The Journal of Systems & Software 171 (2021) 110815

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Enabling consistency in view-based system development — The Vitruvius approach

Heiko Klare *, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger, Ralf Reussner

Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany

ARTICLE INFO

Article history:
Received 19 December 2019
Received in revised form 13 July 2020
Accepted 4 September 2020
Available online 9 September 2020

Keywords:
Consistency
Model-driven software development
Model transformations
Model views

ABSTRACT

During the development of large software-intensive systems, developers use several modeling languages and tools to describe a system from different viewpoints. Model-driven and view-based technologies have made it easier to define domain-specific languages and transformations.

Nevertheless, using several languages leads to fragmentation of information, to redundancies in the system description, and eventually to inconsistencies. Inconsistencies have negative impacts on the system's quality and are costly to fix. Often, there is no support for consistency management across multiple languages. Using a single language is no practicable solution either, as it is overly complex to define, use, and evolve such a language. View-based development is a suitable approach to deal with complex systems, and is widely used in other engineering disciplines. Still, we need to cope with the problems of fragmentation and consistency.

In this paper, we present the Vitruvius approach for consistency in view-based modeling. We describe the approach by formalizing the notion of consistency, presenting languages for consistency preservation, and defining a model-driven development process. Furthermore, we show how existing models can be integrated. We have evaluated our approach at two case studies from component-based and embedded automotive software development, using our prototypical implementation based on the Eclipse Modeling Framework.

1. Introduction

The complexity of systems that engineers develop is ever increasing (Murer et al., 2011). Since no single engineer can understand a modern system in its entirety, many ways have been proposed to cope with the essential complexity and also to reduce the so-called accidental complexity (Brooks, 1987) when modeling large systems. The view-based paradigm proposes the usage of role-specific views, each only representing relevant, partial information of the system. It has proven useful to separate concerns and to offer only relevant information to different stakeholder groups (Brunelière et al., 2017; Cicchetti et al., 2019). Due to the widespread usage of computer-aided design of systems, these views not only represent software parts of the system, but can contain any kind of information, such as hardware layout, distribution of system parts, communication, energy consumption, and so on. Model-driven technologies have made it easier to define domain-specific languages (DSLs) for these views, and to transform instances of these languages into each other.

With a plethora of modeling languages available, new problems arise in the development of complex systems. First, the description of a system is now spread across multiple models of different languages, leading to the *fragmentation* of the system description. Since these models are never fully orthogonal to each other, information has to be repeated and modeled in several models, such that *redundancy* in the system description occurs. Finally, and most severely, this can lead to *inconsistencies* between these models. Although automated transformations can assist developers by creating parts of the models automatically, they are often only used to transform information in one direction at defined points in the development process. In practice, however, development does not follow a strict top-down and waterfall-like process. Developers who work at later stages of the process, such as deployment and testing, may also change the system in such a way that other models need to be updated to regain a consistent system description.

* Corresponding author.
E-mail addresses: klare@kit.edu (H. Klare), max.kramer@alumni.kit.edu (M.E. Kramer), michael.langhammer@alumni.kit.edu (M. Langhammer), dominik.werle@kit.edu (D. Werle), burger@kit.edu (E. Burger), reussner@kit.edu (R. Reussner).
URLs: https://sdq.ipd.kit.edu/people/heiko-klare/ (H. Klare), https://are.ipd.kit.edu/people/dominik-werle/ (D. Werle), https://sdq.ipd.kit.edu/people/erik-burger/ (E. Burger), https://sdq.ipd.kit.edu/people/ralf-reussner/ (R. Reussner).

https://doi.org/10.1016/j.jss.2020.110815

This problem is still poorly addressed in current development processes for large systems, such as automotive system development, as a recent survey has shown (Guissouma et al., 2018). For example, a development process for Electronic Control Units (ECUs) of automobiles (Mazkatli et al., 2017) involves, among others, the three languages SysML (Object Management Group, 2019), used for the description of the software architecture, ASCET (ETAS Group, 2020), used for the definition of component behavior, and AMALTHEA (ITEA, 2020), which describes the deployment on multi-core platforms. Different engineers use different views from these three languages, such as internal block diagrams or block definition diagrams in SysML. The relations between models as instances of these languages are not documented explicitly, which requires manual and potentially error-prone consistency checking (Mazkatli et al., 2017).

In general, criteria for consistency are often not specified explicitly, since there are no standardized ways for the definition of consistency relations and the repair of inconsistent system descriptions (Stevens, 2018). Thus, inconsistencies can only be fixed manually. An inconsistent system description can lead to poor quality of the implemented system concerning correctness, performance, reliability, security, maintainability, and other quality dimensions. Since fixing these problems at late stages of the development process is expensive, they are often not fixed at all. In the research area of Bidirectional Transformations (BX), this essential problem of consistency preservation has already been addressed, but a view- and BX-based development process has not yet been defined. Thus, a systematic approach for the definition of consistency relations and repair, as well as a development process for their specification and application is necessary.

In this paper, we present the model-based Vitruvius approach for enabling consistency in view-based modeling of complex systems. It comprises concepts and a process to define and preserve consistency of models as well as to define views through which they can be modified consistently. Vitruvius combines the advantages of synthetic and projective modeling (International Organization for Standardization, 2011) while preserving compatibility to existing languages and their instances. Central ideas of this approach have first been published in the dissertations of Erik Burger (Burger, 2014), who designed a language for defining views on multiple models, Max E. Kramer (Kramer, 2017), who developed a language family for delta-based consistency preservation, and Michael Langhammer (Langhammer, 2017), who defined and analyzed consistency rules between architecture descriptions and code, and how existing models can be integrated into a consistency-preserving process. This paper presents the overall concept of the Vitruvius approach, for which the dissertations provide selective contributions. We provide a formalization for the central ideas underlying the Vitruvius approach and define a development process for the description of large systems. In detail, this article presents the following contributions:

Concept formalization (C1): We formalize a notion of *consistency*, *inductive consistency preservation* and *view-based modeling* to clarify assumptions and properties of our approach.

Development process (C2): We define a model-driven development process, which consists of a method for the construction of languages to describe systems, called *Virtual Single Underlying Metamodels*.

Consistency preservation languages (C3): We introduce a family of novel delta-based consistency preservation languages that fit to the needs of the envisioned development process.

Existing model integration (C4): We discuss two general approaches to integrate existing languages and instances into the inductive consistency preservation process.

Overall evaluation (C5): We provide an evaluation of the consistency preservation and integration concepts to show the general applicability of our approach.

While the contributions C1 and C2 are completely novel contributions of this article, C3 is a summary of the languages designed in the dissertation of Kramer (2017), C4 is a generalization of the integration strategies in the dissertation of Langhammer (2017) and C5 is an extension of his evaluation.

The Vitruvius approach supports the development of software-intensive systems in different engineering disciplines, which have to cope with the problem of information fragmentation across different tools and consequential inconsistencies. It helps to solve that problem by defining a development process and mechanisms to explicitly represent dependencies, and, as far as possible, automatically preserve their consistency. We have applied the approach to the software engineering domain (Kramer et al., 2015), but we also have preliminary results for other domains such as automation systems (Ananieva et al., 2018a).

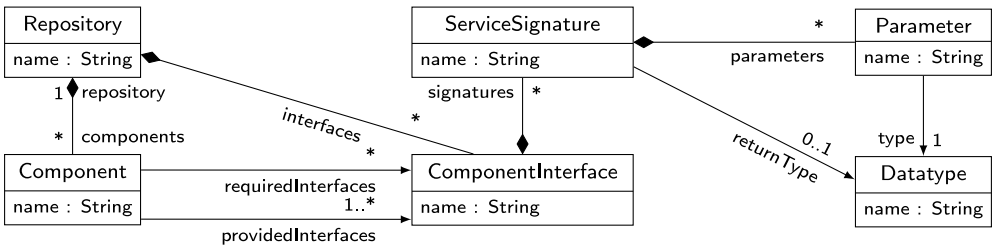
This article is structured as follows: We begin with a running example in Section 2, and foundations in Section 3. In Section 4, we identify drawbacks of current approaches and give a general overview on our Vitruvius approach. We formalize the essential concept of *virtual single underlying models* and their properties in Section 5. Section 6 introduces the development process of the Vitruvius approach, its roles and envisioned scenarios. and Section 7 presents the two languages for consistency preservation used in this process. The integration of existing models into a consistency-aware development process is the topic of Section 8. After an extensive evaluation in Section 9, we close with related work, future work, and conclusion (Sections 10, 11, 12).

2. Running example

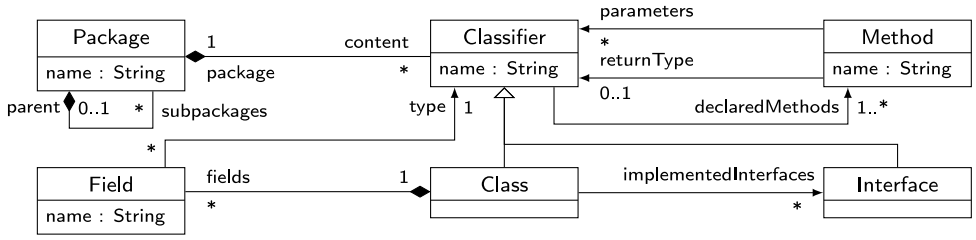
We use a scenario from previous work (Langhammer, 2013) as a running example. It contains a simplified version of the Palladio Component Model (PCM) (Reussner et al., 2016) as an Architecture Description Language (ADL) to describe the component-based architecture of a system, and a simplified metamodel for object-oriented design to represent code, in compliance with UML class diagrams and the class structures in the Java language specification (Gosling et al., 2014).

The simplified component-based ADL is depicted in Fig. 1(a): An architecture model contains a *repository*. It lists component interfaces and reusable components, which are identified by unique names. A *component interface* groups services that are required or provided together and declares a *service signature* for each of these services (Reussner et al., 2011). Such a *service signature* consists of a name, an optional return type, which is a *datatype*, and *parameters*, which have a name and also reference a *datatype*. A *component* references *required* and *provided* component interfaces to denote which services are provided and required by the component. Two service signatures of two different component interfaces can be identical, and a component interface can be provided or required by multiple components. In our running example, a component has no further properties than its name and relations to provided and required interfaces.

The object-oriented design of a system is represented in our running example using *packages*, *classes*, *interfaces*, *methods*, and *fields*, as shown in the simplified metamodel in Fig. 1(b). It is irrelevant whether these elements are defined using a programming language, such as Java, or a modeling language, such as the UML. For the sake of simplicity, we only define *fields* in



(a) Component-based architecture metamodel



(b) Object-oriented design or code metamodel

Fig. 1. Simplified metamodels for the running example.

object-oriented design and also treat associations, known from the UML, as fields. Constructors, method implementations, as well as other concepts of object-orientation, such as class inheritance or interface extension, are not necessary for our examples.

We use a simplification of the *MediaStore* case study (Koziolek et al., 2007; Strittmatter and Kechaou, 2016) as a sample system. It consists of a server with download and upload functionality for media files. In our simplified version, displayed in the left half of Fig. 2, the component repository consists of the two components `MediaStore` and `WebGUI`, and the two interfaces `IMediaStore` and `IWebGUI`, which are required and provided by the components, respectively. Both these interfaces contain methods for uploading and downloading media files.

We employ the mapping defined by Langhammer et al. (2016) to represent the component-based system architecture in object-oriented code. Fig. 2 exemplifies the application of this mapping to the simplified *MediaStore* system. A component repository is represented as a package structure consisting of a root package with the name of the repository. The root package contains dedicated packages for `interfaces` and `components`. Component interfaces are represented by equal object-oriented interfaces within the `interfaces` package of the repository. Their *service signatures* are realized by methods in the corresponding interfaces. The datatypes of parameters and return types are mapped to classes in the object-oriented design model. In the example, the component interfaces `IWebGUI` and `IMediaStore` and their signatures are mapped to interfaces with method declarations in their corresponding object-oriented interfaces.

A component is mapped to a package that has the same name as the component and is placed within the `components` package of the repository, and to a *component-realization class* with the suffix "`Impl`" placed within that package. In the example, the component `WebGUI` is realized by the component-realization class `WebGUIImpl` and is contained in the package `webgui`. The component `MediaStore` is mapped analogously. *Provided interfaces* of a component are mapped to interface implementations of the component-realization class and lead to the implementation of methods provided by the interface within the component-realization class. In case of the example, the provided interface `IWebGUI` of `WebGUI` is realized by an interface implementation in the component-realization class `WebGUIImpl`, as well as an implementation of the methods `httpUpload` and `httpDownload`. Finally, *required interfaces* are represented by an association of the component-realization class to the required interface. This association is realized by an appropriate field that is typed with the component interface and that is set within the constructor of the component-realization class. Therefore, the `WebGUIImpl` expects an instance of the `IMediaStore` as a constructor argument that is assigned to the field `iMediaStore`.

This mapping describes the relations between different abstractions of a software system. One realization of a component-based architecture description language is the PCM. It can be used to predict quality attributes of a software system via simulation of an abstract architecture specification before its implementation. To achieve this, different roles work on different views of the architecture, which concern abstract specifications of service realizations (component developer), the assembly of components (system architect), the resource environment executing the system (system deployer), and typical usage scenarios (domain expert) used for simulation.

3. Foundations

In the following, we introduce the foundations on view-based and orthographic software modeling and the formal constructs and notations that we use throughout this article.

3.1. View-based modeling

View-based development is a paradigm that addresses the problems of separation of concerns and reduction of accidental complexity. It does so by arranging information in *views*. The terms *view*, *view point*, as well as *view type* are used throughout the modeling domain in several ways, going back to approaches as early as the 1980s (Wood-Harper et al., 1985; Finkelstein et al., 1992). The ISO standard 42010 (International Organization for Standardization, 2011) contains a broad definition for "architecture view" and "architecture viewpoint". The standard also contains a distinction between *projective* and *synthetic* approaches: While projective approaches use an underlying repository as the system description and derive views that are only projections, in synthetic approaches a composition of views and their connection with correspondences forms the system description. Recent surveys (Brunelière et al., 2017; Cicchetti et al., 2019) provide an overview of existing view-based modeling approaches.

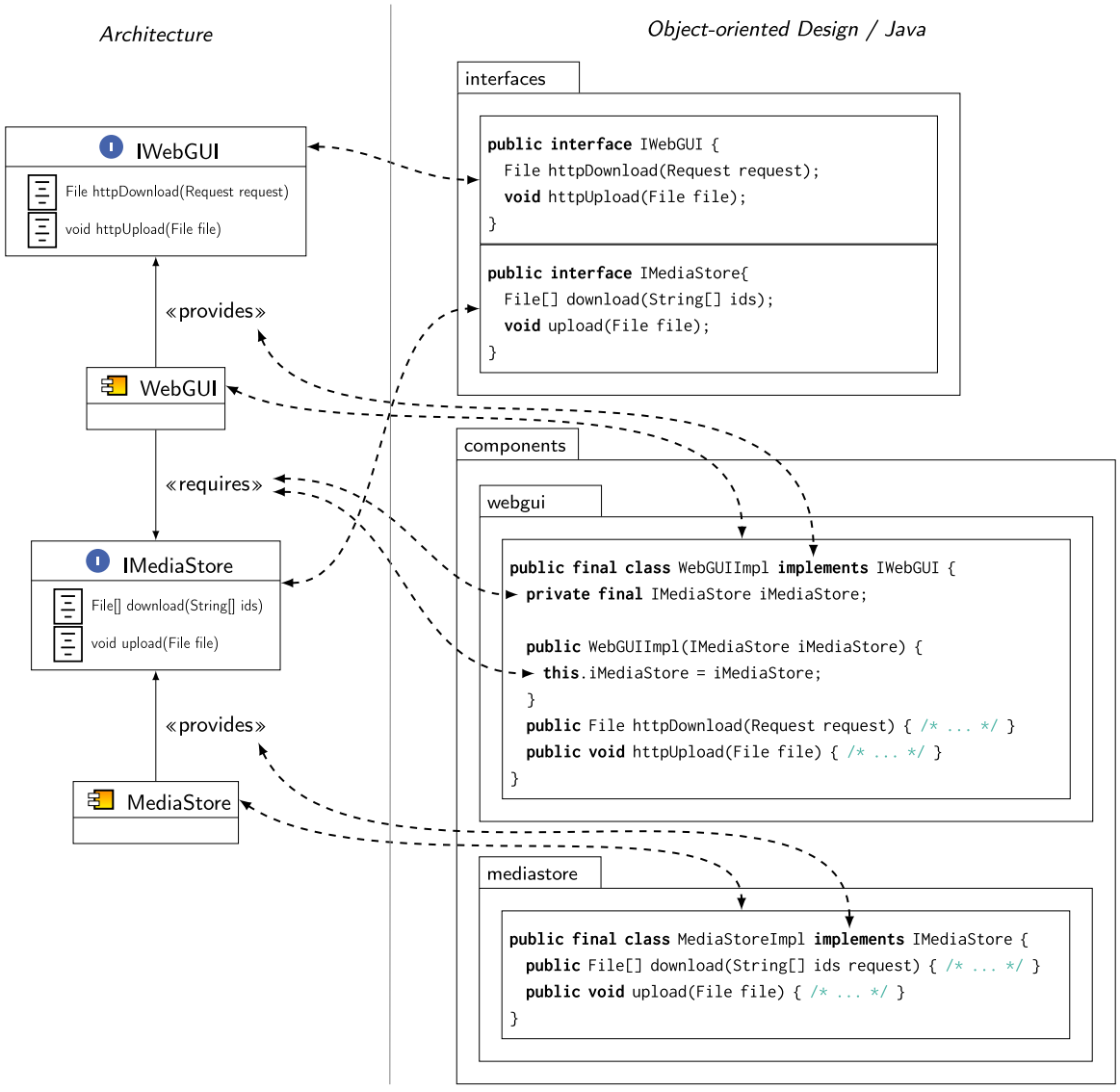


Fig. 2. Mapping between the MediaStore component model and its object-oriented realization. The package structure is depicted in UML syntax, classes and interfaces are represented in Java syntax. The root package and the component repository are omitted for simplicity. Arrows depict elements that are mapped to each other.

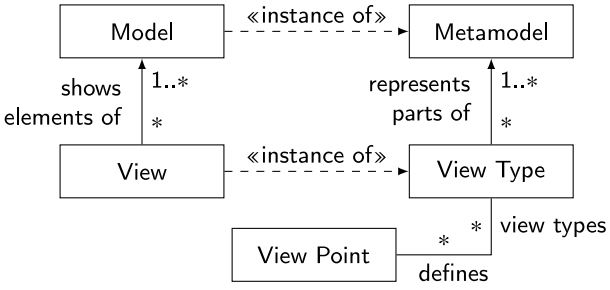


Fig. 3. View and view type terminology, based on Burger (2014).

In this paper, the term *view* denotes a projective concept, as described by Atkinson:

> "a view is a normal model which just happens to have been generated dynamically for the purpose of allowing a user to see the system from a specific viewpoint" (Atkinson et al., 2010, Sec. 3.1)

We use the term *view type*, as introduced by Goldschmidt et al. (2012), for the metamodel of a view. A view type describes the kinds of elements and relations that a view can contain. A *view point* groups one or more view types by the concerns that they serve (International Organization for Standardization, 2011). The relation of these terms is depicted in Fig. 3.

3.2. Orthographic software modeling

Orthographic Software Modeling (OSM) is a view-based development paradigm (Atkinson et al., 2008, 2010). It is built around the concept of a Single Underlying Model (SUM). There is no definition of this term in literature, so we provide one here.

Definition 1 (*Single Underlying Model (SUM)*)**.** A SUM is a complete definition of a system and contains all known information about it. It contains no redundant or implicitly dependent information and is thus always free of contradictions, i.e., inconsistencies.

Definition 2 (*SUM Metamodel*)**.** The formalism that is used to describe a SUM is called a *SUM metamodel*.

Table 1
Elements, sets and notations.

(Meta-)model elements	
e	Metamodel element (meta-class, attribute/reference)
$\underline{e}$	Instance of an element e
$I(e) = \{\underline{e}_1, \underline{e}_2, \ldots\}$	All possible instances of an element $e \in \mathcal{M}$
(Meta-)models	
$\mathcal{M} = \{e_1, e_2, \ldots\}$	Universe of all metamodel elements
$M \subseteq \mathcal{M}$	Metamodel
$\mathcal{I}_M = \bigcup_{e \in M} I(e)$	Universe of all element instances of a metamodel M
$\underline{M} \subseteq \mathcal{I}_M$	Instance of metamodel $M \subseteq \mathcal{M}$, i.e., a model
$\mathfrak{I}_M \subseteq \mathcal{P}(\mathcal{I}_M)$	Set of all valid instances of metamodel M
Notations	
$\langle A \rangle = \langle a, b, \ldots \rangle$	Notation for a tuple $\langle A \rangle$ of elements, models, etc.
Func	Notation for a function Func

A SUM metamodel can be expressed with well-defined description formalisms, such as domain-specific languages, metamodels, or general-purpose programming languages.

In OSM, the SUM is displayed or manipulated exclusively through partial, user-specific, and customizable views. Model transformations create these views dynamically from the SUM (Burger et al., 2014; Tunjic and Atkinson, 2015; Burger and Schneider, 2016; Atkinson and Tunjic, 2017). The views allow the users to make modifications to the system and operate on a temporarily inconsistent state of the system description. While the SUM is always consistent, a view may contain inconsistent information, such as invalid Java code, like it is necessary when developing a system. Whenever a developer wants to share his or her progress with others, he or she can propagate the changes back to the SUM with the transformation used to create the view. The transformation ensures that the performed changes lead to a consistent state of the SUM afterwards. Tolerating inconsistencies beyond the modification of views is a subject of current research and part of future work (see Section 11.2). The view types are organized in dimensions to facilitate navigation through the SUM. Ideally, these dimensions are chosen in such a way that they are orthogonal to each other. OSM defines a development process with two roles: The *methodologist* is responsible for creating SUM metamodels and view types. The *developer* instantiates the SUM metamodel and modifies its instances using the generated views.

As suggested in the OSM approach (Atkinson et al., 2010), the metamodel of a SUM should be specific for the development process, the domain in which the system is to be used, and the modeling standards that have to be supported. It is evident that it is impossible to define a SUM metamodel that serves all possible software and system development scenarios. Many projective view-based approaches, such as DUALLy (Malavolta et al., 2010), or KobrA (Atkinson et al., 2008), rely on a fixed and pre-defined SUM metamodel, which is supposed to represent all concepts necessary for the modeling task. Those SUM metamodels are *monolithic*, as they represent all concepts in one, potentially large metamodel, whereas *modular* approaches separate the concepts into smaller, more specific metamodels.

3.3. Notation

For the formalization of our concepts, we use the mathematical set notation of the Essential MOF (EMOF) established by Burger (2014, Sec. 2.3.2), which is based on the notation of the OCL specification (Object Management Group, 2014, A.1). We introduce the symbols that are relevant for the remainder of this article in Table 1. They describe metamodels and model elements, but deliberately lack the more sophisticated concepts of Burger (2014). In general, a metamodel M is represented as a set of elements. The elements in a metamodel can be meta-classes, attributes, references, and so on. For example, the metamodels in Fig. 1 consist of elements representing the meta-classes such as `Package`, the attributes such as `name`, and the references between them, such as the `subpackages` reference between a `Package` and its `subpackages`. For the following definitions, this distinction is not relevant, so we usually only write $e \in M$ for an element of a metamodel. A metamodel M is a subset of the universe, i.e., the set of all possible metamodel elements $\mathcal{M}$.

Moreover, we use the M3–M1 level hierarchy of the UML to describe elements at different modeling levels: metamodels reside at level M2, while instances are at M1. Elements at level M1 are written in underlined type, e.g., $\underline{e}$ indicating an instance of a metamodel element e. We denote the set of all instances of an element e as $I(e)$. The universe of all possible instances of the elements of a metamodel M is defined as $\mathcal{I}_M$, of which each model is a subset. Without loss of generality, we assume that two metamodels are always disjoint, which in practice is usually achieved by assigning a unique namespace to each metamodel. The set of all models that are valid instances of a metamodel M is characterized by a subset of the power set of all element instances $\mathfrak{I}_M$. We do not discuss how that subset of valid instances may be retrieved or defined but assume it as given according to definitions in existing metamodel formalisms such as the OCL specification (Object Management Group, 2014, A.1) or Ecore (Steinberg et al., 2008).

In addition to sets of elements for describing metamodels and models, we use tuples to describe elements involved in rules that formalize consistency. We extend the subset operator to define a tuple as a subset of a set. We consdier a tuple $\langle a_1, \ldots, a_m \rangle$ a subset of a set A if all its elements are contained in the set:

$$\langle a_1, \ldots, a_m \rangle \subseteq A :\Rightarrow \{a_1, \ldots, a_m\} \subseteq A$$

Similarly, an element a is considered an element of a tuple $\langle a_1, \ldots, a_m \rangle$ if it is contained in the tuple:

$$a \in \langle a_1, \ldots, a_m \rangle :\Rightarrow \exists i \in \{1, \ldots, m\} : a = a_i$$

4. The Vitruvius approach

In this section, we present an overview of the Vitruvius[1] approach for consistency in view-based system development. Vitruvius is based on the OSM paradigm (see Section 3.2). It provides a solution to the problem of how SUM metamodels should be constructed, since defining a SUM metamodel according to Definition 2, which has no redundancies or implicit dependencies, is hard to achieve (Meier et al., 2019, 2020): We propose the concept of a *virtual SUM metamodel (V-SUM metamodel)*, which has an internal structure of modularized, coupled metamodels, but, externally, appears and can be used as if it were a single, *monolithic* (i.e., non-modularized) metamodel. This especially means that its instances are also free of inconsistencies, like ordinary SUMs according to Definition 1. In contrast, it does not achieve this consistency guarantee by being free of redundancies and implicit dependencies, but by preserving consistency of them internally. Using a virtual SUM metamodel over a monolithic SUM metamodel is motivated by the following considerations:

Compatibility to existing metamodels: The development of software and systems usually has to adhere to pre-existing, externally defined languages and standards, with, for example, specific editors, simulators, generators, and other tools. In Vitruvius, existing metamodels can be included as parts of a V-SUM metamodel without modifications. Existing instances of the metamodels, tools, and transformations can be reused.

[1] **Vi**ew-Cen**tr**ic Engineering **U**sing a **Vir**t**u**al **S**ingle Underlying Model, after the Roman architect *Marcus Vitruvius Pollio* (ca. 80-70 B.C.–15 B.C.).

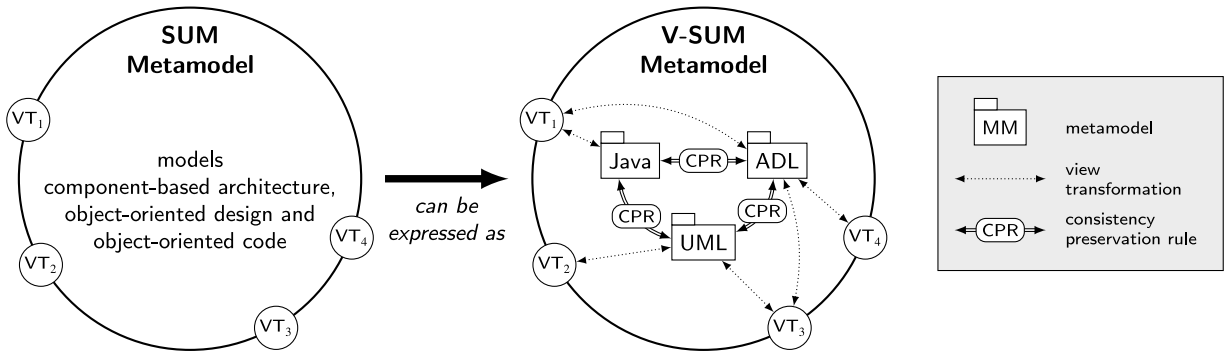


Fig. 4. A monolithic SUM metamodel for the running example in comparison to a V-SUM metamodel.

Explicit consistency definition: It is difficult to define a monolithic SUM metamodel that is free of redundancies (Meier et al., 2020). First, the methodologist defining such a SUM metamodel has to know about all information about the system for all involved roles, whereas defining consistency in terms of (binary) relations supports distributed knowledge about all information dependencies. Second, ensuring absence of redundancies in such large metamodels is overly complex. A V-SUM metamodel may contain redundancies and implicit dependencies, but the coupling between the metamodels specifies how consistency is checked and enforced.

Maintainability and evolvability: A major disadvantage of monolithic SUM metamodels is their poor maintainability due to missing modularity: A methodologist who carries out a modification to such a SUM metamodel must know all possible effects on every element in this metamodel to ensure consistency, correctness, etc. Thus, this person would have to have comprehensive knowledge of all domains represented in the SUM metamodel. In a V-SUM metamodel, the metamodels can be maintained separately by experts of the respective domain.

Reusability: The modular structure of the V-SUM metamodel eases the reuse of parts to build further scenario-specific V-SUM metamodels that are subsets or new arrangements of the metamodels in the V-SUM metamodel.

As an illustration, Fig. 4 depicts a SUM metamodel and a V-SUM metamodel for our running example. The internal structure of the monolithic SUM metamodel (left) is not shown; it contains redundancy-free representations of all concepts in the running example. The V-SUM metamodel (right) internally consists of three metamodels for Java, UML, and ADL, which are coupled by so called *Consistency Preservation Rules (CPRs)*. For the V-SUM metamodel and the SUM metamodel, the same four view types VT_1–VT_4 are defined. They are in part standardized diagram types, such as UML class diagrams (VT_2) or component diagrams (VT_4), and in part custom view types, such as Java source code enriched with architecture information (VT_1) and a combination of class and component diagrams (VT_3). In the V-SUM metamodel, there is also a CPR between Java and the Palladio Component Model ADL (Langhammer, 2017), which is specific for performing performance predictions based on the software architecture. In this case, the implementation of Java methods is related to a control flow abstraction with performance annotations in the PCM model. While this can be achieved with explicit consistency definitions (Langhammer, 2017), finding a representation in a SUM metamodel from which Java code and a suited control flow abstraction can both be projected is obviously not that simple.

In terms of the ISO 42010 (International Organization for Standardization, 2011), Vitruvius is a *hybrid* approach, combining *projective* and *synthetic* concepts. The internal representation constitutes a *synthetic* approach, because the system description is composed of the different models within the V-SUM. However, all views on which users operate are projected from this internal representation, which constitutes a *projective* approach for the outer views. The proposed benefit is to combine the advantage of projective approaches on the one hand, by enabling the definition of arbitrary views ad-hoc, and those of synthetic approaches on the other hand, by modularizing the metamodel. We have observed that, in general, more view types than metamodels are involved in a development process. For example, Fig. 4 depicts four view types for three metamodels, because some view types combine information from different metamodels, such as VT_3 combining object-oriented design in UML with architectural information in an ADL. In contrast to a purely synthetical approach, this separation of metamodels and view types reduces the number of CPRs that have to be defined.

The Vitruvius approach involves a two-part process: First, a methodologist creates the V-SUM metamodel by defining metamodels, CPRs, and view types that derive information from them. Second, one or more developers instantiate the V-SUM metamodel as a V-SUM, and use it to derive views and perform modifications to them. In this article, we focus on the first part (constructing a V-SUM metamodel). In its simplest form, a methodologist selects a set of metamodels that shall be used to develop a system. He or she then defines CPRs between all these metamodels to ensure that information is consistently propagated between all models upon modifications. Finally, he or she defines view types which extract and combine information from the metamodels, and which can be instantiated as views that are shown to or modified by developers of a concrete software system.

In summary, Vitruvius proposes a development process that is based on the V-SUM concept and consists of the following essential parts, which we will explain in more detail in the subsequent sections:

Development process: Vitruvius defines a process for the creation, instantiation, and usage of V-SUM metamodels. In the following, we therefore start with a formalization of the concept of a V-SUM in Section 5 to clarify its properties and its behavioral relation to a SUM according to Definition 1. We continue with an overview of the V-SUM metamodel construction and operation process in Section 6. We also discuss a process for integrating existing models into a V-SUM in Section 8.

Consistency preservation: Vitruvius defines a technique for preserving consistency between instances of the metamodels within a V-SUM metamodel. We discuss the essential part of the V-SUM metamodel construction, the consistency preservation, in more detail in Section 7 and especially propose specialized languages that support the definition of consistency and its preservation.

View generation: Vitruvius defines a mechanism for deriving projective views from a V-SUM. We define our formal notion of views and view types in Section 5.4. Since the view generation is mostly independent from the way the SUM is constructed (i.e., whether it is a monolithic SUM or a V-SUM), we refer to existing and current work, e.g., Burger et al. (2014), Tunjic and Atkinson (2015), Burger and Schneider (2016) and Atkinson and Tunjic (2017), for that topic and do not discuss it in more detail in this article.

5. Modeling concepts for V-SUMs

In this section, we present the V-SUM concept. We formally define our notion of consistency, an inductive approach to its preservation, and its realization in a V-SUM, as well as the properties of view types and views to be defined on a V-SUM. The different concepts for describing a V-SUM and its metamodel are depicted in Fig. 5. The purpose of this formalization is to precisely specify the properties of a V-SUM metamodel, the concepts that are necessary to describe it, and the behavior of its instances. We later use it in Section 6 to emphasize which parts of the practical application process of the approach reflect which necessary parts of the formalization. Additionally, we derive the justification for two specialized languages for consistency preservation presented in Section 7 at proper levels of abstraction, which are inherently induced by the formalization. It constitutes our contribution C1.

5.1. Consistency

The following definitions are in part based on Kramer (2017, ch. 4). They describe under which circumstances we consider models to be consistent with each other.

Definition 3 (*Consistency Rule*)**.** Let M_l and M_r be metamodels and let $\langle E_l \rangle = \langle e_{l_1}, \ldots, e_{l_m} \rangle$, $e_{l_i} \in M_l$ and $\langle E_r \rangle = \langle e_{r_1}, \ldots, e_{r_n} \rangle$, $e_{r_i} \in M_r$ be two tuples of elements of those metamodels. In addition, let $COND_{\langle E_l \rangle} \subseteq I(e_{l_1}) \times \cdots \times I(e_{l_m})$ be a condition for $\langle E_l \rangle$, and let $COND_{\langle E_r \rangle} \subseteq I(e_{r_1}) \times \cdots \times I(e_{r_n})$ be a condition for $\langle E_r \rangle$.

A consistency rule for the element tuples $\langle E_l \rangle$ and $\langle E_r \rangle$ is a relation $CR \subseteq COND_{\langle E_l \rangle} \times COND_{\langle E_r \rangle}$. It contains pairs of instance tuples for $\langle E_l \rangle$ and $\langle E_r \rangle$ that indicate consistency of two models if they co-occur.

In short, a consistency rule is a set that contains pairs of tuples of model elements, each describing that if one of these tuples occurs in a model, another tuple related to it by *CR* has to occur in another model that shall be kept consistent. The relevant element tuples in each of the models are represented by a *condition*. Such a condition enumerates the tuples for which consistency is restrained in some way, i.e., it defines a condition for a consistency rule to be applied. Be aware that it may be allowed that certain instance tuples of $\langle E_l \rangle$ and $\langle E_r \rangle$ may not be restrained regarding consistency and are thus not part of the consistency rule. Such a condition contains a possibly infinite set of tuples of element instances that list model elements that fulfill a condition within one model. In consequence, the consistency rule relating two conditions will, in general, also be infinite.

Example 1. In our running example, introduced in Section 2, we informally introduced a rule that states that each instance of the ADL meta-class `Component` shall be represented by an instance of the object-orientation meta-class `Package` as well as the meta-class `Class`, so that the `Class` is contained in the `contents` reference of the `Package` and that the `name` attribute of all of them is equal, apart from the `Class` additionally having an `Impl` suffix. A consistency rule according to Definition 3 expressing this would consist of two conditions, one containing all components and one containing all pairs of packages and classes, with the class name being the package name with an `Impl` suffix and the class being contained in the `content` relation of the package. Note that all other pairs of a package and a class are not contained in that condition. This is why we need to define a condition that restricts the elements for which consistency is defined. The consistency rule then contains those pairs of these two conditions in which the component name is equal to the package name (and thus the class name without the `Impl` suffix). In consequence, it is an infinite set.

The manifestation of such a rule in concrete models is expressed by the following definition for *fulfilling* a consistency rule. It especially reflects that one tuple of elements may be considered consistent to one of several other tuples, like a specific UML class may be considered consistent to all Java classes with the same method signatures but all kinds of implementations.

Definition 4 (*Consistency Rule Fulfillment*)**.** Let $M_1, \ldots, M_n$ be metamodels, let $\langle \underline{M} \rangle \in \mathfrak{I}_{M_1} \times \cdots \times \mathfrak{I}_{M_n}$ be a tuple of instances and let $COND_{\langle E_l \rangle}$ and $COND_{\langle E_r \rangle}$ be two conditions on which a consistency rule $CR \subseteq COND_{\langle E_l \rangle} \times COND_{\langle E_r \rangle}$ is defined. We say that $\langle \underline{M} \rangle$ *fulfills CR* if, and only if,

$$\begin{aligned}
&\forall \langle c_l \rangle \in COND_{\langle E_l \rangle} : \big(\exists \underline{M}_l \in \langle \underline{M} \rangle : \langle c_l \rangle \subseteq \underline{M}_l \\
&\quad \Rightarrow \exists \langle \langle c_l \rangle, \langle c_r' \rangle \rangle \in CR : \exists \underline{M}_r \in \langle \underline{M} \rangle : \langle c_r' \rangle \subseteq \underline{M}_r \big) \\
&\wedge \forall \langle c_r \rangle \in COND_{\langle E_r \rangle} : \big(\exists \underline{M}_r \in \langle \underline{M} \rangle : \langle c_r \rangle \subseteq \underline{M}_r \\
&\quad \Rightarrow \exists \langle \langle c_l' \rangle, \langle c_r \rangle \rangle \in CR : \exists \underline{M}_l \in \langle \underline{M} \rangle : \langle c_l' \rangle \subseteq \underline{M}_l \big)
\end{aligned}$$

A tuple of models fulfills a consistency rule *CR* if for each of its condition elements that is contained in one model, a condition element to which it is considered consistent, i.e., for which a pair of these two is contained in *CR*, is contained in another model.

Example 2. Referring to Example 1, this means that if a model contains a component, then another model must contain an appropriate package with a contained class, as defined in the consistency rule, and vice versa. You may imagine a consistency rule that states that a component may either be mapped to a package and a class having the component name, or appending an `Impl` suffix like in the example. In this case, the consistency rule would contain two pairs for each component, one having the class with and one without the suffix at the pair's right-hand side. According to Definition 4, if the ADL model contains a component, the object-orientation model has to contain one of these two package and class representations, but not both of them are required.

Remark 1. If the, technically, same metamodel is used to describe two or more disjoint models, this is not explicitly reflected by the given formalism. Virtually duplicating the metamodel, however, solves the problem and allows to describe consistency between multiple instances of the same metamodel. This may, for example, be the case if component and class models shall be kept consistent, although they are instances of the same UML metamodel. The formalism does not support consistency within a single model on purpose, because it should be up to the metamodel itself to ensure that its instances are internally consistent.

One might argue that consistency is usually traced by means of a *trace* or *correspondence model*, which stores the pairs of element tuples in models that fulfill a consistency rule. While the realization of the Vitruvius approach contains an explicit correspondence model, we do not explicitly consider it in this formalism for two reasons. First, a trace model is only necessary in practice if no identifying information for related elements is present or if performance is to be improved. However, without loss of generality we can assume such identifying information in a

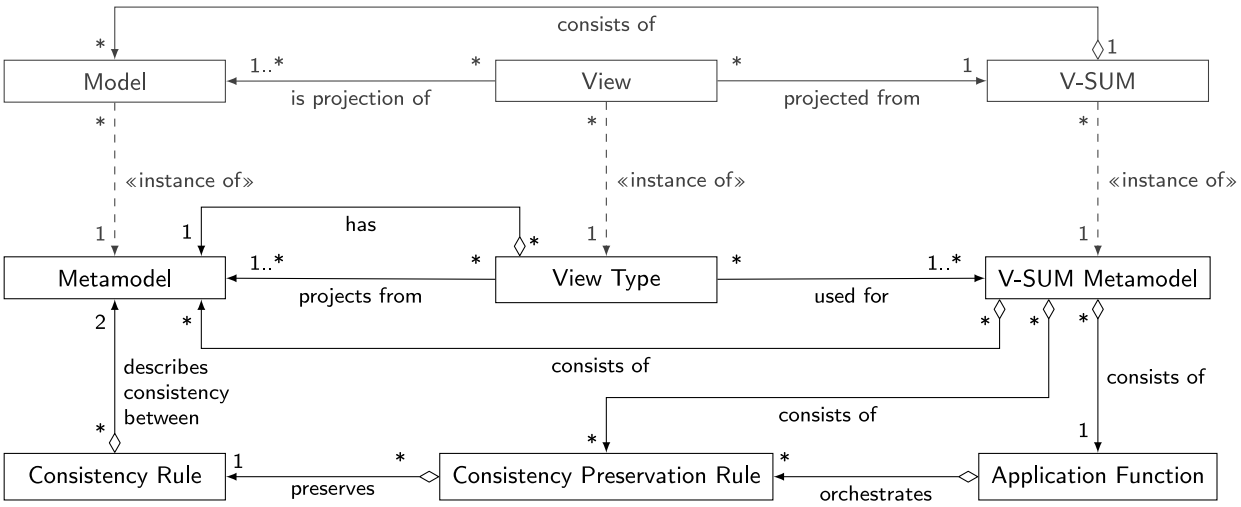


Fig. 5. A conceptual model of the relevant concepts for describing a V-SUM and its metamodel, as well as their relations.

formalism, like it is usually done in formalisms of transformations such as the one used in Stevens (2010). Second, a trace model can, from a theoretical perspective, be treated as a usual model, thus always defining consistency between one concrete and one correspondence model. This conforms to the fact that each n-ary relation can be expressed by binary relations to an additional model (in this case the trace model), as discussed in Stevens (2020) and Cleve et al. (2019).

Based on those definitions of consistency rules and their fulfillment, we can define consistency of a tuple of models.

Definition 5 (*Consistency*)**.** Let $M_1, \ldots, M_n$ be metamodels, let $\langle \underline{M} \rangle \in \mathfrak{I}_{M_1} \times \cdots \times \mathfrak{I}_{M_n}$ be a tuple of instances of them and let $\langle CR \rangle = \langle CR_1, \ldots, CR_m \rangle$ be a tuple of consistency rules. We say that $\langle \underline{M} \rangle$ is *consistent according to* $\langle CR \rangle$ if, and only if,

$$\forall CR \in \langle CR \rangle : \langle \underline{M} \rangle \text{ fulfills } CR$$

5.2. Inductive consistency preservation

The Vitruvius approach uses *delta-based* consistency preservation (Diskin et al., 2011), which means that we define the preservation of consistency inductively and as consequences of changes to a model. The advantage of delta-based over *state-based* approaches is that they provide information about the modifications between two model states, whereas state-based approaches have to estimate these modifications from the (infinite number of) possible change sequences between two model states (Diskin et al., 2011). As discussed by Kusel et al. (2013), delta-based approaches, however, have a drawback over state-based approaches: They depend on the development environment, because they require an observer that tracks model changes, whereas state-based approaches only require the comparison of two model states. Nevertheless, state-based differences can always be converted into a sequence of changes for the transition from the old to the new state. Thus, in cases when a change observer cannot be integrated, a delta-based approach has no advantage over a state-based one, but also no drawbacks.

We consider a change to a model $\underline{M}$ to be any modification transforming one model to another, which can either be the creation or deletion of an element in $\underline{M}$, an attribute or reference update for an element in $\underline{M}$, or any combination of them. More formally, we define a change as follows.

Definition 6 (*Change*)**.** Let $\mathfrak{I}_M$ be the set of all valid instances of a metamodel M. A change is a function:

$$\delta_M : \mathfrak{I}_M \to \mathfrak{I}_M$$

We say that δ_M *is a change in* $\mathfrak{I}_M$.

A change defines how each valid instance of a metamodel is transformed into a new one. For example, assume a change that renames a specific element: For all models that contain the element to rename, this function will return a model that is equal to the input except for the changed name in the specific element. For most input models (those that do not contain the element to rename), it will behave like the identity function. We denote changes that only affect one element value, such as an attribute or reference value, as *atomic changes*. All other changes can be composed of atomic changes and are thus referred to as *composite changes*. Atomic changes according to Definition 6 return the same output model as the input model apart from changing one of the model elements. Modifying several elements, i.e., performing a composite change, can be realized as a composition of such atomic changes, i.e., functions according to Definition 6.

Definition 7 (*Consistency Preservation Rule*)**.** For a tuple of metamodels $\langle M \rangle = \langle M_1, \ldots, M_n \rangle$, let $\mathfrak{I}_{\langle M \rangle} = \mathfrak{I}_{M_1} \times \cdots \times \mathfrak{I}_{M_n}$ be the set of all possible tuples of their instances. In addition, let $\Delta = \{\langle \delta_{M_1}, \ldots, \delta_{M_n} \rangle \mid \delta_{M_i} \text{ is a change in } \mathfrak{I}_{M_i}\}$ be the set of all tuples of possible changes in $\mathfrak{I}_{M_1}, \ldots, \mathfrak{I}_{M_n}$.

A consistency preservation rule for a consistency rule CR is a partial function $\textsc{Cpr}_{CR} : \mathfrak{I}_{\langle M \rangle} \times \Delta \to \mathfrak{I}_{\langle M \rangle} \times \Delta$, such that

$$\begin{aligned}
&\forall \langle \underline{M} \rangle = \langle \underline{M}_1, \ldots, \underline{M}_n \rangle \in \mathfrak{I}_{\langle M \rangle} : \\
&\forall \langle \delta \rangle = \langle \delta_{M_1}, \ldots, \delta_{M_n} \rangle \in \Delta : \exists \langle \delta' \rangle = \langle \delta'_{M_1}, \ldots, \delta'_{M_n} \rangle \in \Delta : \\
&\quad \langle \underline{M} \rangle \text{ fulfills } CR \Rightarrow \big(\langle \langle \underline{M} \rangle, \langle \delta' \rangle \rangle = \textsc{Cpr}_{CR}\big(\langle \underline{M} \rangle, \langle \delta \rangle \big) \\
&\qquad \wedge \langle \delta'_{M_1}(\underline{M}_1), \ldots, \delta'_{M_n}(\underline{M}_n) \rangle \text{ fulfills } CR \big)
\end{aligned}$$

$\textsc{Cpr}_{CR}$ expects originally consistent models and a tuple of changes. It returns the unmodified tuple of models $\langle \underline{M} \rangle$ together with the modified changes $\langle \delta' \rangle$, which, if applied to the original models, again yield models that are consistent according to CR. While returning the set of modified changes would be conceptually sufficient, also returning the unmodified models eases later composition of those consistency preservation rules.

Example 3. For our example regarding consistency between a component and its representation as a package and a class

(see Examples 1 and 2), a consistency preservation rule would return the input (i.e., act like the identity function) if the change does not affect any component, package or class. In cases where the given change affects a component, the function may adapt the representation as package and class in the resulting change. More precisely, if the input change adds a component, the output change could also add an appropriate package and class. If the input change modifies the name of the component, the output change may modify the name of the package and class appropriately. And finally, if the input change removes a component, the output change can remove the package and the class. Changes to the package and the class can be propagated vice versa. It is up to the methodologist to specify what happens if, for example, only the package name is changed, i.e., whether the class name shall also be adapted or whether the component shall be removed, as the consistency rule does not apply anymore, because package and class do not fulfill the name condition. Note that in general there is no restriction regarding how a consistency preservation rule restores fulfillment of its consistency rule. In general there is more than one possibility how to restore consistency.

Using the terminology and properties of bidirectional transformations (Stevens, 2010), $\textsc{Cpr}_{CR}$ is *correct* by definition: The application of δ' to consistent models leads to models that are, again, consistent according to *CR*. $\textsc{Cpr}_{CR}$ is, however, not *hippocratic* (Stevens, 2010) in general, i.e., it may modify δ even in cases when the application of δ would already lead to consistent models. Hippocraticness is not a necessary property for our consistency preservation approach, but may be desirable in practice. Furthermore, Definition 7 does not make any statements on the effects of $\textsc{Cpr}_{CR}$ on consistency rules other than *CR*; we cover these in Definition 8.

5.3. Virtual Single Underlying Models

With the definitions for consistency and its preservation, we can now define a V-SUM metamodel and its instances.

Definition 8 (*V-SUM Metamodel*)**.** Let $\langle M_{VSUMM}\rangle = \langle M_1, \dots, M_n\rangle$ be a tuple of metamodels and let $\mathfrak{I}_{\langle M\rangle} = \mathfrak{I}_{M_1} \times \dots \times \mathfrak{I}_{M_n}$ be the (possibly infinite) set of all possible tuples of their valid instances. Let $\langle \textsc{Cpr}_{VSUMM}\rangle = \langle \textsc{Cpr}_{CR_1}, \dots, \textsc{Cpr}_{CR_k}\rangle$ be a tuple of consistency preservation rules for those metamodels based on their (implicit) consistency rules $\langle CR_{VSUMM}\rangle = \langle CR_1, \dots, CR_k\rangle$, let Δ be the set of all change tuples that can be performed on any of the metamodels and let $\textsc{App} : \mathfrak{I}_{\langle M\rangle} \times \Delta \to \mathfrak{I}_{\langle M\rangle}$ be a consistency-preserving change application function. A virtual SUM metamodel (V-SUM metamodel) is a structure

$$VSUMM := \big(\langle M_{VSUMM}\rangle, \langle \textsc{Cpr}_{VSUMM}\rangle, \textsc{App}\big)$$

such that

$$\begin{aligned}
&\forall \langle \underline{M}\rangle = \langle \underline{M}_1, \dots, \underline{M}_n\rangle \in \mathfrak{I}_{\langle M\rangle} : \forall \langle \delta\rangle = \langle \delta_{M_1}, \dots, \delta_{M_n}\rangle \in \Delta : \\
&\exists \langle \delta'\rangle = \langle \delta'_{M_1}, \dots, \delta'_{M_n}\rangle \in \Delta : \exists \textsc{Cpr}_1, \dots, \textsc{Cpr}_m \in \langle \textsc{Cpr}_{VSUMM}\rangle : \\
&\quad \big(\langle \underline{M}\rangle \text{ consistent according to } \langle CR_{VSUMM}\rangle \Rightarrow \\
&\quad\quad \langle \langle \underline{M}\rangle, \langle \delta'\rangle\rangle = \textsc{Cpr}_1 \circ \dots \circ \textsc{Cpr}_m\big(\langle \underline{M}\rangle, \langle \delta\rangle\big) \wedge \\
&\quad\quad \textsc{App}\big(\langle \underline{M}\rangle, \langle \delta\rangle\big) = \langle \delta'_{M_1}(\underline{M}_1), \dots, \delta'_{M_n}(\underline{M}_n)\rangle \wedge \\
&\quad\quad \textsc{App}\big(\langle \underline{M}\rangle, \langle \delta\rangle\big) \text{ consistent according to } \langle CR_{VSUMM}\rangle \big)
\end{aligned}$$

A V-SUM metamodel consists of a tuple of metamodels, a tuple of consistency preservation rules, and a change application function. The preservation rules normatively imply the underlying consistency rules according to Definition 3. The change application function App can be seen as an interface for performing changes on instances of the V-SUM metamodel. Passing a change to it ensures that if the contained models were consistent before, they will also be consistent afterwards by executing the consistency preservation rules of the V-SUM metamodel. The function is also responsible for determining a proper execution order of consistency preservation rules such that consistency regarding all consistency rules is achieved. Obviously, this orchestration is not trivial, since the execution of one consistency preservation rule may easily lead to the violation of another consistency rule. It is part of current research (Cleve et al., 2019) and especially our current and future work (Klare, 2018; Klare et al., 2019) to investigate how such an orchestration can be generically defined, whether further assumptions have to be made to the consistency preservation rules, and in which sense it has to operate conservatively. Some recent research proposes to find a spanning tree (or a directed acyclic graph) of the transformations for each performed change, e.g. Stevens (2020). However, that approach makes the restrictive assumption that there is always an order such that an early executed transformation does not depend on the results of a later executed one, which is unrealistic. We yet investigated an App function that performs a fixed-point iteration of the consistency preservation rules, which has the benefit that it can be used generically for arbitrary V-SUM metamodels, as reflected by the multiplicity in Fig. 5. Although that realization is theoretically prone to perform an endless iteration, in recent studies we found a categorization of occurring problems that allows to avoid most of them already by proper construction of consistency preservation rules (Klare et al., 2019). This helps the methodologist to define consistency preservation rules in a way that there is always an execution sequence of them that terminates with a consistent state of the models, which is found by that generic App function.

With the definition of V-SUM metamodels, we can now formally define their instances.

Definition 9 (*V-SUM*)**.** A virtual SUM or V-SUM is a structure $\underline{VSUM}$ of a tuple of models $\langle \underline{M}_{VSUM}\rangle \in \mathfrak{I}_{M_1} \times \dots \times \mathfrak{I}_{M_n}$ and its V-SUM metamodel *VSUMM*:

$$\underline{VSUM} := \big(\langle \underline{M}_{VSUM}\rangle, VSUMM\big)$$

with $\langle \underline{M}_{VSUM}\rangle$ being consistent according to $\langle CR_{VSUMM}\rangle$.

To perform modifications on a V-SUM, a change can be applied using the change application function App of its V-SUM metamodel, delivering a new consistent state of the models. Starting with an empty tuple of models, and populating it using this function only, will inductively ensure consistency of the resulting tuple of models. This is why a V-SUM behaves like an ordinary SUM in terms of being free of inconsistencies according to Definition 1. We give a more pragmatic view on how this function works in the process description in Section 6.3.

Remark 2. The definitions of a V-SUM and its metamodel do not restrict the behavior of consistency preservation rules. Therefore, a V-SUM metamodel with a consistency preservation rule that deletes all model elements or at least those that are involved in a violated consistency rule would fulfill the definition. It is the responsibility of the methodologist role to define appropriate consistency preservation rules.

Remark 3. According to Definition 1, a SUM contains no contradicting information and is therefore always consistent. A SUM metamodel according to Definition 2 is a single metamodel without consistency rules, in accordance with Definition 5. In consequence, a SUM metamodel can be seen as a V-SUM metamodel with only one metamodel and an empty set of consistency preservation rules. This does, however, not mean that every metamodel

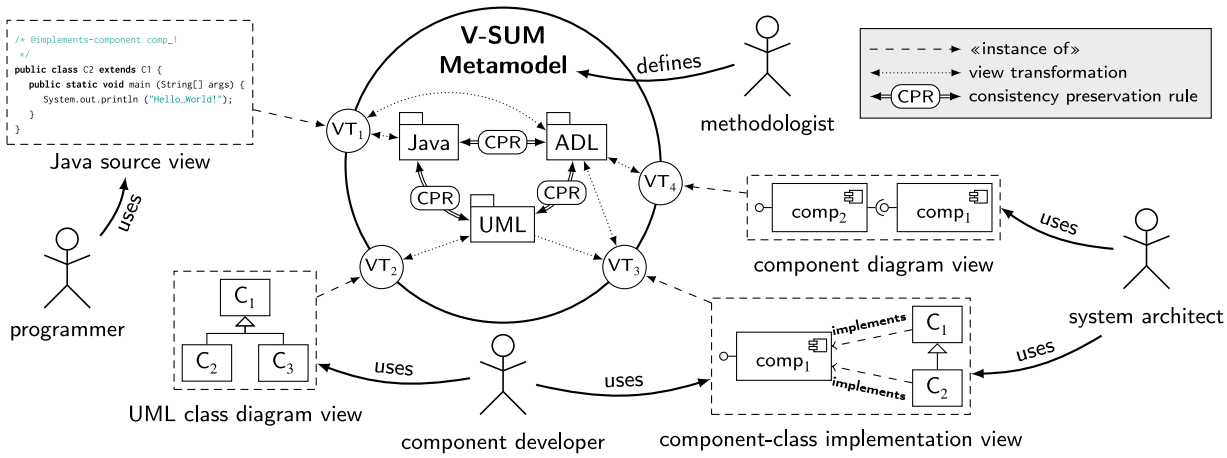


Fig. 6. V-SUM metamodel for the running example with exemplary roles and views on a possible instance.

is a SUM metamodel or V-SUM metamodel. It is up to a methodologist to define a SUM metamodel or V-SUM metamodel, thus if he or she states that there is some metamodel whose instances are free of redundancies and thus inconsistencies, he or she may call it a SUM metamodel or V-SUM metamodel with an empty set of consistency preservation rules. This is inherently the case if there are multiple metamodels whose instances do actually not share any overlap of information, and in other cases it is a matter of definition by the methodologist, because he or she prescribes when models are considered consistent.

5.4. Views and their generation

The definitions for specifying projective views on a V-SUM in this subsection are based on Burger (2014, Sec. 4.3). Such projective views enable a developer to perform changes according to Definition 6, which can then be applied to the change application function App of the V-SUM from which the view was projected to consistently propagate the changes to all models of the V-SUM.

Definition 10 (*View Type*)**.** Let $\mathcal{VT} \subseteq \mathcal{M}$ be the universe of view type elements. A *view type over a V-SUM metamodel* is a metamodel $VT \subseteq \mathcal{VT}$. A view type element $e_{VT} \in VT$ represents information of one or more elements in a V-SUM metamodel $e_{VSUMM} \in \mathcal{M}$.

Definition 11 (*View*)**.** An instance $\underline{VT} \subseteq \mathcal{I}_{VT}$ of a view type $VT \subseteq \mathcal{VT}$ is called *view*. Each element $\underline{e}_{VT} \in \mathcal{I}_{VT}$ represents at least one element $\underline{e}_{VSUM} \in \mathcal{I}_{\mathcal{M}}$.

Views are supposed to be projections from a V-SUM. Thus a function that calculates a view must be able to derive a view only given the models in a V-SUM.

Definition 12 (*View Definition Function*)**.** For a V-SUM metamodel *VSUMM*, let $\langle M_{VSUMM} \rangle = \langle M_1, \ldots, M_n \rangle$ be the tuple of its metamodels and let $\mathfrak{I}_{\langle M \rangle} = \mathfrak{I}_{M_1} \times \cdots \times \mathfrak{I}_{M_n}$ be the set of all possible tuples of their valid instances. A view type is defined by a *view definition function*, which calculates a view $\underline{VT}$ of a specific view type VT from a V-SUM with the model tuple $\langle \underline{M}_{VSUM} \rangle \in \mathfrak{I}_{\langle M \rangle}$:

$$\begin{aligned} \textsc{Def}_{VT} : \mathfrak{I}_{\langle M \rangle} &\rightarrow \mathcal{I}_{VT} \\ \langle \underline{M}_{VSUM} \rangle &\mapsto \underline{VT} \end{aligned}$$

The function induces a relation $\underline{rep}$ that expresses which model element(s) are represented by which view element(s):

$$\underline{rep} \subseteq \mathcal{I}_{\mathcal{M}_\cup} \times \mathcal{I}_{VT}$$

While $\textsc{Def}_{VT}$ is functional, $\underline{rep}$ is not, since a model element can be represented by multiple elements in one or more views.

6. A development process for V-SUMs

In this section, we first introduce the roles associated with the operation and construction of V-SUMs, then we depict the V-SUM operation process and discuss the individual process steps to construct a V-SUM metamodel. Finally, we discuss the domains in which the approach can be applied. This constitutes our contribution C2.

6.1. Roles

The development process for V-SUMs is an extension of the development process of the OSM approach (see Section 3.2). The process distinguishes between the *methodologist* role, responsible for the construction of a V-SUM metamodel, and the *developer* role, responsible for the operation of a V-SUM. An application of the process to the running example is depicted in Fig. 6.

Methodologist: Since each V-SUM metamodel is *method-specific*, i.e., depending on the involved languages (Atkinson et al., 2010), there is a special role for creating and maintaining the V-SUM metamodel, called the *methodologist*. A methodologist executes the three steps described in detail in Section 6.3: selecting metamodels, defining consistency preservation, and defining view types. Thus, the role is concerned with elements at the metamodel level.

Developer: A developer uses the view types that have been predefined by the methodologist to create, access, and manipulate the V-SUM for the system under consideration. He or she derives views from the V-SUM defined by the Def functions for view definition (Definition 12), modifies them and applies the performed changes to the V-SUM using the App function. He or she is thus concerned with elements at the model level and cannot change the V-SUM metamodel. Depending on the domain-specific development process, the developer role can be subdivided into further roles. A component-based development process for our running example may define the roles system architect, component developer, and programmer (see Fig. 6), each using a subset of the provided view types.

6.2. Operation of a V-SUM

A V-SUM metamodel, according to Definition 8, consists of metamodels, consistency preservation rules between them and, in addition, a set of view types. For a practical realization, we additionally use a so-called *correspondence model*, which serves as

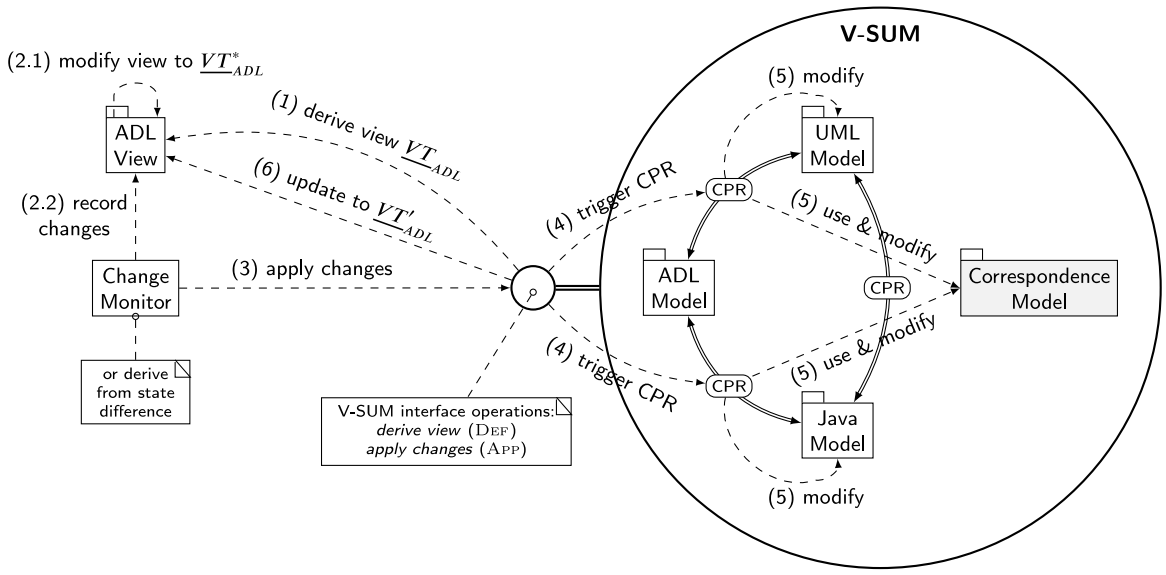


Fig. 7. Editing process with automatic application of changes for a V-SUM of the running example.

a trace model to identify which elements are related to each other in terms of fulfilling a consistency rule according to Definition 4. We already discussed in Section 5 why such a model is only relevant for a practical realization but not in the theoretical foundations discussed before.

When a developer instantiates a V-SUM metamodel, according to Definition 9, he or she can derive views based on the view types of the V-SUM metamodel and perform modifications on them, which are consistently propagated to the models of the V-SUM. Such a process of editing a view with a propagation of the changes to the V-SUM and preservation of consistency can be seen in Fig. 7. The view $\underline{VT}_{ADL}$ is derived from the V-SUM (1) using a *derive view* operation, which is based on the Def function (Definition 12), and modified (2.1) into a *dirty state* ($\underline{VT}^*_{ADL}$) where the elements in the view are not synchronized with the V-SUM. This modification is recorded by an appropriate change monitor (2.2), which may either save editing steps directly or derive them from state differences. Changes can now be applied (3) either manually by the developer, for example by saving the view, or automatically if a certain state of the view is reached. This is achieved by calling the *apply changes* operation provided by the V-SUM. This function takes the changes of the views and calls the appropriate consistency preservation rules (4) for these changes. These rules, in turn, modify the internal models and read and update so called *correspondences* (5) in the correspondence model to trace the relation of consistent elements. In consequence, the *apply changes* function triggering steps (4) and (5) realizes the App function of the V-SUM metamodel according to Definition 8.

If the consistency preservation process terminates in a consistent state, the view updates (6) to a non-dirty state (indicated as $\underline{VT}'_{ADL}$), which is possibly an update regarding $\underline{VT}^*_{ADL}$ due to modifications performed by the consistency preservation rules. This updated view can be edited again.

6.3. Construction of a V-SUM metamodel

The methodologist has to execute the following steps to construct a V-SUM metamodel: selecting metamodels, defining consistency preservation, and defining view types. In the simplest case, they are executed in this order. We present scenarios with different combinations in Section 6.3.4.

6.3.1. Selecting metamodels

A V-SUM metamodel contains a set of metamodels, as defined in Definition 8. Depending on the domain for which the system is developed, the methodologist can include metamodels for different purposes (e.g., security, performance, real-time properties) or development paradigms (e.g., component models, class models). It is often beneficial to reuse existing metamodels, but it is also possible to create new metamodels specifically for the usage within a V-SUM metamodel.

In our running example, three languages are used: an ADL, UML, and Java. The V-SUM metamodel for the example is depicted in Fig. 6. We assume that the ADL is already based on a metamodel definition, as is UML, so the methodologist can included them into the V-SUM metamodel without further effort. Java as a grammar-based textual language can, e.g., be expressed using JaMoPP (Java Model Printer and Parser) (Heidenreich et al., 2010).

If properties of a system shall be described in a language for which a metamodel is not provided, e.g., if existing and especially proprietary tools are used, the methodologist has to create the metamodel or choose a third-party metamodel. For many popular tools, metamodels, especially based on EMOF, have been reverse-engineered, such as MATLAB/Simulink (Heinzemann and Becker, 2013; Son et al., 2012; Armengaud et al., 2011). Additionally, EMF provides an importer for XML-based metamodel specifications (Steinberg et al., 2008, pp. 86). Most tools, even from other engineering domains, provide XML-based representations of their models for interchange purposes, such as the electronic circuit design tool EPLAN (Gischel, 2013) or the exchange format for automation system design AutomationML (AutomationML e.V. (GI), 2018), which eases the integration of such tools into the presented EMOF-based approach.

6.3.2. Defining consistency preservation

Vitruvius uses an inductive approach to preserve consistency of instances of a V-SUM metamodel. Changes passed to the application function App of a V-SUM trigger the appropriate Consistency Preservation Rules (CPRs). The methodologist can realize CPRs by defining incremental model transformations, which update one model after another was modified. Transformations can be defined in a declarative or imperative way, depending on the transformation language. The methodologist can use imperative transformation languages to define CPRs according to Definition 7, which specify how a consistent state regarding some

Consistency Rule (CR) is achieved after a change. Alternatively, the methodologist can use declarative languages for defining CRs according to Definition 3, which specify conditions under which models are in a consistent state, without having to define how to reach this state after a change. In declarative languages, CPRs are derived from a specification of CRs, i.e., constraints are transformed into routines that restore fulfillment of the CRs.

Highly declarative languages restrict the expressiveness of consistency preservation to relations that are covered by the provided language constructs, as stated by Kusel et al. (2013), but improve compactness and usually provide bidirectionality. For example, a simple declarative transformation between ADL components and UML packages and classes requires the definition of only one CR, whereas an imperative approach requires the specification of CPRs for each possible modification of their instances, at least their creation and deletion in both directions. Even in case in which the declarative language is sufficiently expressive, methodologists usually want to be able to affect the way consistency is preserved (Stevens, 2020), and will thus favor a language with imperative concepts.

Therefore, Vitruvius contains specific languages to provide both: The *Mappings language* is used for declarative, bidirectional specification of CRs. The *Reactions language* is used for imperative, unidirectional specification of CPRs. Both languages are discussed in detail in Section 7.

When CPRs are applied, affected elements in other models have to be identified either based on their data, e.g., by searching elements with a matching name, or by using explicitly stored traces, as, for example, employed by ATL (Jouault and Kurtev, 2006) and discussed by Czarnecki and Helsen (2006). We use such explicit traces, called *correspondences*, in Vitruvius to retrieve corresponding elements from a *correspondence model* (see Fig. 7) even when elements do not contain identifying data.

Consistency preservation mechanisms can automatically restore consistency in specific cases. However, depending on the complexity of a consistency rule and the difference in the level of abstraction of the involved metamodels, further information may be required to repair inconsistency: A mechanism can either restrict the possible solutions by implementing a certain strategy, or ask the user for a clarification of intent. Some approaches calculate all possible solutions for restoring consistency, e.g., using answer set programming (Eramo et al., 2012) or satisfiability solving (Macedo et al., 2013). Vitruvius follows a different approach, as it gives methodologists the possibility to specify interactions with the developer who performed the inconsistency-introducing change. This gives methodologists the flexibility to implement different, dynamically selected consistency repair strategies without overwhelming the user with too many resolution options.

Finally, it is up to the methodologist to decide which kinds of inconsistencies the CPRs are supposed to handle. Since the specification of when models are considered consistent is given by the V-SUM metamodel and its CPRs, a methodologist can normatively define the notion of consistency. There is no additional specification to which the CPRs must be somehow correct, although there will sometimes be an informal notion of how consistency should look like, such as for Java and UML. Nevertheless, the methodologist can still decide what the V-SUM is supposed to ensure and which inconsistencies shall not be considered by the mechanisms explicitly, e.g., to support specific workflows that require that some inconsistencies are not handled automatically.

6.3.3. Defining view types

Vitruvius is a view-based approach, so V-SUM elements are exclusively accessed by views. For each metamodel $M \in \langle M_{VSUMM} \rangle$, the methodologist must create at least one view type with a *projectional-complete* view type scope (Goldschmidt, 2011, Sec. 4.4). Such a view type represents all elements in M, which means that an instance of that view type contains all elements of an instance of M. The purpose of those view types is to replace the originally used metamodels. To ensure that existing tools can be used, the original metamodels must still be accessible by providing appropriate view types. The methodologist can then define further view types, such as the *component-class implementation view type* (cf. VT_3 in Fig. 6), which depicts which classes of the UML model implement which components of the architecture.

Although the relations between metamodels and view types could also be described with the same languages as CPRs, it is reasonable to use specialized view definition languages: First, such languages have to support the definition of editability, since not all represented properties may be modified in a view for conceptual or pragmatic reasons. Second, such languages have to support the projection of views from multiple models, whereas CPRs only relate two metamodels. Since the definition of view types is a research area of its own, we do not discuss it here in detail. We discuss existing view definition languages in Section 10.3, including our previous work on multi-model views with *ModelJoin* (Burger et al., 2014).

Since views are the only way of accessing information in a V-SUM, they must be able to abstract from redundancies and to reorganize information. This is why the view types in Vitruvius are *strongly decoupled* from the V-SUM metamodel: It is not necessary (although possible) that the elements in a view type are a subset of the elements in the V-SUM metamodel, such that $\mathcal{VT} \subseteq \bigcup_{M \in \langle M_{VSUMM} \rangle} M$. A view type can reproduce, rearrange, or aggregate information from these metamodels. Additionally, a view type may implement access control techniques. For example, the component developer in Fig. 7 may not be authorized to change information in the code or the UML model that reflects architectural information of the ADL, because this is a responsibility of the system architect. In such a case, a view type can restrict editability of certain information, e.g. of those classes that realize architectural components.

6.3.4. Process scenarios

The order in which the methodologist applies the process steps depend on the scenario. We can distinguish at least three scenarios, which can exemplarily be identified in our running example applied in Fig. 6:

Metamodel-driven: The process steps can be applied in the order in which they were presented. The methodologist starts with the metamodel selection and then defines the CPRs between them, as well as view types for them. The latter two steps can be executed in any order. This applies to the UML part of our running example: Since UML is a well-known standard, the methodologist first selects the metamodel, then the view types (class and component diagrams), since they are part of this standard, and then defines CPRs.

View-type-driven (existing view type): If a view type already exists, the methodologist can start with adding it and afterwards chooses or defines one or more appropriate metamodels representing the information necessary for this view type. Finally, the methodologist defines the CPRs to other metamodels. This applies to the Java part of the running example: the view type *textual syntax for Java source code* is known first, and from there, a suitable metamodel-based representation is chosen in terms of JaMoPP.

View-type-driven (custom view type): It is also possible to start with a notion of elements and relations that are not yet represented in an existing metamodel or view type. The methodologist may be faced with the requirement that elements in

the object-oriented design shall be traced to the architectural elements that they represent. From an informal notion of the rules relating architecture and object-oriented design, he or she may start defining a custom view type that is able to represent classes with annotations showing the architectural elements they implement, which is shown as the *component-class-implementation* view type in Fig. 6. From the concepts in this custom view type, the methodologist then chooses or defines the necessary metamodels for the component and class concepts. Finally, he or she formalizes the consistency rules that are implicitly represented in the custom view type.

6.4. Domains

The Vitruvius approach is supposed to support the development of software-intensive systems in different engineering disciplines by coping with the problem of information fragmentation across different tools and preserving consistency of that information. The engineering disciplines range from traditional software engineering to other fields like electrical and mechanical engineering in the context of cyber–physical systems. We have already applied the approach to the domain of ordinary software engineering (Kramer et al., 2015), but we also have some preliminary results for other domains such as automation systems (Ananieva et al., 2018a).

In general, the Vitruvius approach is limited only by what can be expressed with EMOF-based metamodels. Since EMOF is a generic metadata standard (International Organization for Standardization, 2014), it is not limited to a certain domain. We have already discussed in Section 6.3.1 that for several popular tools from different engineering disciplines, EMOF-based metamodels have been defined, which enable the usage artifacts generated with those tools. The complexity of defining CPRs depends on the quality of the domain metamodels, which is difficult to quantify (Hinkel et al., 2016). Finally, the view types may, in a first step, only represent the information already represented in the metamodel, thus no additional effort to define them is required as they are given by the used metamodels, to already benefit from consistency preservation capabilities. As soon as further view types are defined, developers can even profit from further role-specific views, whose complexity to create depends, like for CPRs, on the domain metamodels to project the information from.

Summarizing, as long as EMOF-based metamodels are given or can be defined for a domain, CPRs and view types for them can be defined. There are no specific requirements that prevent the adoption in a specific domain. The complexity of adoption depends on the quality of the given metamodels and the complexity of the consistency preservation rules to define.

7. Consistency preservation languages

In this section, we introduce the two consistency preservation languages of the Vitruvius approach, which form our contribution C3. The *Reactions language* (Klare, 2016; Kramer, 2017) provides imperative and unidirectional specifications of consistency preservation with declarative constructs for recurring actions, but still provides maximum expressiveness. Methodologists can use it to define CPRs according to Definition 7. The *Mappings language* (Werle, 2016; Kramer, 2017) provides highly declarative, bidirectional specifications of consistency rules, but with reduced expressiveness. A rule in this language conforms to a CRs according to Definition 3. Specifications in the Mappings language are transformed into imperative specifications of the Reactions language, providing a seamless integration of specifications in both languages. Thus, CPRs, as required for operating a V-SUM, are finally derived from specifications in both languages. Additionally, the interaction with the user in cases where consistency cannot be restored fully automated is a feature of the languages.

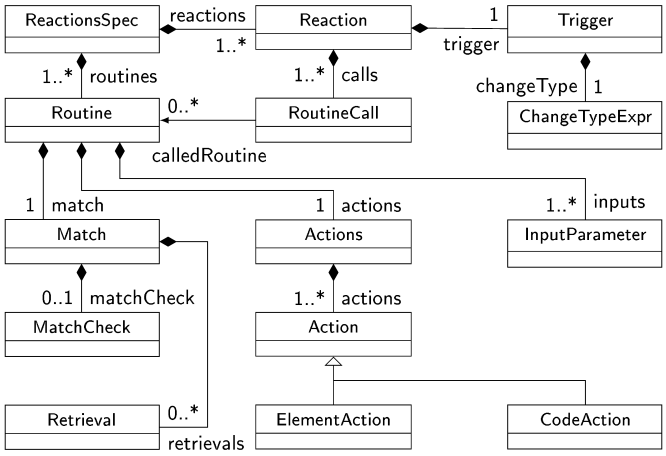


Fig. 8. Simplified class diagram with meta-classes for representing Reactions in terms of an AST.

We developed new languages rather than reusing or extending existing ones because of two reasons: First, we use an incremental and delta-based approach for consistency preservation, as introduced in Section 5.2, whereas most existing languages follow a state-based approach. This allows us to react to actual model changes rather than to potential deltas that were derived from a state difference. Second, the two languages can be used for specifications on two different abstraction levels. Methodologists should define Mappings whenever possible. If their expressiveness is not high enough to derive an appropriate Reaction in a specific situation, they can specify Reactions explicitly. This exceeds the capabilities of hybrid languages such as ATL, because it does not restrict the flexibility of imperative code segments to specific situations, such as the instantiation of a rule in ATL, but allows methodologists to react to arbitrary events with Turing-complete code, if necessary.

The languages rely on traces stored in a *correspondence model*: Each correspondence refers to two elements, and can be tagged with metadata to distinguish different correspondences between the same elements. The correspondence model is an additional artifact that stores explicit relations between elements, but provides no guarantee for consistency of the corresponding elements. While our implementation of the Mappings language automatically creates and uses appropriate correspondences, the methodologist has to manage correspondences manually when using the Reactions language, but is more flexible in specifying them.

The two languages were originally defined by Kramer (2017). In the following, we introduce the core concepts of the two languages to show their capabilities and expressiveness. For a more detailed explanation of the languages and a formalization of their semantics, we refer to Kramer (2017). The syntax definition can be found in Appendix A.

7.1. Reactions language

The *Reactions language* is used to specify unidirectional consistency preservation by updating instances of one metamodel after changes to an instance of another. A *Reaction* constitutes the realization of a CPR (Definition 7). The language provides declarative elements for typical consistency preservation actions, such as the management of correspondences. It also includes check expressions and imperative model manipulations, based on the expression language Xbase (Efftinge et al., 2012).

```
reaction aReaction {
   after // trigger definition
   check { /* further restrictions */ }
   call aSpecificRoutine(...)
}

routine aSpecificRoutine(...) {
   match { /* retrieve corresponding elements */ }
   action { /* perform actions and manage correspondences */ }
}
```

Listing 1: Reaction stub illustrating the main language constructs and steps of change-driven consistency preservation.

7.1.1. Language structure

The top-level structure of the Reactions language is presented in Fig. 8. The three main steps are:

1. *Triggering* Reactions according to the type and properties of a user change
2. *Matching* model elements that correspond to elements of the changed model
3. Performing *actions* on matched elements and managing correspondences

We explain these steps in detail in the following subsections. The top-level keywords of the language are shown in Listing 1. We separate the *match* and *action* steps from the *trigger* step using *Reaction routines* that can be reused by other Reaction routines and Reactions. A routine has a name and a list of parameters. It can be called in the same way a method is called in Java. A Reaction defines only a trigger and the Reaction routines it calls.

7.1.2. Triggers and calling routines

The first element of every Reaction is a trigger definition. It states in reaction to which changes the Reaction is going to be executed. A trigger definition has two parts in which restrictions can be defined, based on the change type and based on the change properties. The user can define the *types of changes* after which a Reaction is to be executed using a concrete textual syntax. This relieves methodologists from performing explicit type checks on the obtained change. We distinguish four main change types:

1. Replacements of a single attribute or reference value
2. List changes which affect a single list entry
3. Insertions and removals of root elements
4. Creations or deletions of model elements

With these change types, we are able to express all possible changes in EMOF-conforming models. Model elements can either be contained in a containment reference of another element, or they are considered *root elements* if they do not have a container. Insertions and removals of list entries or root elements may go along with the creation or deletion of that element. Therefore, the change types 2 and 3 can be combined with change type 4. Apart from that, we only support the specification of Reactions to atomic changes of types 1 to 4 and no further combinations of them. It is possible to restrict a Reaction not only based on the type of a change, but also based on change description properties. These properties yield the type of the modified element, the modified feature, and the type of the new or the old value, if the change type is 1 or 2. A trigger for the insertion of a component into the `components` reference of a `Repository` in the running example can be specified as shown in Line 2 of Listing 2. If it is necessary that the component was not contained in another

```
1  reaction {
2     after element adl::Component
3           inserted in adl::Repository[components]
4     call {
5        val component = newValue
6        createClass(component)
7     }
8  }
9
10 routine createClass(adl::Component component) {
11    match {
12       require absence of oo:Class
13          corresponding to component
14       val componentsPkg = retrieve oo::Package
15          corresponding to component.repository
16          tagged with "componentsPackage"
17    }
18    action {
19       val class = create oo::Class and initialize {
20          class.package = componentsPkg
21          class.name = component.name + "Impl"
22       }
23       add correspondence between component and class
24    }
25 }
```

Listing 2: Reaction to the creation of a component.

element, but was created right before its insertion, the trigger specification can be extended to `created and inserted`.

Further checks on the change can be performed in an optional *check expression*, which is a code block introduced with the `check` keyword, as shown in Listing 1, which returns `true` or `false`. In such an expression, all change properties are available. Depending on the change type, this includes the `affectedObject`, which was created, deleted or whose feature was modified, the `newValue` or `oldValue` that was inserted into or removed from a feature and the `index` if a list entry was changed. These properties are typed according to the specified change type and property restrictions. For example, the new value of a list entry insertion is typed with the element type of the list feature specified in the trigger. The usage of the change type information of the trigger relieves methodologists from explicitly performing type casts.

The second part of every Reaction definition is a code block with one or more Reaction routine calls, which are only executed if the trigger specification matches an occurred change. Within such a block, the actual change properties, which are the same as in a check expression, can be accessed and Reaction routines can be called. Additionally, arbitrary Xbase expressions that cause no side effects may be defined. This is rarely necessary but can be useful, for example, to define local variables used as arguments to Reaction routine calls. Our current prototype, however, does not yet automatically discover unwanted side effects.

7.1.3. Matching

In the first part of a Reaction routine definition, it is possible to specify which elements and conditions have to be matched before actions are executed. For this, retrievals of corresponding elements can be combined with match checks that may realize arbitrarily complex conditions. Retrievals contain a declarative specification of the elements that shall be matched and retrieved based on correspondences. The specification of a retrieval results in a lookup in the correspondence model, which contains the correspondences for element pairs that were created by the actions of previously executed Reactions. These retrievals can have two different types: *Presence retrievals* define which elements have to

be present. They have two subtypes for the retrieval of required and optional elements. A required presence retrieval is successful if exactly one corresponding element exists. Such a presence retrieval is shown for the `components` package of our running example in Line 14 of Listing 2. An optional presence retrieval queries a corresponding element, if existing, but is not required for a successful match. It can be used if elements do not necessarily exist, e.g., because a user was allowed to decide whether it shall exist or not. *Absence retrievals* define which elements have to be absent. They are successful if no corresponding element exists. Such an absence retrieval can be used, for example, to ensure that no elements are created twice. Line 12 of Listing 2 contains an absence retrieval that ensures that no class already corresponds to the created component. Actions are only executed if all except optional retrievals are successful.

All retrievals specify a *source element condition*. It yields an element for which correspondences are inspected. In the component repository retrieval in Line 15 of Listing 2, the source element is defined as `component.repository`. Furthermore, it is necessary to specify the type of the corresponding element as a meta-class, which is an `oo::Package` in our example. In addition, it is possible to restrict the target elements that are to be retrieved using a *retrieve properties check*, which has to be preceded by the keyword `with`. Such a check expression can inspect any of the properties of an element to be retrieved and it can be necessary, for example, if a corresponding element has to be identified in a group of several elements of the same type. Furthermore, a string tag can be assigned to correspondences to distinguish several correspondences of one element. A retrieval can define a *tag expression*, which specifies which string tag had to be used to register the correspondence in a previous Reaction routine. Line 16 of Listing 2 contains a tag expression that identifies the corresponding `components` package among other corresponding elements of the same type, e.g., the `interfaces` repository. In order to make the retrieved elements accessible in the actions, optional and required presence retrievals can be combined with a variable declaration and assignment. The retrieval of the `components` package in Line 14 of Listing 2 is assigned to the variable `componentsPkg`.

We provide declarative retrieval statements in order to relieve methodologists from considering many technical details that have to be dealt with if correspondences are inspected manually to obtain corresponding elements. The generated code performs all necessary operations, including type checks, type casts, variable declarations and assignments.

7.1.4. Actions

The second part of a Reaction routine definition lists all actions to perform for restoring consistency. They can have three different types, which are the creation, deletion or update of model elements, the (de-)registration of correspondences, and the call of other Reaction routines. The statements are executed in the specified order. Each statement has access to the elements retrieved in the match block or defined by previous action statements.

With the first type of actions, instances of meta-classes of the target metamodel can be created, deleted, or updated. An example for an instantiation of a class is shown in Line 19 of Listing 2. An element creation action has to provide the meta-class that is to be instantiated. In the example, this meta-class is `oo:Class`. It may be combined with a variable declaration and assignment as well as with optional initialization code in which values of attributes or references of the element can be set. In Listing 2, the created element is assigned to the variable `class`, and an initialization code block assigns the `package` and the `name` of the created class. With an element deletion action, an existing element of an instance of the target metamodel can be deleted by simply listing the variable name it was assigned to when retrieving the element in the match block. Furthermore, directly or indirectly contained elements are deleted recursively.

Finally, an element update action allows to modify attribute or reference values of an existing model element. To this end, the variable name for the existing element has to be provided together with a block of update code. The code blocks of both creation and update actions allow to group code that modifies an element, e.g., multiple attribute changes, and thus gives methodologists a possibility to structure their code. This can make it easier for them and for future analyses to identify how model elements are initialized or updated.

An action for adding or removing a correspondence is specified by providing the two model elements that shall newly or no longer correspond. Correspondences are added and removed independently of any further correspondences for the same elements. For cases in which several correspondences shall be registered for a single element, it is possible to specify a string tag to identify different correspondences during addition, retrieval, and removal, analogously to the specification in the retrieval. The example in Listing 2 shows the creation of a correspondence between the component and the newly created class in Line 23.

The last type of Reaction routine actions are blocks with calls of other routines, which are syntactically identical to the routine call block of Reactions (see Section 7.1.2). This allows modular composition and reuse of Reaction routines.

We already discussed that preserving consistency is a process that cannot always be automated. Considering the running example for components, as introduced in Example 1, a class (and a package) is supposed to be created for every component that is added to the software architecture. It may, however, not be desired that every class (in combination with its package) is represented as a component in the architecture. It should be up to the methodologist (or a dedicated architect) to decide whether an added class is supposed to represent a component. In this case, interactions with the user are necessary to determine the behavior. The Reactions language provides an internal API for such user interactions, which provides different types of inputs, such as confirmations, single or multiple choice questions, and textual input. This API is accessible within all constructs presented before, such as an initialization, execution or call block.

7.2. Mappings language

In this section, we present the Mappings language that can be used by methodologists to complement the Reactions language in symmetric cases. Symmetric cases are those in which for two metamodels the CPRs for both directions can be derived from a single declarative specification. This is not restricted to bijective cases, but requires that all defined conditions can be enforced in both directions. With these *Mappings*, it is possible to declare under which conditions instances of meta-classes of both metamodels should correspond to each other. It is, however, not necessary to specify after which changes these conditions have to be checked or how they have to be enforced. Instead, unidirectional Reactions that consider these details are automatically generated for both preservation directions from each Mapping. In consequence, a Mapping conforms to a CR according to Definition 3, from which Reactions, which realize CPRs according to Definition 7, are derived.

7.2.1. Language structure

The central concepts of the Mappings language are depicted in Fig. 9. The language provides two first-level constructs for *Mappings* and *Bootstrap Mappings* (see Listing 3). A Mapping defines consistency constraints between meta-classes of two metamodels, whereas a Bootstrap Mapping only defines the instantiation

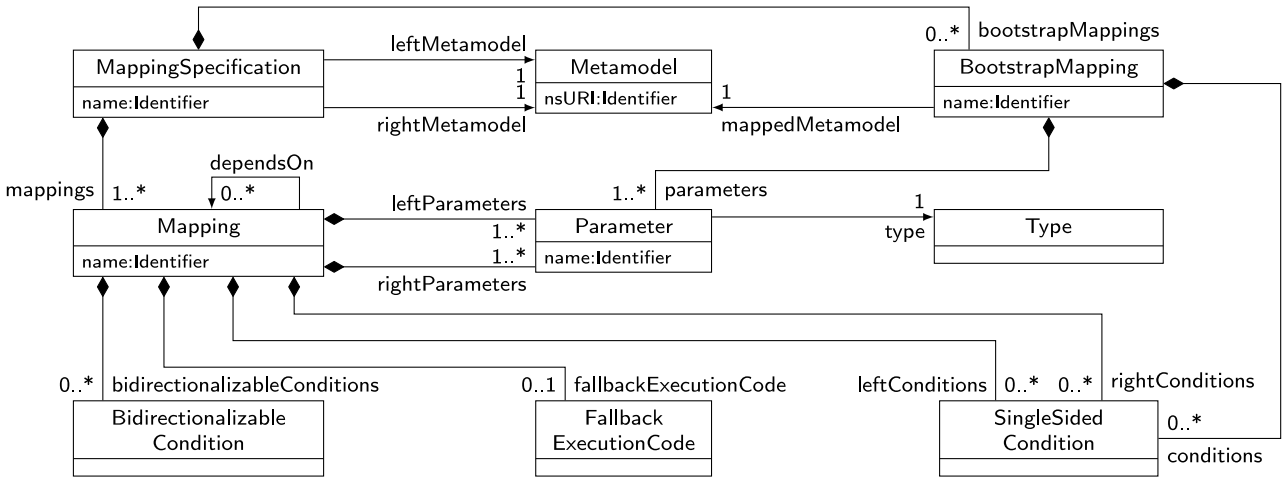


Fig. 9. Simplified class diagram with meta-classes for representing Mappings in terms of an AST.

```
mapping aMapping
  depends on ( /* another mapping name */ ) {
  map ( /* first meta-classes */ )
    with { /* single-sided conditions */ }
  and ( /* second meta-classes */ )
    with { /* single-sided conditions */ }
  such that { /* bidirectional enforcement specifications
      */ }
}

bootstrap mapping aBootstrapMapping {
   create ( /* meta-classes */ )
   with { /* bootstrap conditions */ }
}
```

Listing 3: Mapping stub illustrating the two first class concepts, Mappings for both sides and Bootstrap Mappings for a single side, with their main language constructs.

of meta-classes in one metamodel, which must always exist. Each Mapping relates two metamodels and therefore defines two sets of conditions. Second, it defines a set of conditions that have to hold between the elements in instances of both metamodels, thus representing the consistency constraints.

For the first set of conditions, a Mapping contains two parameter lists in which meta-classes of both metamodels are specified together with identifiers for their instances. For every parameter list, methodologists can specify *single-sided conditions* in a block introduced with `with`, which have to be fulfilled by the instances of the list whenever they are mapped to instances of the other parameter list. This block of conditions can be omitted if it is not necessary. These conditions refer, however, to the metamodel of the parameters in isolation and cannot make any statements about properties of instances of the other side. For the second set of conditions, which are statements that relate elements of both sides, the language optionally provides two kinds of *bidirectional enforcement specifications*, which can be defined in the `such that` block: First, *single blocks of bidirectionalizable conditions* are enforced in both directions, but only support certain operators. In this case, *bidirectionalizable* means that from a condition its preservation in both directions can be derived. Second, *pairs of forward and backward enforcement blocks with arbitrary code* can be defined, which are executed if all Mapping conditions from the left/right metamodel hold after a change in their respective instances. The result is always that the instances of the meta-classes of both parameter lists are mapped to each other.

A Mapping can depend on another Mapping, defined in a `depends on` statement, which makes the elements of that Mapping accessible, e.g., to place elements relative to elements in the Mapping it depends on. A Mapping can only be instantiated when the one it depends on was already instantiated. If a Mapping has no dependencies, this statement can be omitted.

Bootstrap Mappings are similar to ordinary Mappings, but contain only one list of parameters for one metamodel and only one set of single-sided conditions. Such a Bootstrap Mapping defines elements that always have to exist due to the defined consistency rules. Thus, the defined elements are created and the conditions are enforced before any change to the models is made. Therefore, the keyword `create` is used. This can, for example, be necessary to set up some primitive types, which are used throughout other ordinary Mappings and thus have to pre-exist before other Mappings are applied.

Whenever all single-sided conditions of a Mapping get fulfilled for a set of classes in one model, an appropriate set of elements is instantiated in the other model that fulfills both the single-sided conditions as well as the bidirectional enforcement specifications. Additionally, correspondences between the elements are created, which are used to update or remove the corresponding elements after modifications. We say that *a Mapping is instantiated.* Whenever model features are changed that are used in the bidirectional enforcement specifications, the other model is updated appropriately to fulfill that specification. Finally, whenever one of the single-sided conditions of an instantiated Mapping is not fulfilled anymore, the *Mapping is destroyed*. This means that if the single-sided conditions at one side of the Mapping are not fulfilled anymore, the elements that fulfill the parameters at the other side are removed from the model, as well as the correspondences between the elements. To achieve that behavior, Reactions for all relevant changes that affect either the single-sided conditions or the bidirectional enforcement specifications are generated, which appropriately instantiate or destroy the Mapping, or update the features of the bidirectional enforcement specifications.

7.2.2. Example mapping for component repositories

Listing 4 shows the repository consistency rule introduced in Section 2 expressed in the Mappings language. A Mapping between a component repository on the left side and two packages on the right side is defined. The single-sided conditions state that the root package needs no parent package, that all other packages have to be contained in the root package, and that they have to be appropriately named. These single-sided conditions

```
mapping Repository<->Packages {
  map (adl::Repository repository)
  and (oo::Package rootPkg, oo::Package pkg4interfaces,
    oo:Package pkg4components) with {
      null equals rootPkg.parent
      pkg4interfaces in rootPkg.subpackages
      pkg4components in rootPkg.subpackages
      "interfaces" equals pkg4interfaces.name
      "components" equals pkg4components.name
  }
  such that { rootPkg.name = repository.name }
}
```

Listing 4: Mapping between a component repository and its package structure representation in object-oriented design.

decide whether a repository has to be created after a change in the object-oriented design and enforced when a repository is created in the architectural model. Whenever a repository in the architectural model is created, the package structure in the object-oriented model is instantiated. Additionally, when the creation of the package structure is observed in the object-oriented model, a corresponding repository is created. To achieve that, for each change that can lead to the fulfillment of all single-sided conditions of a Mapping, and thus its instantiation, an appropriate Reaction is generated. If the repository in the architectural model is removed or one of the packages in the object-oriented model is removed or renamed, the Mapping is destroyed, thus the corresponding model elements are removed.

Finally, the Mapping contains a bidirectional enforcement specification in the `such that` block, which states that the name of the repository has to be equal to the name of the root package. This condition is ensured by generated Reactions that respond to changes of the package or repository name, respectively. For operators that are not easily bidirectionalizable, such as the aggregation of two values, we refer to our previous work (Kramer and Rakhman, 2016), which proposes an approach for defining inverters for operators in the Mappings language.

7.3. Combination of Mappings and Reactions

The example Mapping in Listing 4 already illustrates the two main advantages of the Mappings language compared to the Reactions language:

1. With Mappings, methodologists only specify once in a mostly *direction-agnostic* way which elements have to correspond, but the Mappings are automatically enforced in *both* directions.
2. Methodologists declare Mapping conditions that have to hold in a *change-agnostic* way, but if a change can lead to the fulfillment of these conditions or require that they are fulfilled, this is automatically checked or enforced.

Methodologists can specify Mappings that abstract away from direction- and change-specific details. To adapt the abstraction level for the consistency preservation directions, they can also consider the direction where this is necessary. This adaptation can be achieved either by specifying separate check-and-enforce code for single-sided conditions or by directly specifying enforcement code for both directions if the abstraction of bidirectionalizable conditions is not sufficient.

Listing 5 shows the Mapping to keeping components consistent with a package and a component-realization class. From this Mapping, at least six Reactions have to be derived: the insertion of a component into a repository (see Listing 2) as well as its removal, the insertion of a class into a package as well as its removal, and the renaming of a component and a class. Each of these Reactions has a Reaction routine that checks the preconditions, e.g., the placement of the class in an appropriate package, and performs modifications of the architecture and object-oriented design models, as well as the correspondence model. This gives an initial impression of the abstraction that can be achieved with Mappings compared to Reactions. We provide one of the Reactions that would need to be implemented using the Reactions language to realize the relations of the Mapping in Listing 4 in Appendix B.

```
mapping Component<->PackageAndClass
 depends on (Repository<->Packages repoPkgs) {
  map (adl::Component component) with {
      component in repoPkgs.repository.components
  }
  and (oo::Package componentPkg, oo::Class class) with {
      componentPkg in repoPkgs.pkg4components.
          subpackages
      class in componentPkg.classifiers
      class.name = componentPkg.name + "Impl"
  }
  such that {
    component.name = componentPkg.name
    component.name + "Impl" = class.name
  }
}
```

Listing 5: Mapping between a component and its representation of a package with a component-realization class in object-oriented design.

This example has emphasized the advantages of Mappings over Reactions. Mappings are however restricted to what can be expressed by means of a bidirectional, declarative specification. For more sophisticated consistency rules, such as asymmetric or complex relations, the expressiveness of Reactions is beneficial. Such scenarios could, e.g., be relating each component to a class but not vice versa, or performing complex processes for checking and enforcing consistency rules, such as running a simulation to check that a performance requirement is fulfilled by the implementation. For that reason, we propose to provide these two languages and always use the one that provides a sufficient level of abstraction for the current consistency rule or part of it to realize. Since we generate Reactions from Mappings, the Reactions generated for a Mapping can be easily extended or adapted. In consequence, the combination of Mappings and Reactions allows to always use an appropriate level of abstraction, improving expressiveness and conciseness at the same time.

8. Integration of existing models

In the previous sections, we have discussed how Vitruvius can be used to keep models consistent during the development of a system. Due to the inductive approach to consistency preservation, this only covers the "greenfield" perspective, where the Vitruvius approach is used from the beginning of the development. In existing systems, developers may already have invested a great amount of time to create models without using the Vitruvius approach. Such models, however, may not adhere to the consistency rules defined in a V-SUM metamodel in general and, additionally, miss the required correspondences. To enable developers of such systems to use Vitruvius with their already existing models, we present two approaches to integrate existing models into the Vitruvius process, constituting our contribution C4.

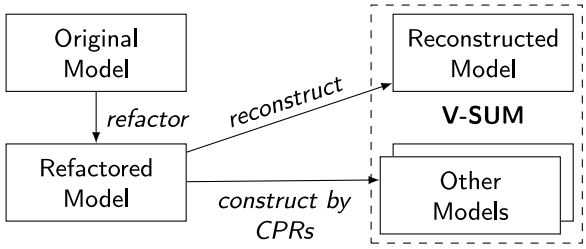


Fig. 10. Reconstructive Integration Strategy (RIS).

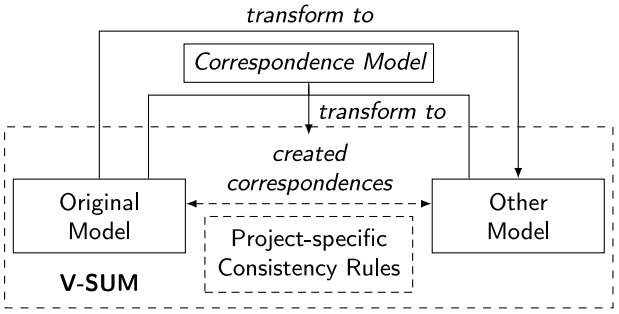


Fig. 11. Linking Integration Strategy (LIS).

The *Reconstructive Integration Strategy* (RIS) simulates the construction of an existing model by generating a change sequence for its construction and applying it to a V-SUM, such that the CPRs for creating other models are applied. The *Linking Integration Strategy* (LIS) links an existing model with another model that is created using a generation or transformation, such as a reverse engineering tool, by generating appropriate correspondences. Both approaches are able to integrate one existing model into the Vitruvius approach. Integrating more than one model is subject of future work.

8.1. Reconstructive Integration Strategy

The Reconstructive Integration Strategy (RIS) integrates an existing model by simulating its creation (see Fig. 10). Therefore, we need an algorithm that describes how instances of metamodels can be built by atomic changes (cf. Definition 6). During this reconstruction process, Vitruvius monitors the changes that are necessary to create the model. The monitored changes are used to create the other models within the V-SUM using the standard process applying changes to its interface (App).

For EMOF-based metamodels, the atomic creation order can be determined by following the containment hierarchy of objects. This means we can start at the root object and follow the containment hierarchy in order to ensure that all objects exist. During the creation, we also create changes for the attributes of an object, as well as its non-containment references.

As an optional initial step before the actual reconstruction, we propose to have a *refactoring* step for conflict resolution. In this step, additional conditions on the models that are introduced by the CPRs have to be fulfilled. For example, the consistency rules between code and ADL, introduced in Section 2, require representations of architectural interfaces in code to be placed in a specific subpackage of the component repository package. Depending on the implemented CPRs, it is possible that those interfaces are otherwise not correctly mapped to the architectural model. Thus, the code should be refactored in advance so that it adheres to this constraint. We will consider a more convincing example in the evaluation in Section 9.4.

8.2. Linking Integration Strategy

A Linking Integration Strategy (LIS) (see Fig. 11) can be used to integrate a source model and a target model that is created by an existing transformation or generation approach. This can, for example, be a reverse engineering tool or a code generator. This transformation or generation needs to create a correspondence model that describes which source element led to the creation of which target elements. If this information is available, we are able to create a model transformation that uses the source model, the target model, and the correspondence model as input, and generates an instance of the Vitruvius correspondence model containing correspondences between source and target model elements. These correspondences can be used by Vitruvius as *normal* correspondences during the execution of CPRs.

One disadvantage of the described approach is that the model generation or transformation tools need to create the same target models as the CPRs, as otherwise the CPRs cannot be properly applied. In general, this precondition does not hold, as an existing transformation approach may not follow the same rules as the defined CPRs for keeping the models consistent. As a simple example, a reverse engineering tool for extracting UML class models from code may represent list-type parameters of Java methods as parameters with the list-type in the UML model, whereas the CPRs keep the list-type Java parameters consistent with parameters in the UML model that have the type of the list elements with infinite multiplicity (*). To overcome this drawback, we need to create *integration-specific* correspondences for elements that cannot be kept consistent with the defined CPRs. These correspondences need to be created by the model transformation that generates the Vitruvius correspondence model, so this transformation needs to be specific for the CPRs. After the creation of *integration-specific* correspondences, during the actual consistency preservation, they need to be treated differently.

To keep those elements with *integration-specific* correspondences consistent, we propose the following solutions:

- A simple solution is to warn users that the performed change cannot be kept consistent automatically, i.e., users need to keep them consistent manually.
- At least parts of the elements can be kept consistent automatically using the standard CPRs, e.g., an element name.
- Special project-specific CPRs for elements with *integration-specific* correspondences can be implemented.

The usage of special CPRs relying on *integration-specific* correspondences is only necessary for the integrated elements. For all elements that are added to the model after the integration, the existing CPRs can be applied.

Example 4. Assume the consistency rule of the running example explained in Example 1, which maps each component to a combination of a package and a class. While this is a rule that could be enforced when starting with a new project, existing projects will usually not follow exactly that pattern to represent components in the object-oriented design. For example, in existing code a component may only be represented by a class that is placed anywhere instead of also being placed in a specific package. This can be represented by integration-specific correspondences and appropriate CPRs that only propagate information of a component to a class and not to a package as well (Langhammer, 2017). Evolving the project, all further added components can then be represented in the object-oriented design by applying the standard consistency (preservation) rules.

As we need to have existing transformation respectively generation approaches between the source model and the target model, a LIS highly depends on those. Hence, a LIS needs to be customized for a specific set of source models and target models. In the following example section, we show how a LIS can be used to integrate source code and an architectural model based on reverse engineering approaches.

8.3. Integration example

In this section, we first explain how a RIS can be used to integrate an architectural model, and second, we explain how a LIS can be used to integrate source code.

8.3.1. Integration of an existing architectural model

To integrate the architectural model from our running example, we use a RIS and the standard algorithm, which traverses the model according to its hierarchy, i.e., every element is visited along the containment hierarchy. During the visit, a creation change for the element is created. In the architectural model from the running example introduced in Section 2, first the repository *MediaStore* itself is visited. The next elements that are visited are the top level elements, which are the interfaces and components. For the running example, the interfaces *IWebGUI* and *IMediaStore* and the components *WebGUI* and *MediaStore* are visited. Afterwards, the signatures of the interfaces in the components are traversed. Finally, the provides and requires relations of the components are visited.

After visiting all elements, the generated changes are applied to the change application function (App) of a newly instantiated V-SUM, which executes the appropriate CPRs, in our case from the architectural model to source code.

8.3.2. Integration of an existing source code base

To show how we can integrate an existing source code base, we explain a LIS that is able to integrate source code with an extracted architectural model. To integrate an existing source code base, we need to have a tool that creates an architectural model from the source code and correspondences as trace information between the created architectural elements and the source code elements. To create the architectural model from source code, we can use the reverse engineering tools *Software Model Extractor (SoMoX)* (Becker et al., 2010) or *Extract* (Langhammer et al., 2016). Both approaches create a PCM model from existing source code as well as a correspondence model.

After running one of the reverse engineering tools and gathering the models, we can integrate them into a V-SUM. For that integration, we create a model transformation that uses the three mentioned artifacts, the Java code as the source model, the architectural model as the target model, and the correspondence model created by the reverse engineering tool, as input to create a Vitruvius correspondence model. If the source code is compliant with the consistency rules from Section 2 between architectural model and code, and the reverse engineering tool created the matching architectural elements, the integration is finished.

Existing source code bases, however, do often not follow strict consistency rules, which are required by Vitruvius. To support such code as well, we need to create *integration-specific* correspondences for such source code bases. In particular, we need to create integration-specific correspondences for the source code elements and the architectural model elements that are not compliant to the already defined CPRs. Therefore, the above-mentioned transformation can be extended in a way that it creates those specific correspondences for non-compliant elements. These integration-specific correspondences are handled differently than standard correspondences by Vitruvius. Instead of using the standard predefined CPRs, a new specific set of CPRs needs to be defined and used. This specific set of CPRs needs to be defined based on the consistency rules, on which the existing code base relies. It is, however, possible to reuse some of the already existing CPRs. For instance, CPRs handling the renaming of elements, as well as those for changing the signatures of methods in architectural interfaces can be reused, as these changes need to be reflected in the corresponding source code method as well. Furthermore, it is possible to not define specific CPRs for a change. For those changes, the architectural model elements and the source code elements cannot be kept consistent automatically, but need to be kept consistent manually. However, Vitruvius is able to (a) notify users that they need to keep a change consistent manually, and (b) point to the changed model elements that need to be kept consistent. The complete process for integrating existing source code with a reverse-engineered architectural model is sketched in Fig. 12.

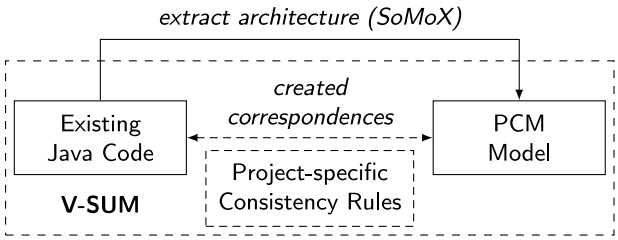


Fig. 12. Integration of existing code with Software Model Extractor (SoMoX).

Hence, to use a LIS to integrate the source code of our running example, we need to perform the following steps:

1. Reverse-engineer the source code using SoMoX.
2. Run the model transformation from source code, architectural model, and the correspondence model to generate the Vitruvius correspondence model.
3. Create specific CPRs for the source code elements and architectural elements that are not matching the defined standard consistency rules.

With the above-created artifacts, we can use the extracted architectural model and the source code within Vitruvius.

9. Evaluation

We have developed a prototypical implementation of the Vitruvius approach in the Vitruvius framework (Vitruvius GitHub, 2020c). Based on this implementation, we have performed different kinds of evaluation with a focus on analyzing the applicability of our approach, forming our contribution C5. We have investigated the applicability of our consistency preservation languages by using them in different case studies and evaluated their benefit. We have validated the capability of the Reconstructive Integration Strategy regarding the reuse of existing models in open source projects. Finally, we have conducted an overall evaluation of the approach, in which we first applied our Linking Integration Strategy to an old revision of an open source code repository, and then applied the changes in the subsequent revisions using the CPRs from our case study.

9.1. Prototypical implementation

Our implementation is based on the *Eclipse Modeling Framework (EMF)* (Steinberg et al., 2008). We support all metamodels that are based on the EMF metametamodel *Ecore*, which implements the EMOF standard (International Organization for Standardization, 2014).

9.1.1. Changes

Changes are represented using a generic metamodel (Kramer, 2017, Sec. 5.4.1). It defines meta-classes for all kinds of changes that are possible in Ecore-based models and thus conforms to Definition 6. These change types are:

1. Model element creation and deletion
2. Attribute value insertion, removal and replacement
3. Reference insertion, removal and replacement
4. Compositions of 1–3

We call the change types 1–3 *atomic* as they only concern a single value of one model element, as defined in Definition 6. All changes, even complex change sequences performed by the user within an editor, can be represented as a sequence of those atomic changes. This means that the set of supported edit operations is not restricted. The specified change types can be mapped to those presented by Hettel et al. (2008) for a generic metamodel definition and to those defined by Koegel et al. (2010c) for versioning Ecore-based models. More details about the change metamodel can be found in Kramer (2017).

9.1.2. Change monitoring

To track those changes, the model of interest has to be monitored. EMF provides an integrated notification mechanism for this. The monitored notifications are converted into representations of the specified changes. These changes are then passed to the implementation of a V-SUM, which executes the appropriate consistency preservation specifications according to Fig. 7.

If a model is not modified in an editor that uses an Ecore-based representation of the model, the EMF notification mechanism cannot be used. This is especially the case for models that are persisted as text and directly modified as such, for example, source code defined in a programming language. To be able to monitor changes to source code, we presented a monitor for the Eclipse Java code editor (Kramer et al., 2015). The monitor detects changes based on the AST changes in the Eclipse IDE. It converts detected changes into instances of the Vitruvius change metamodel and reports them to the Vitruvius framework. Furthermore, the monitor tracks changes that are performed by refactorings, such as renaming elements, within the Eclipse IDE.

9.1.3. Consistency preservation languages

We have defined the consistency preservation languages, the Reactions and the Mappings language, with the Xtext language engineering platform and the Xbase expression language (Efftinge and Völter, 2006). We decided to use Xtext, because it is the most mature language engineering platform available for EMF. Furthermore, we use Xbase, because it is a Java-like expression language defined with Xtext, which can be used within any Xtext-based language. Furthermore, Java developers can easily use it since it compiles to Java, and thus can be integrated into a Java code generator for any Xtext-based language. Both the Reactions and the Mappings language provide a textual editor and a code generator that produces Java code compliant with the CPRs expected by the Vitruvius framework. More information about the implementations of both languages is given in the corresponding GitHub wiki (Vitruvius GitHub, 2020a).

9.2. Case study domains

We have applied our approach in case studies for two different domains: component-based software engineering and embedded automotive software architectures. The case studies comprise six metamodels in total.

For the *component-based software engineering (CBSE)* domain, we have used the following metamodels:

Table 2
Elements (containments as a subset of references) in the case study metamodels. The values for SysML represent the profile, omitting the underlying UML.

Metamodels	Classes	Attributes	References	Containments
PCM	152	27	292	93
UML	256	286	510	194
Java	237	15	123	98
ASEM	20	6	10	6
SysML	56	17	80	0

1. **PCM** to describe an annotated component-based architecture of a software system allowing the prediction of quality attributes of the system
2. **UML component models** to represent the component-based architecture of a software system with the purpose to support a high-level system design
3. **UML class models** to represent the fine-grained system design on a class level
4. **Java code** for the implementation of the system

Although UML component and class models can be integrated into a single model with appropriate relationships, we decided to separate them into different models to foster the independent usage by different roles. To treat Java code as an Ecore-based model, we have used JaMoPP (Heidenreich et al., 2010). We use the consistency rules between architecture and code developed in the dissertation of Langhammer (2017). They constitute a central contribution of that thesis and are the result of a comparison of different possibilities to relate architecture and code. Components can either be represented as packages and classes in Java (Kramer et al., 2015), like in the running example, as Enterprise Java Beans (EJBs), or using a dependency injection framework. A detailed list of the considered consistency rules between PCM and Java classes can be found in Klare (2016, p. 114).

In the domain of *embedded automotive software (EAS)*, developers model the internal behavior of components of an electronic control unit (ECU) and generate code from these models, for example, using ASCET (ETAS Group, 2020). In our case study, we have used the following metamodels:

1. **Automotive Software Engineering Model (ASEM)** (Software Design and Quality, 2020), a structurally equivalent subset of the ASCET metamodel for modeling components of ECU software
2. **Systems Modeling Language (SysML)** (Object Management Group, 2019), block diagrams that allow the domain-independent modeling of the structure of and relations between such building blocks

Initial consistency requirements for these metamodels have been described in Mazkatli (2016) and Mazkatli et al. (2017) and were refined in cooperation with an industrial partner.

To give an impression of the sizes of the metamodels for the case study domains, we list the numbers of classes, attributes, references and containments in Table 2. The values for ASEM are comparatively small, because it only represents a relevant extract of the metamodel of the commercial ASCET tool. The ASCET metamodel consists of several hundred classes and features, comparable to metamodels such as UML. For SysML, we only listed the values for the UML profile, excluding those for the UML metamodel itself on which the profile is applied. The consistency rules that we defined for these metamodels only cover relevant subsets of all of them.

```
1  reaction {
2     after any change
3     call simulateTuringMachine(change)
4  }
5
6  routine simulateTuringMachine(EChange change) {
7     action {
8        execute {
9           // simulation of arbitrary Turing machine
10       }
11    }
12 }
```

Listing 6: Exemplary Reaction to simulate a Turing machine.

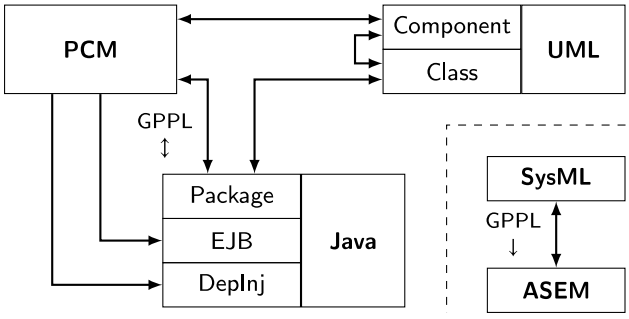


Fig. 13. Domains and realized Reactions in the case studies. The GPPL marks show which transformations were additionally implemented in a GPPL.

9.3. Consistency preservation

We have extensively evaluated the applicability of the Reactions language to preserve consistency in several domains. A detailed evaluation of the Mappings language is presented in Kramer (2017). Since the Mappings only provide an additional abstraction over Reactions, which already provide the necessary expressiveness required for consistency preservation, we focus the presented evaluation on the Reactions language. We provide different types of validation, inspired by Böhme and Reussner (2008, p. 15) for the evaluation of performance models. We have analyzed *completeness* and *correctness* of the Reactions language independent of a concrete usage scenario. Furthermore, we have evaluated its practical *applicability* and discuss potential *benefits* based on two case studies comprising several metamodels and consistency rules. These case studies, the Reactions compiler and the complete Vitruvius framework for consistency preservation are published as open source (Vitruvius GitHub, 2020b).

9.3.1. Completeness and correctness

We have evaluated completeness of the Reactions language in two ways: First, we have shown that the language is *EMOF complete* in the sense that different Reactions can be triggered for all change types that can be differentiated based on metamodels that are expressed using EMOF. Second, we provide a discussion that the Reactions language is Turing complete, because arbitrary Java code can be executed in response to arbitrary changes as a result of the embedding of the Xbase language into Reactions and Reaction routines.

EMOF completeness is given because all changes in EMOF-based models can be represented as a single atomic change of a model element respectively of a model element's property or as a combination of such atomic change representations. This is possible because all characteristics of EMOF-based models are realized in terms of objects and object values for properties that are defined and typed in an EMOF-based metamodel. Thus, the possible change types are induced by the EMOF metametamodel and thus equal for all metamodels. In consequence, everything that can be changed in EMOF-based models can be described with a trigger using the Reactions language.

Computational completeness is given because the language provides a fallback: A developer may express all update behavior in a single block with arbitrary Java code. To execute this code after arbitrary changes, a single Reaction and Reaction routine can be used (see Listing 6). A Reaction to the simple change type `any change`, which reacts to arbitrary changes, can be specified (Line 2), followed by a call to a Reaction routine (Line 3). This routine (Lines 6–12) only calls the arbitrary Java code (Line 9). As the Java language is Turing complete, this reduction shows that the Reactions language is also Turing complete. Such a minimalistic use of the Reactions language as an execution environment for Java code is, however, not necessary. The reason is that the first two of the three main steps of Reactions (see Section 7.1.1) can always be expressed with appropriate language constructs. This means that it is always possible to use constructs of the Reactions language to express after which changes and on which corresponding elements these actions shall be performed. The fallback to Java code is only necessary to express what action shall be performed (Kramer, 2017, pp. 330–331).

To show the correctness of the Reactions language, i.e. the correct behavior of each statement in response to all kinds of changes by the language, we have developed an artificial metamodel that contains elements on which every change type that is possible in Ecore can be performed, and implemented CPRs in terms of 20 CPRs that transfer all modifications in one model to a second model instance. Our considered modification types also conform to the change types identified by Yohannis et al. (2018), which they claim to be complete for Ecore-based models. We defined 22 test cases (Vitruvius GitHub, 2020c) that perform modifications for all modification types that are possible in Ecore-based models in one model instance and checked if the second instance is correctly kept consistent. Their successful execution indicates correctness of the language, since all other types of transformations are compositions of the modifications we made in this evaluation. For more information on this completeness and correctness evaluation, we refer to Klare (2016, pp. 109).

9.3.2. Applicability

To show practical applicability, we have realized Reactions specifications for the case studies, as depicted in Fig. 13. In the CBSE case study, we have implemented the update of Java code after modifications in a PCM model for the consistency rules to Java packages and classes, for the consistency rules to EJBs, and for the consistency rules to the dependency injection framework Guice (Google, 2020). We have also implemented Reactions in the other direction for updating PCM models after changes in a Java application. Additionally, we have implemented Reactions that keep PCM and UML component models, UML component and class models, as well as UML class models and Java code consistent all in both directions. These Reactions can be found in the associated GitHub project (Vitruvius GitHub, 2020d). In the EAS case study, we defined Reactions that preserve consistency of ASEM models after changes in SysML models and vice versa, which is also available on GitHub (Vitruvius GitHub, 2020e).

In total, we have implemented and evaluated CPRs for twelve unidirectional consistency relations. The size of the implemented Reactions is shown in Table 3. We have counted the numbers of Reactions, which is equal to the number of different atomic change events we are able to react to. In all Reactions and helper classes, we have counted the Source Lines of Code (SLoC), which

Table 3
Number and SLoC of Reactions SLoC in each transformation of the case study.

Transformation	#Reactions	SLoC	SLoC Reaction
Java → UML Class	47	756	16
UML Class → Java	51	876	17
UML Class → Component	23	546	24
UML Component → Class	16	368	23
UML Component → PCM	30	690	23
PCM → UML Component	31	653	21
Java Package → PCM	16	683	43
PCM → Java Package	45	1436	32
PCM → Java EJB	39	905	23
PCM → Java DepInj	47	1097	23
SysML → ASEM	17	926	54
ASEM → SysML	23	558	24
Overall	385	9494	25

we defined for the Reactions language as those lines that are neither empty nor only consisting of comments. In sum, we have implemented 9494 SLoC in 385 Reactions and in helper methods to realize the CPRs. As also shown in Table 3, the SLoC count per single Reaction is on average 25 and scarcely differs between the different case studies. The outliers in the SysML to ASEM and in the Java to PCM implementation are caused by comparably large helper classes. This indicates that the reaction to a single change can on average be defined with a small number of statements, promising high comprehensibility of single Reactions.

To evaluate correctness of the implemented CPRs, we employed tests, at least one for each implemented Reaction. Each test creates a model that is as minimal as possible regarding the needed elements to perform a change that triggers the Reaction. After performing this change, the test checks if the corresponding model is in the expected state according to the consistency rules using assertions. We have ensured with those tests that all Reactions update the corresponding models and elements as expected. We were able to successfully express all change property restrictions, retrievals of corresponding elements, model element creations and deletions, and correspondence updates using the language constructs. This indicates that the provided constructs are well suited and sufficient to realize consistency preservation.

9.3.3. Benefits

To decide whether one programming language should be used instead of another for solving a certain problem, potential benefits of the language should be analyzed. These benefits could be software development that is faster, more concise, or less error-prone. Literature on evaluating metrics for general code quality, for example by correlating them with the number of detected faults, have, however, shown that such properties are difficult to measure (Gyimóthy et al., 2005; Yu et al., 2002). These studies did not report significant correlations to code quality for most metrics, but instead identified a high correlation for the two metrics measuring the coupling between object classes (CBO) and lines of code (LOC) (Gyimóthy et al., 2005, p. 907).

To evaluate an indicator for the potential benefit of the Reactions language, we have implemented functionally equivalent consistency specifications using a general-purpose programming language (GPPL). We used PCM models and Java code, as well as SysML and ASEM models. We did not use an existing model transformation language as the baseline for comparison, as none of the existing languages supports incremental and edit-based consistency preservation specification (see also Section 5.1). Our languages thus especially support proper renaming and moving of elements. Operating on changes allows to track that an element that was renamed or moved is still the same. This is,

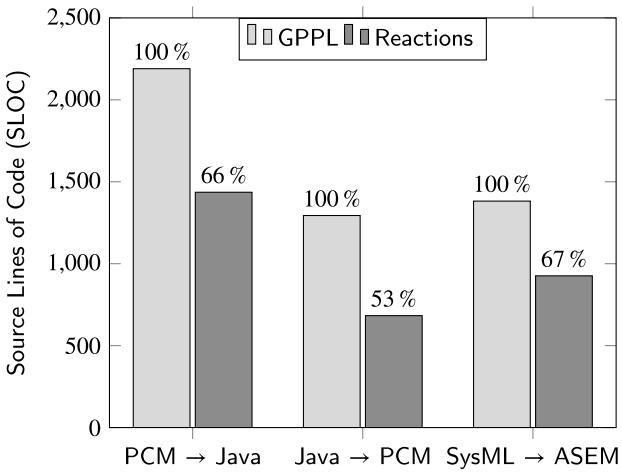


Fig. 14. SLoC for case studies with a GPPL and the Reactions language.

for example, important for the project-specific CPRs explained in Example 4, which relate a component to a class that may be placed in any package. If a class is moved from one package to another, handling the according change allows to identify that the class still belongs to a component, whereas languages that are not edit-based consider this as the removal and addition of a class, resulting in the corresponding component to be removed and newly created. In consequence, all additional information of the component, such as an abstraction of the functionality as represented in PCM, is removed as well. Therefore, a GPPL-based implementation represents the most appropriate evaluation baseline in our opinion. Additionally, we have reduced potential bias, since methodologists will most probably have less experience with a reference transformation language than with a GPPL such as Java. We used Java for the PCM to Java rules and the Java dialect Xtend for SysML to ASEM rules. In both cases, we have compared these GPPL implementations with Reactions. As the Reactions language does not provide classes to measure the CBO metric, we have counted and compared the SLoC of both implementations.

The number of SLoC may depend on the individual programming style. Therefore, we have used the Eclipse formatter with default settings without line wrapping to format Java and Xtend code. Furthermore, we automatically added line breaks to all Reactions to split chained constructs, i.e., before corresponding elements or tags for a retrieval were listed, before properties of changes or corresponding elements were checked, and before element initialization code.

We have counted the SLoC of all classes used for the consistency preservation implementation, including helper classes. In order not to favor the Reactions language, we excluded import statements, since the GPPL code is spread across more classes and therefore contains more of these statements. The SLoC of all implementations are shown in Fig. 14. It is evident that in both case studies, the implementation with Reactions is more concise than the counterparts implemented with a GPPL. All Reactions implementations required at least 33% less SLoC. At least two reasons prohibit a more drastic reduction in SLoC: First, the GPPL implementation was based on an auxiliary framework that provides an internal API that only requires the implementation of an interface for specifying the reaction to a specific type of change. The framework processes the individual changes and automatically dispatches them to the appropriate methods of the implemented interface. In consequence, that auxiliary framework relieves the developer from implementing repetitive tasks. The complete logic that was encapsulated in the interface and the

dispatcher was not counted in the comparison and also provides an abstraction regarding an independent implementation in a GPPL. Second, the Reactions language has the purpose to provide maximum expressiveness while also providing an appropriate level of abstraction for the use case. In consequence, its level of abstraction is not as high as the one provided by the Mappings language, which would yield a more drastic reduction in SLoC. Instead, it especially guides the methodologists in defining consistency preservation by providing constructs for recurring operations and by ensuring transactionality, i.e., that a Reaction is not executed partially. Although the amount of reduced code may depend on the case studies and methodologists, we expect that further studies will also yield code size reductions.

The comparison of GPPL and Reactions code in two case studies indicates that consistency preservation can be expressed more concisely using the Reactions language. This especially results from the language structure and the specialized language constructs it provides, which reduces the necessity to deal with technical and recurring problems.

9.4. Reconstructive integration

In this section, we present an evaluation of the Reconstructive Integration Strategy (RIS), which we described in Section 8.1. It simulates the creation of existing models. During the creation, Vitruvius monitors the changes and reacts to them, such that the corresponding models are created according to the used CPRs. We have evaluated the correct functionality and the practical applicability of the RIS on several case study projects. Its implementation is available as open source (Vitruvius GitHub, 2020f).

9.4.1. Case study projects

We have applied the RIS with two of our consistency preservation specifications between PCM and code to six existing PCM architectural models, varying from simple examples to industrial case study models. Table 4 lists the numbers of architectural elements in the projects. Those projects are:

1. **MediaStore**, a case study intended to show the applicability of the Palladio approach (Koziolek et al., 2007), which was recently extended by an EJB-based implementation (Strittmatter and Kechaou, 2016). We introduced a simplified extract in Section 2.
2. **CoCoME** (Common Component Modeling Example), a case study system to compare approaches for component-based software systems modeling (Herold et al., 2008). The first PCM version (Krogmann and Reussner, 2008) has recently been refined (Heinrich et al., 2016).
3. **Open Reference Case**, a service-oriented variant of the CoCoME system. As stated in Heinrich et al. (2016), an additional web service layer has been introduced to the original CoCoME architectural model.
4. **DPS** (Dynamic Positioning System), a model that can be used to navigate and find the position of a deepwater oil platform (see Duarte et al., 2010; Gouvêa et al., 2011, 2012). In Gouvêa et al. (2011, 2012) a PCM instance of a DPS has been introduced.
5. **Industrial Control System**, an industrial case study for the PCM (Koziolek et al., 2010; Brosch et al., 2012). The system is an industrial size process control system from ABB.
6. **BRS** (Business Reporting System), has been introduced by Koziolek (2011) and is loosely based on a real system introduced by Wu and Woodside (2004).

Table 4
Numbers of elements in existing PCM case study systems.

Project	PCM Elements
MediaStore	125
CoCoME	174
Open Reference Case	244
DPS	46
Industrial Control System	131
BRS	154
Total	874

9.4.2. Results of the RIS case study

As the RIS only simulates the construction of a model, the functionality of the approach can only be restricted by an erroneous implementation. To evaluate the correct functionality, we integrated the introduced existing architectural models via reconstruction, creating the corresponding source code model using two consistency preservation specifications from Section 9.3.2, the CPRs between architectural models and Java packages and classes, as well as those between architectural models and EJBs. Additionally, we evaluated the practical applicability. Since the approach is fully automated, the applicability can only be restricted by its application preconditions. The only precondition for applying the approach is that additional constraints that are introduced by the CPRs are fulfilled by the model to be integrated.

To evaluate functionality, we have validated the Java code created by consistency preservation during reconstruction and additionally compared the expected numbers of classes, interfaces, methods and fields with the actual ones. As the result, all 874 architectural elements were correctly mapped with both consistency preservation specifications, except for *OperationProvidedRoles* using the consistency rules to EJBs. Only 56 out of 73 *OperationProvidedRoles* were correctly mapped in that case. The purpose of *OperationProvidedRoles* is to represent interfaces provided by components. The CPRs for EJBs introduce the constraint that each interface is only allowed to be provided once in a component repository, which is not the case for 17 of the *OperationProvidedRoles*.

To summarize, we were able to show that the RIS operates correctly at least using the two selected consistency preservation specifications applied to six existing projects, which we assume to be a good indicator for the general functionality of the approach. Additionally, the evaluation revealed that the applicability of the approach depends on the strictness of additional constraints introduced by the CPRs. Nevertheless, to make a reliable statement on the practicability of the approach, it has to be applied to further case studies with different CPRs.

9.5. Overall evaluation

In this section, we present an end-to-end evaluation of the Vitruvius approach by showing how source code and architecture can be kept consistent during software development. In particular, we show that Vitruvius is able to

1. integrate existing open source projects using the LIS based on reverse engineering technologies, and
2. to monitor changes to the source code and correctly keep it consistent with the created architectural model.

We show these two points at existing open source projects. We first integrate an old version of each project and then replay changes from the integrated version to a later version based on the commits in the version control system. This simulates a realistic workflow on non-artificial projects. The implementation of the LIS is available as open source (Vitruvius GitHub, 2020f).

Table 5
Integrated versions of the case study projects and result of the reverse engineering process: KSLoC are thousands of SLoC in the project. *Comp* shows the number of extracted components, *Interf* shows the number of extracted interfaces.

Project	Integrated version	Code		Architecture	
		KSLoC	Java files	Comp	Interf
Gora	0.6.0	5.7	76	16	16
Any23	0.90	12.6	190	16	16
Velocity	1.60	26	229	18	18
Xerces	2.10	112	705	20	20

For the evaluation, we needed to extract and replay fine-grained changes between different versions in a Version Control System (VCS). A tool that provides such functionality for Git repositories has been presented by Petersen (2016). The tool performs the following steps in order to replay code changes based on different versions within a Git repository:

1. Extract intermediate versions between the chosen versions.
2. Calculate fine-grained AST changes between the intermediate versions.
3. Replay the fine-grained changes in the IDE.

9.5.1. Case study projects

We used four projects for the evaluation, which were chosen based on the following two requirements: The projects are implemented in Java, as we defined consistency preservation for Java in our case study, and the projects are developed with Git, as we have a change replay tool only for this version control system. We chose the following projects:

1. **Apache Gora** is an open source framework providing an in-memory data persistence for big data.
2. **Apache Any23** (Anything to triples) is a library that allows to extract structured data in RDF format.
3. **Apache Velocity** is a template engine for Java. Users can reference Java objects in a simple template language.
4. **Apache Xerces2** is an XML library that allows users to parse XML files and create XML files.

9.5.2. Integration results

For the extraction of an architectural model from source code, we have used *Extract* (Langhammer et al., 2016). It is a reverse engineering tool, which clusters classes of a software system to components. Table 5 gives an overview of the integrated versions of the projects and the result of the reverse engineering process.

To perform the integration, we ran the LIS integration transformation, which we explained in Section 8.2. We defined a transformation that creates an instance of the Vitruvius correspondence model during the integration, based on the information from the reverse engineering approach Extract. We were able to create an architectural model for all the open source projects with that approach and a Vitruvius correspondence model for each of the used projects. Furthermore, we found that the reverse engineering rules of the *Extract* tool did not conform to the CPRs for Java packages and classes defined in the Vitruvius case study, introduced in Section 9.3.2, such that none of the reverse-engineered components, interfaces, and data type elements can be kept consistent with their corresponding source code elements using these CPRs. This made the development of project-specific CPRs for these elements necessary.

9.5.3. Evolution results

The consistency rules followed by the LIS do not adhere to the generally defined CPRs between PCM and Java. Thus, to apply changes from the Git repositories of the projects, we first had to implement project-specific CPRs, which are similar to the ones described in Section 2 and implemented for the case study in Section 9.3.2, using Reactions. In particular, we implemented specific rules for

- adding and removing import statements (replaced the existing with empty Reactions),
- adding, renaming and removing architecturally relevant methods, as well as
- renaming and removing architecturally relevant classes and interfaces.

Table 6
Integrated versions, numbers of changes to the target version and numbers of created correspondences.

Project	Integrated → target version	Changes	Correspondences
Gora	0.6.0 → 0.6.1	419	336
Any23	0.90 → 1.0	164	555
Velocity	1.60 → 1.64	737	1130
Xerces	2.10 → 2.11	684	3598

All CPRs are also defined from the architectural model to source code. Hence, if users change architectural elements that have corresponding source code elements in the integrated code, consistency is preserved. These rules are only necessary to preserve consistency of the integrated model elements as they do not adhere to the general consistency rules between PCM and Java. For all elements added during further evolution of the system, the general CPRs are applied.

After the integration of each existing project, we applied changes made in the subsequent versions from the Git repository. Table 6 shows the target versions and the numbers of changes between the integrated and the target version. For the target versions, we chose the next minor or major version depending on how many changes occurred between them.

For all projects, we observed that Reactions were executed after 329 changes in total. Most of the changes add a method (69), add an import (62), add a super class (61) and remove an import (45), while only few changes occurred that create a new package (2), add a new interface (2) and create a new class (1). The occurred changes have been kept consistent as follows:

- 2 (0.6%) changes can be kept consistent automatically using the standard CPRs,
- 303 (92.1%) changes can be kept consistent using the defined project-specific CPRs,
- 24 (7.3%) changes cannot be kept consistent automatically, i.e., the users need to keep the architectural model consistent manually.

After applying all changes, the architectural model that is kept consistent was valid, i.e., it was a valid instance of its metamodel and fulfilled the defined consistency rules whose fulfillment was checked manually.

With this case study, we have shown that we are able to integrate existing open source projects into Vitruvius and keep an architectural model consistent with the existing code after more than 92.7% of the performed changes. This gives an initial indication of the overall applicability of the Vitruvius approach. Nevertheless, it also shows that at least in this case the generally defined CPRs do not fit well for the projects they were applied to. For most of the elements, project-specific rules had to be defined. This implies high effort for the integration of existing artifacts and may also be error prone, as the project-specific rules must not conflict with the general rules in the sense that not both kinds of rules are allowed to be applied to the same elements. Since source code and the consistency rules between it and architectural models provide many degrees of freedom,

it may be possible that generic CPRs are more applicable if the integrated models are instances of metamodels with less degrees of freedom or if at least the consistency rules between them are less variable. For example, it may be easier to apply a LIS to a UML class model and Java code. It will be part of future research to investigate that hypothesis. Another approach to investigate will be to provide refactorings to make models compatible with defined CPRs, so that no project-specific rules need to be defined.

9.6. Threats to validity

In this section, we have first evaluated two individual contributions: We have demonstrated the practical applicability of the Reactions language by giving an indicator for the benefit of our language. We measured its conciseness in comparison to GPPLs, and evaluated the functionality of the RIS integration strategy. Afterwards, we provided an end-to-end evaluation of the complete approach by integrating existing projects and applying existing changes from version control to them. There are, however, threats to the validity of our results, which we classify regarding the criteria of Runeson and Höst (2008) and discuss in the following.

9.6.1. Construct validity

It is disputable if SLoC is a reliable indicator for the benefit of using one programming language instead of another. There are, however, no other metrics that are established for that purpose, as stated in Section 9.3.3. Furthermore, we argue that Java, Xtend, and the Reactions language have a similar and partially identical concrete syntax, and therefore yield comparable SLoC results. More specifically, declarative elements of the Reactions language do not replace complex control flow expressions as it is the case, for example, for functional languages. Instead, they replace simple expressions like variable declarations and method invocations. Thus, the conciseness that is gained through the language constructs is not sacrificed by an increase in complexity.

We measured the applicability of our approach in the overall evaluation by counting the numbers of modifications that we were able to keep consistent. Although we claim that this is a reasonable initial indicator for the applicability of the approach, a final statement on its applicability would also require to measure the effort, especially for defining the CPRs. This will be part of controlled experiments that we plan to conduct in future work.

9.6.2. Internal validity

In each of the case studies, the same methodologists were involved in the development of both consistency preservation versions with Reactions and a GPPL. Consequently, the order in which both versions were implemented may have influenced their quality as the second implementation could potentially profit from insights during the first one. The Reactions from PCM to Java, for example, may have been influenced by the preexisting Xtend code. We tried to mitigate this by implementing Reactions in the automotive study first, so that maturity effects may only have improved the subsequent Java implementation as suggested in our evaluation template (Kramer et al., 2016).

Furthermore, all implementations are individual solutions. Due to many degrees of freedom in both languages, especially in the GPPL, different solutions are possible and would potentially lead to other results.

9.6.3. External validity

For the consistency preservation languages, we conducted only two case studies in two different domains so that it can be questioned whether our results can be generalized for further domains. At least, we intentionally selected two case studies from two different domains for our validation. Therefore, we are currently conducting further case studies that integrate more metamodels in the context of component-based software development and in the context of embedded automotive software.

The same threat applies to the overall evaluation, in which we applied the case study for component-based systems to several projects. Although the restriction to the domain of component-based system could be seen a threat to external validity, we at least applied our approach to four different projects to enhance validity. Due to the lack of publicly available projects in the domain of automotive software systems, we were not able to apply that case study to existing projects in the overall evaluation.

The RIS was only applied to PCM models rather than models of other domains and only using one set of CPRs. In consequence, one could argue that it may not applicable to other domains or with other consistency rules. First, the strategy is designed to be correct-by-construction, thus functional correctness in the evaluation only validates that no mistakes were made during its implementation. Second, the strategy is only sensible to additional constraints within a model that are introduced by the CPRs by construction. This means that CPRs can always create a consistent other model as long as the rules do not introduce further constraints to one of the metamodels that further restricts the space of valid models. Without such restrictions the strategy has no domain-specific requirements, thus different results when applying the strategy to other domains are not expectable.

Finally, we briefly discuss why the comparison of our Reactions language with a GPPL and not with a competing approach is *not* a threat to external validity. We already discussed the selection of a GPPL as the evaluation baseline in Section 9.3.3. Existing languages do not support incremental and edit-based consistency preservation. Apart from that, in the literature, several promising approaches that can be used for consistency preservation have been presented. The potential benefit of these approaches over GPPLs have, however, not been evaluated empirically. It is, for example, unclear whether their usage results in code that can be developed or maintained in less time or with fewer errors than GPPL code. Therefore, we have not compared the code size of Reactions to these approaches but to GPPLs. We argue that it is more likely that consistency preservation for software development in an industrial setting would be developed with established languages and not with languages that may pose a higher risk as their benefits have not been shown yet and efforts for learning and using them are unknown.

9.7. Discussion

We presented isolated evaluations for the applicability of our proposed consistency preservation and model integration strategies in the Vitruvius approach, as well as an overall evaluation applying the complete Vitruvius approach to existing open source projects. The results are an initial indicator that the approach is feasible and can be applied to realistic scenarios. Although we applied it especially to component-based software systems, our approach is not limited to this domain. To evaluate the general applicability of the approach, we will also apply it to other domains. We started with an initial application to embedded automotive software development, as described in Section 9.3.2. We also plan to apply the approach to non-software domains, such as energy grids (Burger et al., 2016) and electrical engineering, keeping different models of a circuit board layout in Matlab/Simulink and EAGLE consistent.

The efficiency of our approach is another property that could be evaluated. We do not provide a dedicated evaluation of performance and scalability, since it is out of the scope of this article. We can, however, argue why efficiency is not expected to be

a drawback of our approach. Due to the delta-based operation of our consistency preservation, performance does not depend on the size of the involved models, but only on the number of changes that are performed. In consequence, we expect efficiency not to be an issue of our approach.

10. Related work

In this section, we relate our Vitruvius approach to other research. We discuss the differences to other approaches for holistic system development, consistency preservation and research on view-based development.

10.1. Holistic system development

Approaches for holistic system development consider a software system or a software-intensive technical system as a whole instead of dealing with models that are independently developed by different roles and only kept consistent by manual synchronization. In general, projective approaches, relying on one model describing the system with projective views derived from it, and synthetic approaches, composing the complete system of descriptions in different views, can be distinguished (International Organization for Standardization, 2011).

A projective approach that is most related to Vitruvius is the Orthographic Software Modeling introduced by Atkinson et al. (2010). As introduced in Section 3.2, it relies on a SUM containing all information about the system, from which views can be derived. Nevertheless, the approach does not define a construction process for SUMs and does especially not support the reuse of existing metamodels and tooling defined for them. Another SUM-based approach is OpenMETA (Sztipanovits et al., 2014), which provides a unified design space for multiple models to support a model-spanning design space exploration. Nevertheless, it relies on import and export functionalities rather than a continuous consistency preservation based on fine-grained changes. The DesignSpace approach of Demuth et al. (2015) provides a collaboration platform for keeping different artifacts consistent. It allows to define consistency preservation mechanisms and manages trace links to inform developers about changes. Nevertheless, it does not provide languages to support the specification of consistency rules and to inductively guarantee consistency.

Another research field concerning holistic system development is multi-paradigm modeling, originally introduced by Vangheluwe et al. (2002) and Vangheluwe and de Lara (2003). It investigates the coupling of different modeling paradigms, i.e. metamodels, based on integration and transformation, and is still subject of current research (Amaral et al., 2010). A modeling framework to define relations between existing model types is provided by the macromodel approach (Salay et al., 2008, 2009). This approach focuses on macroscopic relations rather than fine-grained consistency preservation and especially only supports the definition of relations between models, but does not provide a complete framework to abstract from redundancies and to provide a system that behaves like a SUM. Its main purpose is to improve comprehensibility and to help maintain the relations by providing necessary traceability information. It was specifically applied to the development of vehicle control software (Salay et al., 2012). A related approach are megamodels, first applied by Favre and NGuyen (2005). Macromodels are an abstract description of models and relations between them. Usually, they are independent from concrete modeling languages (Diskin et al., 2013). The relations can be used to apply different model management operations to these megamodels (Salay et al., 2015). Similar to macromodels, the focus here is on global relations between models rather than the fined-grained consistency rules that we consider in our approach.

Several holistic system development approaches are specific to one domain. One popular tool is the SysML (Object Management Group, 2019; Delligatti, 2013). It provides a common and extendable language for systems modeling based on the UML. A project focused on modeling and simulating real-time systems is Ptolemy (Ptolemaeus, 2014). It especially focuses on the composition of dynamic behavior models and does not consider structural models. While these approaches provide domain-specific solutions, Vitruvius is a domain-agnostic framework, which supports holistic system design for arbitrary domains and metamodels.

10.2. Consistency preservation

In the following, we give an overview on work that investigates consistency preservation and languages to support its specification. In active database systems (Paton and Díaz, 1999) consistency can be preserved based on rules that specify which updates should lead to further database updates. Such rules are often expressed in terms of an event, a condition, and an action. This overall structure of so-called ECA rules is also similar to the structure of Reactions. In addition to this general structure, the Reactions language supports methodologists specifically in preserving consistency between models for which more structural and typing information is available through metamodels. The language was designed to leverage all this information from Ecore-based metamodels. As a result, Reactions and the changes that trigger them do not need to be described indirectly based on Java objects. Instead, methodologists can stay on the level of the domain concepts and their properties defined in the metamodels.

There are different approaches to consistency preservation for instances of different metamodels. Two models can be kept consistent by updating one if the other was modified, detecting modifications either by monitoring edit operations or by computing differences. For that, incremental model transformation tools are widely used, but many of them are state-based, potentially losing model information in comparison to delta-based approaches, as discussed in Section 5.1. Especially bidirectional transformations (BX), which define consistency constraints and their restoration in both directions (Stevens, 2010), have been extensively researched (Hidaka et al., 2016). Those languages can be separated into rather relational and often bidirectional languages comparable to our Mapping language, like QVT-R or TGGs, and operational and usually unidirectional languages comparable to our Reactions language, such as QVT-O and VIATRA. Some languages follow a hybrid paradigm, expressing relations declaratively as far as possible and providing operational constructs for specific purposes, such as the Atlas Transformation Language (ATL) (Jouault and Kurtev, 2006) and the Epsilon Transformation Language (ETL) (Kolovos et al., 2010b).

10.2.1. Operational transformation languages

Regarding operational approaches, most related to our approach and especially the Reactions language is the VIATRA project (Bergmann et al., 2015). VIATRA is an event-driven model transformation platform that can be used for preserving consistency by describing consistency repair in reaction to events. It defines event-driven transformations that consist of a precondition and an action. A precondition is the creation, update, or deletion of a pattern in the model graph. Such a pattern can be specified with EMF-IncQuery (Bergmann et al., 2012; Ujhelyi et al., 2015). Actions, which are executed if the precondition is fulfilled, have to be defined in Xtend. Preserving model consistency with such transformations requires preconditions with graph patterns and the complete repair logic with

Xtend, whereas our Reactions language offers additional language constructs to ease the repair specification. VIATRA provides an API for a Java library, which is also sometimes called *internal DSL*. The Reactions language, however, is an *external DSL*, so that methodologists benefit from additional type safety and editor features, such as syntax checks and auto-completion. VIATRA supports batch transformations, event-driven consistency preservation, and continuous validation, whereas the Reactions language is tailored to a single scenario, which is change-driven consistency preservation. Furthermore, VIATRA uses an event-driven virtual machine that supports arbitrary Java object graphs and therefore cannot leverage metamodel information, such as explicit containment relations, representations of attributes and references (in terms of fields with appropriate accessors) and so on. It uses the same mechanism to handle programmatic rule activation, model changes, and query result updates. The Reactions language, however, has separate language constructs to define which changes trigger a Reaction and to simply call routines without any indirection.

QVT-O is the imperative language of the QVT standard (Object Management Group, 2016). Like most other languages, it operates in a state-based manner. In consequence, operations such as element renaming or movement are not handled properly but usually result in treating the removal of an old and insertion of a new element, loosing all the information in the elements it was transformed to. The delta-based approach of the Reactions language prevents that behavior and ensures that after all kinds of changes related elements are updated properly.

10.2.2. Relational transformation languages

Regarding relational approaches, highly related to our approach and especially the Mappings language are Triple Graph Grammars (TGGs). TGGs, originally introduced by Schürr (1995), are a commonly used transformation approach. They consist of multiple rules, each consisting of three graphs, which are two model graph patterns and one correspondence graph between the patterns. From these rules, forward and backward batch transformations as well as incremental transformations can be produced. TGG tools were surveyed by Hildebrandt et al. (2013) and Leblebici et al. (2014), the latter with focus on incrementality. Two popular TGG tools with support for incremental executions are MoTE (Giese et al., 2010) and eMoflon (Anjorin et al., 2011). Many other implementations handle move operations as a combination of a deletion and a creation (Leblebici et al., 2014), which potentially results in information loss. TGG tools are generally limited in expressiveness (Hildebrandt et al., 2013). An example where bidirectional approaches like TGGs reach their limits is the partial mapping of classes in object-oriented code to architectural elements: If a user creates a Java class, further information is necessary in order to decide whether a component has to be created in the architectural model. With a bidirectional approach such information has to be encoded, e.g., in naming conventions. While the Mappings language alone suffers from the same restrictions arising from bidirectionality, its operationalization into Reactions allows to adapt the behavior accordingly and especially to integrate user decisions. This makes it possible to combine such conventions in one direction with flexibility in the other direction. A methodologist may, e.g., specify a Mapping that relates a Java class with an appropriate name in an appropriate package to an architectural component. This generates a Reaction that creates a class for each component. The Reaction creating a component for a class can be adapted, such that the user who creates a class is asked whether it shall correspond to a component in order to not require encoding the intent to create a component in specific naming of a package and class.

QVT-R is the relational and bidirectional transformation language of the QVT standard (Object Management Group, 2016). It allows to specify consistency preservation by defining the relations, comparable to Definition 3, that have to hold between two metamodels. Although semantics of that language is problematic at some points (Stevens, 2010), it is comparable to the introduced Mappings language. A central difference between these two languages is that the Mappings language generates a delta-based operationalization in terms of Reactions, while QVT-R operates state-based. In consequence, the Mappings language is able to properly handle operations like renaming and movement, which is not possible when comparing model states like in QVT-R, as already explained for QVT-O. Furthermore, QVT-R provides when- and where-clauses with direction-specific semantics, whereas the Mappings language differentiates between single-sided conditions that are checked in one direction and enforced in the other, and bidirectionalizable expressions that are always enforced.

10.2.3. Other transformational approaches

Finally, hybrid approaches like ATL (Jouault and Kurtev, 2006) and ETL (Kolovos et al., 2010b) can be seen as an improvement of pure operational languages, which provide a declarative framework for defining transformations. They do, however, not provide bidirectionality and thus do not reach the same abstraction as languages like QVT-R, TGGs or the Mappings language. They can be seen as an abstraction between the Reactions and the Mappings language. Like discussed for TGGs, both these languages do not support consistency rules that require user decisions about how elements shall be related, which is a common necessity in practice, thus being restricted in expressiveness.

A specific approach to build networks of bidirectional transformations like in a V-SUM are commonalities models. They introduce further models that contain the information that is shared between models and thus has to be kept consistent. They serve as a hub with bidirectional transformations to the actual models, acting like a multidirectional transformation. Their benefit is that they explicitly express common concepts of metamodels rather than implicitly encoding them in a transformation (Klare and Gleitze, 2019) and that they solve the problem of n-ary relations not being expressible in terms of a set of binary relations (Stevens, 2020). This concept has been considered on a rather theoretical basis (Stünkel et al., 2018; Diskin et al., 2018), discussing which kinds of relations can be expressed with such an approach, and from an engineering perspective (Klare and Gleitze, 2019), discussing the modular specification and composition of commonalities. However, all these approaches do not allow the modular development of transformations based on distributed knowledge about relations between some of the metamodels, which is the goal and central assumption of our work.

10.2.4. Non-transformational approaches

Apart from transformation languages, Xiong et al. (2009) proposed an approach for fixing inconsistencies based on the definition of constraints in an OCL-like syntax. It provides a fixed set of constraint operators, each having an assigned fixing operation that restores the constraint when it gets violated. While the proposed language has well-defined semantics, the expressiveness is restricted to the capabilities of the provided operators, which can neither be extended nor adapted. In contrast, the Mappings and Reactions language provide abstraction as well as expressiveness for serving both needs depending on the context.

The event-driven and reactive programming paradigms are also related to the Reactions language, as they allow to express "what to do, and let the language automatically manage when to do it" (Bainomugisha et al., 2013, p.52:3). In reactive programming, data dependencies rather than control flow dependencies are defined, and dependent values are automatically updated if a value is modified. Specialized approaches for automated model

consistency preservation were, for example, presented by Wimmer et al. (2012). Their approach detects coarse-grained changes rather than the atomic changes that the Reactions language handles. It is based on the assumption that developers want to deal with coarse-grained evolution scenarios rather than atomic, low-level changes.

10.3. View-based development

View-based software development as a term has been proposed as early as the 1990s (Finkelstein et al., 1992). The first object-oriented frameworks, such as OMT (Rumbaugh et al., 1991) and Fusion (Coleman et al., 1994), already contained concepts for several diagram types for structural, behavioral and operational view points, leading to today's standards such as UML (Object Management Group, 2017). These standards describe a decomposition of system descriptions into several views, but often fail to describe the interdependencies between these views, and do not offer synchronization mechanisms between them. Furthermore, the set of view points or view types is fixed and cannot be extended by the user.

When model-driven tools for the definition of domain-specific languages and model transformations became available, the view-based paradigm was implemented in several approaches that have been named *Model View Approaches* in a recent survey (Brunelière et al., 2017). We mention approaches here that provide functionality for the creation of views on arbitrary metamodels (not limited to, e.g., only UML), support editability, and offer a non-intrusive definition of views, meaning that the original metamodels do not have to be modified. They do not necessarily provide multi-model consistency mechanisms. Triple Graph Grammars (Jakob et al., 2006; Jakob and Schürr, 2008; Anjorin et al., 2014) have been extended to an asymmetric case and can be used to create non-intrusive views on models. *EMF Facet* (Eclipse Foundation, 2020) extends models dynamically at run-time with so called facets, whose definition is stored in a separate file. Thus, neither the original metamodel nor its instances need to be modified to apply the facet. *EMF Views* (Brunelière et al., 2015) offers an SQL-like DSL to define views on heterogeneous models. In EMF views, the notion *virtual model/metamodel* is used to describe the views and view types, since they are proxies to other metamodels and models. Thus, EMF Views is a projective approach. This differs from our understanding of "virtual" (as in V-SUM), since we require editability and consistency preservation mechanisms, which EMF views does not offer. The Epsilon-based approach *Epsilon Decoration* (Kolovos et al., 2010a) uses tag-value pairs to define views over models. *VIATRA Viewers* (Semeráth et al., 2016) is an incremental approach to create a view definition framework. The *ModelJoin* language (Burger et al., 2014) offers the definition of editable views types and views on heterogeneous models in a textual DSL. The translatability of views and automatic fixes for untranslatable views in ModelJoin has also been studied (Burger and Schneider, 2016).

The Vitruvius approach is agnostic of the way in which the views are defined. Thus, all of the approaches mentioned above could be used in conjunction with the current implementation of Vitruvius. It should however be noted that it is beneficial if the languages and technologies used for the definition of views are aligned with those used for the definition of consistency preservation rules (see Section 11.3).

11. Future work

We have given an overview on the central ideas and concepts of the Vitruvius approach. It provides a well-defined conceptual and technological basis for future work and research. We will conduct extended evaluations to also investigate efficiency of our approach and the effort to build V-SUMs and especially the consistency preservation specifications, which were out of the scope of this article. Furthermore, we will investigate different conceptual topics, of which we give an overview in the following.

11.1. Evolving the V-SUM metamodel

A V-SUM metamodel can be instantiated multiple times to model different systems that use all or a subset of the metamodels and languages. In the running example, multiple systems can be modeled using the V-SUM metamodel of PCM, UML, and Java. Like every metamodel, a V-SUM metamodel is subject to modifications that affect the internal structure as well as the view types. This can, for example, be due to the evolution of the internal metamodels of the V-SUM metamodel, such as Java. In that case, instances of the V-SUM metamodel have to co-evolve with the metamodel modifications. While there is already research on co-evolution of metamodels and models (Burger and Gruschko, 2010; Herrmannsdoerfer et al., 2011; Demuth et al., 2013a,b; Hebig et al., 2017), it has to be investigated how this can be transferred to the evolution of a V-SUM metamodel, as additionally to the conformity of the instances to the metamodel, the consistency between the different models within the V-SUM must also be preserved according to the defined consistency rules.

11.2. Tolerating inconsistencies

Consistency between models does not always need to be and often even cannot be enforced immediately. Having to restore consistency immediately after fine-grained changes would restrict the developer (Nuseibeh et al., 2001) and is sometimes not possible, because the changed model itself is not consistent after that modification (Kehrer et al., 2013). In such cases, it is desirable to tolerate inconsistencies, e.g., as long as they are restricted to certain model elements, certain model regions, or certain intermediate model states. The idea of *tolerating inconsistency* has been prominently discussed in an article by Balzer (1991).

We allow the temporary toleration of inconsistencies by considering changes of appropriate granularity. According to Definition 8, a V-SUM always stays consistent after applying a change. By appropriately defining the set of possible and potentially deeply composed changes in the models to those resulting in a state, in which consistency to all other models can be restored, the intermediate states when applying only a part of the change induce tolerated inconsistencies. Imagine an element that shall be transformed into another model, but to avoid ambiguities some of its attributes have to be defined first. The change describing the creation and insertion of that element without setting the attributes introduces a tolerated inconsistent state. After also defining the attribute values, that complete change can be processed and consistency can be restored. This can be seen as an approach to handle appropriate transactions of changes. Deciding when and where inconsistencies should be temporarily allowed is, however, a problem that cannot be solved universally. It is up to the methodologists who creates the CPRs to appropriately handle such transactions. Since this only allows a short-term toleration of inconsistencies, it will be part of future work to figure out appropriate kinds of transactions for recording changes and for defining consistency preservation on.

11.3. Coupling view and consistency specification

In the Vitruvius development process introduced in Section 6, the definition of view types and CPRs are two steps that follow each other. First, the definition of view types relies on metamod-

els and consistency rules within a V-SUM. Thus their complexity and the consistency rules influence the difficulty of defining view types. In consequence, we will further research how view type definition depends on the kinds and complexity of consistency rules and how this process can be supported, e.g., by concepts from OSM (Atkinson et al., 2010) (see Section 3.2). Second, to support this, the consistency rules should, in parts, be derived from the view types, since the information needs of developers are persisted in them. As a view type definition consists of a metamodel for the view type and transformations from and to the V-SUM metamodel, this information could be used to extract consistency rules. For example, if two classes from two different metamodels are mapped to the same element of one view type, one can assume that these classes represent the same concept and should thus be kept consistent.

In the current implementation of Vitruvius, the definition of CPRs and view type definitions are technically independent. It could be beneficial to give hints for CPR creation based on information in the view type definition to the methodologist. Vice versa, consistency information should also be respected in the view types, so that it becomes impossible to edit views in a way that consistency rules are violated. Last, the CPRs could be linked to the view type definitions to enable their co-evolution.

11.4. Defining multi-model consistency

Consistency preservation in Vitruvius can be defined with the Reactions and the Mappings language. These languages complement each other, because they provide different levels of abstraction and expressiveness. Additionally, Reactions are generated from Mappings. The languages, however, lack a deeper integration. Mappings and Reactions are specified in different, independent files. We are working on an integration of Reactions into Mappings with checks regarding conflicts of constraints defined in Mappings and CPRs described with Reactions.

Furthermore, consistency preservation is currently considered as the combination of CPRs between metamodel pairs. Defining consistency between more than two metamodels can easily lead to contradictions between the CPRs. First, if CPRs are executed transitively, i.e., the execution of one rule triggers the execution of others, it is unclear if correspondences can be considered transitively to achieve proper results. If rules are contradictory, which can easily occur if they are developed by different domain experts, propagation cycles due to alternating values or dependencies on the execution order of the CPRs can occur. Defining a tree of consistency rules to avoid those issues results in reduced modularity and comprehensibility, and also prevents modular development of consistency rules Klare (2018). If the consistency rules between architectural model and code are defined transitively over UML, the UML metamodel cannot be omitted in a project, which gets even more problematic if a much higher number of consistency rules is involved.

Although one could argue that such n-ary relations should be expressed in n-ary CPRs, Stevens (2020) provides convincing arguments to stick with the definition of networks of binary transformations, as the binary case is hard enough to think about and also fosters the modular definition of CPRs by domain experts. It was also the result of a Dagstuhl seminar that "it seems likely that networks of bidirectional transformations suffice for specifying multidirectional transformations" (Cleve et al., 2019, p. 7). We will therefore investigate how CPRs can be coupled such that they are not contradictory by design and modular in the sense that they can be defined and reused independent from each other. We will also consider the composition of V-SUM metamodels by defining CPRs not only between existing metamodels but also between view types of V-SUM metamodels to stay independent from the internal CPRs of a V-SUM metamodel and allow modular composition. Our initial ideas for supporting multi-model consistency preservation in Vitruvius are discussed in Klare (2018). We also provide a classification of interoperability issues in transformation networks in Klare et al. (2019).

11.5. Enabling consistency-aware versions & variants management

Management of versions and variants has been well researched, especially for code-centric software development projects. While it is basically possible to apply that research also to a V-SUM-based approach and reuse existing versioning tools such as Git or variants management and product line approaches, all of them are not aware of consistency between the artifacts. For example, if two developers perform modifications that shall be merged, conflicts during the merge process are currently resolved based on heuristics or developer decisions, but are not necessarily compliant with the defined consistency rules.

We will therefore develop an approach for consistency-aware versions and variants management, which reuses the change-driven property of Vitruvius to represent variability of software systems. In consequence, navigating between versions and variants will always depend on change sequences and the CPRs defined for them, such that the system description stays consistent according to these rules. This approach can be seen as an extension of the change-based versioning approach in EMFStore of Koegel and Helming (2010a), Koegel et al. (2009, 2010b), and also reuses concepts of the *DeltaEcore* approach by Seidl et al. (2014) and Seidl (2015). We gave an initial overview on our approach in Ananieva et al. (2018b).

12. Conclusion

View-based development is inevitable for creating and evolving complex software-intensive technical systems. Using several views leads to the problem of view consistency. In this paper, we have presented the Vitruvius approach, a view-based, model-driven approach for consistent development of complex systems. The approach enables developers to work with different models (including code-based representations) that represent the system under development from different viewpoints. It supports the preservation of consistency between views. Our approach is not limited to models that describe software, but can be used with any kind of metamodel or textual formal language. We have introduced the concept of the *Virtual Single Underlying Model (V-SUM)*, whose key features are the non-intrusive reuse of existing metamodels and tools, and a modular structure that enables the reuse of parts of the metamodels and models, and which makes the explicit definition of consistency possible.

Consistency preservation rules are first-class entities in the Vitruvius approach. They are used to express the semantic relations between models of different languages/metamodels as well as methods to restore consistency in case a rule is violated. Vitruvius follows a *delta-based* paradigm: Consistency preservation rules are defined in relation to possible changes to the models. We have argued why we deem such an approach beneficial compared to state-based approaches, although it requires additional effort. Besides a formal definition of multi-model consistency, we have presented the *Reactions* and the *Mappings* language, with which consistency preservation rules can be expressed.

We do of course not assume that existing development processes are changed completely and that existing projects are redeveloped using the Vitruvius approach. Thus, we have presented two strategies that allow the integration of existing projects into a changed-based process such as Vitruvius.

We have evaluated our approach using two case study domains (component-based software engineering and embedded automotive software architectures). As a research prototype, we

have implemented Vitruvius's concepts in the Eclipse Modeling Framework. Although it is hard to quantify the benefit of an approach as fundamental as Vitruvius, we have been able to show completeness and correctness of our consistency preservation languages, as well as their applicability and compactness, which give an indicator for the improvements that can be gained through the approach. We have further evaluated the integration strategies and the overall approach with historical data from several open source projects. The results show that a fully automated consistency preservation is hard to reach, but that the Vitruvius approach can aid developers in creating rules that keep consistency in most of the cases.

The Vitruvius approach is a comprehensive approach for the development of systems, in which all the information that is modeled about a system can be connected semantically to create a higher level of consistency. Although consistency is a term that is not central in recent research on system development and model-driven methods, we believe that it is one of the key problems in the development of today's complex systems, and should be treated as a first-level entity in development processes, as done in Vitruvius.

Verifiability

The complete implementation of our evaluation presented in Section 9 is available at GitHub (Vitruvius GitHub, 2020b) and is documented in an associated Wiki (Vitruvius GitHub, 2020a). The Vitruvius framework with the consistency preservation languages, the Reactions language and the Mappings language, are available in the framework project (Vitruvius GitHub, 2020c). The implementations of the two case studies can be found in two repositories, one for the component-based systems case study (Vitruvius GitHub, 2020d) and one for the embedded automotive software case study (Vitruvius GitHub, 2020e). These projects contain the consistency preservation implementations, as well as test cases and test scenarios. All projects deploy to an Eclipse update site from which the projects can be installed. All dependencies, especially to used metamodels, are documented within the projects and the Eclipse update site specifications. The tests, however, have to be manually imported from the repositories. The implementation of the integration strategies LIS and RIS are not deployed to an update site. They can be found in an additional repository (Vitruvius GitHub, 2020f).

In addition, we provide a dedicated reproduction package (Klare, 2020), which contains a prepared Eclipse environment with the necessary, precompiled repositories and workspaces, as explained above. It also provides a script that sets up an environment with all necessary dependencies based on Docker.

CRediT authorship contribution statement

Heiko Klare: Conceptualization, Software, Investigation, Writing - original draft, Writing - review & editing. **Max E. Kramer:** Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Michael Langhammer:** Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Dominik Werle:** Conceptualization, Software, Validation, Writing - original draft. **Erik Burger:** Conceptualization, Methodology, Writing - original draft, Writing - review & editing, Supervision. **Ralf Reussner:** Conceptualization, Methodology, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

```
reaction = "reaction" , [xbase identifier] , "{" ,
   execution time , change type ,
      ["with" , change properties check] ,
   "call" , (routine call | routine call block) ,
"}";

execution time = "before" | "after";

change properties check = xbase expression;

routine call = xbase identifier , "(" , [arguments] , ")";

arguments = argument expression , {"," , argument expression};

argument expression = xbase expression;

routine call block = "{" ,
   {[routine call | routine call expression]} - ,
"}";

routine call expression = xbase expression;

routine definition = "routine" , xbase identifier , "(" ,
   [parameters] , ")" , "{" ,
      {["match" , match block] ,
      "action" , "{" ,
         {action} - ,
      "}" ,
"}";

parameters = typed identifier , {"," , typed identifier};

typed identifier = type expression , xbase identifier;

type expression = xbase identifier , "::" , xbase identifier;

match block = "{" ,
   {(retrieval | match check block)} - ,
"}";

retrieval = (["val" , xbase identifier , "="] , "retrieve" ,
           ["optional"]) | ("requireabsenceof") ,
   type expression , "correspondingto" ,
   source element expression ,
   ["taggedwith" tag expression] ,
   ["with" retrieve properties check];

type expression = xbase identifier , "::" , xbase identifier;

tag expression = xbase expression;

source element expression = xbase expression;

retrieve properties check = xbase expression;

match check block = "check" , "{" ,
   {match check} - ,
"}";

match check = xbase expression;
```

Listing 7: Grammar of the Reactions language with rules for Reaction and Reaction routine definitions in EBNF (without rules for change types), from Kramer (2017).

Acknowledgment

The authors greatly appreciate the constructive and valuable comments of the unknown reviewers.

Appendix A. Consistency languages syntax

We introduced two consistency preservation languages, the Reactions language and the Mappings language, in Section 7. The essential constructs were explained on stub listings, a metamodel of the language constructs and an example for both languages.

```
mapping = "mapping" , xbase identifier ,
 ["dependson(" , mapping dependency, ")"] , "{" ,
   "map (" , parameters , ")" ,
      ["with" , "{" , {single-sided condition} - , "}"] ,
   "and (" , parameters , ")" ,
      ["with" , "{" , {single-sided condition} - , "}"] ,
   ["suchthat" , "{" , {bidirectionalizable condition} -
      , "}"] ,
   ["forwardexecute{" , {xbase expression} - , "}" ,
   "backwardexecute{" , {xbase expression} - , "}"] ,
"}";

bootstrap mapping = "bootstrapmapping" , xbase identifier ,
    "{" , "create(" , parameters , ")" ,
   ["with" , "{" , {single-sided condition} - , "}"] ,
"}";

mapping dependency = xbase identifier , {"," , xbase identifier};

parameters = typed identifier , {"," , typed identifier};

typed identifier = type expression , xbase identifier;

type expression = xbase identifier , "::" , xbase identifier;

single-sided condition = feature condition |
   resource condition | check and enforce code;

feature condition = (multi value condition |
   single value condition | element condition |
   ["not"] , "empty") ,
   feature expression;

multi value condition = {value expression} - ,
   ["not"] , ("equals" | "in");

value expression = xbase expression;

single value condition = value expression ,
   (index expression | num compare expression);

index expression = ["not"] , "atindex" ,
   xbase expression , "in";

num compare expression = "<=" | "<" | ">=" | ">";

element condition = element expression , "defaultcontainedin";

element expression = xbase expression;

feature expression = xbase identifier , "." , xbase identifier;

resource condition = "defaultpathfor" , element expression ,
   "=" , ["pathof" , element expression , "+"] , xbase string;

check and enforce code = "check" , xbase expression block ,
   "enforce" , xbase expression block;

bidirectionalizable condition = xbase expression;
```

Listing 8: Simplified grammar of the Mappings language in EBNF, from Kramer (2017).

```
reaction CreatedRepository {
   after element adl::Repository created and inserted as root
   call {
      val repository = newValue;
      createPackage(repository, null, repository.entityName,
         "repository_root");
      createSubPackages(repository);
   }
}

routine createSubPackages(adl::Repository repository) {
   match {
      val repositoryPackage = retrieve oo::Package
         corresponding to repository
   }
   action {
      call {
         createPackage(repository, repositoryPackage,
            "interfaces", "interfaces");
         createPackage(repository, repositoryPackage,
            "components", "components");
      }
   }
}

routine createPackage(EObject sourceElement,
      oo::Package parent, String packageName, String newTag) {
   match {
      require absence of oo::Package corresponding to
         sourceElement tagged with newTag
   }
   action {
      val pkg = create oo::Package and initialize {
         pkg.name = packageName;
         if (parent !== null) {
            pkg.namespaces += parent.namespaces;
            pkg.namespaces += parent.name;
         } else {
            persistProjectRelative(sourceELement,
               pkg, "model/" + pkg.name + ".oo");
         }
      }
      add correspondence between pkg and sourceElement
         tagged with newTag
   }
}
```

Listing 9: Reactions for the creation of an object-oriented package structure after creating a repository in the ADL.

We provide the EBNFs for both languages to illustrate their complete capabilities. The EBNF for the Reactions language is depicted in Listing 7. It omits the definition of supported change types. The EBNF for the Mappings language is depicted in Listing 8.

Both languages reuse the Xbase expression language (Efftinge et al., 2012). The `xbase identifier` rule enables referencing different kinds of elements, such as meta-classes, methods and so on. The allowed values of such an identifier depend on the context and will not be discussed in detail. For example, statements of the form `xbase identifier, ''::'', xbase identifier` are used for references to meta-classes. The first identifier references the metamodel, while the second identifier references a meta-class within the metamodel. The `xbase expression` rules can either be a single expression or an expression block, such as the

```
reaction PackageCreated {
   after element oo::Package inserted as root
   call {
      createRepository(newValue, "repository_root")
   }
}

routine createRepository(oo::Package pkg, String newTag) {
   match {
      require absence of adl::Repository corresponding to pkg
   }
   action {
      val repository = create adl::Repository and initialize {
         repository.entityName = pkg.name
         persistProjectRelative(javaPackage, repository,
            "model/" + repository.entityName + ".repository")
      }

      add correspondence between repository and pkg tagged
         with newTag
   }
}
```

Listing 10: Reaction for the creation of a repository in the ADL after creating a package in an object-orientation model.

body of a method. Depending on the context, such an expression has different input and return values. For example, the expression in a "check" rule returns a Boolean value.

Appendix B. Reactions language examples

In Section 7.1, we introduced the Reactions language on a simple example for creating a class in an object-oriented representation after a repository in an ADL was introduced. We demonstrate the language capabilities with more complex Reactions in Listings 9 and 10. For even more sophisticated Reactions, we refer to implementations in our code repositories (Vitruvius GitHub, 2020d), from which the depicted Reactions are an extract. The former describes the creation of the package structure in an object-orientation model that corresponds to a repository in an ADL after its creation. The latter, on the other hand, defines the creation of repository in the ADL after a root package is introduced in the object-orientation model. This represents an extract of the Reactions that provide equal behavior than those generated from the Mapping Listing 4, except that using the Mapping, a repository would only be generated after the complete package structure in the object-orientation model is created.

The Reactions can be executed transitively until no further changes occur, i.e. after creating a package the Reaction creates a repository in the ADL model, which, in turn, triggers the creation of the other packages in the object-orientation model. This terminates because of the absence checks in the Reaction routines, which ensure that no duplicate elements are created, according to the strategy proposed in Klare et al. (2019).

The Reactions use the method `persistProjectRelative`, which is an operation provided by an internal API allowing to persist an element as a root element in a file. This is only mandatory if the development environment of the V-SUM may get restarted, requiring a reload of the V-SUM. Otherwise, the model elements could also only persist in-memory.

References

Amaral, V., Hardebolle, C., Karsai, G., Lengyel, L., Levendovszky, T., 2010. Recent advances in multi-paradigm modeling. In: Models in Software Engineering. Springer Berlin Heidelberg, pp. 220–224.

Ananieva, S., Burger, E., Stier, C., 2018a. Model-driven consistency preservation in AutomationML. In: 14th IEEE International Conference on Automation Science and Engineering. IEEE, pp. 1536–1541.

Ananieva, S., Klare, H., Burger, E., Reussner, R., 2018b. Variants and versions management for models with integrated consistency preservation. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems. VAMOS 2018, ACM, pp. 3–10.

Anjorin, A., Lauder, M., Patzina, S., Schürr, A., 2011. EMoflon: Leveraging EMF and professional CASE tools. In: 3. Workshop Methodische Entwicklung von Modellierungswerkzeugen. MEMWe2011, In: Lecture Notes in Informatics, vol. P-192, Gesellschaft für Informatik e.V. (GI).

Anjorin, A., Rose, S., Deckwerth, F., Schürr, A., 2014. Efficient model synchronization with view triple graph grammars. In: Proceedings of the 10th European Conference on Modelling Foundations and Applications. ECMFA, Springer International Publishing, pp. 1–17.

Armengaud, E., Zoier, M., Baumgart, A., Biehl, M., Chen, D., Griessnig, G., Hein, C., Ritter, T., Tavakoli Kolagari, R., 2011. Model-based toolchain for the efficient development of safety-relevant automotive embedded systems. In: SAE 2011 World Congress & Exhibition.

Atkinson, C., Bostan, P., Brenner, D., Falcone, G., Gutheil, M., Hummel, O., Juhasz, M., Stoll, D., 2008. Modeling components and component-based systems in KobrA. In: The Common Component Modeling Example. In: Lecture Notes in Computer Science, vol. 5153, Springer Berlin Heidelberg, pp. 54–84.

Atkinson, C., Stoll, D., Bostan, P., 2010. Orthographic software modeling: A practical approach to view-based development. In: Evaluation of Novel Approaches to Software Engineering. In: Communications in Computer and Information Science, vol. 69, Springer Berlin Heidelberg, pp. 206–219.

Atkinson, C., Tunjic, C., 2017. A deep view-point language for projective modeling. In: IEEE 21st International Enterprise Distributed Object Computing Conference. EDOC 2017, pp. 133–142.

AutomationML e.V. (GI), 2018. Whitepaper AutomationML edition 2.1, Part 1 – Architecture and general requirements. https://www.automationml.org/o.red.c/publications.html.

Bainomugisha, E., Carreton, A.L., Cutsem, T.v., Mostinckx, S., Meuter, W.d., 2013. A survey on reactive programming. ACM Computing Surveys 45 (4), 52:1–52:34.

Balzer, R., 1991. Tolerating inconsistency. In: Proceedings of the 13th International Conference on Software Engineering. IEEE, pp. 158–165.

Becker, S., Hauck, M., Trifu, M., Krogmann, K., Kofron, J., 2010. Reverse engineering component models for quality predictions. In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track. IEEE, pp. 199–202.

Bergmann, G., Dávid, I., Hegedüs, A., Horváth, A., Ráth, I., Ujhelyi, Z., Varró, D., 2015. Viatra 3: A reactive model transformation platform. In: Theory and Practice of Model Transformations. In: Lecture Notes in Computer Science, vol. 9152, Springer International Publishing, pp. 101–110.

Bergmann, G., Ráth, I., Varró, G., Varró, D., 2012. Change-driven model transformations. Software & Systems Modeling 11 (3), 431–461.

Böhme, R., Reussner, R., 2008. Validation of predictions with measurements. In: Dependability Metrics: Advanced Lectures. In: Lecture Notes in Computer Science, vol. 4909, Springer Berlin Heidelberg, pp. 14–18.

Brooks, F.P., 1987. No silver bullet: Essence and accidents of software engineering. IEEE Computer 20 (4), 10–19.

Brosch, F., Koziolek, H., Buhnova, B., Reussner, R., 2012. Architecture-based reliability prediction with the palladio component model. IEEE Transactions in Software Engineering 38 (6), 1319–1339.

Brunelière, H., Burger, E., Cabot, J., Wimmer, M., 2017. A feature-based survey of model view approaches. Software & Systems Modeling 18 (3), 1931–1952.

Brunelière, H., Garcia Perez, J., Wimmer, M., Cabot, J., 2015. EMF VIews: A view mechanism for integrating heterogeneous models. In: 34th International Conference on Conceptual Modeling. Springer International Publishing, pp. 317–325.

Burger, E., 2014. Flexible Views for View-based Model-driven Development. (Ph.D. thesis). KIT Scientific Publishing, Karlsruhe, Germany.

Burger, E., Gruschko, B., 2010. A change metamodel for the evolution of MOF-based metamodels. In: Proceedings of Modellierung 2010. In: Lecture Notes in Informatics, vol. P-161, Gesellschaft für Informatik e.V. (GI), pp. 285–300.

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.

Burger, E., Mittelbach, V., Koziolek, A., 2016. View-based and model-driven outage management for the smart grid. In: Proceedings of the 11th Workshop on Models@run.time co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. MODELS 2016, CEUR Workshop Proceedings.

Burger, E., Schneider, O., 2016. Translatability and translation of updated views in ModelJoin. In: Theory and Practice of Model Transformations: 9th International Conference. ICMT 2016, Held as Part of STAF 2016, In: Lecture Notes in Computer Science, vol. 9765, Springer International Publishing, pp. 55–69.

Cicchetti, A., Ciccozzi, F., Pierantonio, A., 2019. Multi-view approaches for software and system modelling: a systematic literature review. Software & Systems Modeling 18 (6), 3207–3233.

Cleve, A., Kindler, E., Stevens, P., Zaytsev, V., 2019. Multidirectional transformations and synchronisations (Dagstuhl seminar 18491). Dagstuhl Rep. 8 (12), 1–48.

Coleman, D., Arnold, P., Bodoff, S., Gilchrist, H., Hayes, F., Jeremaes, P., 1994. Object-Oriented Development: The Fusion Method. Prentice Hall, Englewood Cliffs, NJ.

Czarnecki, K., Helsen, S., 2006. Feature-based survey of model transformation approaches. IBM Systems Journal 45 (3), 621–645.

Delligatti, L., 2013. SysML Distilled: A Brief Guide To the Systems Modeling Language, first ed. Addison-Wesley Professional.

Demuth, A., Lopez-Herrejon, R.E., Egyed, A., 2013a. Co-evolution of metamodels and models through consistent change propagation. In: Proceedings of the Workshop on Models and Evolution co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems. MODELS 2013, CEUR Workshop Proceedings.

Demuth, A., Lopez-Herrejon, R., Egyed, A., 2013b. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In: Model-Driven Engineering Languages and Systems. In: Lecture Notes in Computer Science, vol. 8107, Springer Berlin Heidelberg, pp. 287–303.

Demuth, A., Riedl-Ehrenleitner, M., Nöhrer, A., Hehenberger, P., Zeman, K., Egyed, A., 2015. DesignSpace: An infrastructure for multi-user/multi-tool engineering. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. SAC '15, ACM, pp. 1486–1491.

Diskin, Z., Kokaly, S., Maibaum, T., 2013. Mapping-aware megamodeling: Design patterns and laws. In: Software Language Engineering. Springer International Publishing, pp. 322–343.

Diskin, Z., König, H., Lawford, M., 2018. Multiple model synchronization with multiary delta lenses. In: Fundamental Approaches to Software Engineering. Springer International Publishing, pp. 21–37.

Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F., 2011. From state- to delta-based bidirectional model transformations: The symmetric case. In: Model Driven Engineering Languages and Systems. In: Lecture Notes in Computer Science, vol. 6981, Springer Berlin Heidelberg, pp. 304–318.

Duarte, F., Pires, C., de Souza, C.A., Ros, J.P., Leao, R.M., e Silva, E.d.S., Leite, J., Cortellessa, V., Mossé, D., Cai, Y., 2010. Experience with a new architecture review process using a globally distributed architecture review team. In: 5th IEEE International Conference on Global Software Engineering. IEEE, pp. 109–118.

Eclipse Foundation, 2020. EMF Facet. URL: https://www.eclipse.org/facet/. (accessed 7 September 2020).

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M., 2012. Xbase: Implementing domain-specific languages for Java. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering. GPCE '12, ACM, pp. 112–121.

Efftinge, S., Völter, M., 2006. oAW xText: a framework for textual DSLs. In: Eclipsecon Summit Europe 2006.

Eramo, R., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A., 2012. A model-driven approach to automate the propagation of changes among Architecture Description Languages. Software & Systems Modeling 11 (1), 29–53.

ETAS Group, 2020. ASCET-DEVELOPER. URL: https://www.etas.com/ascet. (accessed 7 September 2020).

Favre, J.-M., NGuyen, T., 2005. Towards a megamodel to model software evolution through transformations. Electronic Notes in Theoretical Computer Science 127 (3), 59–74.

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.

Giese, H., Hildebrandt, S., Neumann, S., 2010. Model synchronization at work: Keeping SysML and AUTOSAR models consistent. In: Graph Transformations and Model-Driven Engineering. In: Lecture Notes in Computer Science, vol. 5765, Springer Berlin Heidelberg, pp. 555–579.

Gischel, B., 2013. EPLAN Electric P8 Reference Handbook, third ed. Hanser Fachbuch.

Goldschmidt, T., 2011. View-Based Textual Modelling. (Ph.D. thesis). KIT Scientific Publishing, Karlsruhe, Germany.

Goldschmidt, T., Becker, S., Burger, E., 2012. Towards a tool-oriented taxonomy of view-based modelling. In: Proceedings of the Modellierung 2012. In: Lecture Notes in Informatics, vol. P-201, Gesellschaft für Informatik e.V. (GI), pp. 59–74.

Google, 2020. Guice dependency injection framework. URL: https://github.com/google/guice. (accessed 7 September 2020).

Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A., 2014. The Java Language Specification, Java SE 8 Edition, first ed. Addison-Wesley Professional.

Gouvêa, D.D., Muniz, C., Pinto, G., Avritzer, A., Leão, R.M.M., de Souza e Silva, E., Diniz, M.C., Berardinelli, L., Leite, J.C.B., Mossé, D., Cai, Y., Dalton, M., Happe, L., Koziolek, A., 2012. Experience with model-based performance, reliability and adaptability assessment of a complex industrial architecture. Software & Systems Modeling 12 (4), 765–787, Special Issue on Performance Modeling.

Gouvêa, D.D., Muniz, C., Pinto, G., Avritzer, A., Leão, R.M.M., de Souza e Silva, E., Diniz, M.C., Berardinelli, L., Leite, J.C.B., Mossé, D., Cai, Y., Dalton, M., Kapova, L., Koziolek, A., 2011. Experience building non-functional requirement models of a complex industrial architecture. In: Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering. ICPE 2011, ACM, pp. 43–54.

Guissouma, H., Klare, H., Sax, E., Burger, E., 2018. An empirical study on the current and future challenges of automotive software release and configuration management. In: 44th Euromicro Conference on Software Engineering and Advanced Applications. SEAA 2018, IEEE, pp. 298–305.

Gyimóthy, T., Ferenc, R., Siket, I., 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering 31 (10), 897–910.

Hebig, R., Khelladi, D.E., Bendraou, R., 2017. Approaches to co-evolution of metamodels and models: A survey. IEEE Transactions on Software Engineering 43 (5), 396–414.

Heidenreich, F., Johannes, J., Seifert, M., Wende, C., 2010. Closing the gap between modelling and Java. In: Software Language Engineering. In: Lecture Notes in Computer Science, vol. 5969, Springer Berlin Heidelberg, pp. 374–383.

Heinrich, R., Rostami, K., Reussner, R., 2016. The CoCoME Platform for Collaborative Empirical Research on Information System Evolution. Karlsruhe Reports in Informatics 2016,2, Karlsruhe Institute of Technology.

Heinzemann, C., Becker, S., 2013. Executing reconfigurations in hierarchical component architectures. In: Proceedings of the 16th International ACM SigSoft Symposium on Component-Based Software Engineering. CBSE 2013, ACM, pp. 3–12.

Herold, S., Klus, H., Welsch, Y., Deiters, C., Rausch, A., Reussner, R., Krogmann, K., Koziolek, H., Mirandola, R., Hummel, B., Meisinger, M., Pfaller, C., 2008. CoCoME - the common component modeling example. In: The Common Component Modeling Example. In: Lecture Notes in Computer Science, vol. 5153, Springer Berlin Heidelberg, pp. 16–53.

Herrmannsdoerfer, M., Vermolen, S.D., Wachsmuth, G., 2011. An extensive catalog of operators for the coupled evolution of metamodels and models. In: Software Language Engineering. Springer Berlin Heidelberg, pp. 163–182.

Hettel, T., Lawley, M., Raymond, K., 2008. Model synchronisation: Definitions for round-trip engineering. In: Theory and Practice of Model Transformations. In: Lecture Notes in Computer Science, vol. 5063, Springer Berlin Heidelberg, pp. 31–45.

Hidaka, S., Tisi, M., Cabot, J., Hu, Z., 2016. Feature-based classification of bidirectional transformation approaches. Software & Systems Modeling 15 (3), 907–928.

Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schäfer, W., Lauder, M., Anjorin, A., Schürr, A., 2013. A survey of triple graph grammar tools. Electronic Communications of the EASST 57, Bidirectional Transformations 2013.

Hinkel, G., Kramer, M., Burger, E., Strittmatter, M., Happe, L., 2016. An empirical study on the perception of metamodel quality. In: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. SCiTePress, pp. 145–152.

International Organization for Standardization, 2011. Systems and Software Engineering – Architecture Description (ISO/IEC/IEEE 42010:2011(E)).

International Organization for Standardization, 2014. Information Technology – Object Management Group Meta Object Facility (MOF) Core (ISO/IEC 19508:2014(E)).

ITEA, 2020. AMALTHEA4public – An open platform project for embedded multi-core systems. URL: http://www.amalthea-project.org/. (accessed 7 September 2020).

Jakob, J., Königs, A., Schürr, A., 2006. Non-materialized model view specification with triple graph grammars. In: Proceedings of the 3rd International Conference on Graph Transformations. ICGT 2006, Springer Berlin Heidelberg, pp. 321–335.

Jakob, J., Schürr, A., 2008. View creation of meta models by using modified triple graph grammars. Electronic Notes in Theoretical Computer Science 211, 181–190.

Jouault, F., Kurtev, I., 2006. Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Springer Berlin Heidelberg, pp. 128–138.

Kehrer, T., Kelter, U., Taentzer, G., 2013. Consistency-preserving edit scripts in model versioning. In: 28th IEEE/ACM International Conference on Automated Software Engineering. ASE 2013, IEEE, pp. 191–201.

Klare, H., 2016. Designing a Change-Driven Language for Model Consistency Repair Routines Klare, Heiko. (Master's thesis). Karlsruhe Institute of Technology (KIT).

Klare, H., 2018. Multi-model consistency preservation. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2018. ACM, pp. 156–161.

Klare, H., 2020. Reproduction package for evaluating the Vitruvius approach. http://dx.doi.org/10.5445/IR/1000123568.

Klare, H., Gleitze, J., 2019. Commonalities for preserving consistency of multiple models. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 371–378.

Klare, H., Syma, T., Burger, E., Reussner, R., 2019. A categorization of interoperability issues in networks of transformations. Journal of Object Technology 18 (3), 4:1–20, The 12th International Conference on Model Transformations.

Koegel, M., Helming, J., 2010a. Acm/ieee 32nd international conference on software engineering. ICSE 2010, ACM, pp. 307–308.

Koegel, M., Helming, J., Seyboth, S., 2009. Operation-based conflict detection and resolution. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. CVSM '09, IEEE Computer Society, pp. 43–48.

Koegel, M., Herrmannsdoerfer, M., Li, Y., Helming, J., David, J., 2010b. Comparing state- and operation-based change tracking on models. In: 14th IEEE International Enterprise Distributed Object Computing Conference. EDOC 2010, pp. 163–172.

Koegel, M., Naughton, H., Helming, J., Herrmannsdörfer, M., 2010c. Collaborative model merging. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications. SPLASH 2010, ACM, pp. 27–34.

Kolovos, D.S., Rose, L.M., Matragkas, N.D., Paige, R.F., Polack, F.A., Fernandes, K.J., 2010a. Constructing and navigating non-invasive model decorations. In: Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations. ICMT 2010, Springer Berlin Heidelberg, pp. 138–152.

Kolovos, D., Rose, L., Paige, R., Garcia-Dominguez, A., 2010b. The Epsilon Book. Eclipse.

Koziolek, A., 2011. Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes. (Ph.D. thesis). KIT Scientific Publishing, Karlsruhe, Germany.

Koziolek, H., Becker, S., Happe, J., 2007. Predicting the performance of component-based software architectures with different usage profiles. In: Proceedings of the 3rd International Conference on the Quality of Software Architectures. QoSA'07, In: Lecture Notes in Computer Science, vol. 4880, Springer Berlin Heidelberg, pp. 145–163.

Koziolek, H., Schlich, B., Bilich, C., 2010. A large-scale industrial case study on architecture-based software reliability analysis. In: IEEE 21st International Symposium on Software Reliability Engineering. IEEE, pp. 279–288.

Kramer, M.E., 2017. Specification Languages for Preserving Consistency Between Models of Different Languages. (Ph.D. thesis). KIT Scientific Publishing, Karlsruhe, Germany.

Kramer, M.E., Hinkel, G., Klare, H., Langhammer, M., Burger, E., 2016. A controlled experiment template for evaluating the understandability of model transformation languages. In: Proceedings of the Second International Workshop on Human Factors in Modeling Co-Located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. MODELS 2016, vol. 1805, CEUR Workshop Proceedings, pp. 11–18.

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. CBSE 2015, ACM, pp. 21–26.

Kramer, M.E., Rakhman, K., 2016. Automated inversion of attribute mappings in bidirectional model transformations. In: Proceedings of the 5th International Workshop on Bidirectional Transformations. Bx 2016, vol. 1571, CEUR Workshop Proceedings, pp. 61–76.

Krogmann, K., Reussner, R.H., 2008. The Common Component Modeling Example. In: Lecture Notes in Computer Science, vol. 5153, Springer Berlin Heidelberg, pp. 297–326, (Chapter Palladio: Prediction of Performance Properties).

Kusel, A., Etzlstorfer, J., Kapsammer, E., Langer, P., Retschitzegger, W., Schoenboeck, J., Schwinger, W., Wimmer, M., 2013. A survey on incremental model transformation approaches. In: ME 2013 – Models and Evolution Workshop Proceedings. vol. 1090, CEUR Workshop Proceedings, pp. 4–13.

Langhammer, M., 2013. Co-evolution of component-based architecture-model and object-oriented source code. In: Proceedings of the 18th international doctoral symposium on components and architecture. ACM, pp. 37–42.

Langhammer, M., 2017. Automated Coevolution of Source Code and Software Architecture Models. (Ph.D. thesis). KIT Scientific Publishing, Karlsruhe, Germany.

Langhammer, M., Shahbazian, A., Medvidovic, N., Reussner, R.H., 2016. Automated extraction of rich software models from limited system information. In: 13th Working IEEE/IFIP Conference on Software Architecture. WICSA 2016, IEEE, pp. 99–108.

Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., Greenyer, J., 2014. A comparison of incremental triple graph grammar tools. Electronic Communications of the EASST 67, Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques.

Macedo, N., Guimaraes, T., Cunha, A., 2013. Model repair and transformation with echo. In: 28th IEEE/ACM International Conference on Automated Software Engineering. ASE 2013, IEEE, pp. 694–697.

Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.A., 2010. Providing architectural languages and tools interoperability through model transformation technologies. IEEE Transactions on Software Engineering 36 (1), 119–140.

Mazkatli, M., 2016. Consistency Preservation in the Development Process of Automotive Software. (Master's thesis). Karlsruhe Institute of Technology (KIT).

Mazkatli, M., Burger, E., Koziolek, A., Reussner, R.H., 2017. Automotive systems modelling with vitruvius. In: 15. Workshop Automotive Software Engineering. In: Lecture Notes in Informatics, vol. P-275, Gesellschaft für Informatik e.V. (GI), pp. 1487–1498.

Meier, J., Klare, H., Tunjic, C., Atkinson, C., Burger, E., Reussner, R., Winter, A., 2019. Single underlying models for projectional, multi-view environments. In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development. MODELSWARD 2019, SCiTePress, pp. 119–130.

Meier, J., Werner, C., Klare, H., Tunjic, C., Aßmann, U., Atkinson, C., Burger, E., Reussner, R., Winter, A., 2020. Classifying approaches for constructing single underlying models. In: Model-Driven Engineering and Software Development. Springer International Publishing, pp. 350–375.

Murer, S., Bonati, B., Furrer, F.J., 2011. Managed Evolution – A Strategy for Very Large Information Systems. Springer Berlin Heidelberg.

Nuseibeh, B., Easterbrook, S.M., Russo, A., 2001. Making inconsistency respectable in software development. Journal of Systems and Software 58 (2), 171–180.

Object Management Group, 2014. Object Constraint Language – Version 2.4. http://www.omg.org/spec/OCL/2.4/PDF.

Object Management Group, 2016. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification – Version 1.3. http://www.omg.org/spec/QVT/1.3/PDF.

Object Management Group, 2017. OMG Unified Modeling Language (OMG UML) – Version 2.5.1. http://www.omg.org/spec/UML/2.5.1/PDF.

Object Management Group, 2019. OMG System Modeling Language (OMG SysML) – Version 1.6. https://www.omg.org/spec/SysML/1.6/PDF.

Paton, N.W., Díaz, O., 1999. Active database systems. ACM Computing Surveys 31 (1), 63–103.

Petersen, F., 2016. Extending an Architecture and Code Co-Evolution Approach to Support Existing Software Projects. (Master's thesis). Karlsruhe Institute of Technology (KIT).

Ptolemaeus, C. (Ed.), 2014. System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org.

Reussner, R., Becker, S., Burger, E., Happe, J., Hauck, M., Koziolek, A., Koziolek, H., Krogmann, K., Kuperberg, M., 2011. The Palladio Component Model. Karlsruhe Reports in Informatics 2011,14, Karlsruhe Institute of Technology.

Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K., 2016. Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press, p. 408.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., 1991. Objekt-oriented Modeling and Design. Prentice Hall.

Runeson, P., Höst, M., 2008. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14 (2), 131.

Salay, R., Kokaly, S., Di Sandro, A., Chechik, M., 2015. Enriching megamodel management with collection-based operators. In: ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems. MODELS 2015, IEEE, pp. 236–245.

Salay, R., Mylopoulos, J., Easterbrook, S., 2008. Managing models through macromodeling. In: 23rd IEEE/ACM International Conference on Automated Software Engineering. ASE 2008, IEEE, pp. 447–450.

Salay, R., Mylopoulos, J., Easterbrook, S., 2009. Using macromodels to manage collections of related models. In: Advanced Information Systems Engineering. In: Lecture Notes in Computer Science, vol. 5565, Springer Berlin Heidelberg, pp. 141–155.

Salay, R., Wang, S., Suen, V., 2012. Managing related models in vehicle control software development. In: 15th International Conference on Model Driven Engineering Languages and Systems. MODELS 2012, Springer Berlin Heidelberg, pp. 383–398.

Schürr, A., 1995. Specification of graph translators with triple graph grammars. In: Graph-Theoretic Concepts in Computer Science. In: Lecture Notes in Computer Science, vol. 903, Springer Berlin Heidelberg, pp. 151–163.

Seidl, C., 2015. Integrated Management of Variability in Space and Time in Software Families. (Ph.D. thesis). Technische Universität Dresden.

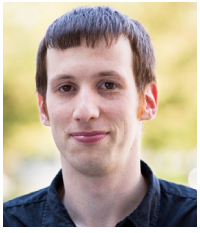
Seidl, C., Schaefer, I., Aßmann, U., 2014. DeltaEcore – a model-based delta language generation framework. In: Modellierung 2014. Gesellschaft für Informatik e.V. (GI), pp. 81–96.

Semeráth, O., Debreceni, C., Horváth, Á., Varró, D., 2016. Incremental backward change propagation of view models by logic solvers. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. MODELS 2016, ACM, pp. 306–316.

Software Design and Quality, 2020. Automotive Software Engineering Metamodel (GitHub). URL: https://github.com/kit-sdq/ASEM. (accessed 7 September 2020).
Son, H.S., Kim, W.Y., Kim, R.Y.C., Min, H.-G., 2012. Metamodel design for model transformation from Simulink to ECML in cyber physical systems. In: Computer Applications for Graphics, Grid Computing, and Industrial Environment. In: Communications in Computer and Information Science, vol. 351, Springer Berlin Heidelberg, pp. 56–60.
Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2008. EMF: Eclipse Modeling Framework, second revised ed. Eclipse Series, Addison-Wesley Longman.
Stevens, P., 2010. Bidirectional model transformations in QVT: semantic issues and open questions. Software & Systems Modeling 9 (1), 7–20.
Stevens, P., 2018. Is bidirectionality important?. In: Modelling Foundations and Applications. Springer International Publishing, pp. 1–11.
Stevens, P., 2020. Maintaining consistency in networks of models: bidirectional transformations in the large. Software & Systems Modeling 19 (1), 39–65.
Strittmatter, M., Kechaou, A., 2016. The Media Store 3 Case Study System. Karlsruhe Reports in Informatics 2016,1, Karlsruhe Institute of Technology.
Stünkel, P., König, H., Lamo, Y., Rutle, A., 2018. Multimodel correspondence through inter-model constraints. In: Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming. Programming 2018, ACM, pp. 9–17.
Sztipanovits, J., Bapty, T., Neema, S., Howard, L., Jackson, E., 2014. OpenMETA: A model- and component-based design tool chain for cyber-physical systems. In: From Programs to Systems. The Systems Perspective in Computing: ETAPS Workshop. FPS 2014, Springer Berlin Heidelberg, pp. 235–248.
Tunjic, C., Atkinson, C., 2015. Synchronization of projective views on a single-underlying-model. In: Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering. MORSE/VAO '15, ACM, pp. 55–58.
Ujhelyi, Z., Bergmann, G., Hegedús, A., Horváth, A., Izsó, B., Ráth, I., Szatmári, Z., Varró, D., 2015. EMF-IncQuery: An integrated development environment for live model queries. Science of Computer Programming 98, 80–99.
Vangheluwe, H., de Lara, J., 2003. Computer automated multi-paradigm modelling: meta-modelling and graph transformation. In: Proceedings of the 2003 Winter Simulation Conference. pp. 595–603 Vol.1.
Vangheluwe, H., De Lara, J., Mosterman, P.J., 2002. An introduction to multi-paradigm modelling and simulation. In: Proceedings of the AIS 2002 conference. pp. 9–20.
Vitruvius GitHub, 2020a. Wiki. URL: http://vitruv.tools. (accessed 7 September 2020).
Vitruvius GitHub, 2020b. Organization. URL: https://github.com/vitruv-tools. (accessed 7 September 2020).
Vitruvius GitHub, 2020c. Framework. URL: https://github.com/vitruv-tools/Vitruv. (accessed 7 September 2020).
Vitruvius GitHub, 2020d. Component-based systems case study. URL: https://github.com/vitruv-tools/Vitruv-Applications-ComponentBasedSystems. (accessed 7 September 2020).
Vitruvius GitHub, 2020e. Automotive software systems case study. URL: https://github.com/vitruv-tools/Vitruv-Applications-AutomotiveSoftwareSystems. (accessed 7 September 2020).
Vitruvius GitHub, 2020f. RIS And LIS integration strategies. URL: https://github.com/vitruv-tools/Vitruv-Applications-PCMJavaAdditionals. (accessed 7 September 2020).
Werle, D., 2016. A Declarative Language for Bidirectional Model Consistency. (Master's thesis). Karlsruhe Institute of Technology.
Wimmer, M., Moreno, N., Vallecillo, A., 2012. Viewpoint co-evolution through coarse-grained changes and coupled transformations. In: Objects, Models, Components, Patterns. In: Lecture Notes in Computer Science, vol. 7304, Springer Berlin Heidelberg, pp. 336–352.
Wood-Harper, A., Antill, L., Avison, D., 1985. Information Systems Definition: The Multiview Approach. Computer Science Texts, Blackwell Scientific.
Wu, X., Woodside, M., 2004. Performance modeling from software components. SIGSOFT Software Engineering Notes 29 (1), 290–301.
Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H., 2009. Supporting automatic model inconsistency fixing. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ESEC/FSE 2009, ACM, pp. 315–324.
Yohannis, A., Rodriguez, H.H., Polack, F., Kolovos, D., 2018. Towards efficient loading of change-based models. In: Modelling Foundations and Applications. Springer International Publishing, pp. 235–250.
Yu, P., Systä, T., Müller, H.A., 2002. Predicting fault-proneness using OO metrics: An industrial case study. In: Proceedings of the 6th European Conference on Software Maintenance and Reengineering. CSMR 2002, IEEE Computer Society, pp. 99–107.

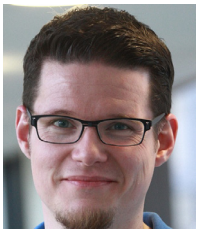
Heiko Klare is a doctoral researcher at the chair for Software Design and Quality (SDQ) at Karlsruhe Institute of Technology (KIT) since 2016. His research interests involve consistency preservation of heterogeneous models, consistency-aware collaborative modeling, as well as view-based modeling and development processes. He researches techniques for coupling independently developed bidirectional model transformations and originally developed the *Reactions language* for model consistency preservation.

Max Kramer is a former doctoral researcher at the chair for Software Design and Quality (SDQ) at Karlsruhe Institute of Technology (KIT). He received his Ph.D. in 2017. His research interests involve model consistency preservation and view-based modeling. He developed a family of languages for specifying consistency of models on different levels of abstraction, including the *Reactions* and *Mappings language*.



Michael Langhammer is a former doctoral researcher at the chair for Software Design and Quality (SDQ) at Karlsruhe Institute of Technology (KIT). He received his Ph.D. in 2017. His research interests involve architecture-based performance prediction, architecture reverse engineering and model consistency preservation, especially between code and architecture description. He developed and evaluated a set of rules for preserving consistency between software architectures and their representation in code.

Dominik Werle is a doctoral researcher at the chair for Architecture-driven Requirements Engineering (ARE) at Karlsruhe Institute of Technology (KIT) since 2016. His research interests involve model-based quality prediction for software systems with a focus on the performance of data-intensive applications. He originally developed the *Mappings language* for model consistency preservation.



Erik Burger is a postdoctoral researcher and head of the model-driven development group at the chair for Software Design and Quality (SDQ) at Karlsruhe Institute of Technology (KIT). He received his Ph.D. in 2014. His research interests are view-based development, metamodel evolution and model co-evolution, as well as distributed development. He developed the *ModelJoin* approach for creating views on heterogeneous models.

Ralf Reussner is a computer science professor at Karlsruhe Institute of Technology (KIT). He holds the chair for Software Design and Quality (SDQ) since 2006 and heads the Institute for Program Structures and Data Organization, which develops the VITRUVIUSapproach. His research group works in the interplay of software architecture and predictable software quality as well as on view-based design methods for software-intensive technical systems.