.3 Use

Using the identified traceability links is possible by analyzing the SUM. Because all information of all artifacts of the project, including traceability links with all traced elements, is integrated within one model conforming to one metamodel, existing modeling techniques and tools like querying [18] and versioning can be reused for traceability. In the same way, arbitrary visualizations can be defined using the SUM (Challenge 2). Another case is the creation of new viewpoints on top of the SUMM to show only selected parts of the SUM. Figure 8 contains an example.

4.4 Maintenance

The *maintenance regarding updates of the traceability links* is realized by updating, adding, or deleting them using the SUM, in the same way like described in the identification activity in Section 4.2.

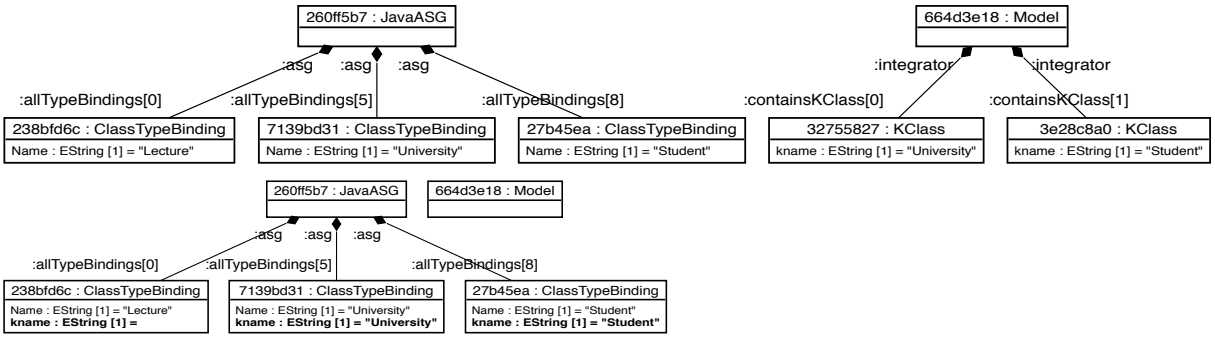

Figure 9: Model before (1st line) and after (2nd line) applying Operator MergeTwoClasses ❺ with new slots (in bold)

For the example of classes represented in class diagrams and source code, the maintenance is solved directly and without additional effort, because the SUMM contains classes only once. Changing names of classes can not lead to inconsistency, because there is no duplicated information anymore.

The *maintenance regarding updates of the initial artifacts* is solved by the design of the used operators as bi-directional (Section 3.2). While Section 4.1 focuses on integration of separated artifacts into the SUM (forward direction), each operator can be executed also in backward direction. Therefore, the current SUM will be taken and the defined chain of operators will be executed the other way round. As result, the SUM will be divided into the separated artifacts again. But the re-created single artifacts contain all the updates executed on the SUM. Therefore, changes in the SUM are submitted into the initial artifacts, which are kept up-to-date to overcome Challenge 1.

5 Conclusion

This paper shows how the integration of metamodels overcomes traceability challenges. Along related work, the integration of all software artifacts with their traceability into one Single Underlying MetaModel (SUMM) was motivated. To create such SUMMs supporting the traceability between separated artifacts, chains of operators combining changes on metamodel and model level with additional decision logic and bi-directionality were introduced. The approach was demonstrated for a software project using requirements, class diagrams, and source code.

As summary, for adding *additional traceability links*, operators like AddRelation are useful. For handling the consistency of *duplicated content*, duplications are reduced by operators like MergeTwoClasses. The elimination of duplicated content prevents inconsistency, introduces a single point-of-truth, and reduces traceability links. In the end, traceability information is stored and updated together with the information of all artifacts in an integrated manner.

References

[1] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software and Systems Modeling*, vol. 9, no. 4, 2010.

[2] O. Gotel and C. Finkelstein, "An analysis of the requirements traceability problem," in *Int. Conference on Requirements Engineering*. IEEE Comput. Soc., 1994.

[3] P. Lago, H. Muccini, and H. van Vliet, "A scoped approach to traceability management," *Journal of Systems and Software*, vol. 82, no. 1, pp. 168–182, 2009.

[4] B. Ramesh and M. Edwards, "Issues in the development of a requirements traceability model," *IEEE Int. Symposium on Requirements Engineering*, vol. 27, no. 1, 1993.

[5] A. Meyer, "A Framework for Abstract Semantic Graphs," Bachelor Thesis, University of Oldenburg, 2016.

[6] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the Gap between Modelling and Java," *Software Language Engineering*, LNCS 5969, pp. 374–383, 2009.

[7] J. Meier and A. Winter, "Towards Metamodel Integration Using Reference Metamodels," *VAO 2016*, 2016.

[8] I. Galvão and A. Goknil, "Survey of traceability approaches in model-driven engineering," *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pp. 313–324, 2007.

[9] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "On-Demand Merging of Traceability Links with Models," *ECMDA 2006 - 3rd European Conference on Model-Driven Architecture: Traceability Workshop*, 2006.

[10] H. Schwarz, J. Ebert, and A. Winter, "Graph-based traceability: a comprehensive approach," *Software & Systems Modeling*, vol. 9, no. 4, pp. 473–492, 2010.

[11] N. Drivalos, D. S. Kolovos, R. F. Paige, K. J. Fernandes, "Engineering a DSL for Software Traceability," *Software Language Engineering*, vol. LNCS 5452, 2009.

[12] A. Espinoza Limón and J. Garbajosa Sopeña, "The Need for a Unifying Traceability Scheme," *ECMDA Traceability Workshop*, 2005.

[13] C. Atkinson, D. Stoll, and P. Bostan, "Supporting View-Based Development through Orthographic Software Modeling," *Evaluation of Novel Approaches to Software Engineering (ENASE)*, pp. 71–86, 2009.

[14] M. E. Kramer, E. Burger, and M. Langhammer, "View-centric engineering with synchronized heterogeneous models," *VAO 2013*, 2013.

[15] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, *An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models*, *Software Language Engineering*, LNCS 6563, 2011.

[16] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, Second Edition. Boston: Addison Wesley, 2009.

[17] D. Kuryazov and A. Winter, *Collaborative Modeling Empowered By Modeling Deltas*, ACM SIGWEB Int. Symposium on Document Engineering, ACM, 2015.

[18] B. Kullbach and A. Winter, "Querying as an enabling technology in software reengineering," in *Eu. Conf. on Software Maintenance and Reengineering*, IEEE, 1999.