Commonalities for Preserving Consistency of Multiple Models

Heiko Klare, Joshua Gleitze
Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
klare@kit.edu, joshua.gleitze@student.kit.edu

***Abstract*—Models are used to describe different properties of a software system. Those models often share information that is represented redundantly and, thus, has to be kept consistent. Defining model transformations between the involved metamodels is a common means to preserve the consistency of their instances. Such transformations specify the relations between instances of metamodels and how to enforce them. However, redundancies are often caused by different models containing representations of the same concept. We propose to make such common, duplicated concepts explicit instead of encoding them in transformations implicitly. We achieve this by defining an additional *concept metamodel* and the relations between it and the existing metamodels, which we call the *Commonalities* approach. We describe a language that allows to define both a concept metamodel and its relations to existing metamodels in one place, in order to achieve conciseness comparable to a direct transformation between the metamodels. Additionally, our approach allows hierarchical composition of concept metamodels to keep multiple models consistent. The expected benefits of our approach are an *improved understandability* of relations between metamodels by making the information about commonalities explicit, *reduced errors* in comparison to the combination of several transformations to keep multiple models consistent, and *improved reusability* because metamodels are not related directly, but only through concept metamodels.**

I. Introduction

Modern software-intensive systems are usually described by different models, such as code, architecture, deployment specifications and other role- or concern-specific models. Since all these models describe the same system, but focus on specific properties or use different levels of abstraction, they typically share information that is represented in the models redundantly, or at least induces dependencies between them. Such redundancies have to be kept consistent to achieve a contradiction-free specification of the system.

In practice, redundancies are often kept consistent manually [7]. A common means to automate consistency preservation are *incremental model transformations*. Such transformations define how an instance of one metamodel has to be updated to restore consistency whenever an instance of another was modified. A transformation can be declared in different ways: it may either specify what has to be changed to restore consistency (*imperative*), or only the consistency constraints that have to hold, from which the rules to restore consistency are derived automatically (*declarative*). But no matter how a transformation is defined, it specifies a *relation* between two metamodels. However, redundant elements are representations of a *common concept* rather than independent elements that have to be directly related. In consequence, we think that it is natural to make the common concept, which the redundant elements are supposed to describe, explicit. We can then specify how this concept manifests itself in the different metamodels, instead of defining directly how the redundant elements are related. For example, it appears to be more natural to say that classes in UML and Java are different manifestations of the concept of a class in object-oriented design, rather than saying that a UML class should be related to a corresponding Java class. Such a common concept is what we call a *Commonality.*

In this paper, we present the *Commonalities approach.* It defines Commonalities between metamodels explicitly and thereby allows to state clearly which common concepts they share. From such a specification, transformations are derived that keep instances of those metamodels consistent. We discuss options how to derive such transformations, strategies to hierarchically compose Commonalities, as well as benefits and limitations of the approach. Additionally, we discuss design options for a language that supports the specification of Commonalities and present the *Commonalities language*, which we have developed as a proof-of-concept. It is based on the Bachelor's thesis of Gleitze [6]. Our main contributions in this paper are:

Commonalities Approach (C1): We propose an approach for making common concepts of different metamodels explicit rather than encoding them implicitly in constraints of a transformation.

Commonalities Language (C2): We discuss design options for a language to define Commonalities and outline one language to specify them.

Proof-of-Concept (C3): We give an indicator for the applicability of the approach by providing a proof-of-concept implementation and applying it to a scenario with four simple metamodels sharing common concepts.

We expect several benefits from our approach, i.e. specifying Commonalities, in comparison to direct transformation specifications between metamodels. First, we claim

Final version published in "2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)"
IEEE Xplore: https://ieeexplore.ieee.org/document/8904619, DOI: 10.1109/MODELS-C.2019.00058

to achieve *better understandability* of relations between metamodels, because common concepts are made explicit. Second, the approach *reduces errors* when more than two metamodels are to be kept consistent. Transformations relate two metamodels and therefore have to be combined to a network of transformations to keep instances of more than two metamodels consistent. Such a network can be regarded as a graph, formed by metamodels as its nodes and transformations as its edges. However, such a network can easily raise compatibility problems if there exists more than one path of transformations between two metamodels. A hierarchy of Commonality specifications is, by design, not prone to such problems. Finally, we *improve reusability*, because an arbitrary subset of metamodels, between which Commonalities are defined, can be selected to keep their instances consistent. In contrast, removing metamodels from a transformation network can easily lead to missing transformation paths between two metamodels.

II. Running Example

We use a running example throughout the paper to explain the Commonalities idea. It relies on three metamodels: UML class models, Java code, whose grammar definition can be treated as a metamodel [8], and the Palladio Component Model (PCM) [19], a component-based architecture description language. The consistency relations between Java and UML class models are mostly one-to-one mappings, as they provide the same concepts of object-oriented design. Consistency relations between PCM and Java were proposed by Langhammer *et al.* [13]. For this paper, only the one-to-one mapping between components in PCM and classes in Java—or generally all object-oriented languages—are relevant, whereby each component is mapped to a class but not vice versa.

For the examples in this paper, a minimalist subset of those metamodels, depicted in Figure 1, is sufficient. It only comprises classes in UML and Java and components in PCM, which all have a name. The name shall be equal between corresponding classes in UML and Java, whereas the classes realizing a component shall have the same name as the component, complemented by an "Impl" suffix (cf. [13]). With state-of-the-art techniques, those constraints could be implemented as relations in declarative or as enforcing routines in imperative transformation languages.

In this paper, we consider metamodels that conform to the Essential Meta Object Facility (EMOF) standard [17]. Such metamodels consist of classes, which we denote as *metaclasses* to avoid confusion with classes in exemplary metamodels such as Java and UML. Metaclasses can in turn contain attributes and associations to other classes, which may be containments.

III. The Commonalities Approach

The state-of-the-art approach to keep models consistent automatically is the application of transformation languages. If instances of multiple (more than two) metamodels are to be kept consistent, one can either use multidirectional transformation approaches, or compose bidirectional transformations to a network of transformations [2]. When an instance of one metamodel is changed in such a network, the transformations are executed successively to propagate the change transitively across all models. There are strategies to find one ordering of transformations to apply [21] and strategies to perform a fixpoint iteration until no further changes are conducted [11].

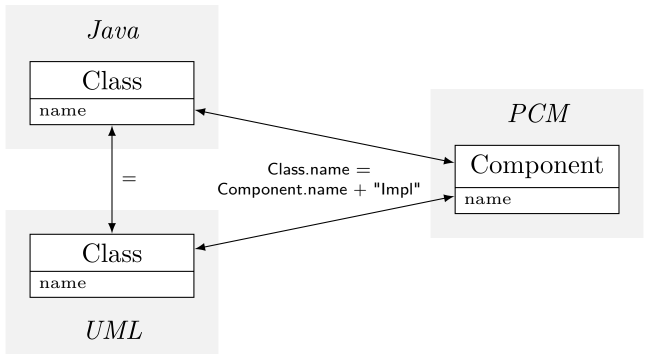

Fig. 1: Metamodel extracts for Java, UML and PCM and consistency relations (↔) between them

In this section, we propose a different approach for keeping two or more models consistent by specifying their common concepts rather than their direct consistency relations. This forms our contribution **C1**.

A. Making Common Concepts Explicit

The redundancies between different metamodels are an expression of common concepts that are represented redundantly. We already gave the example of a class in UML and Java, which are different representations of the common concept of a class in general object-oriented design. We propose to make common concepts explicit rather than encoding them into the rules of a transformation. This can be achieved by creating a *concept metamodel*, which defines those common concepts, and specifying the relations between the concept metamodel and the existing metamodels. We refer to the existing metamodels as *concrete metamodels*. The relation specifications can be used to derive transformations between the concrete metamodels and the concept metamodel.

Figure 2 shows the metaclasses for a `Class` as extracts of the concrete metamodels for UML and Java and the metaclass for the common concept of a `Class` in the concept metamodel for object-oriented design. We denote a single common concept as a *Commonality*. Further Commonalities could, for example, be interfaces or methods. The relation between the `Class` Commonality and its realizations in the concrete metamodels are shown by a *«manifests»* relation. In our simplified example, the relation would especially define that the names of the classes have to equal.

When another concrete metamodel that represent the same concepts shall be added, it is only necessary to define

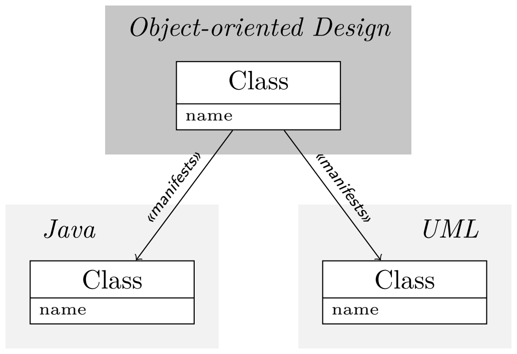

Fig. 2: Concept metamodel for object-oriented design with a `Class` Commonality and its relations to UML and Java

its relation to the concept metamodel. For example, adding `C++` as another metamodel representing object-oriented design would require the definition of the relation between the `Class` Commonality in object-oriented design and its representation in C++. Adding an additional metamodel may require the concept metamodel to be extended by Commonalities that were not relevant for the already considered metamodels. In general, a concept metamodel has to contain Commonalities for redundancies in all concrete metamodels, which—mathematically speaking—can be expressed as the union of all pairwise intersections of the concrete metamodels.

B. Composing Commonalities

We have explained how multiple metamodels can be kept consistent using one concept metamodel. This allows, theoretically, the definition of one large concept metamodel that contains all Commonalities for all concrete metamodels. It would at first sight be similar to a Single Underlying Metamodel (SUMM), as introduced by Atkinson *et al.* [1]. However, it would be less complex than a SUMM, which is able to express all information about the software system and thus contains the union of all concrete metamodels. Nevertheless, one large concept metamodel would still become unmanageably large due to the fact that it had to contain the union of all pairwise intersections of the concrete metamodels, as mentioned before.

To avoid the specification of such a monolithic concept metamodel, we propose to compose Commonalities from different concept metamodels. Instead of having only Commonalities that relate to metaclasses in concrete metamodels, Commonalities may also have relations to other Commonalities. Consider the concept metamodel for component-based design in Figure 3. It contains the Commonality `Component`, which is represented by an equally named metaclass in PCM, as well as an equally named metaclass in UML. Additionally, the `Component` is represented by the Commonality `Class` in the concept metamodel for object-oriented design, conforming to the relations proposed by Langhammer *et al.* [13]. This induces a tree structure with Commonalities as inner nodes and

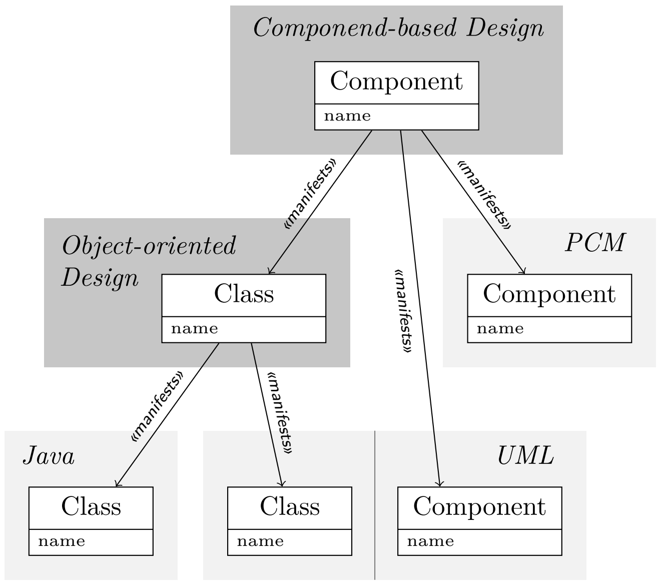

Fig. 3: Concept metamodels (dark) and concrete metamodels (light) of the running example, extended by UML components, with their relations

metaclasses of concrete metamodels as leaves. With such a composition structure, a *«manifests»* relation may not only exist between a Commonality of a concept metamodel and a metaclass in a concrete metamodel but also between two Commonalities. However, a concrete or concept metamodel that is lower in the hierarchy is supposed to represent how a metaclass or Commonality in the higher one manifests, which is why we call it a *manifestation*. For example, the object-oriented design concept metamodel is a manifestation of the component-based design concept metamodel.

A metamodel may have several Commonalities in different concept metamodels with different other metamodels. For example, in Figure 3, the UML metamodel contains a `Class` and a `Component` metaclass, which have two different Commonalities in two different concept metamodels.

C. Transformation Operationalization

To actually keep models consistent, the specification of a hierarchy of concept metamodels has to be operationalized. Two options for operationalization can be distinguished:

Concept metamodels as additional metamodels: The specified concept metamodels are actually instantiated and the transformations are executed as they are defined between the concept metamodels and their manifestations. In consequence, instances of the concept metamodels have to be maintained.

Transformations between concrete metamodels: The concept metamodels and the relations between them and their manifestations are used to derive bidirectional transformations between the concrete metamodels. For example, from the concept metamodel for object-oriented design in Figure 2, a bidirectional transformation between Java and UML is derived.

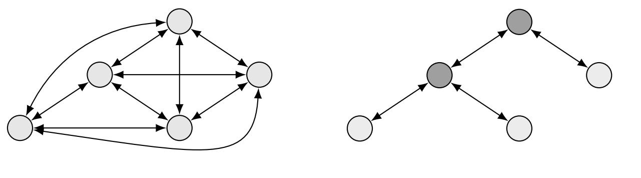

Fig. 4: Extremes of transformation network topologies: nodes represent metamodels, edges represent transformations (concept metamodels in a tree of Commonalities in dark gray), adapted from [10]

A drawback of the first option is that additional models have to be managed and persisted. In consequence, the user has to version these models although they should be transparent to him or her, as long as no appropriate framework abstracts from such tasks. A drawback of the second option is that the types of supported relations that can be described in the transformations are limited. First, only relations may be defined that can be composed with any other relation, such that a direct transformation between two metamodels can be derived. Second, it is possible to define n-ary relations between more than two metamodels that cannot be decomposed into binary relations between them, but only into n binary relations between those metamodels and an additional one [21]. In consequence, the first option provides higher expressiveness.

While the first option can be realized without an additional language by just defining the concept metamodels and the transformations with existing languages, the second option requires a mechanism that generates the transformations between the concrete metamodels from those between the concept metamodels and their manifestations.

D. Benefits of Commonalities

We suppose the Commonalities approach to provide two kinds of benefits: First, we expect that it improves understandability of relations between metamodels, because common concepts are not encoded in transformations implicitly but modelled explicitly. This is even a benefit if instances of only two metamodels shall be kept consistent. Second, it reduces problems that can occur if several bidirectional transformations are combined into a network of transformations to keep multiple models consistent.

Networks of transformations can have two extremes of topologies, as depicted in Figure 4. If transformations between all metamodels are defined, the network forms a dense graph (see Figure 4a). In contrast, if there exists exactly one path of transformations between each pair of metamodels, the network forms a tree (see Figure 4b). Several properties for such networks have been identified by Gleitze [6] and Klare [10]. Two essential properties are *compatibility* and *modularity* [10], which, unfortunately, contradict each other. The Commonalities approach, however, improves both of them. *Compatibility* means that transformations do not define contradictory constraints. Consider the relations introduced for the running example in Figure 1. The names of the same class in Java and UML are defined to be equal. If a class in Java and UML realizes a PCM component, it shall have the same name appended with an “Impl” suffix. If transformations realize the three relations between PCM, UML and Java, and the one between PCM and Java adds that suffix whereas the one between PCM and UML omits it, the constraints can never be fulfilled. In that case, the transformations are considered incompatible. Incompatibility may arise whenever more than one transformation path between two metamodels exists. In consequence, compatibility cannot be guaranteed in dense network, whereas it is inherently high if the network forms a tree. *Modularity* means that any subset of the metamodels can be used without loosing consistency because of missing transformations. Modularity is high if any metamodel can be removed from the network and the remaining transformations still define consistency between all remaining metamodels. In consequence, modularity is high in a dense network, because all metamodels are directly related, while it is low if the network is a tree, because inner nodes cannot be removed without their children not being related by a transformation anymore. Since redundant paths between metamodels improve modularity but reduce compatibility, these properties are inherently contradicting.

The Commonalities approach improves both these properties due to the fact that additional metamodels are introduced in the specification. The transformations between metaclasses in concrete metamodels and Commonalities in concept metamodels induce a tree, thus compatibility is high. Additionally, only the leaves of the tree are concrete metamodels, which are actually used to describe a system and whose instances are modified, whereas the inner nodes only represent auxiliary metamodels, exemplarily marked in Figure 4b. In consequence, taking an arbitrary subset of concrete metamodels removes only leaves and can thus be done without removing any transformations that are necessary to keep instances of the remaining metamodels consistent. This constitutes a major benefit of the Commonalities approach as compared to ordinary networks of transformations.

IV. The Commonalities Language

Having introduced the concepts of the Commonalities approach, we now present the Commonalities language, as proposed by Gleitze [6]. The language is a prototypical realization of the approach. We discuss design options for such a language and outline its syntax and its usage. This forms our contribution **C2**. For a detailed discussion of the language capabilities, we refer to [6]. In this paper, we focus on the presentation of fundamental concepts and therefore only outline the language to give an impression of what our proof-of-concept evaluation is based on.

A. Design Options

The development of a language for realizing the Commonalities approach provides at least two areas of design options. First, there are different possibilities to operationalize the approach, which covers decisions that are not visible to the user of the language. Second, different options regarding how to specify concept metamodels and their relations to their manifestations exist, which are decisions that are visible to the user of the language.

We already discussed in Subsection III-C that there are two different options for operationalizing a specification of Commonalities: one that uses instances of the concept metamodels and defined transformations at runtime, and one that generates direct transformations between the concrete metamodels from the specification. Since the way a specification is operationalized does not affect the user of the approach, it is up to the language and its designer to choose one of the options. We chose to build a language following the former approach, because it does not limit the possible relations that can be expressed.

Regarding the specification of concept metamodels and the transformations, there are two options:

External concept definition: Concept metamodels are defined as ordinary metamodels and the relations to their manifestation are defined in individual specifications for each manifestation.

Internal concept definition: A specialized language allows to define the concept metamodel and the Commonalities it consists of, together with relations of all Commonalities to their manifestations.

A benefit of the first option is that the relation to each manifestation can be specified independently, which reduces dependencies between the different manifestations of one concept metamodel. Additionally, it could be realized without developing a dedicated language. The concept metamodels can be described just like any other ordinary metamodel and one can use any existing transformation language, such as QVT, for the transformation definitions. The second option requires a specific language that enables an integrated specification of the concept metamodel and its relations to manifestations. A benefit is that it improves locality, because all information about one Commonality is represented in one place. This makes it easier to understand the combined transformation logic concerning a Commonality. Additionally, the concept metamodel can be easily extended with this solution when an additional manifestation is related and a necessary Commonality is missing. Finally, it is easier to ensure that for all Commonalities a relation to the manifestation is defined than in the first option, where Commonality specification and relation specification are separated. We chose to build a language following the second option to improve locality and ensure that no elements are specified in the concept metamodel that do have no manifestation in all relevant concrete metamodels.

```
concept ComponentBasedDesign

commonality Component {
  with PCM:Component
  with UML:Component
  with ObjectOrientation:Class

  has name {
    = PCM:Component.name
    = UML:Component.name
    = prefix(ObjectOrientation:Class.name,
        "Impl")
  }
}
```

Listing 1: An exemplary specification of the `Component` Commonality between PCM, UML and the object-oriented design concept in the Commonalities language

B. Language Description

As introduced before, our realization of the Commonalities language provides an internal concept definition and uses the concept metamodels as additional metamodels in the operationalization. An example for the syntax of the Commonalities language is depicted in Listing 1.

The language allows to define concept metamodels by declaring Commonalities, each representing one commonality between different manifestations, such as the `Component` Commonality in our example. Relations between the concept metamodels and their manifestations are supposed to be specified *declaratively*. For every Commonality, the metaclasses in the manifestations that realize them are specified. In the example, the `Component` in PCM and the `Class` in the object-oriented design concept metamodel are related to the `Component` Commonality. In our language, a Commonality is realized by a metaclass in the metamodel that is generated for a concept, so the `Component` Commonality is realized by a `Component` metaclass.

Within a Commonality, attributes and references can be defined, similar to an ordinary metaclass. The relations of an attribute to the manifestation are declared directly at the attribute. In the example, a `name` attribute is specified, which maps to the name of the component in PCM and the name appended with an "Impl" suffix in Java. The language provides several operators for attribute relations, apart from equality relations. The example depicts a prefix operator that allows to compose a String attribute. Such operators can be defined independently and added to the language dynamically. References can be defined comparably to attributes but can be enriched with a definition of containment relations.

The actually conceptualized and implemented language by Gleitze [6] is far more sophisticated than the simple overview we provide here. It supports different kinds of bidirectional operators for attribute mappings, containment specifications (so-called *participations*), attribute checks as preconditions for Commonality instantiation, and more.

V. Proof-of-Concept

We have proposed the Commonalities approach and a realizing language. We have explained that we expect them to improve understandability of transformations and to reduce problems of transformation networks, such as compatibility and modularity. Although we gave arguments that justify this expectation, it has to be evaluated empirically to increase evidence. However, before evaluating the benefits of our approach, we first have to investigate its feasibility. For that reason, we built an initial prototype of the language and applied it to a simple evaluation case as a proof-of-concept. This forms our contribution **C3**.

A. Case Study

We have implemented a prototype of the Commonalities language, which allows to define Commonalities with simple attribute and reference mappings and to compose Commonalities. The syntax is an extension of the example shown in Listing 1. The language comprises a compiler that derives a concept metamodel, as well as a set of transformations from a specification in the language. The generated transformations are defined in the Reactions language [9], which is a delta-based transformation language that is part of the Vitruvius approach [12]. Vitruvius is a view-based development approach that uses transformations to keep models consistent. The implementation of the Commonalities language can be found in the GitHub repository of the Vitruvius project [26].

We have applied the implementation to a simple case study that consists of four metamodels, each containing one metaclass that represents a root element and one that represents a contained element. Both elements have an identifier and a name in all metamodels, and an additional single-valued and multi-valued feature of integers in two of the metamodels. The root metaclass additionally has a containment reference to the contained metaclass. We have defined two Commonalities, one for the root element and one for the contained element, which redundantly represent the same concepts in all the metamodels. The root Commonality references the contained Commonality. This results in one concept metamodel with four manifestations.

To validate that the specifications in the Commonalities language are correctly defined and operationalized, we have defined test cases that perform 21 different model modifications, which create and delete all possible types of elements and modify all attribute and reference values in instances of every metamodel. They cover the set of all possible modifications that can be performed on instances of those metamodels. This also includes change propagation across composed Commonalities. The tests successfully validate that the modifications are correctly propagated to all other models in all cases. The test cases and the used example metamodels are also available in the GitHub repository of the Vitruvius project [26].

B. Discussion

Our proof-of-concept validates the feasibility of the proposed Commonalities approach: It demonstrates that it is possible to apply the concept to define consistency relations between multiple metamodels in a simple scenario and that an operationalization can be derived that preserves consistency of instances of such metamodels. The results only give an indicator that the Commonalities concept can be applied and that a language with an internal concept definition can be designed. To further evaluate the capabilities of such an approach, the language would have to be extended to be able to define more complex relationships. Additionally, the approach has to be applied to larger parts of more complex metamodels and metamodels for different contexts to improve external validity of the results. This could also reveal whether the assumption of having a tree of Commonalities is achievable in realistic scenarios.

Since evaluating functional capabilities of the approach is only an—essential—first step, the evaluation of further properties such as applicability, appropriateness, effectiveness and scalability are part of ongoing work with further case studies. As a central benefit of our approach, we claim to improve understandability of relations between metamodels, but can only give arguments for that by now. An evaluation of that claim would require a user experiment that compares our approach to specifications of direct transformations between multiple metamodels.

Finally, one might argue that defining concept metamodels leads to additional effort, as for two metamodels it is necessary to define one additional metamodel and two transformations rather than only a single transformation. First, this is only true as long as only two metamodels are related by one concept metamodel. If three metamodels shall be related, there would be a network of three transformations, which are not necessarily compatible, without using the Commonalities approach, and one metamodel with three transformations using the Commonalities approach. When the Commonalities approach is applied, the number of necessary transformations increases linearly with the number of metamodels that are related, whereas it increases quadratic without them. Second, using an appropriate language to define concept metamodels and transformations, as we have proposed in Section IV, the effort can be reduced by only requiring to write one specification that contains the concept metamodel and the transformations in one place.

VI. Related Work

The Commonalities approach is related to the highly researched field of model consistency and especially of model transformations. In the following, we compare our approach to others that rely on commonality specifications, to both multidirectional transformations and transformation networks that also allow consistency preservation between multiple models and finally to constraint solving, a different paradigm for preserving model consistency.

Commonality Approaches

The idea of defining commonalities to express consistency of multiple models was especially researched from a theoretical viewpoint. That research is based on the idea of using an additional $n+1$-th metamodel to decompose the n-ary consistency relation between n metamodels into n binary relations [3], [22].

Existing approaches to practically use commonalities for keeping multiple models consistent are domain-specific. The DUALLy approach [4], [16] uses a domain-specific concept metamodel for architecture description languages, which is a fixed metamodel to which relations of arbitrary architecture description languages can be defined.

Multidirectional Transformations

Without defining additional metamodels, multidirectional transformations are an approach to directly define the relations between multiple metamodels. The QVT-R standard [18] considers multidirectional transformations, but Macedo *et al.* [15] reveal several limitations of its applicability and propose strategies to circumvent them. Triple Graph Grammars (TGGs) are a graph-based approach to define transformations, which has been extended to enable the specification of multidirectional rules [23], [24]. In contrast to our work, these approaches support the specification on n-ary relations between n metamodels, but do not provide means to improve their understandability as we expect the definition of Commonalities to do.

Networks of Bidirectional Transformations

We introduced networks of bidirectional transformations as the state-of-the-art for specifying consistency relations between multiple metamodels. Stevens [21] investigates the ability to decompose n-ary relations into binary ones and also discusses confluence issues, which arise from incompatibilites of transformations, as discussed in Subsection III-D. Such a decomposition of relations is not always possible, thus such approaches are restricted to cases, where all n-ary relations can be decomposed into binary ones. Additionally, such networks are prone to compatibility errors or reduced modularity, as discussed in Subsection III-D.

Transformation composition and transformation chains deal with specific problems of transformation networks. Composition techniques deal with internal composition of transformations [27], which are techniques that are integrated into a language, and external composition of transformations, which work independently from the language. Those approaches especially comprise factorization and re-composition of transformations [20] and investigations of compatibility of transformations for different versions of the same metamodels. Transformation chains deal with specific networks that occur when transformations from metamodels with a high level of abstraction to those with a low level of abstraction are defined. Specification languages for transformation chains allow to combine transformations to chains [14] and to treat them as black-boxes [25].

Constraint Solving

Consistency relations between multiple metamodels can also be expressed as logical constraints. Restoring consistency for a set of instances can be achieved by constraint solving. Eramo *et al.* [5] consider the usage of Answer Set Programming (ASP) to define consistency relations between metamodels. The approach derives a set of candidates that fulfill the constraints after a model is modified. However, that research focuses on solving constraints rather than designing an appropriate way how to define them, in contrast to our Commonalities approach.

VII. Conclusion

In this paper, we proposed the Commonalities approach, which allows to make common concepts of different metamodels explicit. The central idea of the presented approach is to define *concept metamodels* that represent the common concepts, i.e., the Commonalities of two or more existing metamodels, and to define the relation of those metamodels to the concept metamodels. Concept metamodels can be hierarchically composed to enable the separate definition and combination of independent concepts. We discussed different options for designing a language that supports the specification of such Commonalities and their relations, as well as for operationalizing such a specification to executable transformations. We outlined a language for the Commonalities approach and explained which of the aforementioned options we chose, and why. Finally, we have applied an implementation of that language to simple scenarios as a proof-of-concept. The results indicate the feasibility of applying the Commonalities approach and implementing an appropriate language.

The expected benefit of our approach is a better understandability of relations between metamodels compared to their implicit encoding in transformations. Additionally, we argued why our approach improves the essential properties *compatibility* and *modularity*, which usually contradict each other in other approaches to keep multiple models consistent, like networks of transformations that define the direct relations between metamodels. In ongoing work, we extend the capabilities of the language to perform a comprehensive evaluation of the functionality of the approach as well as its applicability to more sophisticated case studies. We will validate our claim of improved understandability in a controlled experiment. Nevertheless, the initial results of our proof-of-concept are a promising indicator for the applicability of the Commonalities approach.

References

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

[2] A. Cleve, E. Kindler, P. Stevens, and V. Zaytsev, “Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491),” *Dagstuhl Reports*, vol. 8, no. 12, pp. 1–48, 2019.

[3] Z. Diskin, H. König, and M. Lawford, “Multiple Model Synchronization with Multiary Delta Lenses,” in *Fundamental Approaches to Software Engineering*, Springer International Publishing, 2018, pp. 21–37.

[4] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, “A model-driven approach to automate the propagation of changes among Architecture Description Languages”, *Software and Systems Modeling*, vol. 11, pp. 29–53, 1 2012.

[5] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo, “Change Management in Multi-Viewpoint System Using ASP,” in *Enterprise Distributed Object Computing Conference Workshops*, 2008, pp. 433–440.

[6] J. Gleitze, “A Declarative Language for Preserving Consistency of Multiple Models,” Bachelor’s Thesis, Karlsruhe Institute of Technology (KIT), 2017.

[7] H. Guissouma, H. Klare, E. Sax, and E. Burger, “An empirical study on the current and future challenges of automotive software release and configuration management,” in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018, pp. 298–305.

[8] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, “Closing the gap between modelling and java,” in *Software Language Engineering*, ser. LNCS, vol. 5969, Springer Berlin Heidelberg, 2010, pp. 374–383.

[9] H. Klare, “Designing a Change-Driven Language for Model Consistency Repair Routines,” Master’s Thesis, Karlsruhe Institute of Technology (KIT), 2016.

[10] ——, “Multi-model Consistency Preservation,” in *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings*, 2018, pp. 156–161.

[11] H. Klare, T. Syma, E. Burger, and R. Reussner, “A categorization of interoperability issues in networks of transformations,” *Journal of Object Technology*, vol. 18, no. 3, 4:1–20, 2019, The 12th International Conference on Model Transformations.

[12] M. E. Kramer, E. Burger, and M. Langhammer, “View-Centric Engineering with Synchronized Heterogeneous Models,” in *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, ser. VAO ’13, ACM, 2013, 5:1–5:6.

[13] M. Langhammer and K. Krogmann, “A co-evolution approach for source code and component-based architecture models,” in *17. Workshop Software-Reengineering und-Evolution*, vol. 4, 2015.

[14] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss, “FTG+PM: An Integrated Framework for Investigating Model Transformation Chains,” in *SDL 2013: Model-Driven Dependability Engineering*, Springer Berlin Heidelberg, 2013, pp. 182–202.

[15] N. Macedo, A. Cunha, and H. Pacheco, “Towards a framework for multi-directional model transformations,” in *3rd International Workshop on Bidirectional Transformations - BX*, vol. 1133, CEUR-WS.org, 2014.

[16] I. Malavolta, H. Muccini, P. Pelliccione, and D. A. Tamburri, “Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies”, *IEEE Transactions of Software Engineering*, vol. 36, no. 1, pp. 119–140, 2010.

[17] Object Management Group (OMG), *MOF 2.5.1 Core Specification (formal/2016-11-01)*. OMG, 2016.

[18] ——, *Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification – Version 1.3*. 2016.

[19] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, *Modeling and simulating software architectures – the palladio approach*. MIT Press, 2016, 408 pp.

[20] J. Sánchez Cuadrado and J. García Molina, “Approaches for Model Transformation Reuse: Factorization and Composition,” in *Theory and Practice of Model Transformations*, Springer Berlin Heidelberg, 2008, pp. 168–182.

[21] P. Stevens, “Bidirectional Transformations in the Large,” in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2017, pp. 1–11.

[22] P. Stünkel, H. König, Y. Lamo, and A. Rutle, “Multimodel Correspondence Through Inter-model Constraints,” in *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming*, ser. Programming’18 Companion, ACM, 2018, pp. 9–17.

[23] F. Trollmann and S. Albayrak, “Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models,” in *Theory and Practice of Model Transformations*, Springer International Publishing, 2016, pp. 91–106.

[24] ——, “Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models,” in *Theory and Practice of Model Transformations*, Springer International Publishing, 2015, pp. 214–229.

[25] B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers, “UniTI: A Unified Transformation Infrastructure,” in *Model Driven Engineering Languages and Systems*, Springer Berlin Heidelberg, 2007, pp. 31–45.

[26] Vitruv Tools. (2019). VITRUVIUS Framework (GitHub), [Online]. Available: https://github.com/vitruv-tools/Vitruv (visited on 06/25/2019).

[27] D. Wagelaar, “Composition Techniques for Rule-Based Model Transformation Languages,” in *Theory and Practice of Model Transformations*, Springer Berlin Heidelberg, 2008, pp. 152–167.