Single Underlying Models for Projectional, Multi-View Environments

Johannes Meier[1], Heiko Klare[2], Christian Tunjic[3]
Colin Atkinson[3], Erik Burger[2], Ralf Reussner[2], Andreas Winter[1]

[1]*Software Engineering Group, University of Oldenburg, Germany*

[2]*Software Design and Quality Group, Karlsruhe Institute of Technology, Germany*

[3]*Software Engineering Group, University of Mannheim, Germany*

[1]*{meier,winter}@se.uni-oldenburg.de,* [2]*{klare,burger,reussner}@kit.edu,* [3]*{tunjic,atkinson}@informatik.uni-mannheim.de*

Abstract: Multi-view environments provide different views of software systems optimized for different stakeholders. One way of ensuring consistency of overlapping and inter-dependent information contained in such views is to project them "on demand" from a Single Underlying Model (SUM). However, there are various ways of building and evolving such SUMs. This paper presents criteria to distinguish them, describes three archetypical approaches for building SUMs, and analyzes their advantages and disadvantages. From these criteria, guidelines for choosing which approach to use in specific application areas are derived.

1 Introduction

Due to the ever growing complexity of modern software-intensive systems single developers are no longer able to understand all aspects of a system as a whole. *View-based development* methods are therefore needed to separate system descriptions into individual parts that are relevant to the concerns and responsibilities of single developers. However, the resulting fragmentation of system descriptions leads to *redundancies* and *dependencies* between the information shown in different views, which are difficult and time consuming to manage manually. Automated approaches for ensuring the holistic consistency of multi-view system descriptions are therefore needed.

View-based approaches can be *synthetic* or *projective* (ISO/IEC/IEEE, 2011), depending on where information is stored. In synthetic approaches, the description of the system is spread over all the individual views, whereas in projective approaches, the description is contained in a Single Underlying Model (SUM) (Atkinson et al., 2010), and views are projected from this central store of information as needed. As with all models in model-driven development, a SUM conforms to its metamodel, the Single Underlying MetaModel (SUMM).

The goal of this paper is to illuminate different strategies for supporting projective approaches to view-based software engineering and to highlight their pros and cons. The common underlying property of all projective approaches is that views are considered to be correct by construction and thus inherently consistent with each other as long as they are consistent with the SUM. The problem of maintaining inter-view consistency therefore becomes the problem of maintaining the internal consistency of the SUM and the correctness of SUM-to-view projections. To describe the different approaches in a uniform way and analyze their pros and cons systematically, this paper classifies the different fundamental strategies for constructing SUMs and their corresponding SUMMs, and identifies criteria for evaluating them. More specifically, three existing approaches for constructing SUM(M)s are compared in terms of how they fulfill the identified criteria. Finally, we analyze how the fulfillment of the identified criteria by the different approaches affects their suitability for specific situations.

The insights presented in this paper will allow researchers to classify new approaches for SUM(M) constructions and help developers to choose projectional view-based approaches for their specific project situations using the identified selection criteria.

After introducing a running example and terminology used in this paper in Section 2, classification criteria for SUM approaches are described in Section 3. The three SUM approaches OSM (Section 4), VITRUVIUS (Section 5), and MoConseMI (Section 6) are presented subsequently and are classified using the criteria in Section 7. From this clas-

sification, guidelines for deciding which approach to choose when are derived in Section 8. After discussing related work in Section 9, Section 10 summarizes the findings of this paper.

2 Terminology & Running Example

To motivate the use of several interconnected views in the development of a system, we introduce a highly simplified running example describing the requirements, architecture and implementation of a system. These three views are expressed in *languages* based on metamodels that define the elements (e.g., classes, attributes etc.) that can appear in models. We depict those metamodels in Figure 1. This section also clarifies the terminology used in this paper.

Requirements are represented by natural language sentences (package `Req`). Each `Requirement` within a `RequirementsSpecification` is identified by a unique `id`, which contains the requirement's sentence as simple `text`, and is written by an `author`.

The *architecture* is described by simplified *class diagrams*, which represent system modules as classes (package `UML`). `ClassDiagrams` contain `Classes` with their `className` and unidirectional `Associations`.

The *implementation* realizing the architecture and requirements is represented by *source code* developed in simplified Java (package `Java`). The `JavaASG` contains `ClassTypes`, which in turn contain `Methods` with their caller/callee relations.

These three languages describe different (not necessarily all) facets of the system under development and thus represent three overlapping *viewtypes*. According to Goldschmidt et al. (2012), a *viewtype* is the metamodel of a view, while a *view* is a model that projects information from another model (here: the SUM) for a specific purpose. Since all views share information about the system under development, they are semantically interconnected and contain *dependent information*, which requires updates of other views if one is changed. The interdependence of information can be explicitly defined in *consistency rules*, which define the relations that have to hold be-

tween instances of metamodels.

We define two exemplary consistency rules for the running example: While Consistency Rule 1 covers a situation where existing redundant information needs to be kept consistent, Consistency Rule 2 addresses a different problem of introducing additional information depending on other information. We consider these consistency rules representative, since integrating different views usually refers to merging concepts or introducing additional associations.

Consistency Rule 1: Classes can be defined in the architecture view and in the implementation view: One concrete class can be defined either only in the implementation (`Java.ClassType`), or in both implementation and the architecture (`UML.Class`) if it represents a module. In the latter case, this class has to be kept consistent in the implementation and architecture, e.g., in the case of renaming this class. Therefore, the implementation and architecture are only consistent if the architecture contains a subset of the classes in the implementation.

Consistency Rule 2: Since requirements define goals that the implementation should fulfill, the development progress can be measured by counting the requirements that are supported by the current implementation. Therefore, `Requirements` must be linked to the implementing `Methods`. We thus require that each `Method` has to be automatically linked to those `Requirements` that contain the `Method`'s name in their text. This *additional information* between requirements and implementation has to be stored and kept consistent. Since this is a simplified example for this paper, different rules can be specified instead.

These two consistency rules and three languages are used to motivate criteria for SUM approaches in Section 3. SUM approaches define how SUMs as well as their SUMMs are constructed and are designed by *platform specialist*s, who develop platforms that support SUM-based development. Three such platforms are presented in Sections 4–6 and applied to this running example by a *methodologist*, who uses a SUM platform to define a concrete SUMM to support a particular view-based method (Atkinson et al., 2010).

Depending on the approach, to create the SUMM the methodologist either reuses the existing metamod-
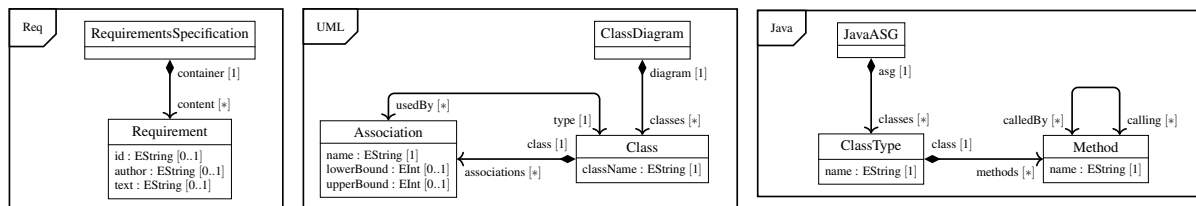

Figure 1: Simplified Metamodels for Requirements (left), Class Diagrams (middle), and Java Source Code (right)

els in Figure 1 or defines a new metamodel for the described concepts. After that, the *developer* instantiates that SUMM defined by the methodologist to develop a software system through views projected from the SUM. The methodologist can also specify *new viewtypes* to provide new views corresponding to the specific concerns of the developer.

3 Classification Criteria

In this section, we identify two groups of criteria for classifying SUM approaches according to how they can be designed once (i.e., the nature of the construction process and resulting SUM(M)) and how they should be selected for various applications (i.e., which approach is best in which context). This list of criteria is the first contribution of this paper. By applying them to different SUM approaches, we evaluate indicators for their appropriateness in Section 7.

3.1 Design Criteria

Design criteria distinguish SUM approaches from each other at conceptual level regarding the structure of created SUM(M)s and their construction process. These criteria are independent from technical design decisions. The goal of this set of criteria is to span the complete solution space of possible SUM approaches. The criteria are not evaluative but rather distinguishing. In other words, the fulfillment of a criterion by an approach does not have implications on whether it is favorable over another approach. They inform *platform specialists* about the possible conceptual degrees of freedom when *designing a SUM approach*.

Criterion C1 (**Construction Process**) covers the *process* of creating a SUM(M) depending on the starting situation. In a *top-down* development approach, a new SUM, and especially its SUMM, is created from scratch. A *bottom-up* approach starts with already existing models and metamodels, which have to be combined into a SUMM and initial SUM.

Criterion C2 (**Pureness**) relates to the absence of internal redundancy in the SUM under construction. An *essential* SUM is "completely free of any internal redundancy" (Atkinson et al., 2015) and dependencies by design. A *pragmatic* SUM contains redundant information (e.g., because it contains different metamodels that define concepts more than once) that has to be interrelated and kept consistent, and thus only behave as if it was free of dependencies due to internal consistency preservation mechanisms. Pragmatic SUMs require additional information to wire the internal models together and thus involve more complex

consistency rules than equivalent essential SUMs.

While **C1** focuses on the *starting point* of the SUM construction process, **C2** focuses on the *results*. Together they allow SUM approaches to be compared at conceptual level.

3.2 Selection Criteria

Selection criteria support the selection of the most appropriate SUM approach for a particular project. This set of criteria addresses the conceptual preconditions and requirements that favor one SUM approach over another in a specific situation. These criteria help *methodologists* to *compare different SUM approaches* for the same application scenario. For example, if existing metamodels need to be reused, it is best to *select and apply* a SUM approach that simplifies the reuse of existing metamodels. The fulfillment of those criteria by a specific SUM approach is affected by the allocation of design criteria for that approach.

Criterion E1 (**Metamodel Reusability**) determines whether concepts to be represented in the SUMM are already available within predefined metamodels and should be reused in the new SUMM. If so, the SUM approach has to accommodate these legacy metamodels by combining them into an initial SUMM. This can either be done directly without additional work or indirectly by providing strategies for migrating the legacy metamodels into the SUMM. Since lots of languages, metamodels and tools with fixed viewtypes are usually already available, approaches fulfilling this criterion support their reuse. Reusing metamodels usually implies a bottom-up approach according to **C1**.

Criterion E2 (**Model Reusability**) establishes whether already existing artifacts (i.e., existing instances of the metamodels to be integrated) need to be incorporated in an initial version of the SUM. If so, the SUM approach has to import these models. This can be done either directly without additional work or indirectly by providing a strategy for migrating the legacy models into views of the SUM or directly into the SUM by some kind of model-to-model transformations. It requires the reuse of the corresponding initial metamodels according to **E1** and usually requires a bottom-up strategy according to **C1**. Reusing models may require that models have to be consistent according to the consistency relations between the integrated metamodels *before* they are integrated into the SUM. This requires additional manual effort to ensure consistency beforehand, in contrast to SUM approaches which offer strategies to handle inconsistent information *during* their integration into the SUM. Existing artifacts developed without a

consistency-preserving SUM approach usually do not initially fulfill the consistency relations, which is why this criterion also checks whether those inconsistencies can be handled automatically during integration.

Criterion E3 (Viewtype Definability) focuses on the task of specifying new types of views on a SUMM. This involves the creation of new viewtype definitions, focused on specific concerns (e.g., managing the traceability links from Consistency Rule 2) whose instances can be used by developers to change the related information in the SUM. Supporting the definition of customized, role-specific viewtypes is an essential capability of view-based development approaches, so the level of difficulty involved has a strong impact on the usability of an approach.

Criterion E4 (Language Evolvability) focuses on the task of maintaining the SUMM in the face of evolved language concepts represented in their metamodels, changed consistency rules, and the integration of new viewtypes. Changes in the metamodel can require corresponding changes in the model (i.e., model co-evolution (Herrmannsdoerfer et al., 2011)) as well as the creation or adaptation of consistency rules. Since languages are subject to change (e.g., new version of Java are regularly introduced) the difficulty of updating the SUMM and its instances after evolution of the integration languages is a relevant criterion, whose importance depends on the probability of languages to evolve in a concrete setup.

Criterion E5 (SUMM Reusability) focuses on the question of whether only a subset of the integrated metamodels and their consistency rules from one project can be reused to construct a SUMM for other projects, or if a SUMM can only be reused as a whole. Additionally, this criterion addresses the amount of effort involved in adding new metamodels to an already existing SUMM. Although this criterion does not target reuse at the model level, it is important since, for example, there are many software development projects that use slightly different languages or consistency rules, which need to be managed.

4 Orthographic Software Modeling

Orthographic Software Modeling (OSM) is a view-based approach, initially developed to support multi-perspective software development (Atkinson et al., 2010) but can be applied to other domains like enterprise architecture modeling (Tunjic et al., 2018) to support methods like Zachman (Zachman, 1987).
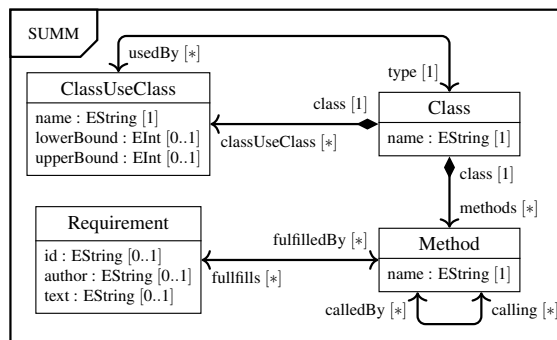

Figure 2: Exemplary Metamodel for SUM in OSM

4.1 Main Ideas

The OSM approach is inspired by the orthographic projection technique used to visualize physical objects in CAD systems. OSM utilizes this principle to define "orthogonal" views on a system under development that present each stakeholder, such as software engineers, with the data he needs in a domain-specific notation. Although stakeholders can only see and manipulate the system via views, the actual description of the system is stored in a SUM. The views are defined to be as "orthogonal" as possible through independent dimensions (i.e., concerns) ranging from behavioral properties and feature specifications to architectural composition. Ultimately, the system description in the SUM can be made formal enough to be automatically deployed and executed on appropriate platforms, thus allowing automatic redeployment when changes occur. In order to support the complete life-cycle of a system, ranging from requirements analysis to deployment, the internal structure of the SUM must be able to store all required data in a clean and uniform way. The data in the SUM should thus be free from dependencies and capture all relationships between its inner elements in a redundancy-free way using approaches like *Information Compression* and *Information Expansion* (Atkinson et al., 2015).

Figure 2 shows an example of an OSM-oriented SUMM corresponding to the information presented in Figure 1. Since a fundamental tenet of the OSM approach is to have a *pure and optimized SUMM*, it is usually created manually from scratch based on the needed viewtypes and concerns of the involved stakeholders. Figure 2 is a reduced version of Figure 1 in which all redundant information, and thus the correspondences that connect duplicate stores of data, have been manually removed. Thus, for example, the two equivalent elements `ClassType` and `Class` have been compressed into one concept `Class` in Figure 2. This is possible because although the two concepts define their own properties for their own contexts, and use

different names (i.e., `name` and `className`), they are in fact equivalent and can be combined. Both attributes are therefore mapped to the single attribute `name` in the SUMM. The two dependencies are distinct, however, and are hence both added to the `Class` element: The first enables `Classes` to have `Methods`, while the second describes dependencies between two `Class` elements (Consistency Rule 1).

Consistency Rule 2 is modelled through a relationship between `Requirement` and `Method`, denoting that the requirement is being fulfilled by the method. In order to allow developers to create instances of the relationship, a new view can be defined, which at least contains the concepts `Requirement`, `Method` and the relationship between these two.

The data structure shown in Figure 2 is simpler and more optimized than the disparate representation in Figure 1. This is achieved by the unified names for dependent concepts (`ClassType` vs. `Class`) and usage of names with more meaning (`Association` vs. `ClassUseClass`). Although the SUMM is built from scratch in the presented example, in principle it is possible to import existing artifacts into the environment using model-to-model transformations.

4.2 Process of Application

In order to make use of OSM, an environment has to be developed which realizes its goals and principles. Both steps, i.e., the definition of the approach and the implementation of a framework which supports the concepts of the approach, are performed by a *platform specialist*. The work involves the development of a framework which can be customized for the used methodology (e.g., KobrA (Atkinson, 2002), MEMO (Frank, 2002), ArchiMate (Iacob et al., 2012)) and targeted domain (e.g., software engineering, enterprise architecture modeling). The different configurations can be reused for projects in the same domain and the same methodology. Tunjic et al. (2018) present a metamodel which is used by the current prototype implementation to support the configuration of OSM environments. In particular, it facilities the configuration of the SUMM and viewtypes, and their integration in a dimension-based view navigation approach using hyper-cubes of the kind used in OLAP (Codd et al., 1993) systems.

The customization of the environment for a specific domain and methodology is performed by a software engineer playing the role of a *methodologist*. In order to be able to configure and customize the environment according to the requirements, the methodologist must have knowledge of the involved domain and the OSM environment. In particular, he is respon-

sible for defining the SUMM and the viewtypes in a way that adheres to the principles of redundancy-freeness and minimality. Defining a viewtype involves the definition of a suitable metamodel as well as a model transformation that maps the concepts from the SUM to those in a view and vice versa. The resulting configuration can be stored in the tooling environment in order to be reused in other projects.

Once a complete configuration of an OSM environment has been defined by the *methodologist*, one or more *developers* can use it to develop a specific system specification. To this end, either an empty SUM is created to start a project from scratch, or existing content is imported into the SUM using model-to-model transformations from external artifacts. When using the OSM platform to develop a system, developers are able to access views using the dimension-based view navigation approach and use them to see and update information from the SUM.

5 Vitruvius

The Vitruvius approach (Kramer et al., 2013) assumes the existence of metamodels that are reused and integrated into a so called *virtual SUMM (V-SUMM)* rather than the development of a new SUMM from scratch. In other words, it focuses on building a pragmatic SUMM in a bottom-up fashion.

5.1 Main Ideas

The Vitruvius approach is based on the projectional SUM idea of the OSM approach. The whole system description is encapsulated in a SUM and only projectional views can be used to modify information in the SUM. Instead of creating a completely new SUMM without dependent information, however, Vitruvius follows a pragmatic approach by coupling existing metamodels using consistency preservation rules (CPRs), which define how consistency is preserved after modifications. The CPRs use and modify *correspondences*, which reference model elements that represent dependent information and can be seen as a trace model. The set of metamodels with their CPRs defines a virtual SUMM (V-SUMM), while instances of them with an actual model of correspondences are denoted as V-SUMs. These CPRs make dependencies between metamodels explicit and ensure that after modifications in one model, all other dependent models are updated consistently. As a consequence, a V-SUM behaves completely like an ordinary SUM since it provides the same guarantees regarding information consistency.
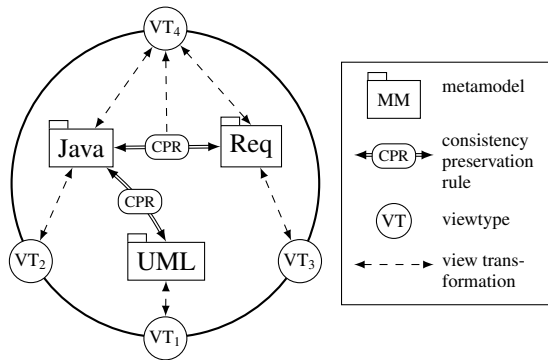
Figure 3: Example V-SUMM in VITRUVIUS

An exemplary V-SUMM for the metamodels from Figure 1 is depicted in Figure 3. It consists of the reused metamodels with CPRs defined between them. For Consistency Rule 1, a CPR defines the creation of a Java class `ClassType` in reaction to the creation of a UML class `Class`. It is up to the methodologist to specify the behavior in the other direction, i.e., whether a UML class is created for a Java class or if the developer shall be asked what to do. Additionally, it propagates all changes on the `name` or `className` to the respective other model. The additional requirements traces in Consistency Rule 2 can be expressed by identifying matching requirements and methods after adding or modifying methods as well as requirements, and by storing them as appropriate correspondences in the existing trace model. Alternatively it is possible to define an additional metamodel that defines links between requirements and methods, which is modified whenever they are changed.

Two types of projectional viewtypes can be defined on a V-SUMM. First, the original viewtypes defined for the existing metamodels, such as a textual editor for Java or a graphical editor for UML, can be reused. In Figure 3, these viewtypes are VT_1, VT_2 and VT_3, which represent the original metamodels from Figure 1. Second, it is also possible to define additional viewtypes that may combine information from different metamodels and their relations defined in the CPRs. Figure 3 contains VT_4, which displays the trace information for Consistency Rule 2 by extracting information from the Java and the requirements model, as well as from the correspondences generated by the CPR. Concretely, this viewtype could, for example, show the Java code with annotations attached to the methods that show the requirements they fulfill. Nevertheless, for defining such viewtypes, specialized languages that support the projection from, and combination of, different metamodels is required. In VITRUVIUS, those can be expressed with the ModelJoin language (Burger et al., 2014).

Consistency preservation in VITRUVIUS is performed in a delta-based manner. In contrast to state-based consistency preservation, edit operations are tracked instead of comparing two model states, which results in less information loss (Diskin et al., 2011). For example, a state-based approach can distinguish the deletion and creation of an element from its renaming only using heuristics, whereas delta-based approaches track the correct operations. To define such delta-based consistency preservation, specific consistency preservation languages have been developed (Kramer, 2017). Initial investigations into consistency preservation in VITRUVIUS have been made on a case study of component-based architectures, Java code and code contracts (Kramer et al., 2015).

5.2 Process of Application

The development of frameworks such as VITRUVIUS first involves a *platform specialist* who defines an abstraction representing the interface of a V-SUM, implements the logic for executing CPRs and defines or selects specific languages or at least an interface to define CPRs. The current implementation of the VITRUVIUS approach (http://vitruv.tools) uses Ecore and contains a Java-based definition of V-SUMs and provides two languages for defining consistency preservation on different abstraction levels.

The *methodologist* then selects a set of metamodels and defines or reuses existing CPRs for the selected metamodel to define a V-SUMM of these artifacts. Finally, the *developer* can instantiate the V-SUMM, derive views according to existing or newly defined viewtypes, and perform modifications of them. Modifications in a view are recorded as sequences of atomic change events (creation, deletion, insertion, removal and replacement) and then sequentially applied to the V-SUM. For each of these changes, the responsible CPRs are executed to restore consistency after each modification, which results in an *inductively consistent* V-SUM.

6 MOCONSEMI

MOCONSEMI (MOdel CONSistency Ensured by Metamodel Integration, (Meier & Winter, 2018)) combines major features of the other two SUM approaches, i.e., creating one SUM by operator-based transformations with reusing existing (meta)models as starting point.
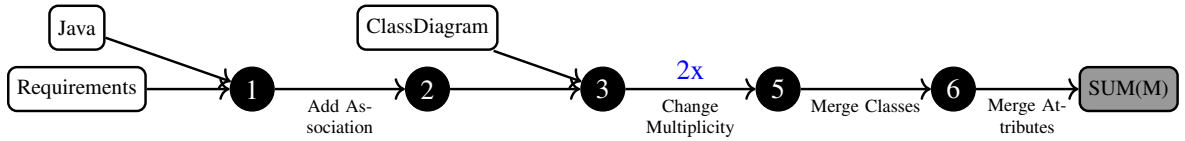
Figure 4: Chain of Configured Operators for Integrating Textual Requirements, Class Diagrams, and Java into a SUM(M)

6.1 Main Ideas

MoConseMI is a SUM approach which starts with existing initial models and conforming metamodels (exemplarily shown in Figure 1) and creates a SUM(M) as suggested by Atkinson et al. (2010). In practice, many models and metamodels already exist in form of DSLs and tools with fixed data schemas. To reuse them even in integrated form, these initial models and metamodels are reused for the integration and the models are kept in sync as views.

The integration of initial (meta)models from the running example in Figure 1 into a SUM(M) resulting in Figure 5 is described by a chain of operators, as depicted in Figure 4. To form the SUM(M) out of the initial (meta)models, theses operators change the current (meta)model in a step-wise way. Starting with the initial model and metamodel representing Requirements, the initial (meta)models for Java and ClassDiagrams are included technically at ❶ and ❸, which require `ProjectData` and its compositions as container without any contentwise integration.

The first operator `AddAssociation` is used to fulfill Consistency Rule 2. To enable the desired traceability links between requirements and methods, a new association between `Requirement` and `Method` is required and created by the operator. In the model, links can be added for this new association. This is also done by the operator corresponding to a decision to control this model change. This ensures that a method is linked with those requirements that contain the name of the method in their requirements text.

Consistency Rule 1 is realized in `MergeClasses` ❺→❻ after the operator `ChangeMultiplicity` is applied twice ❺ as preparation, because the two classes `Class` (from UML) and `ClassType` (from Java) are merged into one single class representing data classes both in UML and Java at the same time. The instances are merged in the same way supported by the model decision that the same instances are identified by same values for `Class.className` and `ClassType.name` specified by the methodologist. As a result of this merge, redundant information is removed from the current (meta)model. The operator `MergeAttributes` is a follow-up treatment, after which the methodologist decided that the integration is done. The last stable model and metamodel are used as the SUM(M), for which Figure 5 marks the contentwise changes in red compared to the initial metamodels in Figure 1.

Summarizing, each operator performs small changes on the current metamodel (e.g., adds a new association) controlled by metamodel decisions (e.g., multiplicities, source and target class of the new association). The operator also changes the current model to keep it consistent to the changed metamodel for model co-evolution (Herrmannsdoerfer et al., 2011). Degrees of freedom of this change are influenced by model decisions, which allow consistency rules to be fulfilled (e.g., specify, when new links should be added). Result is one valid intermediate model conforming to one valid metamodel represented by ❶. To keep the initial models up-to-date, changes in the SUM have to be propagated back to them, which requires operators to be executed backwards. Therefore, each operator is combined with an inverse operator, e.g., `DeleteAssociation` for `AddAssociation`.

In the end, the same operator chain describes the SUMM by collecting the metamodel changes, creates the initial SUM by executing the operators at model
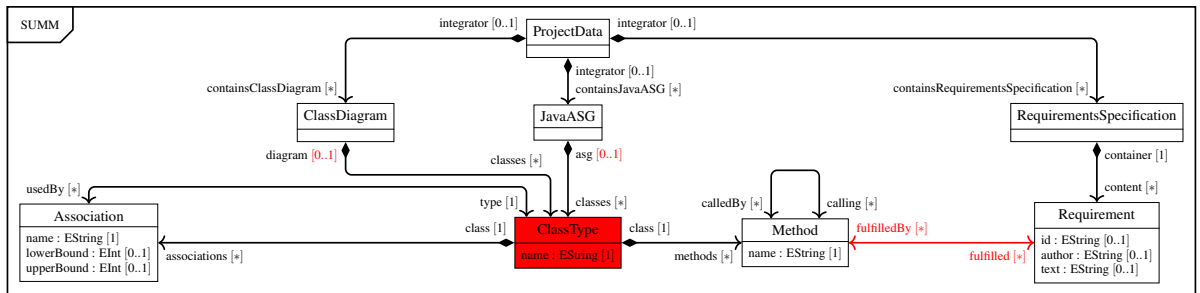

Figure 5: SUMM with Integrated Textual Requirements, Class Diagrams and Java

level reusing the initial models, and ensures consistency between all models by executing operators in both directions. The SUM and SUMM both exist and the SUM is directly usable as a first new view for developers. The SUM is used as a single point-of-truth, from which all initial and new views can be generated.

6.2 Process of Application

The approach and a supporting framework are developed once by the *platform specialist*. This includes the design and the implementation of the operators (currently 20 including inverse ones). The framework is under development using Java and a subset of Ecore, reusing parts of Eclipse EDapt (Herrmannsdoerfer, 2010), and extending some coupled operators (Herrmannsdoerfer et al., 2011).

After that, the *methodologist* creates a chain of operators like in Figure 4 individually for each project by reusing the provided operators and configuring them regarding the specific consistency rules. Additionally, the operators can be used to define new viewtypes on top of the SUMM.

Since the initial (meta)models are "migrated" to view(type)s on the SUM(M), the *developer* can change them as well as the SUM and the newly defined views. These changes are propagated automatically to all other models by executing the operator chain in forward and backward directions, which ensures consistency to the developer's model decisions.

7 Classification of Approaches

This section classifies the three presented SUM approaches regarding the criteria presented in Section 3 as second contribution of this paper. The classification serves both as a comparison of the three approaches, as well as an indicator of the appropriateness of the developed criteria to distinguish SUM approaches. Table 1 summarizes the classification.

7.1 Design Criteria

The overall development strategy, *top-down vs. bottom-up* (**C1**), relates to whether existing models and metamodels are used as starting point. OSM works top-down by creating an essential SUM(M), which, although technically independent, may be conceptually based on existing (meta)models. VITRUVIUS and MOCONSEMI, on the other hand, operate bottom-up, because they are able to reuse initial models and metamodels. While VITRUVIUS keeps them

unchanged inside the modular SUM(M), MOCONSEMI transforms them into an essential SUM(M).

All approaches can lead to either an *essential* or *pragmatic* SUM(M) in terms of **C2**. OSM is designed to have an essential SUM(M) without any internal dependencies. VITRUVIUS is designed to use a pragmatic SUM(M), since it incorporates all initial models with overlapping and, in general, dependent information. MOCONSEMI starts pragmatically, combining all initial models into one, but gradually moves towards an essential SUM(M) by removing dependent information or keeping them consistent through operator application. In special cases, this can lead to an essential SUM(M) without dependent information.

7.2 Selection Criteria

Metamodel Reusability (**E1**) requires that a preexisting set of metamodels and tools is reused to generate the SUMM. OSM supports this feature only conceptually ("hard"), because engineers can always informally draw upon the information contained in existing metamodels when constructing the essential SUMM, either manually or by model-to-model transformations, but this is not a formal part of the approach. VITRUVIUS supports this feature directly ("easy"), because it reuses and keeps the initial metamodels as internal parts of the modular SUMM, but depends on additional logic that builds the consistency preservation mechanisms. MOCONSEMI supports this feature by using the initial metamodels as the starting point for the subsequent transformations into a purer SUMM ("easy"). In general, only bottom-up approaches (**C1**) can easily fulfill **E1**, because in top-down approaches a new metamodel has to be defined, which hampers metamodel reuse.

Model Reusability (**E2**) requires preexisting models to be incorporated into the initial SUM. OSM supports this feature in a semi-automatic way ("hard") by importing data from existing models into the newly constructed SUM using model transformations. Although models do not need to be initially consistent, the transformations have to be defined such that they generate consistent output. VITRUVIUS supports this feature partially ("middle"), because it reuses and keeps the initial models as internal parts of the modular SUM. Nevertheless, this requires the reused models to be consistent according to the consistency rules between the metamodels. This can require high manual effort for the initial integration. MOCONSEMI supports this feature by using the initial models as the starting point for subsequent transformations that create a purer SUM ("easy"). Even if models are not consistent before, the application of operators to inte-

Criterion		OSM	VITRUVIUS	MOCONSEMI
C1	Construction Process	top-down	bottom-up	bottom-up
C2	Pureness	essential	pragmatic	pragmatic → essential
E1	Metamodel Reusability	hard	easy	easy
E2	Model Reusability	hard	middle	easy
E3	Viewtype Definability	easy	hard	middle
E4	Language Evolvability	middle	easy	middle
E5	SUMM Reusability	middle	easy	middle

Table 1: Comparison of the three Approaches regarding Design Criteria and Selection Criteria

grate the models can handle and fix inconsistencies. In general, only bottom-up approaches (**C1**) can easily fulfill **E2**, but the effort for reuse highly depends on the necessity to have initially consistent models.

Viewtype Definability (**E3**) deals with the specification of new viewtypes for the SUMM by methodologists. OSM eases viewtype definition ("easy"), because it provides an essential SUMM containing all concepts in an integrated, redundancy-free way. VITRUVIUS makes viewtype definition more difficult ("hard"), because information is spread across metamodels and has to be combined referring to the defined consistency relations or using specialized languages. MOCONSEMI eases viewtype definition in contrast to VITRUVIUS, but requires slightly more effort than OSM, especially if the SUMM is not essential ("middle"). Rather than arbitrary transformations usable in the OSM approach, the operators for integration are also used for specifying new viewtypes. In general, **E3** is directly influenced by the pureness of an approach (**C2**), as pragmatic approaches always have to deal with the problem of fragmented information, whereas essential approaches provide the necessary information in a minimalistic way.

Language Evolvability (**E4**) deals with maintaining the SUMM in the face of changes to the integrated metamodel elements or the consistency rules. OSM simplifies SUMM evolution, since it is free of redundant information, but has to check that the changes keep the SUMM essential. However, the transformations that generate views to the SUM have to be updated manually to stay up-to-date with SUMM changes ("middle"). VITRUVIUS simplifies metamodel evolution, since the initial metamodels are kept unchanged as sub-parts of the modular SUMM. Therefore, metamodels can evolve directly ("easy"). Additionally, the consistency preservation rules targeting the changed metamodel have to be checked and fixed if required, as well as the defined viewtypes which depend on the effected metamodels. MOCONSEMI supports metamodel evolution, but the effort depends strongly on the kind of change ("middle"),

which is true also for the other approaches to some less degree. If changes in the initial metamodels can be realized by describing the difference between the old and new metamodel version by a chain of operators, the existing operator chain must only be extended by them. The same applies for changed consistency rules. In all other cases, some of the existing operators have to be changed. In general, metamodel evolution is easier to realize in pragmatic approaches (**C2**), since the SUMM is constructed out of existing metamodels, leading to a formal relation between them. A drawback of having dependencies is that their consistency must be preserved after language evolution. On the other hand, in essential approaches the relations between the existing artifacts and the SUMM only exist conceptually. Thus, changes in existing languages must be manually transferred to the SUMM ensuring its redundancy-freeness and minimality, leading to high effort and error potential.

SUMM Reusability (**E5**) addresses the challenge of adding new metamodel elements to or removing some of the already integrated metamodels from the existing SUMM to reuse the SUMM in a different context. Therefore, no models are reused and model-co-evolution is not needed here. OSM makes it easy to add new concepts to an existing SUMM, since they can be inserted directly into the existing structure where they are needed. However, redundancy-freeness must be preserved and removing parts of the SUMM requires related concepts to be checked to ensure consistency ("middle"). VITRUVIUS makes it easy to add a new metamodel "as is" by adding it to the modular SUMM and specifying its consistency to the already integrated metamodels ("easy"). Removing an integrated metamodel works vice versa. Reusing subsets of a SUMM (i.e., a subset of the used metamodels) is easy, since selected parts of metamodels along with their consistency preservation rules can simply be reused. MOCONSEMI makes it easy to add new metamodels by defining a new operator chain that starts with the current SUMM and reuses all existing operators. Removing an already integrated

metamodel requires all operators between the meta-model in question and the SUMM to be removed or fixed ("middle"). Generally, pragmatic approaches (**C2**) allow to easily add or remove metamodels, since these operations are performed on the level of the metamodel as a whole. Additionally, they tend to lead to SUMMs that reflect the structure of the original metamodels, making them easier to remove later. On the other hand, essential approaches can easily fine-tune metamodels to the needs of the project, since the SUMM can be manipulated at the level of individual model elements. However, the absence of dependencies intertwines information and makes the boundaries between the different metamodels less clear.

The application of the classification criteria to the three approaches has shown that all criteria distinguish different properties of SUM approaches, because none of the criteria is fulfilled by all approaches in the same way. Nevertheless, correlations between the reusability of metamodels and models, as well as between the evolution of languages and the reusability of SUMM can be seen, as they all arise from the same conceptual criteria regarding pureness and construction process. From this application of the criteria, it cannot be said that the list of criteria is complete and especially it is unclear whether those presented criteria are the most relevant for selecting an appropriate SUM approach. Nevertheless, we have argued that all these criteria are relevant for certain situations (e.g., whether metamodels shall be reused or not, whether languages can be expected to evolve), which gives an initial indicator for the appropriateness of the criteria.

8 Guideline for Approach Selection

We defined criteria for selecting a SUM approach that is most suitable for a specific situation in Section 3.2, which are derived from the design criteria in Section 3.1. Since the three presented approaches fulfill these selection criteria differently, each fits well for different situations as discussed in the following as third contribution of this paper.

If there are no legacy tools or metamodels describing the system under development to be reused (**E1** and **E2**), the OSM approach is the most suitable. As there is no pressure to reuse existing metamodels, models or tools, defining a new metamodel that is free of implicit dependencies provides the purest solution for describing the system. This makes it most easy to define new viewtypes for specific roles (**E3**). The approach is also the most attractive when dependencies to external tool vendors should be avoided.

If, on the other hand, existing metamodels and tooling are available for reuse, the top-down OSM approach is less suitable in contrast to bottom-up approaches (**C1**) like VITRUVIUS and MOCONSEMI, because they preserve existing viewtypes, compatibility to existing tooling and potentially complete development environments including all instances. Moreover, they achieve this without the need to remodel all dependency-free information and without the corresponding loss of compatibility to existing viewtypes and tooling. VITRUVIUS is the most suitable approach if there are no existing models to be integrated for reuse, because it provides the highest reusability of SUMMs (**E5**) and the best support for evolution (**E4**), since the initial (meta)models are contained in the SUM(M) as separated artifacts. This also allows the modular specification of CPRs by domain experts, their reuse across projects, and the project-specific selection of used metamodels and CPRs.

However, VITRUVIUS is less suitable than MOCONSEMI when existing models need to be reused, for example, in a re-engineering case, as it requires the reused models to be consistent according to the consistency rules (**E2**). When models do not follow these consistency rules, which is especially the case when they are less obvious than those between Java and UML, they have to be adapted initially. The MOCONSEMI approach is most suitable in this case, because it is able to handle inconsistencies in the existing models and resolves them during integration.

Summarizing, if the reuse of existing tools and metamodels is not required, creating an essential SUM(M), as in OSM, is the most suitable solution. If metamodels are to be reused, pragmatic approaches are more suitable. Depending on whether existing instances shall be reused, VITRUVIUS or a combined approach such as MOCONSEMI should be taken. Table 2 summarizes the main advantages and application areas of the different approaches.

9 Related Work

The explicit use of views or perspectives in software engineering can be traced back to the VOSE method in the early 1990s (Finkelstein et al., 1992), which strongly advocated a synthetic approach to views given the state-of-the-art at the time. Most "view-based" software engineering methods that have emerged since then, such as by Kruchten (1995) or the Unified Process (Larman, 2004), assume that views are supported in a synthetic way, although this is usually not stated explicitly (the actual distinction between synthetic and projective approaches to views was first clearly articulated in the ISO 42010 standard (ISO/IEC/IEEE, 2011)). To our knowledge, no general purpose software engineering method avail-

	OSM	**VITRUVIUS**	**MOCONSEMI**
Advantages	Easy Viewtype Definition No Dependencies to Legacy Tools	Reuse of Metamodels / Tools Modular Views	Reuse of Metamodels / Tools Easy Integration of Models
Disadvantages	No Support for Existing Artifacts	Difficult Reuse of Models	No Modularity
Exemplary Application Areas	No Reuse of (Meta-)Models New Domain Description Language	Reuse of Metamodels Combination of Existing Standards for new Projects	Reuse of (Meta-)Models Software Re-Engineering Activities

Table 2: Main Advantages and Disadvantages of the three Approaches with Exemplary Application Areas

able today is based exclusively on the notion of projective views driven by a SUM. However, there are approaches that address the more specific problem of keeping multiple views on a database consistent (Dayal & Bernstein, 1982), or that support a synthetic approach to modeling in a limited context like multi-paradigm modeling (Vangheluwe et al., 2002).

The discipline in which the idea of using views to provide different perspectives on large, complex systems is the most mature is Enterprise Architecture (EA) modeling, characterized by approaches such as Zachman (Zachman, 1987) and TOGAF (Haren, 2011). These all define some kind of "viewpoint framework" defining the constellation of views available to stakeholders and the kind of "models" that should be used to portray them. Some of these, like RM-ODP (Linington et al., 2011), adopt an explicitly synthetic approach, while others such as Archi-Mate (Iacob et al., 2012) and MEMO (Frank, 2002) make no commitment. However, again no EA modeling platform available today explicitly advocates, or is oriented towards the use of projective views.

Therefore, the criteria presented in this paper help to design and select appropriate SUM construction approaches. Additionally, the three sketched SUM approaches under development show the feasibility of projectional, multi-view environments.

10 Conclusion

Ensuring holistic consistency in system development is a growing challenge as systems become larger and more complex. In this paper, we introduced a unifying terminology and developed criteria for classifying approaches that allow to construct single underlying models (SUMs) as solutions to that consistency problem. Based on those criteria, we identified the main conceptual differences between possible solutions, which are the *construction process* to build a SUM and its metamodel, and their *pureness*, i.e., the absence of redundancy. We derived five *selection criteria*, which help to select an appropriate approach

for a specific situation, depending on the necessity to reuse existing metamodels and models, the expected evolution of integrated languages, the need for defining new viewtypes, as well as the required reusability.

We presented three existing approaches, which are OSM, VITRUVIUS and MOCONSEMI, which cover the conceptual solution space for SUM approaches spanned by the criteria *construction process* and *pureness*. On the one hand, OSM creates a pure SUM in a top-down way, while on the other hand, VITRUVIUS leads to a pragmatic SUM by bottom-up reuse of existing (meta)models explicitly keeping dependent information consistent. Between these two ends of the spectrum, MOCONSEMI operates in a bottom-up way like VITRUVIUS, but removes redundant information leading to an improved, possibly essential SUM. There is no known top-down approach that uses pragmatic SUMs. By applying the identified criteria to these different approaches, we were able to give a reasonable indicator for the appropriateness of those criteria, as they are distinguishing for the approaches.

OSM especially replaced the paradigm of refinement of models by model transformation chains by the new paradigm to project models only as views on the complete interconnected information of the whole system. It is the initiator for the idea of constructing SUM that are only defined via projectional views. Based on that idea, VITRUVIUS and MOCONSEMI contribute concrete pragmatic strategies for building SUMs according to this paradigm.

An interesting possibility is to combine the approaches by nesting SUMs developed with different approaches, so that one SUM contains other SUMs by using their provided viewtypes. This, for example, would allow an essential SUM defined for a specific concern of a system to be combined with existing metamodels in a pragmatic SUM using VITRUVIUS or MOCONSEMI. Finally, it also offers the construction of pragmatic SUMs, which can easily become complex when they contain lots of metamodels, to be hierarchically decomposed (i.e nested).

As future work, we plan to define a community case study that describes metamodels, models, and

consistency rules in the application area of software development. Its realization by the three approaches will help to evaluate, compare and improve the approaches, their technical realizations and provided tooling using technical criteria to be proposed.

These criteria and derived evaluations of the three SUM approaches are developed by three groups of SUM researchers. This paper results from that collaboration including discussions from three internal workshops and will be continued with developing the community case study and its application to all three SUM approaches for evaluation in practice.

References

Atkinson, C. (2002). *Component-based product line engineering with uml*. Addison-Wesley object technology series. Addison-Wesley.

Atkinson, C., Stoll, D., & Bostan, P. (2010). "Orthographic Software Modeling: A Practical Approach to View-Based Development." In L. Maciaszek, C. González-Pérez, & S. Jablonski (Eds.), *Evaluation of Novel Approaches to Software Engineering* (Vol. 69, pp. 206–219). Communications in Computer and Information Science. Springer.

Atkinson, C., Tunjic, C., & Moller, T. (2015). Fundamental Realization Strategies for Multi-View Specification Environments. In *19th international enterprise distributed object computing conference* (Vol. 2015-Nov, pp. 40–49). IEEE.

Burger, E., Henß, J., Küster, M., Kruse, S., & Happe, L. (2014). "View-Based Model-Driven Software Development with ModelJoin." *Software & Systems Modeling*, *15*(2), 472–496.

Codd, E., Codd, S., & Salley, C. (1993). *Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate*. Codd & Associates.

Dayal, U. & Bernstein, P. A. (1982). "On the updatability of network views—extending relational view theory to the network model." *Information Systems*, *7*(1), 29–46.

Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., & Orejas, F. (2011). "From state- to delta-based bidirectional model transformations: The symmetric case." *Model Driven Engineering Languages and Systems*, *LNCS 6981*, 304–318.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., & Goedicke, M. (1992). "Viewpoints: A framework for integrating multiple perspectives in system development." *International Journal of Software Engineering and Knowledge Engineering*, *2*(1), 31–57.

Frank, U. (2002). Multi-perspective Enterprise Modeling (MEMO) – Conceptual Framework and Modeling Languages. In *Hawaii International Conference on System Sciences (HICSS)* (pp. 72–81).

Goldschmidt, T., Becker, S., & Burger, E. (2012). Towards a tool-oriented taxonomy of view-based modelling. In E. J. Sinz & A. Schürr (Eds.), *Proceedings of the modellierung 2012* (Vol. P-201, pp. 59–74). GI-Edition – Lecture Notes in Informatics (LNI). GI e.V.

Haren, V. (2011). *TOGAF version 9.1* (10th). Van Haren Publishing.

Herrmannsdoerfer, M. (2010). "COPE - A workbench for the coupled evolution of metamodels and models." *Lecture Notes in Computer Science*, *6563*, 286–295.

Herrmannsdoerfer, M., Vermolen, S. D., & Wachsmuth, G. (2011). "An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models." *SLE*, *LNCS 6563*, 163–182.

Iacob, M., Jonkers, D. H., Lankhorst, M., Proper, E., & Quartel, D. D. (2012). Archimate 2.0 specification: The open group. Van Haren Publishing.

ISO/IEC/IEEE. (2011). *ISO/IEC/IEEE 42010:2011(E): Systems and software engineering – architecture description*. International Organization for Standardization, Geneva, Switzerland.

Kramer, M. E. (2017). *Specification languages for preserving consistency between models of different languages* (Doctoral dissertation, Karlsruhe Institute of Technology (KIT)).

Kramer, M. E., Burger, E., & Langhammer, M. (2013). View-centric engineering with synchronized heterogeneous models. In *Proceedings of the 1st workshop on view-based, aspect-oriented and orthographic software modelling* (5:1–5:6). VAO '13. ACM.

Kramer, M. E., Langhammer, M., Messinger, D., Seifermann, S., & Burger, E. (2015). Change-driven consistency for component code, architectural models, and contracts. In *Proceedings of the 18th international acm sigsoft symposium on component-based software engineering* (pp. 21–26). CBSE '15. ACM.

Kruchten, P. B. (1995). "The 4+1 View Model of architecture." *IEEE Software*, *12*(6), 42–50.

Larman, C. (2004). *Applying uml and patterns: An introduction to object-oriented analysis and design and iterative development (3rd edition)* (3rd). Prentice Hall.

Linington, P. F., Milosvic, Z., Tanaka, A., & Vallecillo, A. (2011). *Building enterprise systems with odp*. Chapman and Hall.

Meier, J. & Winter, A. (2018). "Model Consistency ensured by Metamodel Integration." *6th International Workshop on The Globalization of Modeling Languages, co-located with MODELS 2018*.

Tunjic, C., Atkinson, C., & Draheim, D. (2018). "Supporting the Model-Driven Organization Vision through Deep, Orthographic Modeling." *Enterprise Modelling and Information Systems Architectures-an International Journal*, *13*(SI), 1–39.

Vangheluwe, H., de Lara, J., & Mosterman, P. J. (2002). An introduction to multi-paradigm modelling and simulation. In F. Barros & N. Giambiasi (Eds.), *Proceedings of the AIS'2002 Conference* (pp. 9–20).

Zachman, J. A. (1987). "A framework for information systems architecture." *IBM Systems Journal*, *26*(3), 276–292.