Towards Deep Reactions in Multi-Level, Multi-View Modeling

Thomas Weber
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
thomas.weber@kit.edu

Monalisha Ojha
Software Engineering Group
University of Mannheim
Mannheim, Germany
monalisha.ojha@uni-mannheim.de

Mohammad Sadeghi
Software Engineering Group
University of Mannheim
Mannheim, Germany
mohammad.sadeghi@uni-mannheim.de

Lars König
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
lars.koenig@kit.edu

Martin Armbruster
KASTEL
Karlsruhe Institute of Technology
Karlsruhe, Germany
martin.armbruster@kit.edu

Arne Lange
Software Engineering Group
University of Mannheim
Mannheim, Germany
lange@uni-mannheim.de

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

Colin Atkinson
Software Engineering Group
University of Mannheim
Mannheim, Germany
atkinson@uni-mannheim.de

ABSTRACT

As the scale, complexity, and scope of software-intensive systems continue to grow, so does the importance of synergistically integrating two important emerging paradigms in software engineering - multi-level modeling and multi-view modeling. While stable tooling for both has been developed by research institutions in recent years, to date no tool has attempted to integrate the two at a fundamental level. In this paper, we describe some first steps we have taken in this direction by integrating the Vitruvius V-SUM-based multi-view environment with the Melanee multi-level modeling environment. In particular, we show how Vitruvius's Reactions language, which allows different models in Vitruvius V-SUMs to be kept consistent, can be extended to support multi-level V-SUMs and views represented in Melanee's dialect of multi-level modeling.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages**; *Specialized application languages*; Application specific development environments; • **Information systems** → *Mediators and data integration.*

KEYWORDS

Multi-level modeling, V-SUM, View-based modeling, Vitruvius, Consistency

This work is licensed under a Creative Commons Attribution International 4.0 License.
MODELS Companion '24, September 22–27, 2024, Linz, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688208

ACM Reference Format:
Thomas Weber, Monalisha Ojha, Mohammad Sadeghi, Lars König, Martin Armbruster, Arne Lange, Erik Burger, and Colin Atkinson. 2024. Towards Deep Reactions in Multi-Level, Multi-View Modeling. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3688208

1 INTRODUCTION

As software systems have grown in size and complexity, and are developed as parts of integrated cyber-physical systems, it has become increasingly important to be able to describe and model them using interrelated collections of so-called *views*. View-based modeling environments that keep large numbers of semantically overlapping descriptions of systems consistent over time are therefore receiving growing attention in academia and industry. Of the two basic strategies for achieving inter-view consistency, the so-called projective approach is the most promising at scale, since it reduces the number of pairwise consistency relationships that need to be maintained [9]. However, it requires some kind of central megamodel, or *Single Underlying Model* (SUM) to serve as the source of information and truth from which the views can be projected.

The Vitruvius framework is one such environment that supports the projective approach using a *Virtual* SUM (i.e., V-SUM) rather than a pure, redundancy-free SUM. This obviates the daunting challenge of creating a pure SUM in real-life software engineering projects where it is necessary to work with and integrate, many existing models, based on long-established and utilized metamodels. A Vitruvius V-SUM therefore facilitates the consistent connection of multiple, semantically overlapping models and metamodels by means of *Consistency Preservation Rules* (CPRs) written in a specially designed Reactions language.

While Vitruvius represents the state-of-the-art for supporting scalable, projective view-based modeling, it suffers from the common limitation of most *classic* modeling environments of only directly supporting two levels of classification (i.e., types and instances). While this limitation is manageable in traditional software deployment scenarios, in cyber-physical systems that maintain models of real-world objects over a system's working lifetime (e.g., digital twins) it is desirable to use a multi-level modeling approach that inherently supports multiple classification levels in a level-agnostic way. In the paper we therefore present preliminary work in adapting Vitruvius, especially its Reactions language, to support consistency between multi-level models within a multi-level V-SUM. The particular flavor of multi-level modeling we have adopted is that supported by Melanee [1], but the presented approach could be used with any multi-level modeling language based on deep instantiation.

In addition, by exemplifying our approach using a multi-level model published as a solution to the Collaborative Comparison Challenge [17], we show how the proposed deep Reactions language can also be used to overcome a weakness of Melanee's strict multi-level modeling language which requires the use of a trick (i.e, duplicating model elements) in certain circumstances. By separating the model elements from different *dimensions* within the challenge into different, overlapping models in the V-SUM, and keeping them consistent using the proposed deep Reactions language, we show how the use of the trick can be avoided.

The paper is split into eight parts and ends with concluding remarks. After this introduction, we briefly present the work that provides the foundation of this paper. The next section explains the monolithic solution to the Collaborative Comparison Challenge used to exemplify the approach. We then explain how this monolithic model can be split into two coupled models to obviate the need for workarounds, and how we use Vitruvius' Reactions language to keep the overlapping parts consistent. The next two sections talk about the current state of the Reactions language and its limitations, which are addressed in the section after, where we introduce new concepts to enable the Reactions language to operate in a deep modeling environment. Finally, the related work section embeds the work in the relevant scientific landscape.

2 FOUNDATIONS

This section provides an overview of the two core technologies that are brought together in this paper – the multi-level modeling approach [3] and the consistency preservation mechanisms used in the Vitruvius platform [10].

2.1 Multi-level Modeling

Multi-level modeling was developed to overcome the limitations of the traditional, industry-standard, two-level modeling approach revolving around only classes and their instances. Some modeling domains demand that classes are able to characterize more than just their immediate instances. This requirement is called *deep characterization*, and over the last few years various modeling approaches have been developed to support it, such as *deep instantiation* [3]. One of the main consequences of deep instantiation is that some model elements are types and instances simultaneously, i.e., they have an instance facet and a type facet. Such modeling elements are named *Class-Objects* (clabjects).

The type facets of clabjects can be controlled through a property called *Potency*, which is a non-negative number controlling over how many levels a clabject can have instances and what instances are in its instances extensions, over multiple levels. Over the years, many approaches and languages for multi-level modeling have emerged that offer a variety of features with different advantages and disadvantages in particular scenarios. In this paper, we use a strict, characterization-potency-driven multi-level modeling approach, which is supported by the Level-Agnostic Modeling Language (LML) implemented in a tool called Melanee [1]. The Pan-level Model (PLM) is the linguistic metamodel of Melanee.

Melanee supports strict multi-level modeling which only allows the instance-of relationship to cross level boundaries and applies the principle of *Monotonic Abstraction*. From this, it follows that instances of a clabject must reside exactly on the level below that clabject's level.

Over the years, the notion of potency also evolved to be more flexible in order to allow modeling constructs like *Intermediate Abstract Classes* in classification hierarchies [13]. In contrast to classic potency [3] rules, the characterization potency rules stipulate that the potency value of an instance just has to be lower than the potency value of its type, whereas classic potency demands the potency value to be exactly one lower than the type.

2.2 View-based Modeling and Consistency Preservation

During the development of complex systems, developers create different types of artifacts, or models, to describe the system. Since these artifacts describe the same system, they may contain certain pieces of information that can become inconsistent as changes are made and the system evolves. This is obviously something to be avoided, since freedom from inconsistency is necessary, although not sufficient, to eventually build a correct system [8].

To avoid the need to define explicit consistency rules between all artifacts that could become inconsistent, Atkinson et al. [5] introduced the notion of a *single underlying model (SUM)* which contains all relevant information about the system in a redundancy-free way from which the models required by the developers are generated on demand as projected *views*. Although this avoids inconsistencies by design, designing a SUM for multiple heterogeneous types of artifacts is challenging. A more pragmatic approach is therefore to construct a *virtual single underlying models* (V-SUMs) which, instead of aiming for complete freedom from redundancy, uses explicit consistency specifications between the metamodels to maintain their consistency [6, 16]. Although this requires explicit consistency specifications inside the V-SUM, the views on the V-SUM are still implicitly consistent by creation, as they are projected from consistent models within the V-SUM.

As a pragmatic realization of the V-SUM concept, the Vitruvius approach [10] employs the imperative, unidirectional, and change-driven Reactions language [11, 10] for defining executable CPRs. As shown in Listing 1, Reactions are defined between a source metamodel (`in reaction to`) and a target metamodel (`execute`

actions in) to keep their models consistent. Reactions are triggered by changes of a certain type (here: after [. . .] created and inserted [. . .]) to objects of a certain metaclass (here: uml::Class) and execute Routines in response. Routines consist of three blocks: a match block, a create block and an update block. In match blocks, required objects can be retrieved, while in create blocks new objects can be created. Update blocks can perform arbitrary modifications to the target model to preserve consistency and may contain code in the Xtend language[1], which is a Java dialect.

Listing 1: Example of an imperative consistency specification in the Reactions language. The shown Reaction is triggered by the creation of a UML class and creates a Java class in response.

```
1  reactions: umlToJavaClass
2  in reaction to changes in uml
3  execute actions in java
4
5  reaction CreatedUmlClass {
6      after element uml::Class
7          created and inserted as root
8      call {
9          val umlClass = newValue
10         createJavaClass(umlClass)
11     }
12 }
13
14 routine createJavaClass(uml::Class umlClass) {
15     match { /* retrieve_elements */ }
16     create { /* create_elements */ }
17     update { /* update_models */ }
18 }
```

3 EXAMPLE

As an example of a multi-level model that could benefit from consistency maintenance in the context of a view-based modeling environment, we use the aforementioned Collaborative Comparison Challenge [17] that describes companies, factories, produced devices, and owned artifacts such as intellectual property. The Melanee solution [15], shown in Figure 1, consists of three levels. The most abstract level, *O0*, contains the *FactoryAsModelSupporter* clabject and its subclasses with a potency value of '1', which means that these clabjects can have instances only at the immediate level below. The *CompanyAsOwner* clabject also has a potency value of '1' and has *owns* connections to the factory and device model clabjects, that specialize the respective inheritance hierarchies. The *DeviceModel* hierarchy, with *MobilePhoneModel* and *HuaweiMobilePhoneModel* as its subclasses, all have potency values of '2'. This means these clabjects can influence instances until the *O2* level. The *MobilePhoneModel* is also the powertype of *Device* in the level below.

This example was chosen not only because it shows the key features of a multi-level model, but because it also shows how a strict approach to multi-level modeling can be enhanced by a V-SUM-based modeling approach of the kind supported by Vitruvius. The Melanee solution to the Collaboration Challenge has to use an inelegant "trick" to model a situation where a domain concept is simultaneously related to clabjects that naturally occupy different classification levels. The challenge description includes two examples of this scenario related to *Factory* and *Company* and their instances. For example, the Challenge requires a *Factory* supporting instances of *MobilePhoneDeviceModel* at one level, and then on the level below a *Factory* producing the instances of these instances. A similar situation applies to the *Company* concepts.

Since relationships (other than instantiation) between clabjects at different levels are forbidden in strict multi-level modeling environments, the Melanee solution has to use a "trick" to cope with this scenario involving the introduction of two separate *Factory* and *Company* clabjects that represent the same objects in the real world. This duplication of concepts is necessary because the Challenge deliberately mixes concerns so that concepts of different abstraction levels have to be connected to each other. The former is the concern of supporting device models and the latter is the concern of producing devices that conform to the aforementioned device models. The accidental complexity introduced by the use of the described "trick" can therefore only be untangled by separating these concerns within a model, which is something that can be elegantly achieved using Vitruvius's consistency maintenance capabilities. The result is two models, one for each concern, which have the *DeviceModel* hierarchy in common, where they completely overlap.

4 RUNNING EXAMPLE

This section shows how Vitruvius allows the single, but inelegant multi-level model in Figure 1 to be split into two separate, but synchronized multi-level models that avoid the need for the aforementioned trick. The first model, shown in Figure 2, addresses the concern of *Factory* **producing** mobile phone devices. The second model, shown in Figure 3 describes *Factory* **supporting** mobile phone devices. In both Figure 2 and Figure 3, the color-coding of the hierarchies represents the following:

- *Factory* hierarchy is colored green.
- *Company* hierarchy is colored yellow.
- *DeviceModel* hierarchy is colored blue.

The main reason for the color-coding is to highlight different hierarchies visually and to display the importance of the *DeviceModel* hierarchy which has the same form in both hierarchies and thus represents an overlap. Therefore, it has to be kept consistent. There is a 1-to-1 mapping between the two models, e.g., if another instance of *S400* is created in one model it must be reflected in the other model.

The other hierarchies also have to be kept consistent, although they exist on different levels of abstraction in each model. In Figure 2 the *Factory* concept is introduced at level *O1* whereas in Figure 3 the same concept is introduced at level *O0*. The same is true for the *Company* concepts.

5 VITRUVIUS REACTIONS LANGUAGE

The purpose of the Vitruvius Reactions language [10, 12] is to support the specification of CPRs. CPRs are imperative and consist of different Reactions language constructs. A consistency specification consists of CPRs. This section gives an overview of the language and its constructs and discusses an example that illuminates the current obstacles to defining Reactions for deep models.

[1]https://eclipse.dev/Xtext/xtend/, accessed 03.07.2024

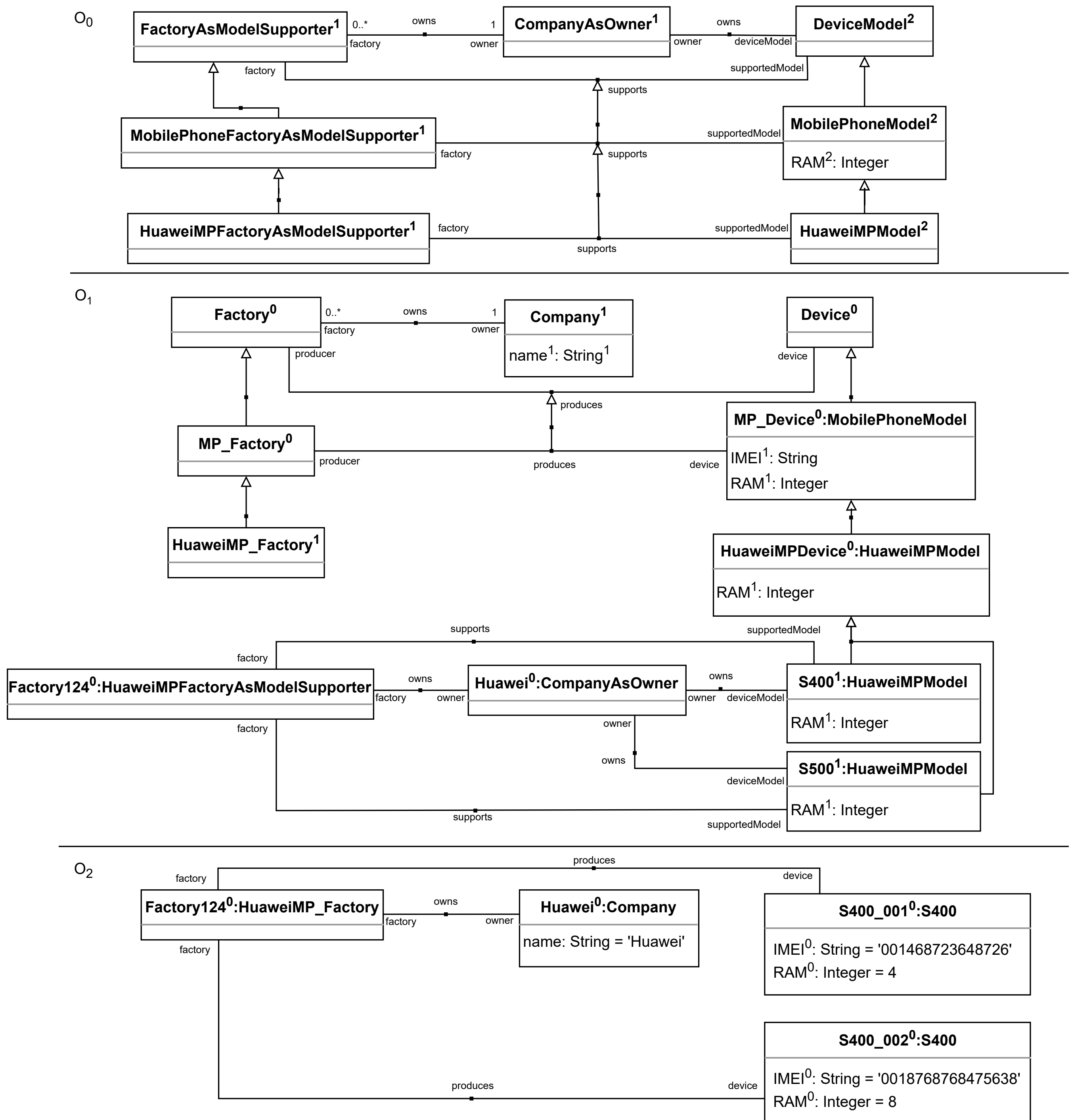

Figure 1: The Melanee solution for the Collaboration Challenge that introduced two separate *Factory* and *Company* concepts to adhere to the strict MLM doctrine [15].

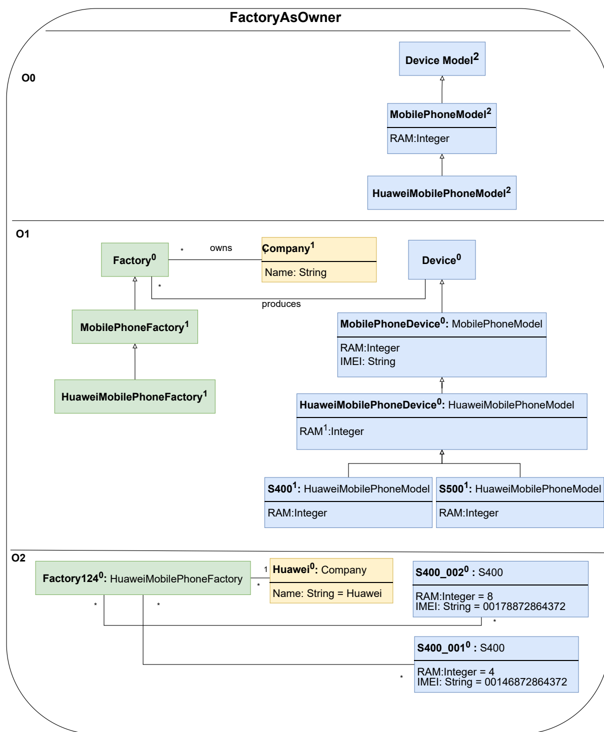

Figure 2: Model where a *Company*'s *Factory* produces Mobile Phone Devices. The *Factory* hierarchy is colored green, the *Company* hierarchy yellow and the overlapping *DeviceModel* hierarchy blue.

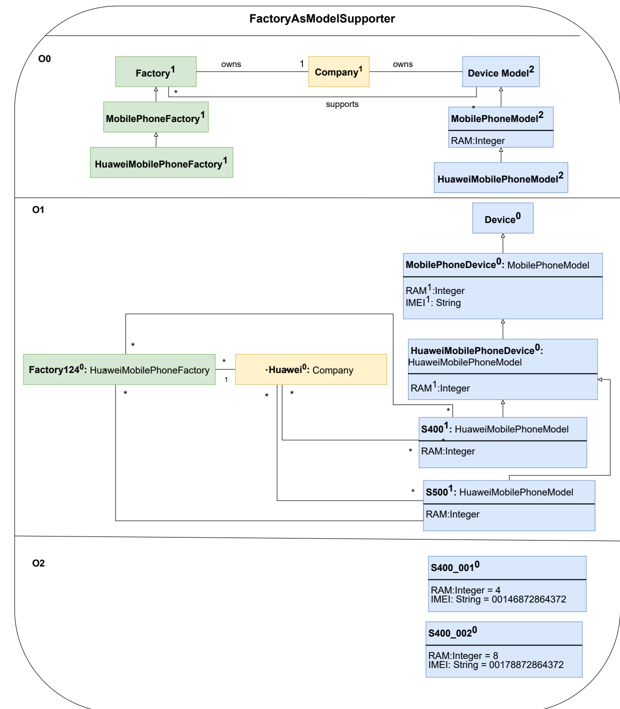

Figure 3: Model where a *Company*'s *Factory* supports Mobile Phone Devices. The *Factory* hierarchy is colored green, the *Company* hierarchy yellow and the overlapping *DeviceModel* hierarchy blue.

Listing 2: Import of used Java elements; Import of metamodels by their namespace Uniform Resource Identifier (URI) into a Reaction.

```
import org.melanee.core.models.plm.PLM.PLMPackage

import "http://melanee.org/PLM" as owner
import "http://melanee.org/PLM" as supporter
```

5.1 Structure

The Reactions language is defined as an *Xtext* [7] grammar, available in our repository[2]. Firstly, CPRs are defined in Reactions files, which start with imports of metamodels and arbitrary Java elements as seen in Listing 2. Note that in the current prototype PLM metamodels have to be imported twice, once for the owner model and once for the supporter model.

The next part of a Reactions file specifies its name to enable the import and reuse of Reactions files inside other Reactions files. Additionally, the source metamodel, i.e., the metamodel whose instances are modified by the developer, and the target metamodel, i.e., the metamodel whose instances are modified by the Reactions, are defined, as illustrated in Listing 3. More than two metamodels can be imported, e.g., if annotation metamodels are used. Modifications in the annotated metamodel, such as the deletion of an element, also necessitate the deletion of their annotation.

Note that we have the same metamodel in both the source and the target definition, and thus Vitruvius is not able to distinguish changes based on their metamodel. A change in the instance of the `supporter` metamodel also triggers reactions for the instances of the `owner` metamodel. We are currently only able to handle this by explicitly mirroring the structure of the metamodel in the Reactions, leading to a lot of specification overhead and duplication. We omitted the duplication in our code examples, but you can view them in our code[3].

Listing 3: A Reaction defines a source metamodel `owner` on which changes are reflected to a target metamodel `supporter` to keep it consistent.

```
reactions: owner2supporter
in reaction to changes in owner
execute actions in supporter
```

The final part of a Reactions file consist of the definitions of concrete Reactions and Routines. A Reaction has a name and three distinct parts. The first part is the definition of the change that the Reaction should react to. In the example in Listing 4, the name is `NewS400Inserted`, and the definition of the change contains the insertion of an element of type `OwnedElement` into a `Level`.

[2] https://github.com/vitruv-tools/Vitruv-DSLs/blob/main/bundles/tools.vitruv.dsls.reactions/src/tools/vitruv/dsls/reactions/ReactionsLanguage.xtext, accessed 03.07.2024

[3] https://doi.org/10.5281/zenodo.13325994

The with-block contains constraints for the triggering of the Reaction. In our example in Listing 4, the Level the OwnedElement (i.e., affectedEObject) needs to be inserted into is two. Additionally, the directType of the newValue, i.e., the OwnedElement that is inserted, needs to have a specific name, here "S400". Note that, due to the current limitations of the Reactions language, we cannot refer to any of the types by their actual type, but only through their name and their PLM metamodel type, e.g., OwnedElement. The last block of a Reaction definition is the definition of Routines to call. In our example in Listing 4, we call a Routine called addNewS400 with the parameter newValue, which is the inserted OwnedElement.

Listing 4: Reaction to describe the insertion of an OwnedElement into a Level. The with block defines constraints and the call block the (re)actions to execute to preserve consistency.

```
1  reaction NewS400Inserted {
2   after element owner::OwnedElement inserted in owner::Level[
        content]
3   with {
4    affectedEObject.level === 2
5    && (newValue instanceof Entity)
6    && (newValue as Entity).directType.name == "S400"
7   }
8   call {
9    insertNewS400(newValue as Entity)
10  }
11 }
```

Routines define parameters and three blocks. The match block retrieves model elements, e.g., containers of elements or corresponding elements. An element corresponds to another element, if a correspondence to that element has been created, e.g., during the execution of a Reaction. Due to the missing support for deep models, we cannot distinguish changes between the owner and supporter model. The corresponding workaround, i.e., adding tagged correspondences which are a mapping from one list of objects to another list of objects, annotated with a string, is omitted from the examples. The tagged correspondences capture information about whether an element is part of the owner or supporter model.

The othertype is used to add the classification of the new element created by this routine, retrieved from the directType used by the input of this routine. The supporterDeepModel element is used as a container for the newly created element. In general, we can rely on a corresponding type in the supporter model, because we have, e.g., set it earlier in the Routine reacting to the creation of the type in the owner model.

The next block, i.e., create, creates new model elements and supports a shorter notation for the usage of EMF factories. The update block then updates model elements in the supporter model and also inserts the newly created newEntity and newClassification. Their attributes are set according to the elements in the owner model. Lastly, they are inserted into the supporter model at level two. For further Reactions, e.g., modifying the inserted elements, we also set correspondences for the Entity and Classification pairs. Other case studies can be found online[4].

5.2 Adding new Ontological Types

In the previous example, we outlined how a Reaction handles the addition of new instances of the S400 clabject. However, what if we want to react to, and mirror, the introduction of a new phone model,

[4]https://github.com/vitruv-tools/Vitruv-CaseStudies, accessed 03.07.2024

Listing 5: A routine creating a S400 in the supporter model for a given S400 in the owner model.

```
1  routine insertNewS400(owner::Entity ownedElement) {
2   match {
3    val othertype = retrieve supporter::Clabject corresponding to
          ownedElement.directType
4    val supporterDeepModel = retrieve supporter::DeepModel
          corresponding to ownedElement.deepModel
5   }
6   create {
7         val newEntity = new supporter::Entity
8         val newClassification = new supporter::Classification
9   }
10  update {
11   newEntity.name = ownedElement.name
12   newClassification.instance = newEntity
13   newClassification.type = othertype
14   val level = supporterDeepModel.getLevelAtIndex(3)
15   level.content += newElem
16   level.content += newClassification
17   addCorrespondenceBetween(ownedElement, newEntity, "")
18   addCorrespondenceBetween(ownedElement.classificationsAsInstance.
          get(0), newClassification, "")
19         ...
20  }
21 }
```

Listing 6: Test code for inserting a new S400_003 of type S400

```
1 var entity = PLMFactory.eINSTANCE.createEntity();
2 entity.setName("S400_003");
3 var classification = PLMFactory.eINSTANCE.createClassification();
4 classification.setInstance(entity);
5 classification.setType((Clabject) Util.getElementWithName("S400",
      owner));
6 getSecondLevel(owner).getContent().add(entity);
7 getSecondLevel(owner).getContent().add(classification);
```

for instance, a S600? In this case, a new Reaction is needed for which we apply the same principle as in the existing Reaction: if a new element is inserted into level one of the owner model with the direct type HuaweiMobilePhoneModel, a Routine creates a corresponding element in the supporter model with the same name. If there is a Factory which supports such a newly introduced phone model in the supporter model, the developer needs to connect the Factory with the phone model.

By adding the S600 instance to the owner and supporter models, we also want a Reaction which reacts to the addition of instances of the S600. One possibility is partial reuse of the Reaction for the S400. The Routine shown in Listing 5 is independent of the concrete phone model. Thus, we can reuse it without modification for the S600. We only need to copy and adjust the Reaction in Listing 4 so that it checks the direct type name to be equal to S600. Since Vitruvius does not support the addition of new Reactions during runtime, Vitruvius would have to be restarted to add the Reaction for instances of the S600.

Another possibility is to extend the checks in Listing 4. Instead of testing the name of the direct type of the new element, the name of the direct type's direct type can be tested on equality to HuaweiMobilePhoneModel. As a consequence, the Reaction for new instances of the S400 would also apply to all instances of new HuaweiMobilePhoneModels such as the S600. This can avoid a restart of Vitruvius when this possibility is considered beforehand. Otherwise, Vitruvius still needs to be restarted to extend the Reaction of the S400.

5.3 Complete Example Reaction

The complete example[5] to react to the insertion of a new `S400`, consists of the preceding examples. To test the addition of a new `S400`, we create a new `S400_003` with its `classification` as a `S400`, seen in Listing 6. We then add both to the second level in the `owner` model and assert that they also exist in the `supporter` model after we execute the consistency preservation.

5.4 Consistency between Domains

The Routine in Listing 7 shows a comprehensive example of how consistency is preserved in the separated models. The previous section was concerned with the *DeviceModel* hierarchy and the ability to react to new instances of the *S400* device at the same level of abstraction. This Routine, however, shows the consistency preservation of model elements that exist on different levels of abstraction. For instance, the *Factory* clabject in Figure 2 exists at a lower level of abstraction than the *Factory* clabject in Figure 3. The Routine would almost be the same for the *Company* hierarchy.

With this Routine, we can split the original model into two separate models while preserving their consistency at different levels of abstraction. This solution addresses the original challenge while avoiding the accidental complexity present in the initial approach.

6 TOWARDS DEEP REACTIONS

In the previous section, we have shown that we can write and execute Reactions for the two separate multi-level models. For example, we have shown how we can react to newly instantiated *S400* phone models by instantiating a new instance in the other view. Additionally, we described how we can react to adding a new phone model, for example, a *S600*. In the current prototype, Vitruvius needs to be stopped, and a newly created Reaction has to deal with the newly created type *S600*. Furthermore, the Reactions language in its current state is not aware of the deepness of the model, which means that a modeler cannot exploit the idiosyncrasies of a deep modeling environment.

Below, we identify the requirements that a deep Reactions language for a deep virtual SUM environment needs to fulfill.

- **R1:** The language should be **aware of the deepness** of the models and meta-models.
- **R2:** Since the deep modeling approach used in this paper is **level-adjuvant**, the Reactions language should also be **aware of levels**.
- **R3:** The deep Reactions language should have **reflective** capabilities that allow the methodologists to write precise and rich queries.

The first requirement states that the new features should be aware of the deepness of the models. The language needs to be aware of clabjects influencing more than one instantiation step, i.e., one level of type-instance relationships and has to provide language features that allow the modelers (or methodologists) to specify that changes to a clabject should be reacted to in a deep manner. Consider the scenario of having to react to a new instance of a *S400* phone device with only one Reaction definition in the context of *DeviceModel* in level *O0*.

[5]https://doi.org/10.5281/zenodo.13325994

Listing 7: A routine creating a `Factory` in the `supporter` model for a given `Factory` in the owner model (we omitted the Reaction definition here, it is similar to Listing 4).

```
1  routine insertNewHuaweiMobilePhoneFactory(owner::Entity
       ownedElement) {
2   match {
3    val othertype = retrieve supporter::Clabject corresponding to
         ownedElement.directType
4    val supporterDeepModel = retrieve supporter::DeepModel
         corresponding to ownedElement.deepModel
5    val supporterConnection = retrieve supporter::Entity
         corresponding to ownedElement.connections.findFirst[it.name
          === "owns"]
6    val othercompany = retrieve supporter::Entity corresponding to
         ownedElement.connections.findFirst[it.name === "owns"].
         participants.findFirst[it.directType.name === "company"]
7   }
8   create {
9          val newFactory = new supporter::Entity
10         val factoryClassification = new supporter::Classification
11         val connection = new supporter::Connection
12   val newFactoryConnectionEnd = new supporter::ConnectionEnd
13   val companyConnectionEnd = new supporter::ConnectionEnd
14         val connectionClassification = new supporter::
               Classification
15  }
16  update {
17   newFactory.name = ownedElement.name
18   val level = supporterDeepModel.getLevelAtIndex(2)
19   factoryClassification.type = othertype
20   factoryClassification.instance = newFactory
21   newFactoryConnectionEnd.destination = newFactory
22   connection.allConnectionEnd += newFactoryConnectionEnd
23   companyConnectionEnd.destination = othercompany
24   connection.allConnectionEnd += companyConnectionEnd
25   connectionClassification.type = supporterConnection
26   connectionClassification.instance = connection
27   level.content += newFactory
28   level.content += factoryClassification
29   level.content += connection
30   level.content += connectionClassification
31  }
32 }
```

The second requirement states that the language should be aware of levels (in a level-adjuvant MLM approach). The current state of the prototype is level aware in the sense that a level is a container for model elements that are of interest and have to be reacted to. The new Reactions language should be aware of the multiple classification levels so that the modeler can control over how many levels the Reaction should be executed.

The third requirement aims at the reflective capabilities of the language. This feature helps the modeler write richer and more concise type queries that are essential for controlling the scope of what the Reaction should react to and the scope of the actionable model elements in the other views of the V-SUM system.

To fulfill these requirements, we propose the following syntax changes, accompanied by a corresponding implementation.

6.1 Support for Deep Types

The first syntax change we propose relates to the handling of deep types in Reactions. As Reactions are defined between metamodels, their syntax supports only the types defined in the metamodel. For deep models, however, the PLM metamodel only describes the structure of deep models in general, i.e., the linguistic metamodel. This newly introduced feature relates to the requirement **R1**.

The actual types a Reactions developer would be interested in, from an Ecore point of view, are instances of the PLM model. In the current prototype, shown in Listings 2, 3, 4 and 5, this means that developers can only use the types from the PLM metamodel (e.g., `OwnedElement` or `Entity`) and have to check manually whether the Ecore model elements have the correct level and deep type, as shown in lines 4-6 of Listing 4.

In order to allow developers to use deep model types in their Reactions, we propose to specify whether the Reactions in a file are defined between Ecore metamodels or between deep models when importing the metamodels or models. For deep models, instead of, e.g., `import "http://melanee.org/PLM" as owner` (Listing 2, line 3), developers would then write `import deep "<pathtolml-model>" as owner` (Listing 8, line 1). Note that the `import deep` statement is followed by a path to an LML model rather than a namespace URI because we cannot distinguish LML models in the Eclipse metamodel registry (as both have the namespace URI "http://melanee.org/PLM"). Thus, we have to distinguish LML models differently.

For all deep models imported using the **deep** keyword, the types defined in the model become available using the common type syntax of the Reactions language, as shown, e.g., in line 9 of Listing 8 (`owner::S400`).

Listing 8: Example Reaction for consistency preservation between two deep models using our proposed syntax for supporting deep models in the Reactions language.

```
import deep "pathtolml_model" as owner
import deep "pathtolml_model" as supporter

reactions: owner2supporter
in reaction to changes in owner below level 1
execute actions in supporter below level 1

reaction NewS400Inserted {
    after direct element owner::S400 inserted in owner at level 2
    call {
        addNewS400(newValue)
    }
}

routine addNewS400(owner::S400 oldS400) {
    match {
        val supDeviceLevel = retrieve level 2 in supporter
    }
    create {
        val supDevice = new supporter::S400
    }
    update {
        supDevice.name = oldS400.name
        addCorrespondenceBetween(oldS400, supDevice)
        supDeviceLevel.content.add(supDevice)
    }
}
```

6.2 Level Restriction for Changes

Our proposed new syntax for supporting deep models, as described in subsection 6.1, allows developers to use deep types when defining Reactions. While this seems useful, using types from an Ecore model comes with certain problems. With classical Ecore models, Reactions are defined between metamodels and changes occur only at the model level. This is not the case with deep models, however, where there is no distinction between a fixed meta-level and a changing model level. It would therefore be possible for Reactions, using types from a deep model, to become invalid after changes to the deep model at the runtime. As this would result in a system unable to react to further changes to the deep model, we propose to limit the changes to which Reactions can react to certain levels. This newly introduced feature relates to requirement **R2**.

We propose to limit the levels on which changes are permitted using the syntax `in Reaction to changes in <deep source model> below level <x>` and `execute actions in <deep target model> below level <y>` at the beginning of a Reactions file, as shown in lines 5-6 of Listing 8. In the example, the Reactions only react on changes at the meta-level *O2*, specified by `below level 1` for the source model, while changes on meta-levels *O0* and *O1* would be rejected by Vitruvius. The keyword `below` refers to the representation of the level in the figures, i.e., *O1* is below *O0*. The same is true for the target model, which can receive changes as well. The types at the meta-levels *O0* and *O1* from the source and target models can therefore be used as types in the definition of the Reactions. If it becomes necessary to perform changes at the meta-levels *O0* or *O1* of either model, Vitruvius would have to be shut down and restarted with the adapted Reactions. In terms of the introduction of new phone models, this would be necessary in order to add the Reaction for new phone models at *O1* and to raise the level so that Vitruvius can execute the Reaction.

To support the separation of concerns and re-usability of the Reactions, we would propose to allow Reactions to be used for different levels of the same source or target model together. Changes on a deep model would, however, only be accepted if all Reactions defined for the deep model permit them. We would not limit the co-existence of Reactions to Reactions on deep models but allow Reactions to be used on deep models together with Reactions defined on conventional Ecore metamodels, which includes the PLM metamodel.

6.3 Explicit Level Support

As the changes to which Reactions between deep models can react target elements at different meta-levels, with the restrictions described in subsection 6.2, it is necessary for developers to specify the meta-level of changes. This newly introduced feature also relates to the requirement R2. In the current version of the Reactions language, this requires checking or setting the level of model elements manually. Examples are shown in line 4 of Listing 4 and lines 14-16 of Listing 5. As with explicit support for deep types, as described in subsection 6.1, we want to add explicit support for deep meta-levels to the Reactions language.

To make working with deep meta-levels more intuitive, we propose two new syntax elements for the specification of meta-levels for the changes that a Reaction reacts to and for levels used during the creation of consequential changes in **update** blocks. For the first case, we propose to extend the syntax `after` in the definition of a Reaction to include the deep meta-level of the changes. For insertion changes, e.g., the complete syntax would be: `after element <element type> inserted in <deep model> at level <x>`, as shown in line 9 of Listing 8. This is also beneficial in Reactions for adding new phone models and their instances. When the Reaction in Listing 8 specifies `owner::HuaweiMobilePhoneModel` as the inserted element type, it can react to all new instances of

new phone models. Instead of specifying a single level, it would also be possible to react to changes in a range of levels, e.g., using the syntax `at level 2-3`. A Reaction with this trigger would not only react to changes to instances of the specified element type but also to instances of instances of it. For the second case, we propose to retrieve the PLM object representing the meta-level in the **`match`** block of routines. The retrieved objects can then be used in the Xtend code of the **`update`** block of routines as before, e.g., for adding elements to a deep meta-level. An example of this mechanism can be seen in line 17 and line 25 of Listing 8. This newly introduced feature relates to the requirement **R3**.

6.4 Triggers for Changes to Direct and Indirect Instances

With the proposed changes in subsection 6.3, it would be possible to react to changes of instances of a deep type, as well as instances further down the instance-of hierarchy, by specifying the permitted levels at which changes are reacted to. Often, however, it is desirable to target instances of subclasses of the specified deep type as well. Since we expect this to be the default case for developers, we do not want to introduce a new syntax for including instances of subclasses, but would instead offer a syntax to prevent the inclusion of instances of subclasses. In this case, developers who only wish to react to changes to instances would use the syntax `after direct element` instead of `after element`, as can be seen in line 9 of Listing 8.

6.5 Implicit Creation of Classifications

When creating a new element, in a deep model, as well as in traditional Ecore models, it is necessary to define the type of the element. In the PLM metamodel, this is represented by an instance of the metaclass `Classification` with attributes for the new `instance` and its `type`. In the current implementation of the deep Reactions language, the classification needs to be created manually, as shown in lines 8 and 12-13 of Listing 5. With our proposed changes to the Reactions language, described in subsection 6.1, however, this becomes unnecessary in most cases, as the types from the deep model can be used when creating elements, as shown in line 20 of Listing 8. This allows Vitruvius to implicitly create a classification relationship where the newly created element is the `instance` and the deep type is the `type` of the classification. This newly introduced feature relates to the requirement **R1**.

Creating classification relationships implicitly depends on the availability of the types from the deep model. As discussed in subsection 6.2, however, only types from a certain range of levels can be made available, as changes to types that are used in Reactions would break the Reactions at runtime. When creating objects of types that are allowed to change, and which therefore are not available as types for the definition of Reactions, classifications cannot be created implicitly, as the type information is not available. In these cases, the classifications would have to be created manually, as done in the current version of the Reactions language in Listing 5. For example, new phone models such as `S600` are not available, so the classifications for their instances would need to be created manually.

7 RELATED WORK

Kühne [14] introduced an approach to combat the introduced accidental complexity of strict, level-adjuvant multi-level modeling paradigms by separating the emerging orthogonal, ontological dimensions. clabjects from different dimensions can be connected regardless of their level of abstraction so that the "Factory" clabject can be connected to mobile phone models and/or devices. In every dimension, the strict multi-level rules apply so that there are rules and checks in place to prevent the user from modeling well-known antipatterns, like meta-bombs or meta-cycles [2].

Another already deep, view-based, modeling approach is Orthographic Software Modeling (OSM) [4], which uses a deep dialect of ATL (Atlas Transformation Language) to keep the views and the SUM consistent. It is an approach that expects the SUM to be redundancy-free in comparison to the V-SUM approach presented in this paper, which is a pragmatic method to assemble the metamodel information needed to generate the views in a projective manner. In the current version of OSM, the solution to the challenge would also include the "trick" in the SUM because it is based on the strict multi-level modeling variant that forbids any connections other than instance-of relationships from crossing level boundaries.

Generated views in OSM would look the same as in the presented model in Figure 3 and Figure 2. Although using deep models is already possible with OSM, the V-SUM approach supports the integration of multiple, pre-existing metamodels from development tools [6]. This requires explicit consistency specifications between the metamodels, which can be achieved for example by using the extended Reactions language [11] we have proposed in this paper.

Stevens [18] introduced consistency management between models as a challenging aspect of Model-Driven Development (MDD). To model large-scale software, megamodelling, like a V-SUM, is employed to encapsulate dependencies between models and to ensure consistency through heterogeneous transformations including unidirectional, bidirectional, and multi-directional transformations. By applying a build system to the metamodel, it is possible to determine which models are affected by a change and ensure that the minimum necessary transformations are performed to maintain the state in a coherent manner, thereby addressing the challenge of Multi-Model Consistency Management.

8 CONCLUSION

In this paper, we showed how the original Melanee solution to the Collaboration challenge, can be separated into two separate models, to avoid the accidental complexity introduced by the "trick" employed to overcome the restrictions of strict modeling. We explained how we split the model along its natural dimensional boundary to remove the unwanted accidental complexity. As a result, we claim that the resulting models are easier to understand and maintain in the future.

We implemented a part of the consistency preservation specifications to resolve the overlap with the Reactions language. Additionally, we illustrated the shortcomings of the language for use with deep models by showing example specifications in the current version. Based on the observed shortcomings, we created requirements for a deep Reactions language. Additionally, the proposed features allow us to be more dynamic when reacting to changes,

especially changes at the levels *O1* or *O0*, which can not currently be handled at runtime in Vitruvius.

The new `deep` keyword will make the Reactions language aware of the deepness of the models used, and thus fulfill our first requirement **R1**. The `level 0x` and `below level 0x` keywords allow us to write level-aware Reactions as described in requirement **R2**. The combination of these new language features allows us to access the ontological type of the models to write precise, dynamic, and rich deep queries, thus fulfilling requirement **R3**. A deep version of Vitruvius would allow Melanee's deep modeling approach to be combined with the pragmatic V-SUM approach, and thus open new application areas for both. On the one hand, methodologists using the Vitruvius approach are empowered to use the features of deep models for their existing V-SUMS, as deep models can be seamlessly integrated into a V-SUM that provides deep Reactions. On the other hand, existing deep models with overlapping information can be kept consistent using deep Reactions in the Vitruvius framework.

In the future, we plan to implement our proposed new keywords and features and evaluate their usability in a user study. We aim to rework Vitruvius so that the whole platform can deal with multiple instantiation levels instead of the regular EMF two-level dichotomy. Understanding how reactions could work in a multi-level modeling setting is one of the first steps toward realising a deep version of Vitruvius.

ACKNOWLEDGMENTS

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – CRC 1608 – 501798263 and supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF), and funded by the Topic Engineering Secure Systems of the Helmholtz Association (HGF) and supported by KASTEL Security Research Labs.

REFERENCES

[1] Colin Atkinson and Ralph Gerbig. 2012. Melanie: multi-level modeling and ontology engineering environment. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, 1–2.

[2] Colin Atkinson and Thomas Kühne. 2001. Processes and Products in a Multi-Level Metamodeling Architecture. en. *International Journal of Software Engineering and Knowledge Engineering*, 11, 06, (Dec. 2001), 761–783. doi: 10.1142/S0218194001000724.

[3] Colin Atkinson and Thomas Kühne. 2001. The Essence of Multilevel Metamodeling. en. In *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools* (Lecture Notes in Computer Science). Martin Gogolla and Cris Kobryn, (Eds.) Springer, Berlin, Heidelberg, 19–33. isbn: 978-3-540-45441-0.

[4] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2010. Orthographic software modeling: a practical approach to view-based development. In *Evaluation of Novel Approaches to Software Engineering*. Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 206–219. isbn: 978-3-642-14819-4.

[5] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2010. Orthographic Software Modeling: A Practical Approach to View-Based Development. en. In *Evaluation of Novel Approaches to Software Engineering* (Communications in Computer and Information Science). Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski, (Eds.) Vol. 69. Springer Berlin Heidelberg, Berlin, Heidelberg, 206–219. isbn: 978-3-642-14819-4. doi: 10.1007/978-3-642-14819-4_15.

[6] Colin Atkinson, Christian Tunjic, and Torben Moller. 2015. Fundamental Realization Strategies for Multi-view Specification Environments. en. In *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. IEEE, Adelaide, Australia, (Sept. 2015), 40–49. isbn: 978-1-4673-9203-7. doi: 10.1109/EDOC.2015.17.

[7] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[8] Istvan David, Hans Vangheluwe, and Eugene Syriani. 2023. Model consistency as a heuristic for eventual correctness. *Journal of Computer Languages*, 76, 101223.

[9] 2011. ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000). eng. (2011).

[10] Heiko Klare, Max E Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. 2021. Enabling consistency in view-based system development—the vitruvius approach. *Journal of Systems and Software*, 171, 110815.

[11] Max Emanuel Kramer. 2017. *Specification Languages for Preserving Consistency between Models of Different Languages*. PhD Thesis. Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. doi: 10.5445/IR/1000069284.

[12] Max Emanuel Kramer. 2019. *Specification languages for preserving consistency between models of different languages*. Vol. 24. KIT Scientific Publishing.

[13] Thomas Kühne. 2018. Exploring Potency. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (MODELS '18). event-place: Copenhagen, Denmark. ACM, New York, NY, USA, 2–12. isbn: 978-1-4503-4949-9. doi: 10.1145/3239372.3239411.

[14] Thomas Kühne. 2022. Multi-dimensional multi-level modeling. en. *Software and Systems Modeling*, 21, (Jan. 2022), 543–559. doi: 10.1007/s10270-021-00951-5.

[15] Thomas Kühne and Arne Lange. 2022. Melanee and dlm: a contribution to the multi collaborative comparison challenge. In (MODELS '22). Association for Computing Machinery, Montreal, Quebec, Canada, 434–443. isbn: 9781450394673. doi: 10.1145/3550356.3561571.

[16] Johannes Meier, Christopher Werner, Heiko Klare, Christian Tunjic, Uwe Aßmann, Colin Atkinson, Erik Burger, Ralf Reussner, and Andreas Winter. 2020. Classifying approaches for constructing single underlying models. In *Model-Driven Engineering and Software Development: 7th International Conference, MODELSWARD 2019, Prague, Czech Republic, February 20–22, 2019, Revised Selected Papers 7*. Springer, 350–375.

[17] Gergely Mezei, Thomas Kühne, Victorio Carvalho, and Bernd Neumayr. 2021. The multi collaborative comparison challenge. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 495–496.

[18] Perdita Stevens. 2020. Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels. *Software and Systems Modeling*, 19, 4, 935–958. doi: 10.1007/s10270-020-00788-4.