Model Consistency ensured by Metamodel Integration

Johannes Meier
Software Engineering Group
University of Oldenburg, Germany
meier@se.uni-oldenburg.de

Andreas Winter
Software Engineering Group
University of Oldenburg, Germany
winter@se.uni-oldenburg.de

ABSTRACT

To keep semantically interrelated and physically separated artifacts consistent to each other is challenging. Consistency between such artifacts, described by models and conforming to metamodels, targets mainly synchronizing overlapping information and additional relations between them. This paper depicts an approach to synchronize models by integrating models together with their metamodels into an integrated (meta)model using special operators. These operators are used to keep all models consistent to each other, which is discussed for two consistency rules along three use cases.

1 MOTIVATION

In modern software systems the heterogeneity of languages describing certain aspects of systems grows. This includes models, domain specific languages for different but overlapping concerns, and data produced by tools in arbitrary data formats. All these languages produce data, which are often separated technically from each other by different data formats, different technical spaces, or different tools, because of historical growth, or different users with different working locations, concerns or access rights. In this paper, all artifacts are described by *model*s and conforming *metamodel*s.

These technical separations rise issues, if these models describe information which are interrelated contentwise: If two models are overlapping and describe the same information, this information has to be kept consistent to each other. If two models describe different information which are related to each other, these relations have to be described, changed and stored. These interrelations between models are specific to each project and are called *consistency rules*. If all consistency rules are fulfilled, the models are consistent to each other. Therefore, the goal of this paper is to depict an approach that ensures *model consistency* for arbitrary models conforming to arbitrary, but appropriate metamodels, by using an integrated data structure containing all information of all models.

As ongoing example throughout this paper (the transferability is discussed in Section 4.3), a strongly simplified software development project environment is depicted: Requirements are described textually. The design is developed by class diagrams, while the sourcecode is written in Java. These three languages are supported by three different tools and result in three technically separated models and metamodels. Figure 1 shows three simplified metamodels for the ongoing example. Requirements are depicted by a set of `Requirements` with an `id` and its requirements sentence stored in `text` (top left). Class diagrams contain `Classes` with their `classNames` and `Associations` which are unidirectional, have a role `name` and point to one class as `type` (top right). Java sourcecode depict `ClassTypes` representing classes with `name` and containing `Methods` with `name` and `calling`-`calledBy`-relationship (bottom).

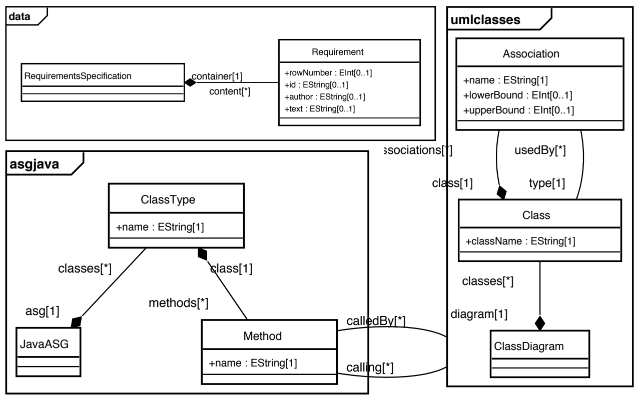


Figure 1: Initial simplified Metamodels for Requirements (top left), Class Diagrams (top right), and Sourcecode (below)

Since these three models describe three different concerns originating in different tools within the same project, there are the following two illustrative consistency rules:

Consistency Rule 1: Data classes are described by class diagrams as well as by Java: If an instance of `asgjava.ClassType` has the same value for `name` like an instance of `umlclasses.Class` for `className`, these two instances describe the same information and should be renamed in parallel. Sourcecode and class diagram are consistent to each other, if they describe the same set of data classes, identified by their class names.

Consistency Rule 2: Requirements describe functionality, which is realized by business logic written in Java. Since the manager wants to know, which requirements are already fulfilled in Java, requirements should be linkable with Java methods. Requirements and sourcecode are consistent to each other, if (at least) each requirement is linked with all methods whos names are contained in the text of the requirement.

Up to now, the required consistency assurance is often done with hand-written glue transformations or even manually. In the latter case, even the consistency rules are not made explicit, which causes potential misunderstandings between users and introduces new inconsistencies. Therefore, an elaborated approach is required to *ensure consistency automatically* between technically separated models and will be presented in Section 3. As preparation, there are the following *four main challenges* to overcome for model consistency approaches, which are used for the design of the new approach of Section 3 and form its main characteristics:

Formalize Consistency Rules (Challenge 1): All consistency rules have to be specified explicitly and formally. This is required for the automation of consistency assurance. This formalization has to be done on metamodel level to support all conforming models.

This is done by a stakeholder called *methodologist* who knows the current project like the ongoing software development example with its consistency rules and who has modeling experience [1].

Create explicit SUM(M) (Challenge 2): An integrated model and conforming integrated metamodel describing all information of all initial and independent models has to be developed. This integrated model is called Single Underlying Model (SUM) [1] conforming to the Single Underlying MetaModel (SUMM). Since overlapping models describe information twice, which leads to inconsistencies easily, the SUM contains such information only once without possibility for inconsistencies and can be used as single point-of-truth. SUMs are helpful for analyses, refactorings, and visualizations targeting information from more than one initial model.

Support initial (Meta)Models (Challenge 3): Reusing existing models and conforming metamodels is the normal case, since there are already lots of metamodels for domain specific languages and legacy models. In particular, already existing models have to be reused in ongoing projects, which requires follow-up challenges:

Reuse initial Models (Challenge 3a): Initially, these models have to be imported into the new SUM.

Fix initial Inconsistencies (Challenge 3b): Since these initial models are synchronized by hand or even not synchronized at all, they can contain inconsistencies, which have to be fixed.

Consistent initial Models (Challenge 3c): Since existing tools and environments work only with these initial (meta)models due to missing tool interoperability, they have to be kept up-to-date and should be changable, as discussed in the next challenge.

Ensure Model Consistency after User Changes (Challenge 4): If *users*, who work with the initial models or the SUM, change something, these changes have to be propagated to all other models to keep them consistent to the changed model. Additionally, these user changes can lead to inconsistent models afterwards, which have to be fixed according to the consistency rules. As result, all models reflect the changes of the users, are consistent to each other and fulfill all consistency rules.

Contribution of this paper is the description of a new approach ensuring model consistency regarding explicit consistency rules, supporting initial (meta)models, and providing an integrated (meta) model. Section 2 discusses advantages and disadvantages of existing model consistency approaches along the challenges. Section 3 describes the new approach, which is applied to the ongoing example with discussion in Section 4 and summarized in Section 5.

2 RELATED WORK

Following ISO Standard for Architecture Description 42010:2011 [5], model synchronization approaches are split into synthetic and projectional approaches. *Synthetic* approaches keep the models separated from each other and introduce pair-wise transformations between them to ensure consistency between them. Therefore, initial models can be supported easily (Challenge 3), but no integrated model is used (Challenge 2). The transformations are the formalization of consistency rules (Challenge 1) and ensure model consistency (Challenge 4). Here, several approaches exist using different transformation techniques like Triple Graph Grammars [12] or QVT-R [11], or using explicit correspondences [3].

	synthetic	projectional	
		OSM	Vitruvius
1. Formalize Consistency Rules	yes	yes	yes
2. Create explicit SUM(M)	no	explicit	virtual
3a. Reuse initial Models	yes	no	yes
3b. Fix initial Inconsistencies	yes	–	no
3c. Consistent initial Models	yes	–	yes
4. Ensure Model Consistency	yes	yes	yes

Figure 2: Comparison of Model Consistency Approaches

Projectional approaches introduce a new data structure containing all information of all models (Challenge 2), which is used to synchronize all models only with this integrated data structure (Challenge 4). Additional to the reduced amount of consistency rules between (meta)models, another benefit of projectional approaches is the integrated data structure, which allows analyses, refactorings, and visualizations using all information of all models in an integrated manner (Challenge 2).

The *Orthographic Software Modeling* (OSM) approach [1] introduced the idea of a *Single Underlying Model* (SUM) conforming to a Single Underlying MetaModel (SUMM) which contains all information of the current project without any redundancies (Challenge 2). Users change the SUM not directly but through views which are subsets of the SUM and which correspond to the initial models to integrate in this paper. Since OSM is a top-down approach starting with the development of the SUMM with high quality, initial models are not supported (Challenge 3). Therefore, there is no distinction between initial views and newly created views.

The *Vitruvius* approach [6] follows the SUM idea, where users change the initial views, but realizes the model consistency internally by a modular SUM which keeps the models separated and which are synchronized (Challenge 4) by an own language for consistency rules (Challenge 1). Since this language needs consistent models as precondition, there is no build-in support to fix initial inconsistencies (Challenge 3b). In the end, the SUM exists only virtual and is not usable (Challenge 2). New views containing information of several initial views are defined with ModelJoin [2].

Figure 2 summarizes the main characteristics of the mentioned approaches. Since OSM does not support initial separated models, while Vitruvius and all synthetic approaches have no explicit SUMs, which are helpful for analyses, refactorings, and visualizations targeting information from more than only one model, this paper describes a new projectional approach with explicit SUM and support for the initial models, presented in Section 3. This includes fixes of inconsistencies in the initial models (Challenge 3b) shown in Section 3.2 and support for changing the SUM directly by users in contrast to OSM and Vitruvius depicted in Section 3.3.

3 METAMODEL INTEGRATION

The new approach of this paper follows the idea of having a projectional and explicit SUM [1] (Challenge 2) which takes the initial models into account (Challenge 3). The main challenges, motivated in Section 1, are fulfilled by this new approach, which will be presented along three use cases applying this approach to the ongoing example. This approach is *operator-based*, since operators are used to describe transformations between the initial models and the SUM (Use Case 1). Chains of operators are executed to initialize the SUM (Use Case 2) and to keep all initial models and the SUM consistent

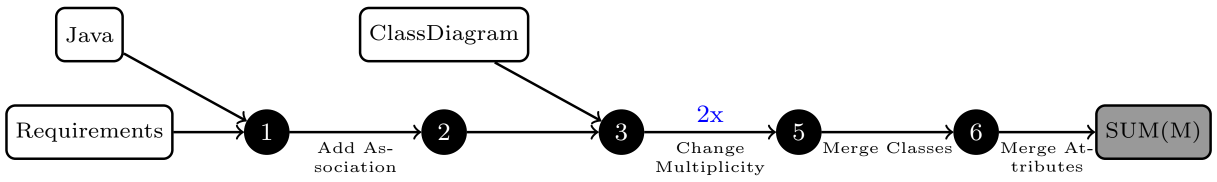


Figure 3: Chain of configured Operators for integrating Requirements, Class Diagrams, and Sourcecode

to each other (Use Case 3). Operators split long transformations into clear and manageable steps or mini transformations, which allows for iterative development and simplifies debugging, because the models before and after have to be consistent. As shown in detail later, the SUMM can be developed and improved step by step. Additionally, the consistency rules can be assigned explicitly to single operators and are therefore distinguished from each other.

3.1 Use Case 1: Configuration of Operators

The challenges overcome by this first use case are the formalization of consistency rules (Challenge 1) and the bottom-up definition of the SUMM (Challenge 2). Therefore, *preconditions* for this use case are the initial metamodels and the consistency rules. Initial metamodels are reused as starting points. After that, the methodologist selects step-wisely special operators which are provided by the approach, and applies them to the current metamodel to form an improved metamodel. After selecting and configuring enough operators, the current metamodel is used as final SUMM (Challenge 2).

A possible chain of configured operators for the example is shown in Figure 3: Starting with the metamodels for Requirements and Java, the first step is to include the two metamodels ❶. This is done only for technical reasons and results in a new container class `ProjectData`, but changed nothing regarding the contentwise integration. The same counts for ClassDiagram included at ❸. Operators are selected and configured, which are denoted along the edges of the operator chain and introduce ❷, ❺, and ❻. Each sign *i* describes one stable and consistent metamodel with one conforming model, which is created by executing all previous operators starting with the initial metamodels.

The methodologist selected the operator `AddAssociation` ❶→❷ to fulfill the second consistency rule (Section 1), which requires to link requirements and methods fulfilling them. Since links are represented by an association on metamodel level, which is missing in the initial metamodels (Figure 1), a new association between `Requirement` and `Method` has to be created. This new association is drawn in red in the final SUMM in Figure 8. Conforming to the consistency rule, on model level links have to be created between each requirement and that methods whose names are contained in the text of the requirement.

The operator `ChangeMultiplicity` is applied twice ❺ (❹ is hidden because of space) which change the multiplicities for the associations `asg` and `diagram` (red in Figure 8). They change nothing in the model and provide the basis for the next operator.

The methodologist selected the operator `MergeClasses` ❺→❻ to fulfill the first consistency rule (Section 1), which requires the same set of data classes described by class diagrams and in Java. Since data classes are described by two different classes (`ClassType` and `Class`), in the SUMM only one class should be used (`ClassType`, marked red in Figure 8). Doing the same on model level unifies duplicated instances representing the same data classes and prevents inconsistencies in the SUM. After applying the last operator `MergeAttributes`, the final SUMM is available in Figure 8.

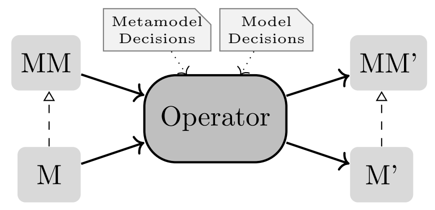


Figure 4: Signature of Operators

To support this, the used *operators* have the following five characteristics, depicted along the signature of operators in Figure 4. To create a new association into the metamodel for the first consistency rule along the operator `AddAssociation` ❶→❷, each operator executes a small **(1) change in the metamodel** on the given input metamodel MM and changes it in-place with MM' as result. Thus the initial metamodels are changed step by step along the operator chain resulting finally in the SUMM.

Since initial models should be reused and transferred into the SUM (Challenge 3a), additional **(2) changes in the model** are required to migrate the input model M in-place into the output model M'. These model changes keep the model conform to the changed metamodel and solve the model-co-evolution problem. Additionally, these model changes reflect the consistency rules. As example, the operator `AddAssociation` has to do nothing regarding model-co-evolution, but could create links following consistency rules ❶→❷ (discussed later). These two characteristics are the same as in *coupled operators* introduced in [4].

Since theses operators should be provided as library by the approach and reused by methodologists for arbitrary projects, the operators are designed in generic way to work with arbitrary metamodels and models. Therefore, they provide **(3) metamodel decisions** which describes the metamodel changes in more detail. The operator `AddAssociation` provides metamodel decisions to control source and target class of the new association together with wanted role names and multiplicities. In the example for the second consistency rule, the methodologist configures `AddAssociation` with `Requirement` as source class and `Method` as target class (❶→❷).

The **(4) model decisions** describe, how the corresponding change on model level look like in detail regarding degrees of freedom during model-co-evolution and consistency rules. Here in the strongly simplified example, a link should be created automatically between one requirement and one method, if the name of the method is contained in the text of the requirement. Therefore, the methodologist adds a model decision to `AddAssociation` ❶→❷ which checks all methods and links them with requirements according to this consistency rule. While the metamodel decisions are also used in [4], the model decisions are newly introduced in this approach. Both decisions allow the methodologist to create individual configurations fulfilling the specific consistency rules of the current project, while the operators are designed in general only once.

The last characteristic, **(5) bidirectionality** is required to keep the initial models up-to-date (Challenge 3c): Up to now, the operators are *configured* by the methodologist, while these operators are *executed* later to keep all models consistent to each other (Sections 3.2, 3.3). While the operator chain of Figure 3 shows, how the initial (meta)models are transferred into the SUM(M), also changes in the SUM has to be propagated into the initial models.

To support this "backward direction", bidirectionality is required for all operators, which is realized by supplemented each operator by an inverse operator. As example, `AddAssociation` is supplemented by the operator `DeleteAssociation` which removes one configurable association from the current metamodel and removes all links conforming to this association from the current model. After executing one operator and then its inverse operator, the current metamodel should be the same as before both executions. An inverse operator is always configured together with its forward operator. This bidirectionality is requested only for the metamodel, but not for the model to allow repairing model inconsistencies, as described in Section 3.2 in more detail. Regarding these five characteristics, the approach provides several operators. The currently available operators are depicted in Figure 5 and are still increasing.

	Name	Description
Add / Del.	`Add/DeleteClass`	creates / deletes a class
	`Add/DeleteAssociation`	creates / deletes an association
	`Add/DeleteAttribute`	creates / deletes an attribute
Change	`RenameClassifier`	renames a class
	`RenameFeature`	renames an association or attribute
	`ChangeAttributeType`	exchanges the type of an attribute
	`ChangeMultiplicity`	changes the multiplicity of an attribute or association
	`MakeClass(Non)Abstract`	adds / removes `abstract` to / from a class
Refactoring	`Merge/SplitClasses`	merges two classes into one / splits one class into two
	`Merge/SplitAssociations`	merges two associations into one / splits one association into two
	`Merge/SplitAttributes`	merges two attributes into one / splits one attribute into two
	`Extract/InlineSubClass`	extracts /inlines a sub-class

Figure 5: Current Set of Operators

Summarizing, in this use case the methodologist specifies by configuring operators, how the SUMM is created out of the initial metamodels. Thereby, the consistency rules for the current project are specified explicitly within the decisions of the used operators. The result is a chain of configured operators in form of a tree, as shown exemplary in Figure 3. Therefore, *postconditions* for this use case are having the operator chain containing the formalized consistency rules, and the derived SUMM.

3.2 Use Case 2: Initialization of SUM

The challenges overcome by this second use case are to create the SUM (Challenge 2) reusing the initial models (Challenge 3a) and to fix inconsistencies within the initial models (Challenge 3b), both corresponding to the operator chain containing the consistency rules. Therefore, *preconditions* for this use case are the initial models and the operator chain (Section 3.1). Since this is done automatically, this use case is started by either the methodologist or users.

For creating the SUM reusing the initial models, the configured operators are executed in forward direction one after another, in the order shown in Figure 3. The edges without operator names include the additional model into the current model technically. The contentwise integration is done using the operators `AddAssociation` ❷ which adds links between requirements and Java methods (more details in Section 3.3), `MergeClasses` and `MergeAttributes`, which are described in more detail later. The final SUM is reached after executing all operators and conforms to the SUMM depicted in Figure 8. Since all information of all initial models are stored in the explicit SUM, the initial models can be thrown away, because they are reconstructable always out of the SUM, or can be kept for performance issues.

The other challenge, fixing inconsistencies within the initial models following the consistency rules, is fulfilled during the execution of operators and will be described using the operator `MergeClasses` ❺→❻, whose changes are depicted in Figure 6: The input for `MergeClasses` (top left) has the data classes "University" and "Student" of the current software development project in the Java sourcecode (column "Java", `ClassType`), but only "University" as part of the class diagram (column "ClassDiagram", `Class`). Therefore "Student" is missing in the class diagram and should be fixed regarding the consistency rule, that class diagram and sourcecode should contain the same data classes (Section 1). This inconsistency is solved in the output of `MergeClasses` (column "SUM(M)", top "Model Changes"), because there is no distinction between data classes in class diagrams and sourcecode any more. Therefore, the output model contains only two objects, one for "University" and another one for "Student", since the model decisions specified, that `University : ClassType` and `University : Class` describe the same information and should be merged. Since the resulting model contains now the name of the class twice in the slots `name` and `className`, this duplicated information is removed using the operator `MergeAttributes` ❻→❼ (details skipped). This is done during the execution in forward direction to create the SUM.

To solve this inconsistency also in the initial models, the complete operator chain is executed in backward direction from SUM to all initial models using the inverse operators afterwards: Now, the inverse operator `SplitClass` (bottom part of Figure 6) takes the current model stemming from the SUM as input (bottom right) and produces the output (bottom left). This output now contains "University" and "Student", both for the sourcecode and the class diagram, because the methodologist specified for the model decision of `SplitClass`, that each input data class should be used both in class diagram and sourcecode. Therefore, the additional object is created (marked in red in Figure 6) and makes the model consistent.

In general, inconsistencies are fixed during the model-co-evolution following the consistency rules, which are specified within the model decisions by the methodologist. These fixes of inconsistencies are possible, because bidirectionaly is requested only for metamodels, but not for models (Section 3.1). Therefore, *postconditions* for this use case are having the fixed initial models and the newly

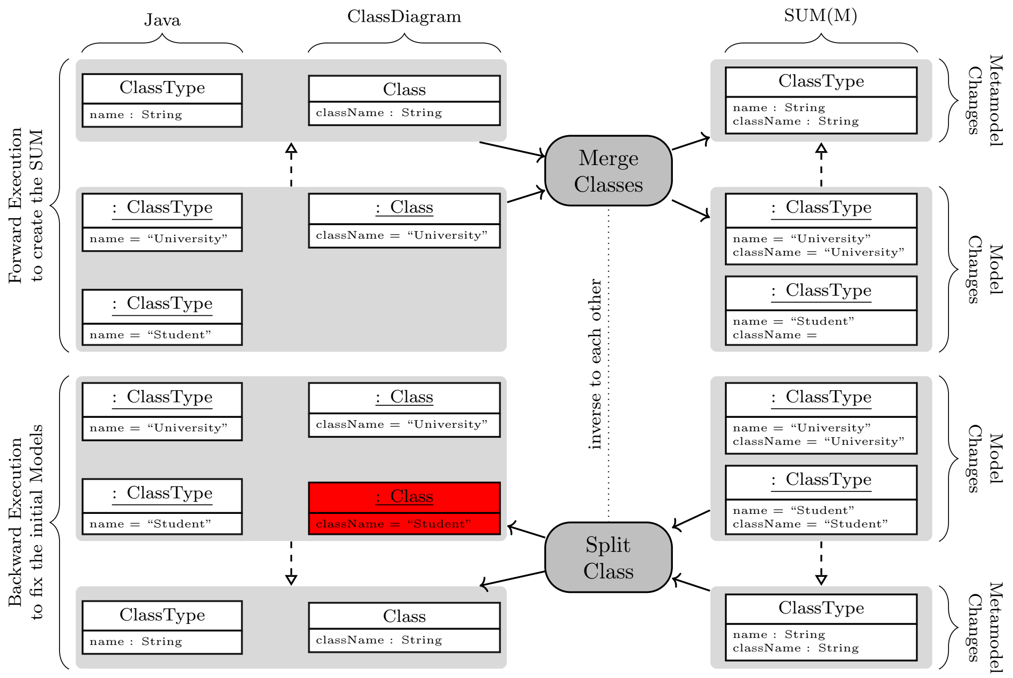


Figure 6: Operator `MergeClasses` ❺→❻ fixes Inconsistencies during the Initialization

created SUM. After executing these first two use cases only once, users change the SUM and the models often in the last use case.

3.3 Use Case 3: Consistency Assurance

After configuration (Section 3.1) and initialization (Section 3.2) of the SUM once, this use case is triggered by users who change one of the models or the SUM itself, which might introduce new inconsistencies in both cases. Therefore, the challenges overcome by this third use case are to propagate the changes into all other models and to fix new inconsistencies according to the operator chain and its consistency rules (Challenge 4). This includes the SUM as well as all initial models to keep them up-to-date (Challenge 3c). *Preconditions* for this use case are current versions of initial models and SUM and the operator chain.

The user changes are propagated into the SUM and all other models automatically by executing the chain of operators. This execution is done in both directions, because all operators including inverse operators can have model decisions which ensure model consistency. Since changes in the initial models can be mapped to changes in the SUM by executing the operators between this model and the SUM, only the case of changes in the SUM is discussed here. Since the SUM provides all information of all initial models in an integrated way, the SUM itself form a model which can be seen as new view containing all information and which can be changed by users as they are changing the initial models.

The ideas of this model change propagation are depicted along the operator `AddAssociation` ❶→❷ and Figure 7. Fulfilling the consistency rule, that methods should be linked with requirements, if the text of the requirement contains the name of the method, the execution of `AddAssociation` during the initialization (Section 3.2) integrates `Requirements` (left column "Requirements") and `Methods` (column "Java") by adding a new association between them in the SUMM (right column "SUM(M)", marked in red) on the metamodel level (row "Metamodel Changes"). In the model (row "Model Changes"), one link between requirement R1 and method M1 is created in the SUM due to the keyword "register" (marked in red in row "1. Initialization"). Now the user does two changes the SUM (depicted in row "2. Run" and column "SUM(M)", in red), with following expected impacts due to the consistency rule:

1. The user creates a new and additional link between requirement R2 and method M2 in the SUM. Since these links conform to the new association which exists only in the SUM, no changes will appear the initial models for Java and Requirements.

2. The user renames the method M1 from "register" to "enrole" in the SUM. Since `Methods` are part of the initial model for Java, the method should be renamed also in this initial model (column "Java", marked in red). To realize that, the operator chain has to be executed in backward direction, here with the inverse operator `DeleteAssociation`. According to the consistency rule inside the model decision of `AddAssociation`, the SUM should be enriched by a link between requirement R2 and the renamed method M1 due to the "enrole" keyword (row "3. Run", column "SUM(M)", marked in red). At the same time, the existing link between R1 and M1 has to be deleted in the SUM, since there is no match regarding the methods name anymore. To realize that, the operator chain is executed again in forward direction, now using `AddAssociation`.

In general, to handle changes in the SUM, the operator chain has to be executed in backward direction using the inverse operators to update the initial model. Since also the operators in forward direction realize consistency rules, the operator chain has to be executed in forward direction afterwards. This rises the problem *not to loose information, which is persisted only in the SUM, but not in the initial models*: An example is the manually added link between R2 and M2. During the 2. Run, it is deleted by `DeleteAssociation`.

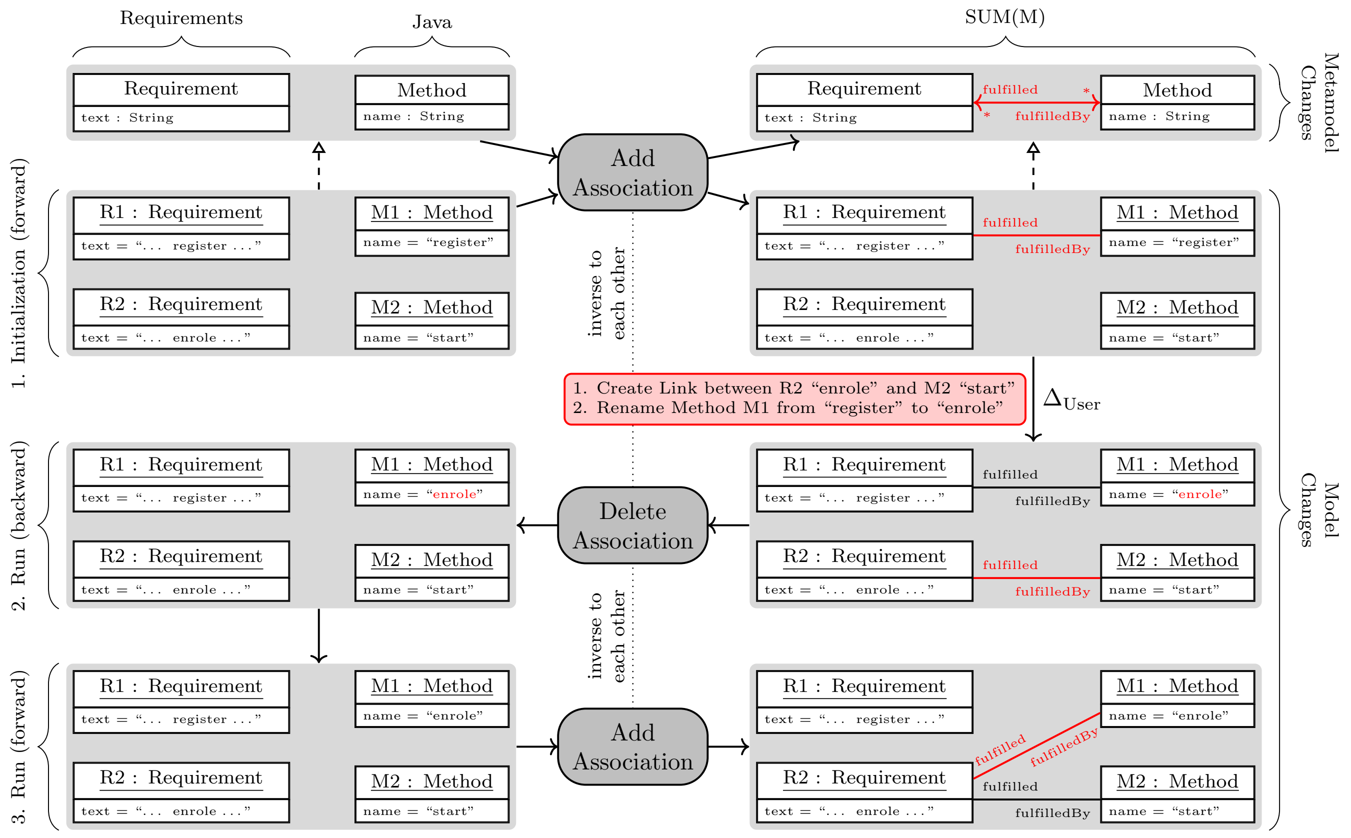


Figure 7: Operator `AddAssociation` ❶→❷ ensures Model Consistency during Consistency Assurance

During the 3. Run, is is not recreated directly, since this link does not base on a keyword triggering the consistency rule and is not contained within the initial models. To solve this problem, missing information which is stored only in the SUM and not in the current model has to be added again. This missing information is calculated by comparing the model differences which are recorded during the previous execution of `DeleteAssociation` (2. Run) and the current execution of `AddAssociation` (3. Run). In the example, this recreates the link between requirement R2 and method M2, which was added manually to the SUM and was temporary gone.

Additionally, it also recreates the link between R1 and M1, which should be deleted, since it was introduced automatically basing on the keyword "register", which was changed to "enrole" now. This problem is caused by the design of the operators to work with complete models and not only with model differences, which is required for the initial creation and clean-up of the SUM (Section 3.2). That means, the *operators react only on existing and added information, but not on changed and removed information*. To solve this problem, the execution of operators is more complex as suggested up to now: After executing `AddAssociation` the second time (row "3. Run") and recreating missing information (see paragraph before), the model differences of the current execution of `AddAssociation` (3. Run) are compared with the model differences of the previous execution of `AddAssociation` (1. Initialization). Changes of the previous run of this operator in this direction which are not created again by the current run of this operator, are not valid any more. Therefore, these differences have to be inverted. For this example, the old creation of that link between R1 and M1 leads now to a deletion of that inconsistent link.

In general, after executing the operator, adding missing information and reverting previous changes which are invalid now, the resulting model is consistent and complete, so that the next operator can be executed. Since the SUM contains all information, the step to add missing information is required only on the way from models to the SUM. In the end, *postconditions* for this use case are updated versions of initial models and the SUM.

4 APPLICATION

To solve the motivated consistency problems in the ongoing software development example, the methodologist selected and configured the operators depicted in Figure 3 according to Section 3.1. Linking requirements and methods with each other is supported by the operator `AddAssociation` ❶→❷, as described in more detail in Section 3.3. Since the two meta-classes describing data classes are merged by the operator `MergeClasses` ❺→❻ (detail in Section 3.2), possible inconsistencies between class diagram and sourcecode are fixed. Additionally, this prevents new inconsistencies in the SUM, because this information is stored only once.

The resulting SUMM is shown in Figure 8 and represents the complete content of the three initial metamodels shown in Figure 1. Additionally, it contains the new association between `Method` and `Requirement`, and the two classes `umlclasses.Class` and `asgjava.Class` are merged into `data.ClassType` which represents now data classes both in sourcecode and class diagrams.

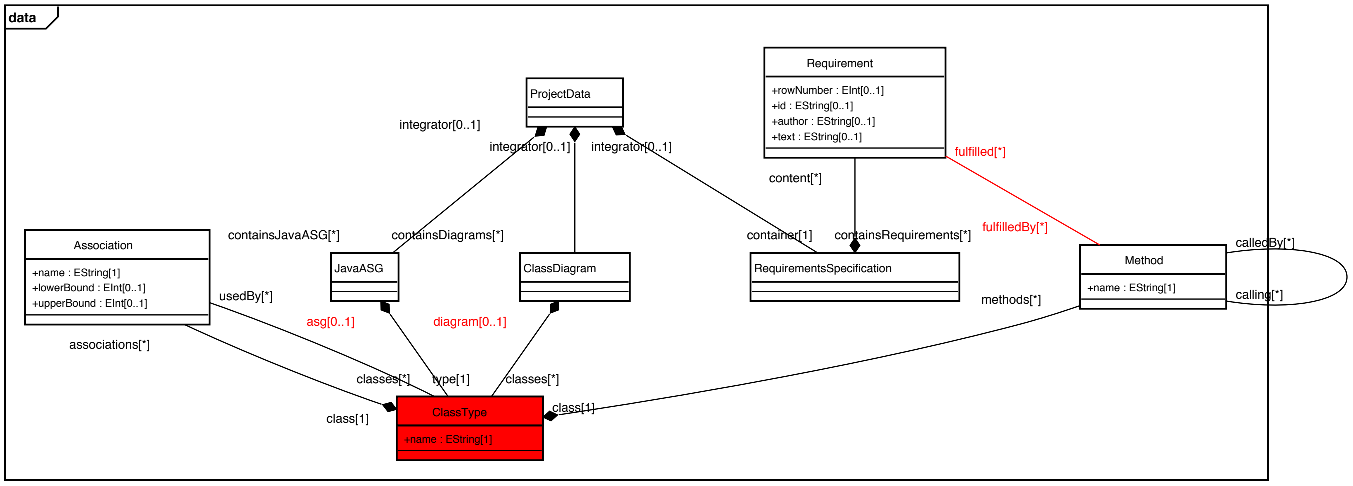
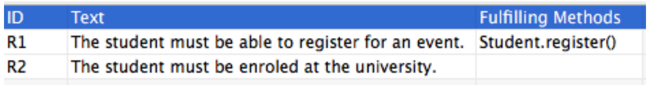


Figure 8: Single Underlying MetaModel (SUMM) integrating Requirements, Class Diagrams and Sourcecode

Now the user can change the models for Requirements, Java and ClassDiagram as well as the SUM: The SUM and all models will be updated and possible inconsistencies will be fixed according to Section 3.3, like discussed along Figure 7.

4.1 New Viewpoints

If the user want to change information which is new in the SUM and represented in none of the initial models, he has to change the SUM directly. Although that is possible as shown, it is not very feasible, since the SUM contains all information and not only the information the user is interested in. As example, the links between requirements and methods are neither contained in the requirements model nor in the Java model, but only in the SUM. Since the SUM contain also all information about classes and class diagrams, which are not required here, the user wants to use a *new view* like exemplary depicted in Figure 9: It lists all requirements



ID	Text	Fulfilling Methods
R1	The student must be able to register for an event.	Student.register()
R2	The student must be enroled at the university.	

Figure 9: New View to manage Requirement-Method-Links

together with their linked methods as subset of the SUM. Now, users can easily change links between requirements and methods.

Therefore, *new viewpoints* have to be specified by the methodologist showing parts of the SUMM, in particular, information stemming from more than one initial model. This new viewpoint is specified again by selecting and configuring some more operators on top of the SUMM, which defines an additional chain of operators to describe SUM(M)⟷New View(Point). The same set of operators can be reused and configured according to the concerns for the viewpoints here again. The users change the corresponding *new views* and the changes are synchronized with the SUM and all other models by executing the operators (Section 3.3).

4.2 Evolution of Metamodels

Up to now, only the evolution of models by user changes was discussed roughly in Section 3.3. Additionally, the metamodels of Requirements, Java and ClassDiagram as well as of the SUMM could change [9]. The methodologist handles this evolution of the metamodels by updating the existing operator chain accordingly.

Reasons for *changes in the SUMM* can be fixes for found bugs, refactorings or new information to store. As before, where the SUMM was created bottom-up in step-wise way by applying operators, these changes can be realized by selecting and configuring additional operators. Finally, the methodologist describes with operators the relationship SUMM⟷SUMM’, resulting in an explicit SUMM’ and further consistency rules.

Changes in the metamodels of the *initial artifacts* arise because of updates of the tools providing these metamodels or because of new versions for Java or UML. The realization effort for the methodologist depends on the amount of changes and its impact on already integrated structures: If the changes contain only additional elements in the metamodel without any additional consistency rules, the current operator chain stays the same. If the changes can be mapped to the old version of the metamodel, the methodologist can describe this mapping by additional operators, for example Java’⟷Java. More advanced changes require changes of the configured operator chain, which means in the worst case, that the complete integration has to be specified again.

4.3 Transferability

The approach was demonstrated using only one simplified example stemming from a software development project. This section discusses, why the complete approach is transferable to other software development projects and even to other application domains.

Different software development projects will have different consistency rules or different initial metamodels. This issue is handled by the methodologist, who specifies different chains of configured operators for different projects, with different SUMMs as result. The model decisions are substantiated regarding project-specific consistency rules. As example, in another project with the same set of initial metamodels as in the ongoing example, no links between requirements and methods should be created automatically ❶→❷: Then the methodologist specifies the corresponding model decision to create no links automatically. As contrast, arbitrary complex algorithms stemming from research of the requirements community can be included into model decisions. Another case, same consistency rules with slightly different initial metamodels, is discussed in Section 4.4.

Other application scenarios are traceability issues, solvable by this approach [10]: Traceability links between, for example, methods and requirements can be stored and maintained within the SUM by new associations, introduced by operators like `AddAssociation`.

The approach is intended to work even in application domains outside of software development. This is possible, since the operators to specify the integration are independent from concrete metamodels, because the metamodel decisions of the operators are designed to be generic for reuse and to be substantiated regarding the predefined degrees of freedom. Therefore, precondition for the approach are not specific application domains, but the representation of artifacts in form of models and conforming metamodels. This technical issue can be overcome by developing adapters realizing the transformation of artifacts into models and vice-versa.

4.4 Integration on Reference Level

The example integrates requirements and Java by supporting links between Java methods and textual requirements. If another project uses C++ instead of Java with the same consistency rules, the result is the same operator chain like in Figure 3, but with $\boxed{\text{C++}}$ instead of $\boxed{\text{Java}}$, since the current integration uses Java methods and not C++ methods. Now the methodologist spends again effort to realize the same consistency rules for similar models.

Instead of integrating concrete Java methods or C++ methods, Java methods and C++ methods are shifted to "reference" methods representing methods written in arbitrary programming languages. Instead of using concrete metamodels (CMM) for Java or C++, a reference metamodel (RMM) describing object-oriented general purpose programming languages is required. An example reference metamodel is the Dagstuhl Middle Model [7], because it describes only essential elements like packages, classes and methods, but ignores specific aspects like pointer handling.

Instead of integrating concrete metamodels (CMM) into the SUMM like in Figure 3, reference metamodels (RMM) are integrated into a Reference SUMM (RSUMM) using operators, resulting in a similar operator chain. Instead of linking textual requirements with Java methods, now arbitrary requirements are linked with methods of arbitrary programming languages. The methodologist describes the required mappings $\boxed{\text{CMM}}\longleftrightarrow\boxed{\text{RMM}}$ again by configuring operator chains with the CMM like Java as starting point and the RMM like the Dagstuhl Middle Model as end point.

To get the $\boxed{\text{SUMM}}$, for each RMM one mapped CMM is selected. Since the integration of the RSUMM is defined on reference level using the RMM, the integration is executable again using the mapping between CMM and RMM and the wanted SUMM is created automatically together with its consistency rules.

In general, for similar integration projects, the integration should be done once on reference level using reference metamodels and not on concrete level using concrete metamodels as before. Therefore, the integration on reference level eases the configuration of SUMMs for the methodologist compared to Section 3.1 [8], while their use for initializing the SUM (Section 3.2) and for ensuring model consistency (Section 3.3) is the same as for regularly created SUMMs for the users. This is another example, where the generic operators can be reused and applied even to different levels on abstraction.

5 CONCLUSION

To ensure the consistency of technical independent and separated models conforming to different metamodels, which are overlapping and have relations to each other contentwise, a new bottom-up approach was depicted in this paper. The central idea is to create a Single Underlying (Meta)Model containing all information of all initial (meta)models as single point-of-truth. Additionally, the initial (meta)models are not thrown away, but migrated to projectional view(points) on this SUM(M) and kept up-to-date. This is reached by introducing operators which, formed as chain, define, how the SUMM is created out of the initial metamodels, and which executes transformations to ensure consistency between all initial models and the SUM. Since these operators are generic regarding metamodel and model changes and support arbitrary consistency rules explicitly, these operators are reusable for new viewpoints on top of the SUMM, for integrations on reference level and for ensuring model consistency in different application domains.

A prototypical framework to support this approach for model consistency ensured by metamodel integration is currently under development with Java, ECore as language to describe metamodels, reuse of parts of the model migration structure of Eclipse EDapt and extension of coupled operators described by [4]. Currently, the configuration of the operator chain is supported by a Java API and model decisions are implemented in Java. In future work, both steps could be supported by domain specific languages.

REFERENCES

[1] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2009. Supporting View-Based Development through Orthographic Software Modeling. *Evaluation of Novel Approaches to Software Engineering (ENASE)* (2009), 71–86.
[2] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. 2014. View-based model-driven software development with ModelJoin. *Software & Systems Modeling* (2014).
[3] Mahmoud El Hamlaoui, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, and Adil Anwar. 2014. Heterogeneous models matching for consistency management. In *2014 Int. Conf. on Research Challenges in Information Science*. IEEE, 1–12.
[4] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. 2011. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. *Software Language Engineering* LNCS 6563 (2011), 163–182.
[5] IEEE. 2011. ISO/IEC/IEEE 42010:2011 - Systems and software engineering - Architecture description. 2011, March (2011), 1–46.
[6] Max E Kramer, Erik Burger, and Michael Langhammer. 2013. View-centric engineering with synchronized heterogeneous models. *1st VAO* (2013), 1–6.
[7] Timothy C. Lethbridge, Sander Tichelaar, and Erhard Ploedereder. 2004. The Dagstuhl Middle Metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science* 94, 1 (2004), 7–18.
[8] Johannes Meier and Andreas Winter. 2016. Towards Metamodel Integration Using Reference Metamodels. *4th Workshop VAO* (2016), 19–22.
[9] Johannes Meier and Andreas Winter. 2018. Towards Evolution Scenarios of Integrated Software Artifacts. *Softwaretechnik-Trends* 38, 2 (2018), 63–64.
[10] Johannes Meier and Andreas Winter. 2018. Traceability enabled by Metamodel Integration. *Softwaretechnik-Trends* 38, 1 (2018), 21–26.
[11] J. R. Romero, Juan Ignacio Jaén, and Antonio Vallecillo. 2009. Realizing correspondences in multi-viewpoint specifications. *EDOC 2009* (2009), 163–172.
[12] Andy Schürr and Felix Klar. 2008. 15 Years of triple graph grammars: Research challenges, new contributions, open problems. LNCS 5214 (2008), 411–425.