Classifying Approaches for Constructing Single Underlying Models

Johannes Meier[1], Christopher Werner[2], Heiko Klare[3][0000−0002−9711−8835], Christian Tunjic[4], Uwe Aßmann[2], Colin Atkinson[4][0000−0002−3164−5595], Erik Burger[3][0000−0003−2832−3349], Ralf Reussner[3][0000−0002−9308−6290], and Andreas Winter[1]

[1] Software Engineering Group, University of Oldenburg, Germany
{meier,winter}@se.uni-oldenburg.de
[2] Software Technology Group, Technische Universität Dresden, Germany
{christopher.werner,uwe.assmann}@tu-dresden.de
[3] Software Design and Quality Group, Karlsruhe Institute of Technology, Germany
{klare,burger,reussner}@kit.edu
[4] Software Engineering Group, University of Mannheim, Germany
{tunjic,atkinson}@informatik.uni-mannheim.de

Abstract Multi-view environments for software development allow different views of a software system to be defined to cover the requirements of different stakeholders. One way of ensuring consistency of overlapping information often contained in such views is to project them "on demand" from a Single Underlying Model (SUM). However, there are several ways to construct and adapt such SUMs. This paper presents four archetypal approaches and analyses their advantages and disadvantages based on several new criteria. In addition, guidelines are presented for selecting a suitable SUM construction approach for a specific project.

Keywords: projectional, SUM, model consistency, integration, metamodeling

1 Introduction

The increasing complexity of modern software-intensive systems means that individual developers are no longer able to cope with every detail of their structure and functionality. *View-based software development* approaches are therefore useful for allowing individual aspects of a system to be considered independently by separate developers. However, the resulting fragmentation of system descriptions leads to *redundancies* and *dependencies* between the information shown in different views which are difficult to resolve manually. Therefore, automatic mechanisms are needed to ensure holistic consistency between views and the system they portray.

View-based approaches can be characterized as either *synthetic* or *projective* [16] based on the primary source of information for the views. Synthetic approaches distribute information about the system over all the separate views, whereas projective approaches centralize the description in a Single Underlying Model (SUM) [2] from which views are *projected* when needed. As with all model-driven development approaches, a SUM is constructed in terms of instances of concepts defined in a metamodel, which we refer to as a Single Underlying MetaModel (SUMM). Many of the challenges faced in defining generic mechanisms for creating and synchronizing SUMs therefore need to be solved at the SUMM level.

This paper compares the advantages and disadvantages of different strategies for realizing SUM-based approaches to software engineering. The common feature of all projective approaches is that views are regarded as constructively correct, and thus inherently consistent with one another, as long as they agree 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 the 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 strategies for constructing SUM(M)s and identifies criteria for evaluating them. Four existing approaches for constructing SUM(M)s are then compared in terms of how they fulfill the identified criteria. Finally, the suitability of the approaches for different situations is analyzed based on the identified criteria. The presented results allow researchers to classify new approaches for SUM(M) construction and help developers select SUM-based approaches for their specific requirements based on the identified criteria.

After discussing related work in Section 2, the running example and terminology used in this paper are introduced in Section 3, followed by classification criteria for SUM approaches that are described in Section 4. The four SUM approaches OSM (Section 5), VITRUVIUS (Section 6), RSUM (Section 7), and MOCONSEMI (Section 8) are presented subsequently and are classified using the criteria in Section 9. In addition, this section describes guidelines for deciding when to choose each approach. Section 10 summarizes the findings of this paper.

2 Related Work

The explicit use of views or perspectives in software engineering can be traced back to the VOSE method in the early 1990s [9], 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 [20] or the Unified Process [22], 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 [16]). To our knowledge, no general purpose software engineering method available 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 [7], or that support a synthetic approach to modeling in a limited context like multi-paradigm modeling [28].

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 [31] and TOGAF [12]. These all provide 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, like RM-ODP [24], adopt an explicitly synthetic approach, while others such as Archi-

Mate [15] and MEMO [10] make no commitment. However, again no EA modeling platform today explicitly advocates, or is oriented towards, the use of projective views.

Bruneliere et al. [4] conducted a systematic study of model view approaches and from it distilled a detailed feature model of the different capabilities they offer. However, they mainly focused on mechanisms and languages rather than fundamental architectural choices, and did not specifically consider the "synthetic versus projective" distinction of importance here. Atkinson and Tunjic [3], on the other hand, focused on exactly this distinction when they identified several fundamental design choices for realizing multi-view systems. However, they were concerned with the fundamental differences between SUM-based and non-SUM-based approaches rather than between individual SUM-based approaches. In contrast, in this paper we explicitly focus on four distinct SUM-based approaches.

Given the growing importance of projectional approaches, one goal of this paper is to support the evolution of SUM construction methods based on criteria to specify the conceptual solution space (Section 4.1). The aims is for developers to be able to use these criteria (Section 4.2) to help select a concrete SUM approach for a specific situation. Four existing SUM approaches are therefore classified in terms of the criteria characterizing the feasibility of projective, multi-view approaches and examples of how to design and apply SUM approaches are presented.

All four groups actively developing SUM-based approaches at the present time are contributing authors of this paper, which is an extension of a MODELSWARD2019 paper [25]. The main additions are the inclusion of a fourth SUM-based method (RSUM) in Section 7, and a new, independent set of "technical criteria" for classifying SUM approaches in Section 4.3. The explanation of the use of the different SUM approaches in the context of the running ongoing example is also extended and made more explicit. Finally, further insights about how the different SUM approaches can be combined, arising from an ongoing series of joint meetings, have been added to Section 9.5. The progress made in these meetings and VAO international workshops will be continued at the International Workshop on View-based Software Engineering (VoSE) in September 2019 with further researchers.

3 Running Example and Terminology

In this section we introduce a running example of a highly simplified software development project in which requirements, architecture models and implementation (i.e. code units) need to be kept consistent. These three views are described by *languages* using metamodels that define the elements (e.g., classes, associations etc.) that can appear in models. Figure 1 sketches metamodels for the three views. Since this example is used to demonstrate all four SUM approaches, it is directly taken from the initial paper [25].

As shown in Figure 1, *requirements* contain natural language sentences (package `Req`). The `RequirementsSpecification` consists of `Requirements` which are identified by a unique `id` and contain the corresponding natural langauge sentence as simple `text`, written by an `author`. Simplified *class diagrams* are used for the *architecture* and represent system modules as classes (package `UML`). `Classes` have a `className`, one or more unidirectional `Associations` and are collected in `ClassDiagrams`. The *imple-*

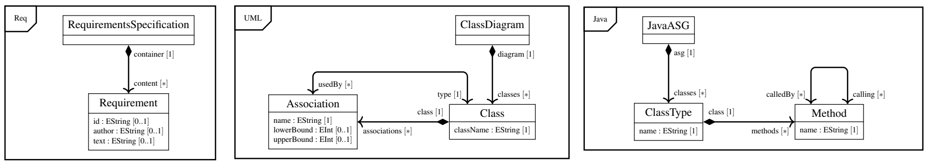


Figure 1. Simplified Metamodels for Requirements (left), Class Diagrams (middle), and Java Source Code (right), taken from [25]

mentation is represented by simplified Java (package `Java`) and realizes the architecture and requirements. The `JavaASG` (Abstract Syntax Graph) contains `ClassType`s with their `name`, which in turn contain `Method`s with their call relations.

These three languages describe different (but not necessarily all) facets of the system under development and thus represent three overlapping *viewtypes*. According to Goldschmidt, Becker, and Burger [11], 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 terms of *consistency rules* which define the relations that have to hold between instances of metamodels.

In the running example, two exemplary consistency rules can be found, which are directly taken from [25] – Consistency Rule 1 targeting redundant information which needs to be kept consistent and Consistency Rule 2 controlling the way additional information is derived from other, already existing information. These two consistency rules are considered representative, because overlapping views usually contain redundant concepts or special interrelations.

Consistency Rule 1: Classes can be defined in the architecture view and in the implementation view. A class can be defined only in the implementation (`Java.ClassType`), or in both the implementation and the architecture (`UML.Class`) if it represents a module. In the latter case, the class has to be kept consistent in the implementation and architecture views (i.e. if it is renamed). 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 progress of the development project can be measured by counting the requirements that are supported by the current implementation. Therefore, `Requirement`s must be linked to the implementing `Method`s. We thus require that each `Method` has to be automatically linked to those `Requirement`s that contain the `Method`'s name in their text. This *additional information* between the requirements and implementation has to be stored and kept consistent. Only these two consistency rules are considered to keep the example as simple as possible. However, in general many further rules could also be envisaged. These two consistency rules and three languages help to show the application of the SUM approaches in Section 5–8. *SUM approaches* specify conceptually how SUMs and corresponding SUMMs are constructed. *Platform specialist*s design SUM approaches and implement supporting platforms. Sections 5–8 show how four such

SUM platforms are applied to this running example. SUM approachesare applied by a *methodologist* who uses a SUM platform to construct a concrete SUM(M) fulfilling the needs of the particular multi-view-project [2].

Depending on the approach, the methodologist creates the SUMM either by reusing the existing metamodels in Figure 1 or by defining a new metamodel from scratch. After that, the *developer* works with views projected from the SUM which is an instance of the SUMM developed by the methodologist. To provide views to cope with all the concerns of developers *new viewtypes* can be configured by the methodologist.

4 Classification Criteria

In order to classify SUM approaches, this section describes classification criteria grouped into three categories with the following objectives: *design criteria* which target the SUM construction process in Section 4.1, *selection criteria* which help users select an appropriate approach for the current application in Section 4.2, and *technical criteria* which focus on technical realization strategies to implement an already conceptually designed SUM approach in Section 4.3.

Because the first two criteria groups were already defined in the initial version of this paper, those definitions have been directly taken from [25] and repeated here. The extended criteria and their grouping represent the first new contribution of this paper. They are used to classify the four SUM approaches (Section 5.3, Section 6.3, Section 7.3, Section 8.3) and to compare them with each other (Section 9).

4.1 Design Criteria

Design criteria capture how a SUM is constructed independently of technical issues (Section 4.3). They describe the main conceptual design decisions for SUM approaches, which span the solution space of possible approaches from the problem perspective. They do not evaluate the quality of SUMs, but help *platform specialists* decide on the conceptual degrees of freedom when *designing a SUM approach*.

Criterion C1 (**Construction Process**) covers the *process* of creating a SUM(M) depending on the initial situation. In a *top-down* approach, a new SUM and SUMM are created from scratch. A *bottom-up* approach, on the other hand, starts with existing models and metamodels and combines them 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” [3] 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 behaves as if it is 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. In between these extremes, some initial redundancies could be resolved targeting a more essential SUM.

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 a conceptual level, while the details of the approaches are designed individually.

4.2 Selection Criteria

When there are several concrete SUM approaches available, the selection criteria help to select the most appropriate SUM approach for a particular situation. These criteria reflect the conceptual preconditions and requirements that favor one concrete SUM approach over another for the application in hand. They therefore help *methodologists compare different SUM approaches* when selecting one to use for a particular project. For example, if many new viewtypes have to be defined on top of the SUM, an approach should be selected that eases the definition of new viewtypes (see following **E3**).

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 ("easy"). The value "middle" indicates that some manual effort is required, while "hard" indicates no support from the approach. Since numerous 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) 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 directly without additional work or indirectly using a strategy for migrating the legacy models into views or into the SUM by some kind of model-to-model transformations ("easy"). It requires the reuse of the corresponding initial metamodels according to **E1** and usually requires a bottom-up strategy according to **C1**. To reuse models they may have to be consistent according to the consistency relations between the integrated metamodels *before* they can be integrated into the SUM. This requires additional manual effort to ensure consistency beforehand ("middle"), in contrast to SUM approaches which offer strategies to handle inconsistent information *during* their integration into the SUM ("easy"). Existing artifacts developed without a consistency-preserving SUM approach usually do not initially fulfill the consistency relationships, which is why this criterion also checks whether those inconsistencies can be handled automatically during integration. The value "hard" indicates no support from the approach.

Criterion E3 (**Viewtype Definability**) focuses on the task of specifying new types of views on a SUMM for 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. The degrees of difficulty depend on whether there are no redundancies ("easy") or all initial redundancies still exist ("hard") and how many models internally exist to query information from. This is because redundant and distributed information makes it harder to collect all relevant information and to propagate changes back into the SUM.

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 [14]) as well as the creation or adaptation of consistency rules. Since languages are subject to change (e.g., new versions 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 that languages will evolve. The degrees of difficulty, “easy”, “middle”, and “hard” depend on how many of the unchanged parts of the SUMM can be reused unchanged in the new SUMM version.

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 there are many software development projects that use slightly different languages or consistency rules, which need to be managed. The degrees of difficulty depend on whether each single part of the SUMM can be reused without any manual effort (“easy”), some manual effort is required with some restrictions (“middle”) or the SUMM is non-reusable and unstructured (“hard”).

4.3 Technical Design Decisions

Technical Design Decisions describe the degrees of freedom available in the technical realization of a single approach. These technical realization choices are orthogonal to the conceptual design decisions (Section 4.1) and the related conceptual selection criteria (Section 4.2) since they can be realized independently of the actual SUM approach used. In other words, they form degrees of freedom for realizing different technical aspects of a particular SUM approach after deciding on the design criteria. They help platform specialists identify and address technical challenges during implementation.

Criterion T1 (**Configuration Languages**) addresses the platform specialist’s challenge of providing languages that can be used by methodologists to customize a SUM approach to the needs of the current project. In particular, languages to specify project-specific consistency rules are required that allow methodologists to tailor SUM approaches to different projects in different application domains. Methodologists need languages to consider and realize project-specific consistency rules, manipulate the SUM, define additional viewtypes and support additional needs of the developers.

Criterion T2 (**Meta-Metamodel**) addresses the issues of choosing a language to describe metamodels in the implementation of a SUM approach. This meta-metamodel defines the possible language elements available to methodologists to describe the SUMM and the viewtypes to be integrated.

5 Orthographic Software Modeling

Orthographic Software Modeling (OSM) is a view-based approach initially developed to support software development [2] using multiple perspectives. However, OSM can be applied to other domains like enterprise architecture modeling [27] in order to support methods like Zachman [31].

5.1 Design Objectives

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 or she needs in a domain-specific notation. Although stakeholders can only see and manipulate the system using the views, since the actual description of the system is stored in a SUM. The views are defined to be as "orthogonal" as possible using 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 on changes. To support the complete life-cycle of a system, ranging from requirements analysis to deployment, the internal structure of the SUM must support the storing of 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 inner elements in a redundancy-free way using approaches like *Information Compression* and *Information Expansion* [3].

In order to use the OSM approach, an environment has to be developed which realizes its goals and principles. Both steps, 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 [1], MEMO [10], ArchiMate [15]) and the targeted domain (e.g., software engineering, enterprise architecture modeling). The configurations can be reused for projects in the same domain and the same methodology. Tunjic, Atkinson, and Draheim [27] present a metamodel which is used by the current prototype implementation to support the configuration of OSM environments. In particular, it facilitates 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 [6] systems.

A software engineer, playing the role of a *methodologist*, performs the customization of the environment for a specific domain and methodology. 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 or she is responsible 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 for reuse in other projects.

Once a complete configuration of an OSM environment has been defined by a *methodologist*, *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.

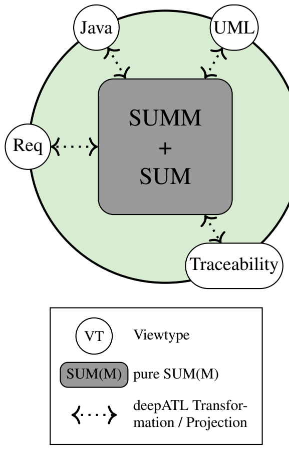


Figure 2. SUM Approach OSM

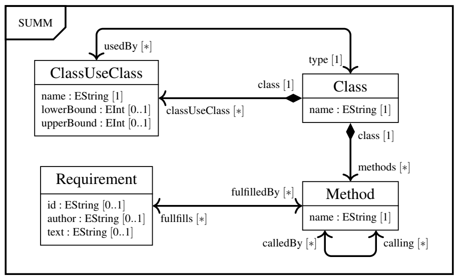


Figure 3. Exemplary Metamodel for SUM in OSM (taken from [25])

5.2 Application to the Running Example

Figure 3 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 3 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 3. 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 and are hence both added to the `Class` element: The first allows `Class`es to have `Method`s, while the second describes dependencies between two `Class`es (Consistency Rule 1). Consistency Rule 2 is captured by a relationship between `Requirement` and `Method`, representing the fact 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 containing at least the concepts `Requirement`, `Method` and the relationship between them. While Figure 3 shows the integration of the 3 domains into a SUMM, Figure 2 shows the arrangement of the viewtypes resulting from the integrated domains. Each viewtype is related by a one-to-one projection to the SUMM, or more precisely to the relevant concepts from the SUMM. The data structure shown in Figure 3 is simpler than disparate representation in Figure 1. This is achieved by unifying names for equivalent concepts (`ClassType` vs. `Class`) and using 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.

5.3 Classification Based on the Criteria

In OSM the SUM is built following a *top-down* approach (**C1**), based on the domain and applied methodology. Since the SUM is created from scratch, it can be constructed in an optimal way by avoiding any internal redundancies and dependencies, resulting in an *essential* SUM (**C2**) (see Table 1).

The **E1** selection criterion is only conceptually supported by the OSM approach ("hard") since 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. However, this is not a formal part of the approach. The OSM approach supports the **E2** selection criterion in a semi-automatic way, i.e. by importing data from existing models into the newly constructed SUM using model transformations ("hard"). The models do not need to be initially consistent as long as the transformations are defined to generate consistent output. As the essential SUM provides an integrated and redundancy-free structure, the **E3** selection criterion can be easily fulfilled by the OSM approach ("easy"), since the information relevant for views is contained in one single artifact, the SUM. The **E4** selection criterion, related to model evolution, is supported quite well ("middle") by OSM's essential SUM principle, since it is free of redundant information but has to check that the changes keep the SUMM redundancy-free. However, the transformations that generate views from the SUM have to be updated manually to stay up-to-date with the SUMM changes. Finally, the **E5** selection criterion is supported by OSM, since a SUMM can easily be extended by adding new concepts directly into the existing structure where they are needed. However, redundancy-freeness must be preserved and when concepts are removed from the SUMM, related concepts have to be checked to ensure consistency ("middle").

The configuration of the current OSM prototype environment is realized using the ECORE modeling language (**T1**), which is used to define the dimension-based view-navigation feature of OSM. The SUMM and the view-types are defined using the PLM modeling language (**T2**), which supports the usage of multiple classification levels using ontologies, while the relationships between the SUMM and the view-types are defined using the DeepATL transformation language.

6 VITRUVIUS

The VITRUVIUS approach [18] is based on a so called *virtual SUMM (V-SUMM)*, which composes a SUMM of existing metamodels instead of creating it from scratch. Therefore, VITRUVIUS relies on pragmatic SUMMs that are defined in a bottom-up fashion.

6.1 Design Objectives

In the VITRUVIUS approach, the whole description of a system is encapsulated in a SUM, which may only be modified via projectional views. This conforms to the projectional SUM idea of the OSM approach. VITRUVIUS follows a pragmatic approach by composing a SUMM of existing metamodels that are coupled by Consistency Preservation Rules (CPRs), which specify how consistency of dependent information in instances of those metamodels is preserved after one of them is changed. The CPRs use and modify a trace model that contains so called *correspondences*, which reference model elements that have to be kept consistent. A set of metamodels with a set of CPRs consitutes a virtual SUMM (V-SUMM), while instances of them with an actual correspondence model 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. A V-SUM operates *inductively*, i.e., it is always consistent before a modification and ensures that it is again consistent after modifications by executing the CPRs. As a consequence, a V-SUM behaves completely like an essential SUM in the OSM approach since it provides the same guarantees regarding consistency.

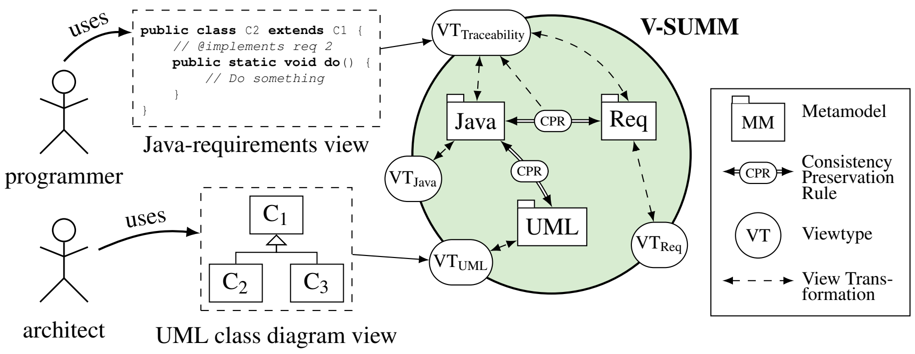
Consistency preservation in VITRUVIUS is performed in a delta-based manner. In other words, it tracks edit operations instead of comparing two models states like in state-based consistency preservation. This results in less information loss [8]. For example, a state-based approach cannot reliably distinguish the deletion and creation of an element from renaming it, whereas a delta-based approach tracks the actual operations. Specific languages have been developed that support the definition of such delta-based consistency preservation in the VITRUVIUS approach [17]. Consistency preservation in VITRUVIUS was first investigated on a case study of component-based architectures, Java code and code contracts [19].

The development of a framework such as VITRUVIUS first involves a *platform specialist* who defines 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`) based on Ecore contains a Java-based definition of V-SUMs and provides two languages for defining consistency preservation at different abstraction levels.

The *methodologist* selects metamodels and reuses or defines CPRs for those selected metamodels to define a V-SUMM. Finally, one or more *developers* can instantiate the V-SUMM, derive views according to existing or newly defined viewtypes, and perform modifications of them. A change recorder tracks modifications in a view and applies them to the V-SUM as a sequence of atomic change events (creation, deletion, insertion, removal or replacement). After each of these changes is applied, the responsible CPRs are executed to restore consistency, which results in an *inductively consistent* V-SUM.

6.2 Application to the Running Example

We depict an exemplary V-SUMM for the metamodels from Figure 1 in Figure 4. It consists of the reused metamodels and a set of CPRs 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`. The methodologist is free to specify the expected behavior in the other direction, i.e., whether a UML class is created for each Java class or if the developer shall be asked what to do. Additionally, the rule 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 matching requirements and methods after adding or modifying methods as well as requirements, and by storing them as correspondences in the existing trace model. Alternatively, such links could be specified in an additional model, which is modified by a CPR whenever a requirement or method is changed.



```
public class C2 extends C1 {
    // @implements req 2
    public static void do() {
        // Do something
    }
}
```

Figure 4. Example V-SUMM in VITRUVIUS (extended version of Fig. 3 of [25])

Two types of projectional viewtypes can be defined on a V-SUMM. First, existing viewtypes for the existing metamodels, such as a textual editor for Java or a graphical editor for UML, can be reused. In Figure 4, these viewtypes are VT_{Java}, VT_{UML} and VT_{Req}, which provide concrete syntaxes for the original metamodels from Figure 1. Second, the methodologist and the developers can defined additional viewtypes, which may combine information from different metamodels and their relations defined in the CPRs. Figure 4 contains $\text{VT}_{\text{Traceability}}$, which displays the trace information for Consistency Rule 2 by extracting information from Java and the requirements model, as well as from the correspondences generated by the CPRs. This viewtype could, as exemplarily sketched in Figure 4, show the Java code with annotations attached to the methods that show the requirements they fulfill. Such viewtypes may combine information from multiple metamodels, which needs to be supported by an appropriate language. In VITRUVIUS, that can be expressed with the ModelJoin language [5].

6.3 Classification Regarding the Criteria

VITRUVIUS follows a *bottom-up* construction approach (**C1**) to build a *pragmatic* SUM (**C2**). This V-SUMM may contain redundancies, but keeps them consistent by explicit consistency preservation mechanisms. In general, VITRUVIUS provides good support for reusability and evolvability of metamodels and SUMMs, but requires considerable effort to reuse existing models and to define new viewtypes (see Table 1).

Metamodel Reusability (**E1**) is well supported ("easy"), because existing metamodels can be integrated into a V-SUMM as is without further effort. They serve as an unmodified component of a V-SUMM. On the contrary, **Model Reusability** (**E2**) is only moderately ("middle") supported by VITRUVIUS. Although existing models can be integrated into a V-SUM, they first need to be consistent according to the consistency rules, and second, have to be enriched with the correspondences that would have been created if the consistency preservation rules were executed during the creation of the models. This is necessary because of the inductive characteristics of the approach. **Viewtype Definability** (**E3**) is comparatively difficult ("hard") using VITRUVIUS, because information that is to be projected into a view is potentially spread across several models so that information has to be combined and aggregated. This may require high effort and especially high knowledge about the involved metamodels from the person who specifies a viewtype. To ease viewtype definition, VITRUVIUS provides the specialized ModelJoin language for defining viewtypes on several models.

One well supported feature is **Language Evolvability** (**E4**) ("easy"). Due to the integration of original metamodels into the V-SUMM, their evolution can be easily supported. Necessary adaptations after the evolution of a metamodels concern the defined CPRs, as well as defined viewtypes. Finally, **SUMM Reusability** (**E5**) is high ("easy"), because it is easy to add or remove single metamodels from the V-SUMM by just adding or removing the metamodels with associated CPRs. For that reason, the reuse of a subset of the metamodels in a V-SUMM is well supported and enables the reuse of parts of a V-SUMM in a different context and the combinations with other V-SUMMs.

Regarding technical design decisions, the configuration languages (**T1**) in VITRUVIUS consist of the declarative Mappings language and the imperative Reactions language for specifying CPRs, as well as the ModelJoin language for defining view types on several metamodels. All concepts of VITRUVIUS are designed for a meta-metamodel (**T2**) that conforms to the EMOF standard. This especially includes the Ecore meta-metamodel that is used for the current implementation of VITRUVIUS.

7 RSUM

The Role-based Single Underlying Model (RSUM, [30]) follows the same basic idea as the VITRUVIUS approach where several metamodels are kept consistent with consistency preservation rules. However, the metamodels are no longer regarded as separate, but can be reconnected and combined with new relations as visualized in Figure 5.

7.1 Design Objectives

Regarding the design objectives, the RSUM approach differs only slightly from the VITRUVIUS approach and follows the projectional SUM idea, whereby all information is stored in a pragmatic SUM of different combined models. Additionally, only projectional views can be generated on this SUM. Compared to the VITRUVIUS approach, which works in the background with object-based programming, the RSUM approach uses role-based programming as introduced by Kühn *et al.* [21] in the form of the Compartment Role Object Model (CROM). The idea behind CROM is a new partition of elements into natural types, role types, and compartment types. Naturals describe fixed objects that play roles in compartments. A role adapts the behavior of a natural in a compartment and interacts with various other roles in it. The concept of compartments is used in the RSUM approach to explicitly describe consistency preservation rules (CPRs) and build relationships [29] between metamodel elements that blur the boundaries of the base metamodels. The compartments for consistency assurance follow an incremental approach and propagate all changes directly to the related elements. Such incremental direct propagation is also implemented between views and the RSUM, and back. It is planned to extend the change propagation with the use of transactions.

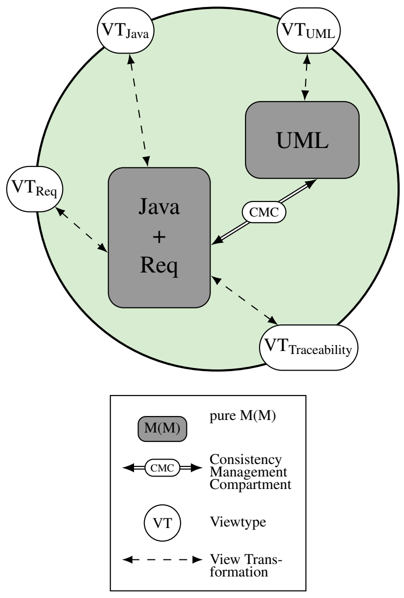


Figure 5. SUM Approach RSUM

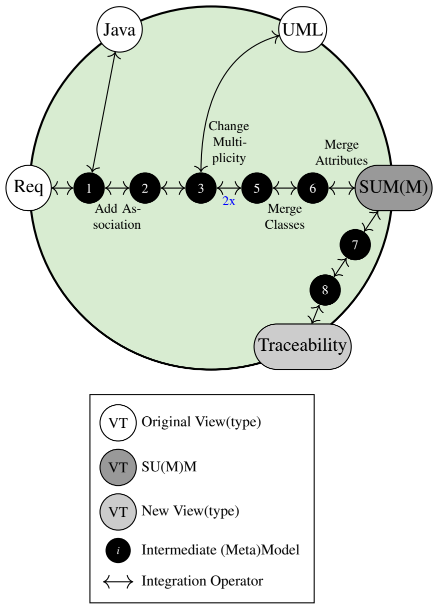


Figure 6. SUM Approach MoConseMI with Chain of Configured Operators to integrate Requirements, UML and Java, adapted from [25]

For the development of the RSUM framework, a *platform specialist* is required that provides the basic functionalities for the integration of new views, metamodels, models, and CPRs. The direct implementations can be automatically generated with Domain Specific Languages (DSLs), created by the *platform specialist*, and integrated into the framework. The current RSUM implementation uses CROM and is based on the SCala ROLes Language (SCROLL) [23], a role-based programming language in Scala.

The *methodologist* selects the needed metamodels and then defines all CPRs between these metamodels in predefined DSLs. Consistency rules differ in two classes, as exemplified in Section 3 with “Consistency Rule 1 & 2”. For rules of the first category, special consistency relationships are defined using extra consistency management compartments (CMCs) in RSUM. The second category of rules creates relational compartments that blur the separation between the integrated metamodels, as described in more detail in the next paragraph. After that, the developer creates new viewtypes and instantiates views from them on the RSUM. To minimize the learning effort for different SUM approaches, the RSUM approach uses the syntax of ModelJoin [5] to define viewtypes, which is used to generate view compartments with incremental change propagation to the RSUM. In the RSUM, consistency is automatically ensured by the defined CPRs.

7.2 Application to the Running Example

Figure 7 shows an RSUM at the metamodel level resulting from the running example in Figure 1. The basic concept consists of separating all relations from the classes and

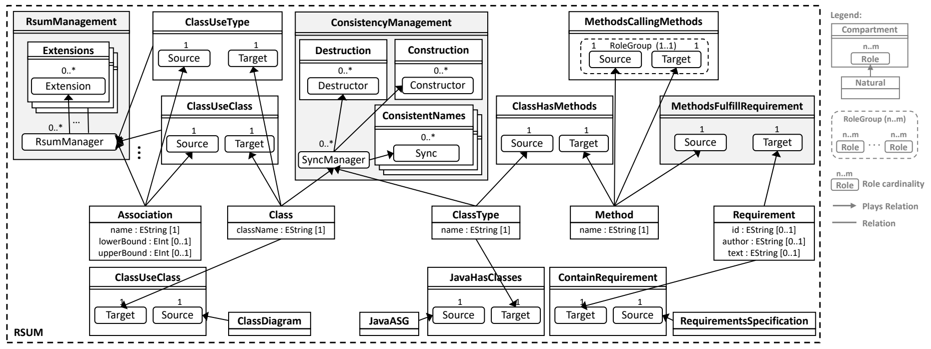


Figure 7. RSUM metamodel of the running example.

managing them in extra relational compartments. This leads to a certain additional overhead but simplifies the administration of the elements in the RSUM and the views. Furthermore, they are automatically generated when integrating metamodels in the RSUM, whereby this design decision remains hidden from the developer. Figure 7 highlights three special compartments in grey that are not created by integrating metamodels. The `RsumManagement` compartment is the central component of the RSUM approach and manages the internal elements, the active and inactive views, and the extensions where currently only one extension for recording changes is pre-implemented. The other two compartments serve to ensure consistency in RSUM and are only created when a consistency rule is defined and integrated. In this case, the `ConsistencyManagement` compartment (CMC) ensures consistency between the naturals `Class` and `ClassType`, defining what happens to the other element when deleting, inserting, or modifying one of these elements. This compartment is automatically generated after describing the CPRs from a *methodologist* in the predefined DSLs. The last highlighted compartment is the `MethodsFulfillRequirement` compartment, which represents a new relation between the naturals `Method` and `Requirement`. This compartment is created by a DSL and then integrated into the RSUM. This new relation merges the requirements model with the Java source code model as shown on an abstract level in Figure 5.

For the generation of views, projections can be generated on all relations (relational compartments) and naturals. In this approach, the views are implemented as compartments, which means that the elements in the RSUM only play roles in the views and therefore do not generate materialized views. The flexibility arising from the fact that roles can be played by other elements at instance level is a big advantage of the role concept and makes the approach useful.

7.3 Classification Regarding the Criteria

When considering the design criteria, the RSUM approach looks the same as the VITRUVIUS approach. It follows a *bottom-up* design approach (**C1**) to create a *pragmatic* SUM (**C2**) and does not resolve inconsistencies but provides consistency management through CPRs. If there are certain 1-to-1 mappings between model elements in the

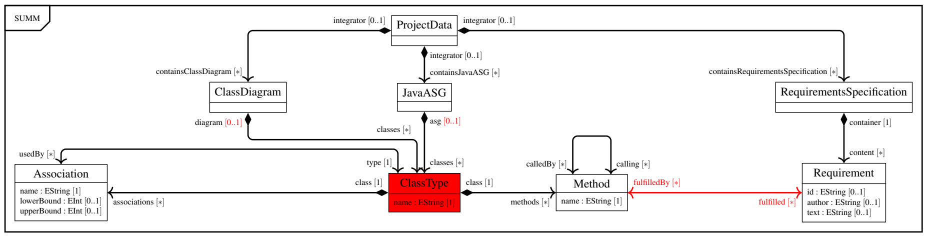


Figure 8. SUMM with integrated Requirements, Class Diagrams and Java (taken from [25])

CPRs, it would be possible to merge them into one natural type. However, this leads to disadvantages in a language evolution step.

The reusability of metamodels (**E1**) is supported as the metamodels have no special dependencies in the RSUM and new ones can be added without much effort ("easy"). Regarding the reusability of models (**E2**), it looks like the reusability of metamodels without the possibility to compare input models with existing instances ("middle"). Currently, (meta)models can be automatically integrated into RSUM as Ecore and XMI files. Since the definition of viewtypes (**E3**) requires knowledge of the integrated metamodels, this could lead to some problems ("middle"). If, however, enough knowledge of the underlying models is available or only predefined viewtypes of the *methodologist* are used, the use of ModelJoin no longer poses a problem. The evolution of languages (**E4**) is relatively easy due to the integration of metamodels ("middle"). However, not only the consistency compartments have to be modified, all additional relational compartments must be adapted. As in VITRUVIUS, the SUMM (**E5**) is easy to reuse since metamodels and consistency relationships can easily be integrated and removed because of the loosely coupling of all elements provide by the role concept.

Regarding the technical criteria, a DSL is used to generate the CPRs and also viewtypes are generated with ModelJoin queries (**T1**). The management of the created elements is done by the `RsumManagement` compartment. In the background the RSUM approach works with the role-based programming language SCROLL on the CROM model, which is modeled with ECORE (**T2**).

8 MoConseMI

MOdel CONSistency Ensured by Metamodel Integration (MoConseMI, [26]) combines the bottom-up reuse of existing (meta-)models with their operator-based improvement into one more essential SUM.

8.1 Design Objectives

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, Stoll, and Bostan [2]. In practice, many models and metamodels already exist in the form of DSLs and tools with fixed data schemas. To reuse them, these initial models and metamodels are integrated and kept in sync as views of the SUM. To achieve this, the initial models and conforming metamodels have to be transformed into the final SUM and conforming SUMM. Therefore, the required transformations have to target *models and metamodels together*.

After creating the initial SUM(M) as the output of the executed transformations with the initial (meta)models as input, the initial models have to be kept consistent with respect to changes made by the developer in the SUM. Therefore, the transformations have to be executable in an inverse direction from the SUM to the initial models, as well as in a forward direction. The last main design objective of MoConseMI is that the required transformations are reusable in different projects, not only in software engineering projects.

To fulfill these design objectives, MoConseMI uses *operators* which divide the whole transformation between initial models and the SUM into chains of small and reusable parts. Each operator does a small change 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). To achieve the required model co-evolution [14], the operator also changes the current model to keep it consistent with the changed metamodel. Degrees of freedom in the model-co-evolution process are influenced by model decisions, which allow consistency rules to be fulfilled (e.g., specify, when new links should be added, or if at all). The concatenation of several operators builds the whole transformation between initial models and SUM. Additionally, the operators can be used to define new viewtypes on top of the SUMM. The required backward executability of operators is attained by combining each operator with an inverse operator, e.g., `DeleteAssociation` for `AddAssociation`.

8.2 Application to the Running Example

The operators are developed once by the *platform specialist* and are provided as a reusable library (currently 23 operators including inverse ones). A supporting framework is under development using Java and a subset of Ecore, reusing parts of Eclipse EDapt [13], and extending some coupled operators [14] for the needs of MoConseMI.

The methodologist reuses the provided operators by combining them as chain of operators to describe the transformation between initial (meta-)models (Figure 1) and SUM(M) (Figure 8) in step-wise way as shown in Figure 6 individually for each project. The methodologist uses the metamodel and model decisions to configure the operators to support the consistency rules for the current project.

The first step is to combine the initial models and metamodels for $\boxed{\text{Requirements}}$, $\boxed{\text{Java}}$ and $\boxed{\text{ClassDiagrams}}$ only technically at ❶ and ❸. For this, the used EMF framework requires `ProjectData` and its compositions as container (Figure 8). After that, Consistency Rule 2 is realized by the operator `AddAssociation` ❶→❷ as the first contentwise integration regarding traceability links between requirements and methods.It creates a new association between `Requirement` and `Method` whenever required and thereby enables traceability information to store. In the model, the operator adds links between some methods and requirements as configured by the methodologist to ensure that a method is linked with those requirements that contain the name of the method in their text.

Before fulfilling Consistency Rule 1, the operator `ChangeMultiplicity` is applied twice ❺ as a technical preparation. After that, `MergeClasses` ❺→❻ merges the two classes `Class` (from UML) and `ClassType` (from Java) into one single class in the metamodel representing data classes both in UML and Java at the same time. In the model, matching UML and Java instances are merged into each other, again controlled by a model decision, which was configured to identify matching instances when they have the same values for `Class.className` and `ClassType.name`. The motivation for this merging step is the unification of redundant information which ensures consistency and makes the resulting SUM more "essential". Finall,y `MergeAttributes` merges the two redundantly name attributes. The last stable model and metamodel are used as the SUM(M). The SUMM in Figure 8 marks the changes done by the operators in red compared to the initial metamodels in Figure 1.

Ultimately, MoConseMI "migrates" the initial (meta)models to view(type)s on the SUM(M). Therefore, the *developer* can change the initial models, the SUM and the newly defined views in the same way. To propagate the changes automatically into all other models in order to ensure their consistency, the operator chain is executed in forward and backward directions.

8.3 Classification Regarding the Criteria

Since MoConseMI starts with the initial (meta-)models, it is *bottom-up* regarding **C1** (see Table 1) and inherits all their initial redundancies. Because the operators can resolve redundancies in a step-wise way, pureness can be improved until, in the best case, a SUM(M) without any dependencies is attained (**C2**).

Metamodel Reusability (**E1**) is well supported ("easy"), since the initial metamodels are used as the starting point of the operator chain. The same counts for the **Model Reusability** (**E2**) ("easy"), since even initial inconsistencies can be resolved by executing the operator chain in both directions. **Viewtype Definability** (**E3**) benefits by the explicit and integrated SUMM, since all information is collected and integrated inside one metamodel ("middle"). In contrast to OSM, the SUM(M) in MoConseMI might still contain redundant information, which makes the definition of new viewtypes harder, since the same information can be found at different places. The **Language Evolvability** (**E4**) in MoConseMI highly depends on the kind of change. Additional elements in the metamodels are directly added to the SUMM without any changes in the operator chain, while big refactorings in the metamodels require lots of changes in the operator chain ("middle"). **SUMM Reusability** (**E5**) is easy when adding new metamodels, because the existing chain of operators is lengthen by some more operators. Removing an already integrated metamodel requires all the operators which were needed for its integration to be removed or changed ("middle"). As a result, **E5** depends on the order of integrated initial (meta-)models ("middle").

Regarding technical design decisions, MoConseMI uses chains of operators as a configuration language which change metamodels and models together to create the initial SUM(M) and to keep all models consistent (**T1**). The same operators allow new views to be defined. The current implementation supports a subset of ECore (**T2**).

Criterion	**OSM**	**VITRUVIUS**	**RSUM**	**MOCONSEMI**
C1 Construction Process	top-down	bottom-up	bottom-up	bottom-up
C2 Pureness	essential	pragmatic	pragmatic	pragmatic $\rightarrow$ essential
E1 Metamodel Reusability	hard	easy	easy	easy
E2 Model Reusability	hard	middle	middle	easy
E3 Viewtype Definability	easy	hard	middle	middle
E4 Language Evolvability	middle	easy	middle	middle
E5 SUMM Reusability	middle	easy	easy	middle
T1 Configuration Languages	Ecore, DeepATL	Mappings/React., ModelJoin	RCs, ModelJoin	Bidirect. Operators
T2 Meta-Metamodel	PLM	Ecore	CROM	Ecore

Table 1. Comparison of the four Approaches regarding Design Criteria, Selection Criteria, and Technical Design Decisions (extended version of Table 1 from [25])

9 Discussion and Comparison of SUM Approaches

This section summarizes the classification of the four presented SUM approaches in Table 1 regarding the three groups of classification criteria. Each approach's classification is described in detail in its respective section, so this section contains a more abstract discussion about the dependencies of the criteria on each other. It also answers the general question of which approach fits best for which situation and why. We conclude this section with the idea of combining SUM approaches to create more flexible ones.

9.1 Design Criteria

Design criteria form the foundation of every approach and therefore have considerable influence on the manifestation of the selection criteria and limit the technical implementation possibilities. Figure 9 shows the solution space spanned by the design criteria and the placement of the four presented approaches. Each approach can be distinguished as a *top-down or bottom-up* approach (**C1**), whereby the decision to use existing models is considered as a starting point. Three of the presented approaches (VITRUVIUS, RSUM, and MOCONSEMI) follow a bottom-up approach while one (OSM) follows a top-down approach. The top-down variant has the advantage that a new model is created and can be adapted to the requirements of users. This facilitates the definition of new viewtypes and avoids redundancy. However, it offers no reuse opportunities and makes adaptation and evolution more difficult. Bottom-up approaches, on the other hand, offer increased reusability, but require more effort to manage the (meta)models.

The use of an *essential or pragmatic* SUM(M) approach (**C2**) is directly influenced by the decision whether a top-down or bottom-up approach is used (**C1**). In a bottom-up approach, it is hard to achieve a SUM that is free of redundancy. The MOCONSEMI approach is the only of the four approaches that offers a way of moving from a pragmatic to an essential SUM(M) by creating a redundancy-free SUM(M) bottom-up. In a top-down approach, redundancy can already be avoided by construction.

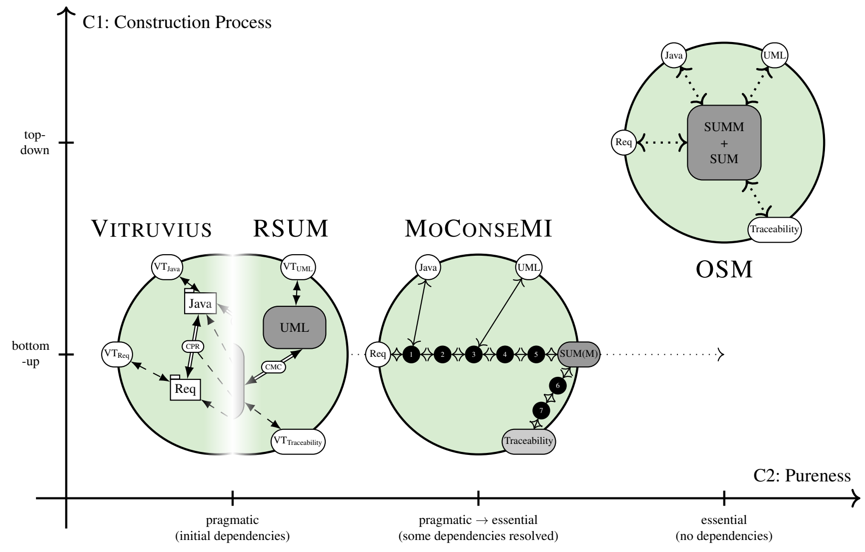


Figure 9. Conceptual Classification of SUM Approachs

9.2 Selection Criteria

The selection criteria depend mainly on the underlying design criteria as mentioned in the previous section. **Metamodel Reusability** (**E1**) is better supported by bottom-up approaches than by top-down approaches, since they are already based on the metamodels and no new ones are generated. The same applies to **Model Reusability** (**E2**). However, it depends on the consistency of the models to be integrated and the already integrated models. In addition, the mechanisms for reusability depend mainly on the automatic modification of the basic metamodels because only changes that can be automatically undone promote reusability. In contrast, **Viewtype Definability** (**E3**) depends on the type of SUM(M) approach. Due to the absence of redundancy and the minimality of an essential SUM(M) approaches, the creation of viewtypes is much easier. However, if pragmatic approaches can provide reliable consistency mechanisms and viewtypes are only defined on partial models, they can also support the creation of viewtypes in a relatively simple way. The creation of viewpoints only gets complicated when they cross model boundaries.

Language Evolvability (**E4**) means in general the evolution of metamodels. In pragmatic approaches this should be easier than in essential ones because they are constructed from existing metamodels and have formally defined relationhsips between them. In contrast, these approaches must preserve consistency after the evolution steps, i.e., the adaptation of the internal model operators and consistency preservation rules. In essential approaches, there exists only a conceptual relation between the existing artifacts and the SUMM, i.e., a manual adaptation of the new metamodel must be done to ensure redundancy-freeness and minimality that leads to high effort and error proneness. **SUMM Reusability** (**E5**) does not have much to do with criteria E1-E4, since it is about adding and removing single metamodel elements in the SUMM. Pragmatic and essential approaches achieve this in a different way. Pragmatic approaches facilitate the simple addition and removal of metamodels, as the structure of the original metamodels still exist separately and the change operations are performed on the abstract metalevel. In contrast, essential approaches make it easy to fine-tune metamodels to project needs, since the manipulation of the essential SUMM is possible at the level of individual model elements. The existence of only one model without dependencies removes the clear boundaries between the starting metamodels.

The application of the selection criteria highlights the general differences between the four SUM approaches. None of the presented criteria is fulfilled by one approach best, i.e., each approach has its pros and cons regarding all considered criteria. In general, each selection criteria can be realized in a simple way if the corresponding design criteria is selected, or in complex way using a lot boilerplate code if a different (non-corresponding) design criteria is selected. From the correlation of design and selection criteria, dependencies can also be determined within the selection criteria. The list of criteria presented here cannot be considered complete and it is unclear whether these are the most important criteria for selecting a SUM approach. However, the criteria can be used as an initial indicator of which approach would be best, since relevant situations such as the evolution and reusability of metamodels and other points are covered.

9.3 Technical Design Decisions

The technical design decisions (**T1** and **T2**) are independent of the two categories described before, since the implementation of each approach depends on the preferred languages but could be done with different technical choices. The current implementations for each of the four approaches are presented here, all of which are constantly evolving. The used **Configuration Languages** (**T1**) apply the generic approaches to specific application projects. These configuration languages contain transformation languages to manage the consistency in, and the integration of models into, the SUM. In addition, they imply query languages to realize initial and new view(type)s on top of the SUM. If we consider the **Meta-Metamodel** (**T2**) of the four approaches they cover the complete space between object-oriented modeling via role-oriented modeling to deep modelling, which show that each technology allows the realization of SUM approaches.

9.4 Process for Approach Selection

After describing the overall differences between the four SUM approaches regarding the different criteria, in this section we describe a process for selecting a SUM approach, as illustrated in Figure 10. This process considers the choice of an approach from a technical and a conceptual point of view.

From a conceptual point of view the main question is about the existence and degree of reuse of legacy tools or metamodels (**E1** and **E2**). If there are no tools or metamodels to reuse, the *top-down* OSM approach fits most because it defines a new metamodels without redundancy that can avoid dependencies to external tool vendors. In addition, OSM provides a simple viewtype definition strategy (**E3**). If the reusability of (meta-)models or viewtypes is important, *bottom-up* approaches (**C1**) like VITRUVIUS,

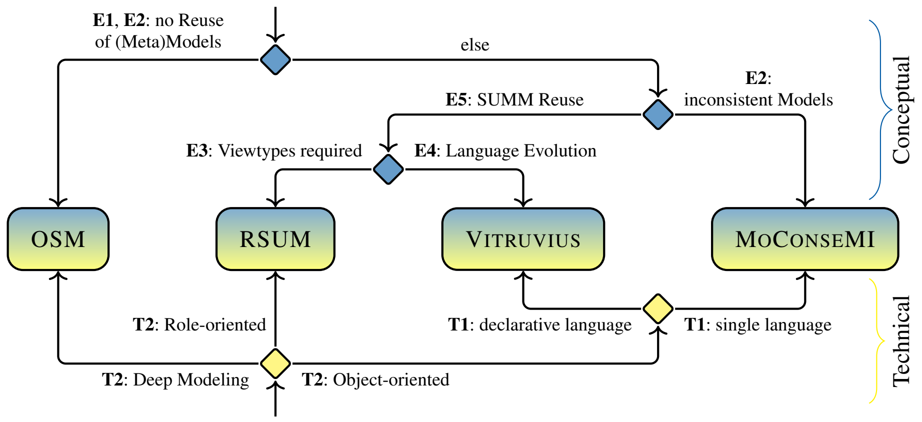


Figure 10. Process for selecting SUM Approaches

	OSM	VITRUVIUS	RSUM	MOCONSEMI
Advantages	Easy Viewtype Definition No Dependencies to Legacy Tools	Reuse of Metamodels / Tools Modular Views	Reuse of Models + Metamodels Modular Views	Reuse of Models + Metamodels Easy + incremental Integration
Disadvantages	No Support for Existing Artifacts	Difficult Reuse of Models	Overhead for role-modeling and programming	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 Runtime adaptation and integration of (Meta-)Models	Reuse of (Meta-)Models Software Re-Engineering Activities

Table 2. Main Advantages and Disadvantages of Approaches with Exemplary Application Areas (extended version of Table 2 from [25])

RSUM, or MOCONSEMI are a better choice. These approaches are compatible to existing tools or even to complete development environments and facilitate integration without remodeling models. When the models have inconsistencies, theMOCONSEMI is best because the models contained therein do not have to be conform to any specific consistency rules and can be initially adapted. If, the reusability of the SUMM (**E5**) is more important, VITRUVIUS or RSUM are the most appropriate. These approaches allow the modular definition of consistency relationships and the reusability of these and the (meta-)models across projects. VITRUVIUS and RSUM differ in the complexity of view definition (**E3**) (use RSUM) and language evolution (**E4**) (use VITRUVIUS).

If the selection of a SUM approach is based on technical specifications, the question of the implementation paradigm (**T2**) arises first. The OSM prototype implementation currently uses deep modeling for the implementation, whereby RSUM is based on the role-based programming paradigm. In contrast, the approaches MOCONSEMI and VITRUVIUS use object orientation. One important difference between VITRUVIUS and MOCONSEMI is the type of configuration languages (**T1**) supported. VITRUVIUS uses multiple,c declarative languages in while MOCONSEMI only uses a single language.

In summary, the guideline in Figure 10 offers a decision-making aid for selecting the most suitable SUM approach from a technical or conceptual point of view. In addition, Table 2 summarizes the main advantages and disadvantages of the four SUM approaches.

9.5 Combination of SUM Approaches

As well as selecting a single SUM approach (Section 9.4), there can sometimes also be advantages in combining several SUM approaches. Since SUMs can be accessed by well defined views, their provided viewtypes can be used to merge several SUMs into a single unified SUM. If an essential SUM was defined to support the modeling of a specific aspect of a system using the OSM approach, it could be combined with other already existing metamodels in pragmatic SUMs using the VITRUVIUS, RSUM or MOCONSEMI approach.

A similar strategy helps to ease the construction of pragmatic SUMs. If several (meta-)models are combined, pragmatic SUMs can become incomprehensible because of the growing number of interrelations between the (meta-)models. Instead, combining only small numbers of metamodels into a pragmatic SUMs and hierarchically composing these pragmatic SUMs into larger reduces the complexity of each individual SUM and improves their reusability. For example, it may be reasonable to combine highly related metamodels, such as object-oriented programming languages and UML class diagrams, into one SUM, which is equivalent to create an essential SUM, and to combine that SUM with other, less related metamodels instead of combining them all together. Nevertheless, this currently only represents a conceptual possibility and its feasibility and applicability have to be further investigated in future work.

10 Conclusion

Larger and more complex systems require mechanisms to ensure holistic consistency during system development. This paper presents uniform terminology and a criteria catalog for the classification of approaches that use a SUM-based approach as a solution to the consistency problem. These are then used to define a set of guidelines that can be used to select which one of the four presented SUM approaches, OSM, VITRUVIUS, RSUM, and MOCONSEMI, is the most suitable for a particular project. To this end, the selection of an approach can be considered either from a technical point of view based on the programming paradigm or from a conceptual point of view based on the selection criteria including metamodel reusability and viewtype definability.

The four presented approaches cover the entire solution space available at the present time. The OSM approach describes a *top-down* approach where a *pure* SUM is created without redundancies. On the other hand, the RSUM and VITRUVIUS approaches are based on *pragmatic* SUMs that take a *bottom-up* approach to keep multiple models consistent through defined relationships. MOCONSEMI also introduces a *bottom-up* approach, but can move between *pragmatic* and *essential* SUMs since redundant information can be removed. We are not currently aware of an implementation of a pragmatic top-down approach.

OSM is regarded as the initiator of the initial SUM idea, where models are only views of an entire model and are projected from it by transformations. RSUM, VITRUVIUS, and MOCONSEMI are concrete strategies for constructing a pragmatic implementation, since the use of a single, redundancy-free model has some disadvantages as described in the discussion.

References

1. Atkinson, C.: Component-based Product Line Engineering with UML. Addison-Wesley (2002)
2. Atkinson, C., Stoll, D., and Bostan, P.: "Orthographic Software Modeling: A Practical Approach to View-Based Development." In: Evaluation of Novel Approaches to Software Engineering, pp. 206–219. Springer (2010)
3. Atkinson, C., Tunjic, C., and Moller, T.: Fundamental Realization Strategies for Multi-View Specification Environments. In: 19th International Enterprise Distributed Object Computing Conference, pp. 40–49. IEEE (2015)
4. Bruneliere, H., Burger, E., Cabot, J., and Wimmer, M.: "A feature-based survey of model view approaches." Software & Systems Modeling 9764, 138–155 (2017)
5. Burger, E., Henß, J., Küster, M., Kruse, S., and Happe, L.: "View-Based Model-Driven Software Development with ModelJoin." Software & Systems Modeling 15(2), 472–496 (2014)
6. Codd, E., Codd, S., and Salley, C.: Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate. Codd & Associates (1993)
7. Dayal, U., and Bernstein, P.A.: "On the updatability of network views—extending relational view theory to the network model." Information Systems 7(1), 29–46 (1982)
8. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., and Orejas, F.: "From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case." Model Driven Engineering Languages and Systems LNCS 6981, 304–318 (2011)
9. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M.: "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development." International Journal of Software Engineering and Knowledge Engineering 2(1), 31–57 (1992)
10. Frank, U.: Multi-perspective Enterprise Modeling (MEMO) – Conceptual Framework and Modeling Languages. In: Hawaii International Conference on System Sciences (HICSS), pp. 72–81 (2002)
11. Goldschmidt, T., Becker, S., and Burger, E.: Towards a Tool-Oriented Taxonomy of View-Based Modelling. In: Proceedings of the Modellierung 2012. GI-Edition – Lecture Notes in Informatics (LNI), pp. 59–74. GI e.V. (2012)
12. Haren, V.: TOGAF Version 9.1. Van Haren Publishing (2011)
13. Herrmannsdoerfer, M.: "COPE - A workbench for the coupled evolution of metamodels and models." Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6563 LNCS, 286–295 (2010)
14. Herrmannsdoerfer, M., Vermolen, S.D., and Wachsmuth, G.: "An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models." Software Language Engineering LNCS 6563, 163–182 (2011)
15. Iacob, M., Jonkers, D.H., Lankhorst, M., Proper, E., and Quartel, D.D.: ArchiMate 2.0 Specification: The Open Group, (2012). `http://doc.utwente.nl/82972/`
16. ISO/IEC/IEEE: ISO/IEC/IEEE 42010:2011(E): Systems and software engineering – Architecture description. International Organization for Standardization, Geneva, Switzerland (2011)
17. Kramer, M.E.: Specification Languages for Preserving Consistency between Models of Different Languages. PhD thesis, Karlsruhe Institute of Technology (KIT) (2017).
18. Kramer, M.E., Burger, E., and Langhammer, M.: View-centric engineering with synchronized heterogeneous models. In: Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling. VAO '13, 5:1–5:6. ACM (2013)
19. Kramer, M.E., Langhammer, M., Messinger, D., Seifermann, S., and Burger, E.: Change-Driven Consistency for Component Code, Architectural Models, and Contracts. In: 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering. CBSE '15, pp. 21–26. ACM (2015)

20. Kruchten, P.B.: "The 4+1 View Model of architecture." IEEE Software 12(6), 42–50 (1995)
21. Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., and Aßmann, U.: "A Metamodel Family for Role-Based Modeling and Programming Languages." In: Software Language Engineering: 7th International Conference, Västerås, Sweden. Springer International Publishing, 2014, pp. 141–160.
22. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition). Prentice Hall (2004)
23. Leuthäuser, M., and Aßmann, U.: Enabling View-based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-based Programming. In: Joint MORSE/-VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering, pp. 25–33. ACM (2015)
24. Linington, P.F., Milosvic, Z., Tanaka, A., and Vallecillo, A.: Building Enterprise Systems with ODP. Chapman and Hall (2011)
25. Meier, J., Klare, H., Tunjic, C., Atkinson, C., Burger, E., Reussner, R., and Winter, A.: Single Underlying Models for Projectional, Multi-View Environments. In: 7th International Conference on Model-Driven Engineering and Software Development, pp. 119–130. SCITEPRESS - Science and Technology Publications (2019)
26. Meier, J., and Winter, A.: "Model Consistency ensured by Metamodel Integration." 6th International Workshop on The Globalization of Modeling Languages, co-located with MODELS 2018 (2018)
27. Tunjic, C., Atkinson, C., and Draheim, D.: "Supporting the Model-Driven Organization Vision through Deep, Orthographic Modeling." Enterprise Modelling and Information Systems Architectures-an International Journal 13(SI), 1–39 (2018)
28. Vangheluwe, H., Lara, J. de, and Mosterman, P.J.: An introduction to multi-paradigm modelling and simulation. In: AIS'2002 Conference, pp. 9–20 (2002)
29. Werner, C., Schön, H., Kühn, T., Götz, S., and Aßmann, U.: Role-Based Runtime Model Synchronization. In: 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 306–313 (2018)
30. Werner, C., and Aßmann, U.: "Model Synchronization with the Role-oriented Single Underlying Model." MODELS 2018 Workshops (2018)
31. Zachman, J.A.: "A framework for information systems architecture." IBM Systems Journal 26(3), 276–292 (1987)