Variants and Versions Management for Models with Integrated Consistency Preservation

Sofia Ananieva
FZI Research Center for Information Technology
Karlsruhe, Germany
ananieva@fzi.de

Heiko Klare
Karlsruhe Institute of Technology
Karlsruhe, Germany
heiko.klare@kit.edu

Erik Burger
Karlsruhe Institute of Technology
Karlsruhe, Germany
burger@kit.edu

Ralf Reussner
Karlsruhe Institute of Technology
Karlsruhe, Germany
reussner@kit.edu

ABSTRACT

Modern software systems are often developed and maintained by describing them in several modeling and programming languages. To reduce complexity and improve understandability of such systems, models represent specific *views* on the system. These views have semantic interrelations (e.g., by sharing common or dependent information) that need to be kept consistent during evolution of the system. Apart from that, modern systems need to run in many different contexts and be highly configurable to satisfy the demand for fully customizable products. Such *variable* systems often comprise various dependencies from which inconsistencies may arise. Combining solutions for consistency management with variants and versions management, however, comes with many challenges.

In this research-in-progress paper, we introduce the `VaVe` approach which makes variants and versions management aware of automated consistency preservation in the context of multi-view modeling. We explain core features of the approach and reason about its benefits and limitations.

CCS CONCEPTS

• **Software and its engineering** → **Software configuration management and version control systems**; **Reusability**; **Software product lines**;

KEYWORDS

Software Product Lines, Variability Management, Delta-Based Consistency Preservation

ACM Reference Format:
Sofia Ananieva, Heiko Klare, Erik Burger, and Ralf Reussner. 2018. Variants and Versions Management for Models with Integrated Consistency Preservation. In *VAMOS 2018: 12th International Workshop on Variability Modelling of Software-Intensive Systems, February 7–9, 2018, Madrid, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3168365.3168377

VAMOS 2018, February 7–9, 2018, Madrid, Spain

ACM ISBN 978-1-4503-5398-4/18/02…$15.00
https://doi.org/10.1145/3168365.3168377

1 INTRODUCTION

Coping with the high complexity of today's software systems often involves several models describing different views on the entire system. This technique is commonly known as *Multi-View Modeling* [4]. In the scope of *Model-Driven Software Development (MDSD)*, models do not only serve documentation and communication purposes but are also treated as first-class artifacts to generate executable code from. Different models presenting views on a software system may contain redundant or dependent information, which consequently has to be kept consistent. For example, deleting an element in a specific model has to propagate to other dependent models and result in the deletion of the respective element to preserve consistency. Various approaches concentrate on automated consistency management, i.e., model consistency specification [16], validation [3, 21] and repair [10, 30].

Consistency is an important property for systems with *variability*. Variability of a system reflects its ability to be configured (or customized) according to different customer requirements. The result of a configuration denotes a particular *product* of a variable system. This product can be represented through different views. *Software Product Line Engineering (SPLE)* has a long tradition in managing variability of a software system by making the variable and common parts of a family of software products explicit [20]. Variability in SPLE can be distinguished into *variability in space* and *variability in time*. The former denotes the configurable functionality of an SPL represented by *variants*. The latter denotes the evolution of an SPL due to new requirements or bug fixes represented by *versions* whereby versions are generally related to the common or variable parts of an SPL.

Dependencies between variants and versions of a variable system lead to increased complexity from which inconsistencies may arise. The detection and repair of inconsistencies among model elements is extensively researched, while consistency management for variable systems is less addressed [19]. However, it is essential for variants and versions management to take consistency constraints into account in order to avoid inconsistencies. Such inconsistencies manifest, for example, in a configured system with incompatible versions of variable system parts.

To enable consistency preservation through change propagation, there are two basic approaches to describe changes: The *delta-based* and the *state-based* approach. The *delta-based* approach uses atomic

edit operations to describe changes performed in a model. In a *state-based* approach, two versions of a model are compared to determine their differences. In the latter case, consistency preservation is less challenging since prevailing consistency constraints can be checked for arbitrary model versions. However, change information cannot be derived unambiguously. For example, renaming an element is indistinguishable from deleting and creating a new element with that name, whereas delta-based approaches recognize the renaming operation and process this information correctly: a dedicated recorder mechanism monitors sequences of changes so they can be propagated appropriately to dependent models in order to restore consistency. To the best of our knowledge, there is no approach combining delta-based consistency preservation with multi-view modeling and SPLE regarding versions and variants management.

In this research-in-progress paper, we introduce a variability management approach called `VaVe` which combines management of variability in space and time with consistency preservation. The approach supports configurations containing variable assets in selected versions, and utilizes delta-based consistency preservation mechanisms to keep variants and versions consistent. In summary, we make the following contributions:

(1) We present a non-invasive approach for managing variants and versions that can be used with arbitrary EMOF-based metamodels.
(2) We utilize a delta-based consistency preserving mechanism to keep views on a product consistent.

The paper is structured as follows: First, we introduce a motivating example and provide basic knowledge about the view-based consistency preserving framework Vitruvius. In Section 3, we describe core concepts of the proposed approach and discuss its beneficial properties and limitations in Section 4. Subsequently, we outline related work in Section 5 and conclude the paper with an overview on future work in Section 6.

2 RUNNING EXAMPLE & BACKGROUND

In this chapter, we provide basic definitions and introduce a running example. Furthermore, we describe main concepts of the consistency preserving framework Vitruvius.

2.1 Variants and Versions in SPLE

As we mentioned at the beginning, variability in SPLE concerns two dimensions: *variability in space* and *variability in time* [20, 28]. Variability in space deals with the configurable functionality of an SPL. Variants represent concrete configuration decisions that lead to different co-existing products of an SPL. Variability in time considers the evolutional functionality of an SPL. Versions categorize the unique state of an SPL development artifact at a particular point in time of development. Hence, a configuration consists of a set of selected variants in particular versions. A valid configuration conforms to constraints imposed by the topology of a *variability model* [20] and by explicitly specified constraints between different versions. The result of a configuration is a product of the SPL.

As described by Seidl et al. [28], a separated consideration of variability in space and time is not always possible, because new versions of variable system parts may introduce additional dependencies or incompatibilities. Therefore, both variability in space and variability in time must be considered jointly.

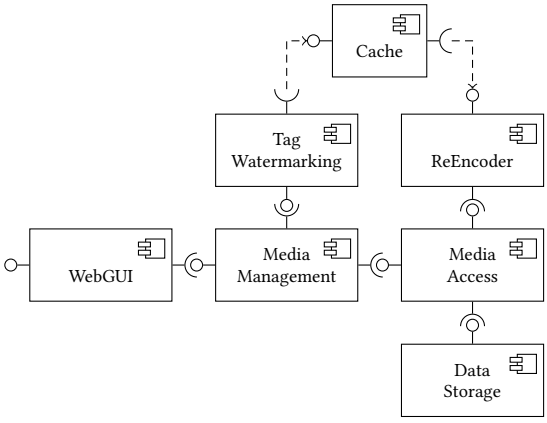

Figure 1: UML component diagram of a simplified version of the Media Store (based on [23])

2.2 Media Store Example

The Media Store system has been developed to serve as a case study for the *Palladio Component Model* (PCM). Both the PCM and the Media Store case study are published in [23]. The PCM is a metamodel that can be used to develop a component-based architecture description of a software system. It allows to simulate the system behavior and to predict its quality attributes, e.g., performance properties such as response times, at design time. The essential elements of a PCM model are *components*, which encapsulate certain functionality, and *interfaces*, which specify signatures of services and which can be provided and required by components. Those components can be assembled to a system, which defines the used components and their connections. Such a structural description of a PCM system model is comparable to a UML component diagram [22, p. 184].

The Media Store is a PCM model that describes a web-based file hosting system for audio files. Figure 1 presents the component-based architecture of a simplified version of the Media Store depicting the components of the system and their required (sockets) and provided (lollipops) interfaces. The *WebGUI* component represents the frontend of the Media Store. The *Media Management* contains business logic, e.g., to upload and download audio files. The *Tag Watermarking* component adds a comment tag to a downloaded audio file and is a rather simple but insecure way to realize a digital watermark. The *ReEncoder* component reencodes an audio file after retrieving it from the storage upon download. Often accessed files are stored in a *Cache* to reduce overhead for reencoding them repeatedly. The data access is realized by the components *Data Storage* and the *MediaAccess*.

Due to the component-based architecture, the Media Store system allows for different design alternatives by selecting individual components in the final system, for example, removing the *Cache* component or replacing the *Tag Watermarking* with an *Audio Watermarking* component, which provides a more secure watermark.

To realize a design alternative, the respective components need to be integrated manually into the system; a variability mechanism to select desired components of the Media Store and, accordingly, derive a particular product is not available.

The variety of design alternatives of the Media Store can be represented with a *feature diagram* as illustrated in Figure 2. A feature diagram is a graphical tree-like representation of a *feature model*. Such models are commonly used to define the features of an SPL [12]. A feature model consists of a hierarchically arranged set of features, each representing a user-visible aspect or quality of a software system. The selection of desired features (e.g., a configuration) is used to derive an SPL product. Feature models capture dependencies between features in two ways. First, according to the relationships between parent and child features which may be of different categories:

(1) Mandatory: a child feature always appears together with its parent feature in a configuration.
(2) Optional: a child feature may or may not appear together with its parent feature in a configuration.
(3) Alternative: exactly one child feature of a parent feature must appear in a configuration.
(4) Or: at least one child feature of a parent feature must appear in a configuration.

Second, cross-tree constraints (boolean expressions) across features of the feature tree govern the validity of a configuration and are usually located below the feature tree.

Figure 2 depicts a feature diagram of the Media Store. A configuration of a Media Store must contain a *Watermarking* feature. One can choose between the *Tag* or the *Audio* watermarking. The selection of the *Tag* watermarking automatically selects the *ReEncoder* in the current configuration. The selection of *Audio* watermarking feature leads to a deselection of the *ReEncoder* feature (since this functionality is already comprised by the *Audio* watermarking feature). In addition, one may select the *Cache* feature.

After describing the running example according to the Media Store case study and foundations of SPLE, we will next concentrate on the Vitruvius framework and its mechanisms of automated consistency preservation in multi-view modeling.

2.3 Vitruvius

Vitruvius is an approach for view-based development of software-intensive systems [14]. It is based on the idea of the *Orthographic Software Modeling (OSM)* approach by Atkinson, Stoll, and Bostan [1] for developing a software system using a *single underlying model (SUM)*. The SUM combines all information about the software systems within one model, for example about requirements, architecture description and implementation, as well as their relations. The goal of using such a SUM instead of different models to represent a system is to avoid redundant representations of the same information in order to prevent inconsistencies. A user, for example a software developer, can then modify and evolve the system using views representing only an extract of the SUM concerning a specific aspect, for example the implementation, of the SUM.

A SUM instantiates an appropriate metamodel, the so called *single underlying metamodel (SUMM)*, which has to be explicitly defined before developing the system. Since it is hard to define

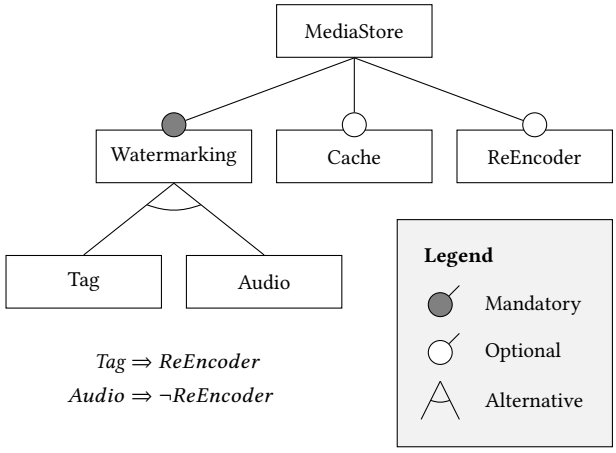

Figure 2: Feature diagram of the Media Store

and evolve a metamodel that can describe any aspect of a software system redundancy-free and, additionally, results in losing all tool support for the existing metamodels, the Vitruvius approach replaces the SUMM with a virtual one, the V-SUMM. A V-SUMM consists of multiple existing metamodels, for example the PCM, a Java metamodel, etc., and consistency preservation specifications that define how dependencies between instances of those metamodels are kept consistent. This makes a V-SUMM behave like a redundancy-free SUMM, but as existing metamodels are integrated, they can evolve independent from the SUMM and existing tool support for them can be obtained. A V-SUMM instance is a V-SUM and consists of models that instantiate the models of the V-SUMM.

The consistency preservation mechanisms used to keep the models of a V-SUM consistent rely on a *delta-based* paradigm and an explicit definition of consistency relations between the metamodels. Assuming that all models in a V-SUM are in a consistent state, modifications are recorded by monitoring the views and used for restoring consistency to other models. Such modifications are represented through instances of a *change descriptions metamodel*, which defines *atomic change types* that concern only one value of one model element. Atomic changes can be composed to *compound changes*. For the specification of consistency relations, two languages are provided: imperative rules for repairing consistency after a change can be specified in the Reactions language [13], declarative specifications of metamodel relations can be specified in the Mappings language [15]. The languages allow to retrieve and update a model of *correspondences*, which stores the elements that contain dependent information that has to be kept consistent.

The consistency preservation process can usually not be fully automated. Often user decisions are required to determine how consistency should be enforced if there are several valid options. This is especially the case if a model with a low level of abstraction is modified and has to be kept consistent with a more abstract model. For example, when adding a class in the implementation it is unclear whether it shall represent a component on the architectural level or only an implementation class.

The delta-based paradigm used in Vitruvius makes this approach only applicable in a greenfield scenario starting with empty

models. Otherwise, the models are not initially consistent, as consistency can only be guaranteed if changes are tracked and respective consistency preservation specifications are executed. Vitruvius is based on the Eclipse Modeling Framework (EMF) [29]. EMF provides a metametamodel called *Ecore* as a formalism for all metamodels supported by tools based on EMF. Ecore is compatible with EMOF [11] in the sense that instances of both can be converted into instances of the other without information loss. In consequence, all Ecore-based or EMOF-based metamodels can be used within Vitruvius and its V-SUMM mechanism.

A currently available case study in the Vitruvius framework targets the consistency preservation between instances of the PCM metamodel (i.e., the Media Store) and the Java Model Parser and Printer (JaMoPP) metamodel [9]. The JaMoPP metamodel conforms to the language specification of Java. Consistency relations between the metamodels are specified through the Reactions language. For example, components of the PCM model are mapped to a package and a facade class in the Java code while an interface within a PCM model is mapped in a straightforward manner to a Java interface in a specific package. This case study complements the running example which will be used throughout this paper.

Listing 1 represents a simplified consistency rule that creates a Java interface whenever a PCM interface is introduced as follows:

```
1  reaction CreatedInterface {
2    after element inserted in pcm::Repository[interfaces]
3    call createInterface(newValue)
4  }
5  routine createInterface(pcm::Interface pcmInterf) {
6    match {
7      val contractsPackage = retrieve java::Package
8         corresponding to pcmInterf.repository
9         tagged with "contracts"
10   }
11   action {
12     val javaInterf = create java::Interface and initialize {
13          javaInterf.namespace = contracsPackage.namespace;
14          javaInterf.name = pcmInterf.entityName;
15          javaInterf.addModifier(
16             ModifiersFactory.eINSTANCE.createPublic());
17     }
18     add correspondence between javaInterf and pcmInterf
19   }
20 }
```

Listing 1: Reaction to the creation of a PCM interface.

3 APPROACH

In this chapter, we explain and illustrate the concepts of the variability management approach and its integration with delta-based consistency management of Vitruvius.

3.1 Delta-Based Variability Management

Variability management approaches either extend a metamodel with variability constructs resulting in a new dedicated metamodel (generally defined as the *amalgamated* approach) or the connection with variability constructs is made across model elements leaving the original metamodel unchanged (the *separated* approach) [6]. One particular advantage of the latter approach is that invasive changes of a metamodel are not required, making it possible to apply variability management for arbitrary EMOF-based metamodels. For this reason, we follow the separated approach to manage variability in space and time within the VaVe approach being presented in this paper. VaVe is an extension to the Vitruvius framework and utilizes its delta-based consistency preserving mechanisms to keep views on a particular product consistent. The mechanisms of the VaVe approach are defined within the VaVe metamodel.

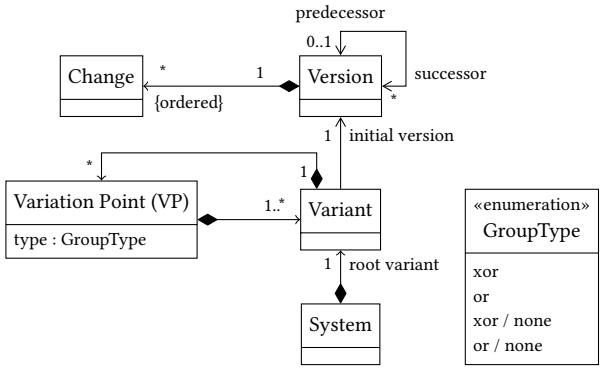

Figure 3: The metamodel of the VaVe approach

Figure 3 illustrates the metamodel of the VaVe approach. The *System* metaclass contains the *root variant* which represents the core features of the system that are not subject to variability in space. The *Variant* metaclass is a central part of the metamodel. It consists of variation points that, in turn, contain variants enabling a hierarchical structure between variation points and variants. Variation points capture variability in space by specifying locations for variation and listing all possible configuration options (i.e., variants) for that point. There are four different types of variation points similarly to feature categories in a feature model (see Subsection 2.2). The group type *xor* represents an alternative variant group and indicates that exactly one variant must be selected. The group type *or* requires the selection of at least one variant. The group type *xor / none* implies the selection of at most one variant. The group type *or / none* indicates the selection of none or at most all variants. The *Variant* metaclass has one initial version. A version is related to multiple successors and at most one predecessor. Furthermore, a version consists of an ordered sequence of changes that differ it from the predecessor: these sequential changes need to be performed to move from a previous version to the next one.

The VaVe model is an instance of the described VaVe metamodel. Figure 4 exemplifies the VaVe model of the Media Store. Within the VaVe model, we differentiate between two types of features, i.e., core features and variable features. Core features represent reusable assets commonly known in SPLE as the *integrated platform*. Such features may be versioned collectively. In the Media Store, core features comprise, for instance, a registration and a login feature. Variable features belong to a variation point and are versioned individually. In the Media Store, the variation point *Watermarking*

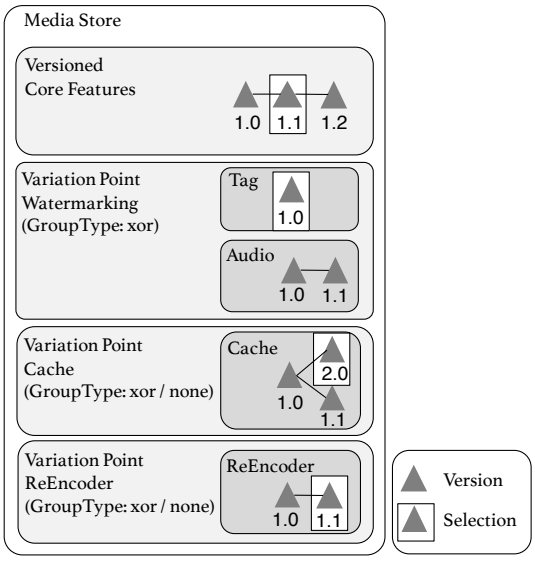

Figure 4: The VaVe model of the Media Store including an exemplary configuration

includes, for example, both variants *Tag* and *Audio* in specific versions. Logical relations govern dependencies between variants, for example, the mutual exclusiveness (GroupType: xor) between both variants *Tag* and *Audio*. To manage variability in time, constraints need to be version-aware to capture knowledge about dependencies between particular versions of variants. For example, the cache feature in version 1.1 depends on the core features in version 1.0. After the variability has been specified in the VaVe model, a configuration can be created. In Figure 4, a particular configuration of the Media Store is illustrated by white boxes. It involves core features of the Media Store in version 1.1, the tag feature in version 1.0, the cache feature in version 2.0 and the reencoder feature in version 1.1.

The derivation of a product according to a particular configuration involves the use of *delta modeling* [24]. Delta modeling realizes the delta-based paradigm of Vitruvius. The general idea behind delta modeling is to apply delta operations (add-, change- and remove-operations) to a product in order to transform it into another one. Thus, the product derivation according to a particular configuration requires to apply a sequence of respective delta operations on each particular variant to create the desired product. In Subsection 3.2, we explain how to switch between different products.

3.2 Integrating Consistency Preservation

So far, we have explained the VaVe approach and how to manage variability in space and time using the delta-based paradigm of Vitruvius. To summarize the consisteny preserving mechanisms of Vitruvius, models in the V-SUM are modified through specific views which, in turn, are monitored for the performed changes. Those changes are recorded and propagated to dependent models in order to restore consistency (see Subsection 2.3).

Automated consistency preservation in multi-view modeling provides favourable properties when dealing with product derivation or switching between different products. To explain how the consistency preserving mechanisms of Vitruvius are utilized within the VaVe approach, we answer three main questions while referring to the Media Store example for illustration:

(1) How is consistency during configuration ensured?
(2) How to switch between different products?
(3) How is the VaVe approach technically integrated within the Vitruvius framework?

(1) During the selection of a versioned variant, the following consistency checks are performed: On the one hand, explicit consistency constraints between versions or version ranges of variants must be met. For instance, in the Media Store, the tag variant in version 1.0 may require the reencoder variant in version 1.0 or greater. On the other hand, the following implicit consistency constraint given by the topology of the VaVe metamodel must be fulfilled: a selected version of a variant requires the version of the "parent" variant based on which it was created (up to the root variant representing the core features). For example, if the cache variant in version 2.0 was developed on the core features in version 1.2, then it requires core features in this version. Only if both checks are successful, consistency is ensured and selection of the versioned variant is allowed.

(2) A product is based on a particular configuration. For simplicity, we assume that this configuration is always consistent. As mentioned earlier, each version defines the sequence of delta operations that differ it from the predecessor version. Since a configuration refers to particular versions of core and variable features, the VaVe approach needs to determine the respective delta operations that are required to create the desired product. On the one hand, respective delta operations must be applied on each variant whose version has been changed. On the other hand, all variants that have been removed or added require delta operations to be applied on their container variants. For instance, in the Media Store example, switching to a product of the PCM model which involves the cache component in version 1.1 instead of version 2.0 requires to revert the respective delta operations that are associated with the cache variant in version 2.0 and apply all delta operations that are associated with it in version 1.1 (the recording mechanism of Vitruvius provides the possibility to roll back the recorded changes that define a specific version). The respective Java model of the Media Store is automatically updated due to the predefined consistency relations between the PCM metamodel and the Java metamodel (see Subsection 2.3). The consistency check should be postponed after all delta operation have been performed in order to tolerate intermediate inconsistent states that can occur meanwhile.

(3) Finally, we address the last question which deals with the technical integration of the VaVe approach into Vitruvius. On metamodel level, Vitruvius maintains metamodels that can be divided into two groups: the metamodels that represent views on the software system, for instance, Java, UML or PCM. Orthogonal to that, there are Vitruvius-specific metamodels like the *correspondences metamodel* and the *change descriptions metamodel* to manage the consistency preserving mechanisms (see Subsection 2.3). The VaVe metamodel belongs to the latter group.

4 DISCUSSION

So far, we have described main principles of the proposed VaVe approach and illustrated its integration into the Vitruvius framework. In this section, we discuss the introduced approach and reason about its beneficial properties and limitations.

4.1 Beneficial Properties

A particular benefit of the VaVe approach is its non-invasive mechanism of making a language variability-aware. Hence, the approach can be applied to arbitrary metamodels based on the *Ecore* metametamodel to enable variability mechanisms on the one hand and to preserve compatibility with existing tool-chains on the other hand. To achieve this benefit, we provide a separate definition of the variability mechanisms within the VaVe metamodel.

Considering variants and versions management for models with integrated consistency preservation, we propose two beneficial properties that we consider a novel contribution of the proposed approach.

(1) **Variability propagation.** Due to the consistency preservation between models used to describe the software system, changes to variability in space and time are automatically propagated appropriately into all models. This variability propagation can be viewed from two perspectives: from a static point of view, the derivation of a product according to a specific configuration is reflected within all dependent models without the need for the developer to be aware of any change impact. For example, a replacement of the watermarking component in our Media Store architecture is propagated to the Java code, automatically exchanging the used component there as well. From a dynamic point of view, new variability, e.g., an optional *Shop* component of the MediaStore to buy or rent audio files, is represented by a new variation point with the respective variant within the VaVe model. This implicitly propagates the variation point introduced in the PCM model to the Java code.
(2) **Automated constraint derivation.** The VaVe approach supports the explicit specification of constraints between versions or version ranges of variants. Additionally, an automated and implicit constraint derivation is performed from the operational tracking. For example, if the *Shop* variant in its first version was developed on the core assets in version 1.2, then it requires core assets in this version. Hence, a product with the selected *Shop* variant in its first version will automatically contain core features in version 1.2. The automated constraint derivation is a mechanism to ensure consistency between versions in an implicit way.

Along with the beneficial characteristics, the VaVe approach is subject to certain limitations which we present in the next subsection.

4.2 Limitations

Some may argue that the VaVe approach is not standalone but rather only applicable within the Vitruvius framework. While certain design decisions were made that are necessary for the integration in the Vitruvius framework, general concepts of the approach constitute an independent nature. This includes, for example, the delta-based variant derivation mechanism and the definition of variability within a dedicated variability model. Nevertheless, these aspects have been addressed in prior research [27]. Instead, little research has been devoted to the integration of variants and versions management with consistency preservation in multi-view systems. Therefore, concepts of the Vitruvius framework are vital for the contribution of the conducted research. A stronger limitation is that the delta-based paradigm used in Vitruvius makes this approach only applicable in a greenfield scenario starting with empty models. This assumption has been chosen to initially guarantee the consistency between the used models. For future work, this limitation needs to be addressed to ease adoption and to enable migration from other systems. Furthermore, no formalization of user decisions can currently be reapplied when switching between products.

5 RELATED WORK

The following related work has been performed in the research area of managing variability in multi-view modeling: Gomaa and Shin [5] propose a metamodeling approach for managing variability and consistency checking between UML model views of an SPL. The proposed metamodel describes the metaclasses in each view, their attributes and how each view relates semantically to other views. Similar to our work, Gomaa and Shin consider consistency checking between multiple views at metamodel level. Contrary, the authors propose a single metamodel containing the definition for all predefined views whereas our approach involves a V-SUMM that can be flexibly extended by further views on the system. As another difference, Gomaa and Shin propose a metamodel that combines definitions for both UML and variability constructs. Pohl et al. [20] propose an *Orthogonal Variability Model (OVM)* to explicitly model the variability of the SPL in one central model enabling a consistent view on variability. Variation points and variants in the OVM are linked to respective variable elements in different development artifacts like use cases, components and test models. Although this approach is similar to ours, an OVM only contains variability definitions (i.e., variation points and variants) apart from core features. As another difference, the VaVe metamodel captures variation points and variants in a hierarchical way supporting implicit consistency constraints between versions given the topology of the VaVe metamodel (see Subsection 3.2). Conducted research by Lopez-Herrejon et al. [19] closely relates to our work dealing with detection and repair of inconsistencies in models with variability (a model with variability contains elements that are part of a feature, modularized in so-called *feature modules*). The authors investigate whether all possible feature combinations of an SPL are consistent through applying *safe composition* (i.e., a guarantee that all programs, which are derived from an SPL feature model, are type safe). In contrast, the presented approach by Lopez-Herrejon et al. determines where fixes for detected inconsistencies should be placed, whereas we concentrate on automated consistency preserving mechanisms to keep views on a particular product consistent.

In the field of *Software Configuration Management (SCM)*, Conradi and Westfechtel [2] provide a fundamental survey about configuration management approaches. Within the delta-based versioning approach, the authors distinguish between extensional versioning (i.e., a version can be retrieved based on a unique number; changes

applied to the retrieved version lead to the creation of a new one) and intensional versioning (i.e., a consistent version set is automatically constructed based on the properties of specific versions given some query). With this regard, the VaVe approach utilizes extensional versioning as the creation of any new version requires a predecessor version (or none) to determine the starting point from which changes have to be applied to reach the new version. Linsbauer et al. [18] investigate a variability aware configuration management and revision control platform of highly configurable systems. Major contributions comprise, for example, a repository allowing the commit and checkout a particular configuration and the storage of automatically computed traces from features to fine-grained implementation artifacts. Similarities to our approach comprise versions management for variants individually instead of the complete variable system. Furthermore, the authors also consider heterogeneous implementation artifacts like UML models, source code or CAD diagrams targeting multiple views of a system. Linsbauer et al. [17] present a classification of *Variation Control Systems (VarCS)* for integrated management of features, variants and variation points and elaborate why VarCS are not widely used as common version control systems like Git or Subversion. One of the reasons found is that most VarCS do not perform consistency checks (or even consistency repair) on the created products.

Significant related work has been conducted in the research area of managing variability in a model-driven way. DeltaEcore [27] is a language generation framework to automatically derive *delta languages* to alter source languages like Java programmatically. Such delta languages are language-dependent and comprise transformation operations to alter a product to another one by adding, removing or changing parts. Although we don't target the generation of delta languages, there are many similarities between the concepts of DeltaEcore and the VaVe approach. First, both approaches deal with variability management in space and time in a model-driven way. Second, the variant derivation mechanisms of both approaches encompass delta modeling. Third, the VaVe model captures configuration and evolution knowledge of a variable software system as does the *hyper feature model* [26] utilized in DeltaEcore. Although DeltaEcore considers multiple artifacts representing different views on a variable software system, it does not provide explicit functionality for consistency preserving mechanisms. As another difference, the VaVe approach provides a recorder mechanism to appropriately propagate changes to dependent models. Schwägerl et al. [25] propose *SuperMod* which combines concepts of MDSE, SPLE and Version Control (VC). SuperMod provides basic VC constructs like a central repository, a local workspace and commonly known VC commands like commit and checkout. A difference to our work is that SuperMod assigns features to *visibilities* which serve as a generalization of arbitrary propositional formulas called *presence conditions*. If evaluated to true, features are included in the desired product. Closely related to our work, Haugen et al. [7] introduce the *Base Variability Resolution models (BVR)* language which is an enhanced and simplified version of the *Common Variability Language (CVL)* [8]. Both the BVR and the VaVe approach share the goal of enabling variability support for arbitrary EMOF-based models. However, BVR maintains several variability assets while the VaVe approach captures variability in space and time within one main variability artifact. As another difference, BVR is primary concerned with handling variability in space. BVR, along with all previously mentioned approaches, does not provide consistency preserving mechanisms in variable systems if inconsistencies has been detected.

6 CONCLUSION AND FUTURE WORK

In this research-in-progress paper, we have presented the VaVe approach. The goal of our research is to support versions and variants management for view-based modeling with automated consistency preservation. The combination of these concepts enables developers to describe systems in specialized views that are aware of the versions and variants of the system under development. Furthermore, the VaVe approach supports the resolution of inconsistencies, which always arise when complex systems are modeled with multiple heterogeneous and partial views.

VaVe is designed as an extension to the Vitruvius framework and is based on its delta-based consistency preservation mechanisms. Like the Vitruvius approach, VaVe is non-intrusive and can be used with any kind of EMOF-based metamodel.

We have described the concept for our VaVe approach in this paper. Currently, we are planning the realization and implementation of our approach within the existing Vitruvius prototype, which is based on the Eclipse Modeling Framework. Furthermore, we are investigating the integration of concepts from DeltaEcore. To evaluate our approach, we plan to apply it first to a case study from component-based software development using the running example from this paper. Then, we extend the evaluation to further domains, e.g., industrial automation systems (AutomationML), embedded systems for automotive (SysML), and others to which the Vitruvius approach has been applied successfully.

ACKNOWLEDGMENTS

This work was partially funded by the Federal Ministry of Economy and Energy (BMWi), following a decision of the German Bundestag in context of the INTEGRATE project (grant agreement 01MA17001B).

REFERENCES

[1] C. Atkinson, D. Stoll, and P. Bostan (2010). Orthographic Software Modeling: A Practical Approach to View-Based Development. In: *Evaluation of Novel Approaches to Software Engineering*. Vol. 69. Berlin/Heidelberg: Springer, pp. 206–219.

[2] R. Conradi and B. Westfechtel (June 1998). Version Models for Software Configuration Management. In: *ACM Comput. Surv.* 30.2, pp. 232–282. URL: http://doi.acm.org/10.1145/280277.280280.

[3] Z. Diskin, Y. Xiong, and K. Czarnecki (2010). Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In: *Proc. of the 1st International Workshop on Model-Driven Interoperability*. Oslo, Norway: ACM, pp. 42–51.

[4] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke (1992). Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. In: *International Journal of Software Engineering and Knowledge Engineering* 2.1, pp. 31–57.

[5] H. Gomaa and M. E. Shin (2004). A Multiple-View Metamodeling Approach for Variability Management in Software Product Lines. In: *Proc. of the 8th International Conference on Software Reuse: Methods, Techniques, and Tools*. Berlin, Heidelberg: Springer-Verlag, pp. 274–285.

[6] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen (2008). Adding Standardized Variability to Domain Specific Languages. In: *Proc. of the 12th International Software Product Line Conference*. IEEE Computer Society, pp. 139–148.

[7] Ø. Haugen and O. Øgard (2014). BVR – Better Variability Results. In: *Proc. of the 8th International Conference on System Analysis and Modeling: Models and Reusability*. Cham: Springer International Publishing, pp. 1–15.

[8] Ø. Haugen, A. Wąsowski, and K. Czarnecki (2013). CVL: Common Variability Language. In: *Proc. of the 17th International Software Product Line Conference*. Tokyo, Japan: ACM, pp. 277–277.

[9] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende (2010). Closing the Gap Between Modelling and Java. In: *Proceedings of the Second International Conference on Software Language Engineering*. Denver, CO: Springer-Verlag, pp. 374–383.

[10] T. Hettel, M. Lawley, and K. Raymond (2008). Model Synchronisation: Definitions for Round-Trip Engineering. In: *Proc. of the 1st International Conference on Theory and Practice of Model Transformations*. Zurich, Switzerland: Springer-Verlag, pp. 31–45.

[11] ISO/IEC 19508:2014(E) (2014). Information technology – Object Management Group Meta Object Facility (MOF) Core. International Organization for Standardization, Geneva, Switzerland.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep. CMU/SEI-90-TR-21. SE Institute.

[13] H. Klare (2016). Designing a Change-Driven Language for Model Consistency Repair Routines. MA thesis. Karlsruhe Institute of Technology (KIT).

[14] M. E. Kramer, E. Burger, and M. Langhammer (2013). View-centric engineering with synchronized heterogeneous models. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. Montpellier, France: ACM, 5:1–5:6.

[15] M. E. Kramer (2017). Specification Languages for Preserving Consistency between Models of Different Languages. PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT). 278 pp. URL: http://nbn-resolving.org/urn:nbn:de:swb:90-692845.

[16] P. F. Linington (2007). Black Cats and Coloured Birds – What do Viewpoint Correspondences Do? In: *Workshops Proceedings of the 11th International IEEE Enterprise Distributed Object Computing Conference, ECOCW*. Annapolis, Maryland, USA, pp. 239–246.

[17] L. Linsbauer, T. Berger, and P. Grünbacher (2017). A Classification of Variation Control Systems. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Vancouver, BC, Canada: ACM, pp. 49–62.

[18] L. Linsbauer, A. Egyed, and R. E. Lopez-Herrejon (2016). A Variability Aware Configuration Management and Revision Control Platform. In: *Proc. of the 38th International Conference on Software Engineering*. Vol. 38. Austin, Texas: ACM, pp. 803–806.

[19] R. E. Lopez-Herrejon and A. Egyed (2012). Towards Fixing Inconsistencies in Models with Variability. In: *Proc. of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*. Leipzig, Germany: ACM, pp. 93–100.

[20] K. Pohl, G. Böckle, and F. J. v. d. Linden (2005). Software Product Line Engineering: Foundations, Principles and Techniques. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

[21] A. Reder and A. Egyed (2012). Incremental Consistency Checking for Complex Design Rules and Larger Model Changes. In: *Proc. of the 15th International Conference on Model Driven Engineering Languages and Systems*. Innsbruck, Austria: Springer-Verlag, pp. 202–218.

[22] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg (2011). The Palladio Component Model. Tech. rep. Karlsruhe: KIT, Fakultät für Informatik. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022503.

[23] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann (Oct. 2016). Modeling and Simulating Software Architectures – The Palladio Approach. Cambridge, MA: MIT Press. 408 pp.

[24] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella (2010). Delta-oriented Programming of Software Product Lines. In: *Proc. of the 14th International Conference on Software Product Lines: Going Beyond*. Jeju Island, South Korea: Springer-Verlag, pp. 77–91.

[25] F. Schwägerl, T. Buchmann, and B. Westfechtel (2015). SuperMod - A model-driven tool that combines version control and software product line engineering. In: *Proc. of the 10th International Joint Conference on Software Technologies*. IEEE Computer Society, pp. 1–14.

[26] C. Seidl, I. Schaefer, and U. Aßmann (2013). Capturing Variability in Space and Time with Hyper Feature Models. In: *Proc. of the 8th International Workshop on Variability Modelling of Software-Intensive Systems*. Sophia Antipolis, France: ACM, pp. 1–6.

[27] — (2014a). DeltaEcore – A Model-Based Delta Language Generation Framework. In: *Proc. of Modellierung*. Vienna, Austria, pp. 81–96.

[28] — (2014b). Integrated Management of Variability in Space and Time in Software Families. In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. Florence, Italy: ACM, pp. 22–31.

[29] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks (Dec. 2008). EMF: Eclipse Modeling Framework. second revised. Addison-Wesley Longman, Amsterdam.

[30] M. Wimmer, N. Moreno, and A. Vallecillo (2012). Viewpoint Co-evolution Through Coarse-grained Changes and Coupled Transformations. In: *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*. Prague, Czech Republic: Springer-Verlag, pp. 336–352.