Exploring Design Alternatives for Multi-model Consistency

Lars König[1], Fabian Eger[1], Martin Armbruster[1], Colin Atkinson[2], Mattias Ulbrich[1] and Erik Burger[1]

[1]Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2]University of Mannheim, Mannheim, Germany

Abstract

One of the central challenges in model-driven development is the preservation of consistency between multiple heterogeneous models of a system. The Virtual Single Underlying Model (V-SUM) approach is a conceptual framework to express consistency between multiple models, which combines projective views with explicit consistency preservation. However, there are different ways to handle the semantic overlap that exists in the meta-models of a V-SUM. In this work we explore two strategies: using explicit Consistency Preservation Rules (CPRs) and modifying the meta-models to remove redundancies. Both strategies support the same view types, but require different model-view transformations to provide them, with potentially different effects on the complexity of defining a V-SUM. We use an existing case from the automotive domain and extend it to cover both strategies. We then discuss the complexity of both strategies and give recommendations on when to apply them based on usage scenarios for V-SUMs.

Keywords

consistency preservation, model management, view-based development

1. Introduction

During the development of cyber-physical systems, developers use different tools, representations, and formalisms to work on the models of the developed system. As these models describe the same system, they might contain the same information, i.e., they have a *semantic overlap*. Keeping this overlapping information consistent, is one of the central challenges of model-driven development [1].

In contrast to consistency checking, consistency preservation is concerned with semi-automatically keeping the models consistent upon changes to one of them. Different approaches have been proposed [2, 3], such as the SUM approach [4] and the more general V-SUM approach, implemented by the Vitruvius framework [5]. While the SUM approach assumes a single, redundancy-free model, from which the individual developer views are projected, the V-SUM approach extends this by allowing developers to combine multiple models with explicitly defined CPRs.

In this work, we want to explore these two approaches as two strategies for designing multi-model consistency in the form of a V-SUM. For that, we consider an example case [6] from the automotive industry, which we present in section 2. In this case, developers from different domains use models conforming to different meta-models, such as a *CAD* meta-model and the meta-model of the tool *Simulink*. For comparing the two approaches, we build on an existing V-SUM implementation [6] using the Reactions [7] language, which is part of the Vitruvius framework. We extend this implementation by creating redundancy-free versions of the involved meta-models, as well as view definitions using the NeoJoin [8] language. We describe the two approaches in detail in section 3. We then compare the two strategies regarding the necessary assumptions and limitations of the two approaches, as well as regarding their complexity and access behavior.

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29–July 3, 2026
lars.koenig@kit.edu (L. König); fabian.eger@kit.edu (F. Eger); martin.armbruster@kit.edu (M. Armbruster); colin.atkinson@uni-mannheim.de (C. Atkinson); ulbrich@kit.edu (M. Ulbrich); burger@kit.edu (E. Burger)
0000-0002-1751-1291 (L. König); 0009-0009-6845-1071 (F. Eger); 0000-0002-2554-4501 (M. Armbruster); 0000-0002-3164-5595 (C. Atkinson); 0000–0002–2350–1831 (M. Ulbrich); 0000-0003-2832-3349 (E. Burger)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-26

CEUR-WS.org/Vol-4338/ICMM-paper4.pdf

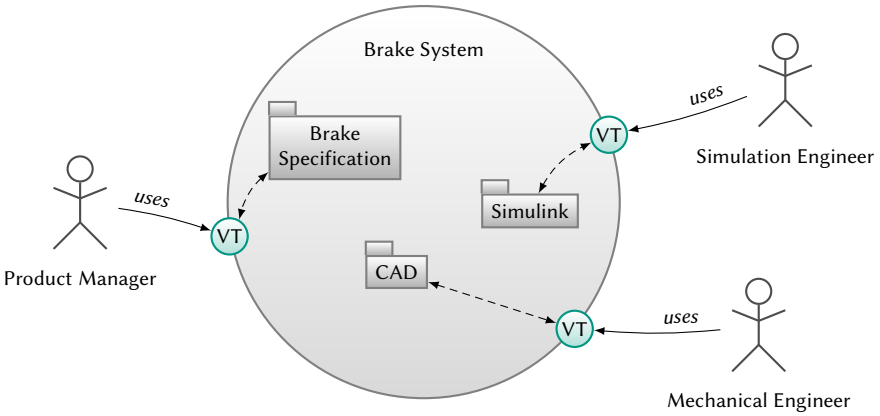


Figure 1: MM = meta-models, VT = view types, ◂- - -▸ = view generation transformations

2. Running Example: Brake Development

In order to demonstrate the various design options, we present a concise, illustrative example, which details the development of a brake system. In this short example, there are three roles working on parts of the brake system as explained by Hagel et al. [6]. The first role is the product manager, who is responsible for the specification of the brake system and, consequently, has a meta-model that represents the required components for the specification. The second role is that of a mechanical engineer, who utilizes a CAD (Computer-Aided Design) modelling tool, and thus its meta-model, to create a representation of the brake system. The final role is that of the test or simulation engineer. They utilize the Simulink tool to create simulations of the brake system, which also has a meta-model. Figure 1 shows the brake system development with the three meta-models, each only referring to the identity view type for the role utilizing the meta-model. *Identity View Type* in this case means that the view type has a one-to-one mapping for each class of the meta-model.

As these three roles only operate within the context of the identity views associated with their respective meta-models, they may model the same concepts across different meta-models, yet without the presence of trace links between them. This scenario potentially results in inconsistencies during the development of the brake system. The aforementioned overlaps are defined as semantic overlap between the meta-models. For instance, one may consider the following excerpt from the meta-models in Figure 2, where a brake component and the namespace, both classes in different meta-models, have the same attribute *id*, thus the corresponding CAD namespace to a brake component should have the same *id*.

A more general point is the necessity for each brake component to have a corresponding namespace. The rationale behind this requirement is as follows: in the event of the addition of a brake pad to the system in the brake specification, the creation should also be reflected in the CAD drawing by the creation of a corresponding namespace in the CAD model.

In the following sections, we discuss the different design alternatives for managing the semantic overlaps and consequently removing possible inconsistencies in this example.

3. Design Alternatives

When designing appropriate consistency preservation measures for multi-model consistency, there are often alternative designs with potentially different effects on, e.g., their complexity. In this work, we explore two strategies by applying them to the brake case, presented in section 2. Although these strategies could be combined, we want to show the two extremes here. In subsection 3.1, we use explicit consistency preservation rules to incrementally keep the models consistent. On the other hand, in subsection 3.2, we use projective views on the models instead.

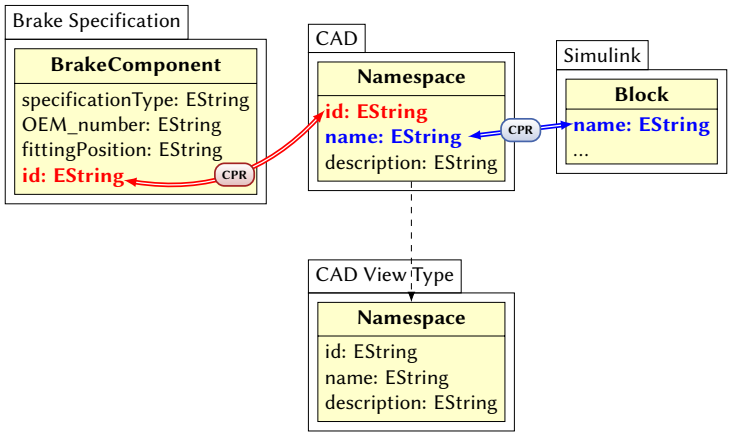


Figure 2: Three meta-classes with semantic overlap (red, blue) and consistency preservation rules (◄►) between them. The view generation (- -►) of the view type *CAD View Type* is the identity function.

3.1. Strategy A: Explicit Consistency Preservation

In this section, we describe the strategy for preserving consistency between multiple models via specific CPRs. In this approach with explicit CPRs, the meta-models of the different domains do not have to be adapted or merged in order to achieve overall consistency. Accordingly, for each identified semantic overlap, rules are established to represent how a change in one model has to affect the model of the other meta-model, in order to keep the models consistent. So to refer to the examples mentioned in section 2, if a *BrakeComponent* is added in the *Brake Specification* model, than a CPR would create a *Namespace* in the *CAD* model. And as shown in Figure 2, if the *id* is changed either in the *Namespace* or in the *BrakeComponent* the *id* in corresponding class is also changed to keep them consistent.

In the Vitruvius framework [5] with the underlying V-SUM approach these CPRs are defined via the Reactions language. The Reactions [7] language is an imperative language, which allows the developer to define rules to react to changes on one model by defining the Reaction in line 1 and the trigger in line 2 of Listing 1. In this example the Reaction's trigger is the change of the name attribute in the *SimuLink Block*. If the trigger is matched by an update in the model the defined routine to update the other model is called in line 3. In the update routine itself the corresponding *Namespace* is retrieved in the match block (line 6–8 in Listing 1). After that the *Namespace name* is updated to the new name in the update block in line 9–11. It is important to note that Reactions are defined unidirectionally; in the aforementioned example, the Reaction from the *Namespace* to the *Block* is still missing.

From a process perspective, after identifying the semantic overlap, the CPRs between two meta models are created. When another meta-model is added, new semantic overlap has to be identified and new CPRs have to be created between the new meta-model and each already existing meta-model. This process can be parallelized as for each CPR only the two concerned meta-models are necessary.

```
1  reaction SimulinkBlockNameUpdated {
2      after attribute replaced at simulink::Block[name]
3      call updateCADNamespaceName(affectedEObject, newValue)
4  }
5  routine updateCADNamespaceName(simulink::Block block, String newName) {
6      match {
7          val mNamespace = retrieve cad::Namespace corresponding to block
8      }
9      update {
10         mNamespace.name = newName
11     }
12 }
```

Listing 1: A Reaction defining the CPR for a SimuLink Block name change, which propagates to the corresponding CAD namespace name.

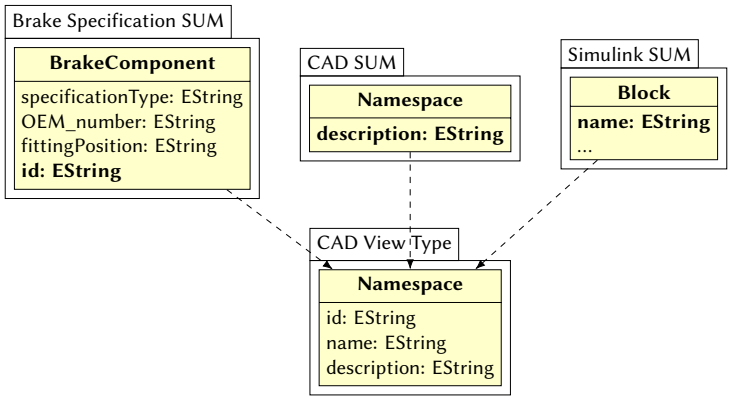


Figure 3: Three meta-classes without semantic overlap. The view generation (- -►) of the view type *CAD View Type* combines information (bold) from all three meta-classes.

For the view type generation in this scenario, only the identity function to the required meta-model is necessary (as presented in Figure 1) and no complex calculation or manipulation of other meta-models are needed because the consistency mechanisms are handled via the CPRs and the original meta-models of the different roles exist without changes.

The running example, as well as the evaluation data, for this strategy was taken from the existing Vitruvius BrakeCaseStudy repository[1]. In this repository, the different meta-models and CPRs, defined in the Reactions Language can be found.

3.2. Strategy B: View Types on Redundancy-Free Meta-Models

The second strategy is based on the idea of a single underlying model (SUM), as proposed by Atkinson et al.. While the purpose of a SUM is also to keep the views on it consistent, it does so by removing redundancies in the underlying meta-model, instead of using explicit CPRs. As we believe it is infeasible to create a single meta-model for cross-domain projects, we use a V-SUM instead. However, removing redundancies can serve as an effective strategy to avoid using explicit CPRs in a V-SUM as well.

Here, we first describe how we created redundancy-free meta-models from the meta-models present in the brake case, presented in section 2. Building on the created, redundancy-free meta-models, we describe view definitions that combine their information to recreate view types that match the original meta-models from the brake case. Thus, we ensure that the developers can use the same views as before, while avoiding inconsistencies in the internal models.

Process for Removing Redundancy

In order to remove the redundancy of the different meta-models, we first have to identify the semantic overlap of the meta-models. If we detected the semantic overlap, like the *name* in the *SimuLink Block* and the *CAD Namespace*, we then have to decide where to store the overlapping information. This is not always an easy decision, which is why we divided it into three different cases.

In the first case the attribute detected as semantic overlap between the two meta-models is not inherited to classes which are not part of the semantic overlap, and therefore we can decide where to put the attribute. For example consider Case A in Figure 4, where it does not matter if the name is stored in class *A* or class *B*.

The second case is where in one meta-model the attribute is also inherited to other classes which are not part of the semantic overlap and the other meta-model has no inheritance for that attribute. In this case we need to shift the attribute to the meta-model with the inheritance. To also demonstrate this in an example, consider Case B in Figure 4. In this particular case, we can not store the *name* in class *A*, because the name of class *B* is inherited from another class, which also has subclasses (i.e. *D*) inheriting

[1] https://github.com/vitruv-case-studies/BrakeCaseStudy (accessed 2026-04-03 17:00)

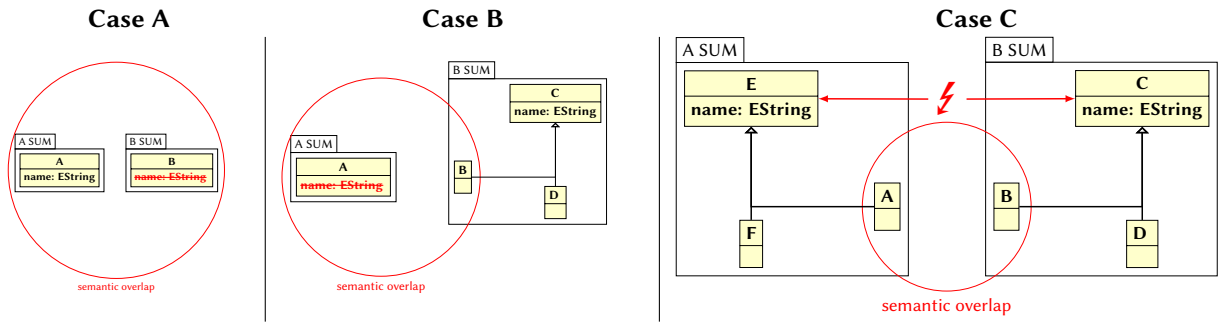


Figure 4
Three cases of how semantic overlap can be distributed between meta-models. (A) It can be decided where to store the attribute. (B) Based on the inheritance hierarchy the attribut has to be stored in the B SUM. (C) Based on the inheritance in both SUMs, no easy distribution of the attribute is possible

the attribute, which are not in the semantic overlap of the two meta-models. Of course it is also possible to shift the attribute *name* down the inheritance hierarchy and then keep it in the B SUM, but as we want to change the metamodel as little as possible, the first solution would be the better choice.

In the last case, both meta-models have the inheritance of the attribute to also classes which are not part of the semantic overlap, which means that in that case the redundancy cannot be removed in the first place (see Case C in Figure 4).

As previously outlined in subsection 3.1, the present study also explores how redundancy removal works in scenarios involving more than two meta-models. In contrast to the process outlined in subsection 3.1, it is not possible to parallelize the steps involved in the creation of redundancy-free meta-models. This is due to the fact that the newly added meta-model can potentially add Cases B and C (see Figure 4) to the redundancy free meta-models before, which than requires reconsideration of the decisions before. Thus, it is not possible to ascertain whether the redundancy removal between the two earlier meta-models is sufficient for the third meta-model, given that the addition of a new meta-model is undertaken without prior knowledge. Consequently, the integration of a new meta-model must be approached with consideration for the overarching context of the existing meta-models. It is evident that this process can be very overwhelming when large meta-models are employed. Consequently, it is not possible to carry out the process in parallel.

Creating the View Definitions

Building on the redundancy-free meta-models, which we use inside the V-SUM, we need to define view types that represent the original meta-models, as required by the developers using the V-SUM. For our example presented in Figure 1, we define the view types *Brake Specification*, *CAD*, and *Simulink*. In addition to the domain meta-models, we utilize the same correspondence meta-model as used by the CPRs to connect related model elements. However, as the views are responsible for integrating the information from the underlying models, they are responsible for creating all corresponding elements. In this extreme scenario, no CPRs are necessary, although in practice, we would expect a combination of view definitions and CPRs to be used.

Combining information from multiple meta-models is realized in NeoJoin [8] using *join* statements. In Figure 3, we show the result of joining the three meta-classes *BrakeComponent*, *Namespace*, and *Block* to create the meta-class *Namespace*. The attributes *id*, *description*, and *name* of the view type meta-class *Namespace* are selected from the joined meta-classes. On the model level, one instance of each class is selected to create an instance of the meta-class *Namespace* in the view. Note that in NeoJoin, we use a single view definition to create both, the view type meta-model and the model-view transformations.

In NeoJoin, a join combines selected model elements on the left-hand side with a model element on the right-hand side based on a join condition. As related meta-classes in this scenario by design contain no redundant information, we use explicit correspondence objects to relate them. We show the NeoJoin query to create the meta-class *Namespace* as an example in Listing 2. In lines 1–3, we join a *Namespace*

```
1  from Namespace namespace
2  join Correspondence corr           on corr.leftEObjects.contains(namespace)
3  join BrakeComponent brakeComponent on corr.rightEObjects.contains(brakeComponent)
4  join Correspondence corr2          on corr2.leftEObjects.contains(namespace)
5  join Block block                   on corr2.rightEObjects.contains(block)
6  create Namespace {
7    ^id = brakeComponent.ID
8    block.name
9    namespace.description
10   parameters: Parameter = namespace.parameter
11 }
```

Listing 2: NeoJoin query for creating the view type meta-class *Namespace*, shown in Figure 3, as well as the associated model-view transformations.

with a *BrakeComponent*. We do so by first joining the *Namespace* with a *Correspondence* on the condition that the *Correspondence* element contains the *Namespace* in its reference *leftEObjects*. After that, we join the two joined objects with a *BrakeComponent* on the condition that the *Correspondence* element contains the *BrakeComponent* in its reference *rightEObjects*. By that, we have the three related model elements available for the creation of the view element of the meta-class *Namespace*. In lines 4–5, we proceed by joining a *Block* using another *Correspondence*. We finally select the attributes from the joined meta-classes in lines 7–10, which are required for the meta-class *Namespace* in our *CAD* view type.

In the basic case, we can join one element from each meta-model, as demonstrated in Figure 3. However, there are cases where the same information is modelled differently. In our example, presented in section 2, this is the case for attributes of the subclasses of the meta-class *BrakeComponent* in the meta-model *Brake Specification*, which are modelled as standalone model elements of subclasses of the meta-class *Parameter* in the *CAD* meta-model. We solved this by adding joins for each required *Parameter* instance on the condition that the *Parameter* is referenced by the already joined *Namespace* and has the correct *name*. We illustrate this for the *numberOfPins* attribute of the *BrakeComponent* subclass *ABSSensor* in Figure 5. Here, the model element of the meta-class *IntegerParameter* is joined using the referencing model element of the meta-class *Namespace* and its *name* “Number of Pins”. The resulting view element of the meta-class *ABSSensor* then contains the attribute *numberOfPins* with the value 6. For the remaining attributes, we proceed in the same way, choosing the respective names and subclasses of the meta-class *Parameter*, which are: *IntegerParameter*, *NumericParameter*, *StringParameter*, and *BooleanParameter*.

The usage of different subclasses of *Parameter* in the *CAD* meta-model leads to another problem when creating the view definition for the *Simulink* view type. As the *Simulink* meta-model contains only one, untyped *Parameter* meta-class, we would need to project all *Parameter* subclasses from the *CAD* meta-model to this untyped meta-class. In NeoJoin, this is not allowed, as it would not be possible to propagate newly created view elements back to the underlying models, as the concrete subclass in the *CAD* meta-model would be unknown. We solved this by creating additional subclasses also in the Simulink view type. We argue that this is a valid solution, as tools or representations could treat the created subclass elements as elements of the *Parameter* superclass. In general, this solution builds on a notion of *view type conformity* [9], i.e., the resulting view type conforms to the desired meta-model, instead of being equal to it.

4. Comparison of the Two Strategies

Here, we compare the two strategies presented in section 3: using explicit consistency preservation rules or redundancy-free meta-models with projective views. First, we discuss necessary assumptions and limitations of the two strategies. Then, we compare the two strategies using lines of code (LoC) and number of meta-model changes as metrics for complexity and discuss the results. While our

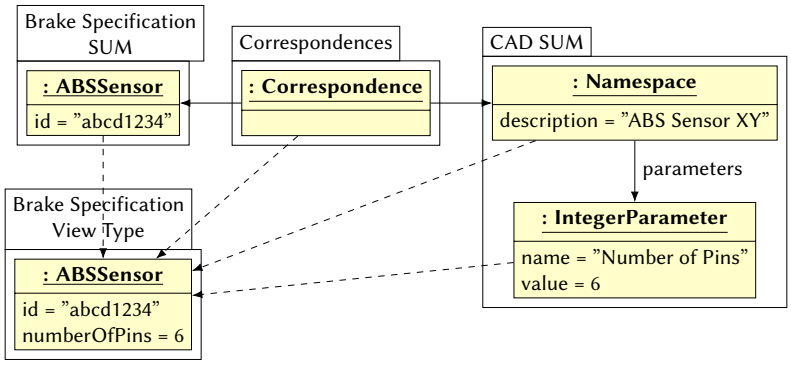


Figure 5: Example of how model elements from the meta-models *Brake Specification SUM* and *CAD SUM* are combined to produce a view element of the meta-class *ABSSensor*. Note that additional attributes and the Simulink SUM model elements are left out to simplify the diagram.

first strategy, using CPRs, is always applicable, we compare it with our second strategy, using view definitions, to explore whether its use can reduce the complexity of creating a V-SUM for keeping the used models consistent.

4.1. Assumptions and Limitations

As CPRs are typically Turing-complete transformation languages, such as the Reactions [7] language, they can be applied in all relevant scenarios. On the other hand, query-based view definition languages are by design restricted, to enable a more concise, easy-to-use syntax [8]. Thus, we discuss the relevant assumptions and limitations for using view definition languages to remove redundancies.

One assumption, which underlies the second strategy, is that the original meta-models can be changed to create redundancy-free meta-models. This is possible when, for instance, the meta-models are self-created or easy to access and integrate. However, there are cases, in which the meta-models cannot be modified. This includes meta-models, which are developed by third parties, proprietary, or provide additional features such as analyses requiring models conforming to the original meta-model. As a consequence, the second strategy cannot be applied in all cases.

In our example in section 2, we were able to create view types conforming to the original meta-models from the redundancy-free meta-models.We describe the process of creating the redundancy-free meta-models and view definitions in subsection 3.2. To check the differences between the resulting view types and original meta-models, we used a small, self-written program, which we include in our replication package [10]. All reported differences are expected, as we explain below.

First, one difference between the generated view types and the original meta-models is the presence of a single root meta-class. As this is a design choice of NeoJoin, we do not consider this as an error in our view definitions. The generated view types contain additional subclasses required to distinguish, e.g., elements of the *Parameter* meta-class in the *Simulink* view type, as explained in subsection 3.2. In addition, we argue that these subclasses preserve the conformance of the view types to the original meta-model. Finally, the original *Simulink* meta-model contains additional meta-classes that we did not recreate, as they are not part of the semantic overlap between the *Simulink*, *CAD*, and *Brake Specification* meta-models. These additional meta-classes are also not considered in the implemented CPRs of our first strategy.

While we were able to create conforming view types for all original meta-models, we noticed a couple of issues with NeoJoin [8], which we used as a view definition language. First, the Xtend[2] expressions supported by NeoJoin for feature definitions do not support instance-of expressions. This could limit their expressiveness when, e.g., using a query for a superclass with a value that depends on the concrete subclass of an element. In addition, it is currently not possible to access Enum values in expressions, which would be required to filter over Enum features. In our example, this would be required, if the

[2]https://eclipse.dev/Xtext/xtend/ (accessed 2026-04-03 17:00)

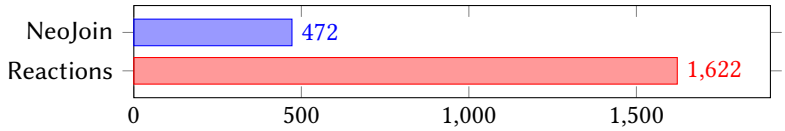


Figure 6: Lines of Code (LoC) required for implementing the first strategy using the Reactions [7] language and second strategy using the NeoJoin [8] language.

elements of the *Parameter* meta-class in the *CAD* meta-model could have different units, represented by an Enum value. We assume both issues could be fixed with reasonable effort. Last, there is an open issue in the implementation of NeoJoin, which prevents developers from creating abstract meta-classes when a meta-class with the same name is inferred from another query. This is a rather technical issue, which we expect to be fixed soon.

Another relevant aspect for choosing one of the strategies regards the expressiveness of meta-concerns (i.e., concerns that span multiple meta-models, result from their combination in a V-SUM, and target the consistency management). In both strategies, these meta-concerns need to be handled differently.

The correspondences are the first meta-concern. While the first strategy allows developers to set correspondences fine-grained between two elements, the second strategy assumes that all instances of a meta-class have a correspondence to an instance of another meta-class if they are kept consistent. This results from combining these meta-classes to eliminate redundancy. For example, all instances of the *BrakeComponent* correspond to an instance of the *Namespace* when employing the second strategy.

Further meta-concerns include the handling of additional, annotation-based features (e.g., uncertainty, inconsistency, or maturity). In the first strategy, elements in one model can be annotated, for which developers decide how and when to propagate them. For example, if a new *BrakeComponent* is in development and has an uncertain *fittingPosition*, the developer can prevent the direct propagation of the new component to the CAD model until the uncertainty is resolved. In contrast, in the second strategy, a new *BrakeComponent* leads directly to a new *Namespace*, independent of potential annotations. As a consequence, their influences and resolutions are different, assuming that this can be handled in redundancy-free meta-models.

4.2. Complexity of the Implementations

In an effort to explore whether our second strategy, using view definitions with redundancy-free meta-models, reduces the complexity of creating a V-SUM, we implemented both strategies for our example system, as we describe in subsection 3.1 and subsection 3.2. We include the implementation artifacts in our replication package [10]. We estimate the complexity of both approaches using lines of code (LoC) and number of meta-model changes as metrics. We calculated the LoC with the tool cloc[3] and included the used language configuration in our replication package as well. We did not count lines that were empty or contained only comments.

In Figure 6, we show our first complexity metric *LoC* for the implementations of the two strategies. The first strategy, using CPRs written in the Reactions [7] language, resulted in 1622 lines of code. On the other hand, the second strategy, using view definitions written in the NeoJoin [8] language, resulted in 472 lines of code. These values are not completely accurate, however, as we did not consider meta-classes without a semantic overlap between the meta-models when implementing both strategies. In addition, the Reactions from the *BrakeSpecification* to the *Simulink* meta-model are incomplete and the Reactions for the reverse direction are missing completely. For our second complexity metric *number of meta-model changes*, we counted additions and deletions of classes, attributes, and references, as well as changes to the type of attributes. We counted 30 changes in total (Brake Specification: 22, CAD: 6, Simulink: 2).

While the results do not allow us to draw any general conclusions, we believe they support our hypothesis that view definitions can reduce the complexity of creating a V-SUM compared to CPRs.

[3] https://github.com/AlDanial/cloc (accessed 2026-04-02 16:00)

Of course, this is only true when our second strategy is applicable, as we discuss in subsection 4.1. We assume the reduced complexity stems from the fact that view definitions written in a query-like style are more high-level, as they only select meta-classes and compute their features. On the other hand, CPRs require explicit implementations for handling different kinds of changes to elements of a meta-class, such as creation, modification, and deletion of elements.

Another aspect of complexity involves the meta-models a developer needs to consider for creating the CPRs or view definitions. While CPRs in the Reactions language are written between exactly two meta-models, one Reaction for each direction, there is only one view definition per view type required. However, this view definition needs to relate all meta-models that have a semantic overlap with the view type. Depending on the number of meta-models, this could increase its complexity or require additional domain knowledge for including the meta-models.

4.3. View Access Behavior

Our second strategy, using view definitions, essentially shifts effort from explicit synchronization logic toward view construction, which may be perfectly reasonable when model changes are relatively infrequent and views are only queried occasionally. However, if integrated views are accessed frequently, repeatedly reconstructing them from multiple underlying models may introduce nontrivial overhead, especially when the joins are complex or when multiple models have to be consulted to produce one developer-facing artifact. In contrast, the CPR-based strategy appears to push more effort toward write-time synchronization, since updates need to be propagated explicitly whenever one of the underlying models change. Thus, the practical trade-off is not just one of applicability and specification complexity, but also one of read/write behavior. A strategy that requires fewer lines of code may still be less attractive in a setting with frequent view access, while a more verbose CPR-based solution may be justified if it keeps developer-facing artifacts readily available after each update. Conversely, when updates are rare and combined views are only needed occasionally, the overhead of maintaining explicit CPRs may outweigh their benefits.

5. Conclusion

In this work, we explored two strategies for specifying multi-model consistency preservation. The first strategy, represented by the V-SUM approach, uses explicit consistency preservation rules (CPRs) between the involved meta-models to keep their models consistent. The second strategy, inspired by the SUM approach, builds on redundancy-free meta-models and uses view definitions to combine their information and provide developers with the required models. We implemented both strategies on an example case [6] from the automotive industry using the Reactions language [7] for the CPRs and the NeoJoin language [8] for the view definitions.

We finally compared the two strategies and discussed the assumptions and limitations of the second strategy, which is more restricted. As query-based view definition languages are by design restricted to selecting meta-classes and defining their features, they are only applicable in cases where the semantic overlap can be expressed in these terms. If they are applicable, however, our comparison of the complexity of the two strategies suggests that they can reduce the complexity of defining consistency preservation measures by a considerable amount. We assume this to be due to CPRs requiring an explicit handling of different change types that affect model elements, such as adding, modifying, and removing. In subsection 4.1, we discuss further arguments for choosing either strategy.

While in this work, we only explored the two strategies by implementing them on a single example case, we see potential for considering both strategies when designing consistency preservation measures. Further work is necessary to systematically analyze the effects of choosing either strategy. In addition, we plan to investigate how to efficiently combine the two strategies to benefit from their respective advantages, and to compare additional approaches, such as the Commonalities [11] approach for factoring out the semantic overlap of meta-models in a new meta-model.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — SFB 1608 — 501798263, and supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF).

Declaration on Generative AI

The authors have employed Grammarly, DeepL Write, and LanguageTool to check the spelling, grammar, and style of their text.

References

[1] I. David, H. Vangheluwe, E. Syriani, Model consistency as a heuristic for eventual correctness, Journal of Computer Languages 76 (2023) 101223. doi:10.1016/j.cola.2023.101223.

[2] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, C. Guychard, Addressing modularity for heterogeneous multi-model systems using model federation, in: Companion Proceedings of the 15th International Conference on Modularity, MODULARITY Companion, ACM, New York, NY, USA, 2016, pp. 206–211. doi:10.1145/2892664.2892701.

[3] A. Demuth, M. Riedl-Ehrenleitner, A. Nöhrer, P. Hehenberger, K. Zeman, A. Egyed, DesignSpace: an infrastructure for multi-user/multi-tool engineering, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC, ACM, New York, NY, USA, 2015, pp. 1486–1491. doi:10.1145/2695664.2695697.

[4] C. Atkinson, D. Stoll, P. Bostan, Orthographic Software Modeling: A Practical Approach to View-Based Development, in: Evaluation of Novel Approaches to Software Engineering, volume 69 of *Communications in Computer and Information Science*, Springer, Berlin, Heidelberg, 2010, pp. 206–219. doi:10.1007/978-3-642-14819-4_15.

[5] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, R. Reussner, Enabling consistency in view-based system development — the Vitruvius approach, The journal of systems and software 171 (2021). doi:10.1016/j.jss.2020.110815.

[6] N. Hagel, J. Mäkelburg, C. Hammann, T. Weber, T. A. Völk, F. P. Urbano, P. Grycz, K. Bause, M. Kaushik, V. Scotti, A. Bairy, M. Schwammberger, M. Acosta, A. Albers, A. Koziolek, T. Düser, Explainability in Automated Cross-Domain Model-Driven Brake System Development, in: 2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), 2025, pp. 204–211. doi:10.1109/ASEW67777.2025.00047, iSSN: 2151-0849.

[7] M. E. Kramer, Specification languages for preserving consistency between models of different languages, volume 24, KIT Scientific Publishing, 2019. doi:10.5445/IR/1000069284.

[8] L. König, T. Stickling, E. Burger, Towards dynamic views on heterogeneous models – the NeoJoin view definition language, in: Joint Proceedings of the STAF 2025 Workshops, co-located with the International Conference on Software Technologies: Applications and Foundations (STAF 2025), 2025. doi:10.5445/IR/1000188774.

[9] E. Burger, A. Toshovski, Difference-based Conformance Checking for Ecore Metamodels, in: Modellierung 2014, Gesellschaft für Informatik eV, 2014, pp. 97–104. URL: https://dl.gi.de/items/fdaacc2c-41a1-4e06-91d8-ce7728d73b7c.

[10] L. König, F. Eger, Replication package for "exploring design alternatives for multi-model consistency", 2026. doi:10.5281/zenodo.19390915.

[11] H. Klare, J. Gleitze, Commonalities for Preserving Consistency of Multiple Models, in: ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, 2019, pp. 371–378. doi:10.1109/MODELS-C.2019.00058.