Towards Consistency Preservation for Multi-Domain Variability Modeling

Dirk Neumann
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
dirk.neumann@kit.edu

Tobias Pett
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
tobias.pett@kit.edu

Ina Schaefer
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany
ina.schaefer@kit.edu

Abstract

Modern software-intensive systems comprise multiple engineering domains such as mechanics, electronics, and software. Traditionally, variability modeling approaches address variability by focusing primarily on a single domain. As a consequence, the management of multi-domain variability becomes a difficult task. One frequent outcome here is inconsistency, which means that the variability models of the different domains are incorrect. To address this problem, we propose a novel approach that combines consistency preservation for multi-domain models and delta-oriented variability modeling. By applying changes in form of deltas in one domain, our approach systematically derives corresponding deltas in other domains. In this way, we are able to ensure a consistent variant derivation process across domains. Our approach paves the way for further research on multi-domain variability management.

Keywords

Variability Modeling, Multi-Domain Variability, Incremental Consistency Preservation, Delta Modeling

ACM Reference Format:
Dirk Neumann, Tobias Pett, and Ina Schaefer. 2025. Towards Consistency Preservation for Multi-Domain Variability Modeling. In *29th ACM International Systems and Software Product Line Conference - Volume A (SPLC-A '25), September 01–05, 2025, A Coruña, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3744915.3748466

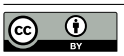
1 Introduction

Variability management across multiple domains is essential to meet customer needs in software-intensive systems, such as cyber-physical systems (CPS) [6]. For example, consider the variability for a car which integrates various features like exterior mirror heating or power windows. Each feature has its own realization in structure, implementation, mechanics, and electronics [6, 19]. In this context, variability must be realized in multiple domains, and crucially, the variability must be kept consistent across domains, meaning that domain artifacts are compatible to each other. Traditionally, Software Product Line Engineering (SPLE) [18] has focused mostly on handling the variability of only one domain: the software. Expanding to multiple domains requires manual development, which may result in the problem of inconsistencies. Advantageously, dependencies between domains are often well understood and a formulation of consistency between them may even be available.

This insight offers the starting point for a novel approach for consistency preservation in multi-domain variability management. The representation of variability in one domain can be used to retrieve information concerning the representation of the same variability from another domain. Our approach is based on two key concepts: incremental consistency preservation [7] and delta-oriented variability modeling [20, 21]. Incremental consistency preservation, as implemented in the Vitruvius approach [11], allows keeping multiple models from different domains consistent. Vitruvius defines triggers in a source domain which activate repair routines to change the target domain correspondingly to preserve consistency. Delta-oriented variability modeling [20, 21] allows expressing variability as deltas, which consist of delta operations. Deltas are (re-)used to implement new variants based on existing variants. In our approach, we leverage the fact that a delta represents variability as change, which has the advantage that it matches the change-based incremental consistency preservation approach. We extend variability modeling from a single domain to multiple domains. The definition of variability in one domain now becomes usable as a starting point for retrieving variability for another domain through consistency. By applying deltas in a consistency-preserving environment to one domain, corresponding changes in other domains are identified. These changes are then captured by deltas and applied in the target domain, thus making it possible to propagate the variability from one to another domain. We envision an approach to multi-domain variability management by consistent propagation of variability between multiple domains. This will reduce the risk of errors by developers working independently in their respective domains.

2 Background

Multi-Domain Modeling and Consistency. In model-driven engineering, a multi-domain system may be represented by several models, which are instances of their respective domain metamodel. Splitting a system description into multiple (meta)models benefits the separation of concerns but at the same time consistency preservation becomes essential. To address this need, *Orthographic Software Modeling* (OSM) [5] introduces the *Single Underlying Model* (SUM) which treats multiple (meta)models as one and use projections and a commit mechanism to keep consistency. The *Vitruvius* approach [11] generalizes this concept to a *Virtual Single Underlaying Model* (V-SUM). The V-SUM can reuse the existing domain metamodels and introduces the so-called *Consistency Preservation Rules*



SPLC-A '25, A Coruña, Spain

ACM ISBN 979-8-4007-2024-6/25/09
https://doi.org/10.1145/3744915.3748466

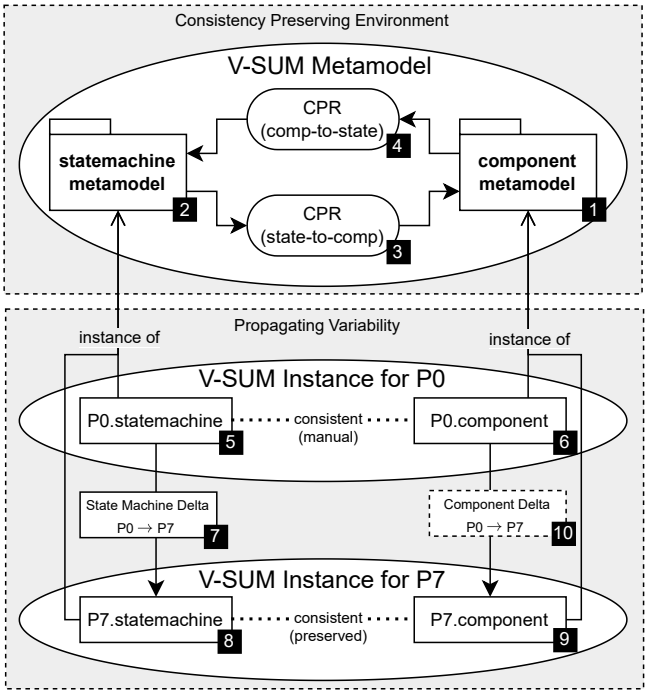


Figure 1: Consistency-preserving variability propagation across domains using the Vitruvius approach [11].

(CPR). These CPRs are unidirectional. They listen to consistency-breaking changes in a source domain by defining triggers, called *reactions*. If such a reaction is triggered, consistency can be restored by identifying and executing consistency repairing changes, called *routines*, in a target domain.

Delta-Oriented Variability Modeling. Delta-oriented variability modeling [20, 21] is an approach for managing variability by applying incremental changes, called *deltas*, used to derive one product variant from another. A delta consists of one or more atomic *delta operations* (representing additions, removals, or modifications) and can express a specific user intent, such as adding an end user-visible feature. Delta modeling integrates extremely well with consistency management in the Vitruvius approach, as both approaches rely on a semantically similar notion of change (in contrast to other variability realization mechanisms).

3 Propagating Variability between Domains

The goal of our multi-domain variability management approach is to develop variability only for a subset of source domains, and then propagate the variability to the remaining target domains. In general, the workflow of this approach consists of two major phases: Preparing the consistency preserving environment and subsequently propagate variability to multiple domains.

To illustrate our approach, we use the *Body Comfort System* case study (BCS) [14] as a running example. It consists of two domains: First, the structural composition of ECUs described in component diagrams similar to SysML. Second, the behavior of each component is described by a state machine. The BCS[1] is also a product line, defining a feature model with 19 variable features and corresponding deltas which allow for the derivation of 11.616 valid variants. The core product P0 includes the minimal number of features possible, which is only the feature *Manual Power Window*. A second product, P7, differs from P0 only in this single feature. Unlike P0, it uses the *Automatic Power Window*. These features translate into the selection of their corresponding components: `ManPW` and `AutoPW`. With this variant, the user does not have to push and hold the buttons for the power window, but instead has to push the respective button only once and can stop the power window by pressing the button for the opposite direction. Goal of this running example is to derive the delta in one of the domains from the predefined delta in the other domain, for both directions.

3.1 Preparing the Consistency Preserving Environment

In order to propagate variability through multiple domains, these domains, the variability within them and the cross-domain consistency relations have to be defined. These together build the consistency preserving environment shown in the upper part of Fig. 1. The source domains have to be chosen, which in our running example is the state machine domain. The remaining domains are the target domains, here the component domain.

Define Metamodels. For the considered domains, we need to define the metamodels as a uniform representation of their respective instances. This standardization forms the foundation for the V-SUM. For the running example, the component metamodel **1** and state machine metamodels **2** are defined (cf. Fig. 1).

Define Delta Languages. We define the set of delta operations that characterize how variability can be expressed in the domain. These operations are tied to the defined metamodels, each detailing how its elements can be modified. Adding a transition to a state machine region is such a delta operation, for instance.

Define Consistency Preservation Rules. CPRs have to be specified which each relate two domain metamodels. Between the two domains metamodels, there are consistency relations that must be maintained in order to represent a fully functional product. These CPRs allows for variability to be propagated from a source to a target domain. In the running example, CPRs link the state machine to the component domain **3** and vice versa **4**. An example reaction is shown in Listing 1.

For example, in a state machine, a transition uses a signal as trigger. The corresponding component must have an input port which accepts said signal. In the `ManPW` component of P0, transitions `t1` and `t3` are triggered by the signal `pw_but_dn?`. Therefore, `ManPW` must offer an input port that can send this signal. The reaction for this example is shown in Listing 1. It first imports the metamodels between which it preserves the consistency and assigns them aliases in lines 1 and 2. In line 3, a `reaction` is defined. It consists of a trigger in line 4 (in this case the addition to `transitions` in the `Region`) and the call of the routine `addPort` in line 5, to repair a possible inconsistency. The parameters `newValue` and `affectedObject` are

[1] https://github.com/TUBS-ISF/BCS-Case-Study-Full

```
1  import "statemachine_metamodel.ecore" as STMA;
2  import "component_metamodel.ecore" as COMP;
3  reaction TransitionAdded {
4      after adding to STMA::Region[transitions]
5      call addPort(affectedObject, newValue)
6  }
7  routine addPort(STMA::Region region, STMA::Transition
       trans){
8      match {
9          assert(trans.trigger != null);
10         val comp = retrieve COMP::Component corresponding
         to region;
11         assert(comp.hasNoInput(trans.trigger));
12     }
13     update {
14         val newPort = new COMP::Port(trans.signal);
15         comp.inputPorts.add(newPort);
16 }   }
```

Listing 1: Example reaction to create a port after a transition was added to a state machine region.

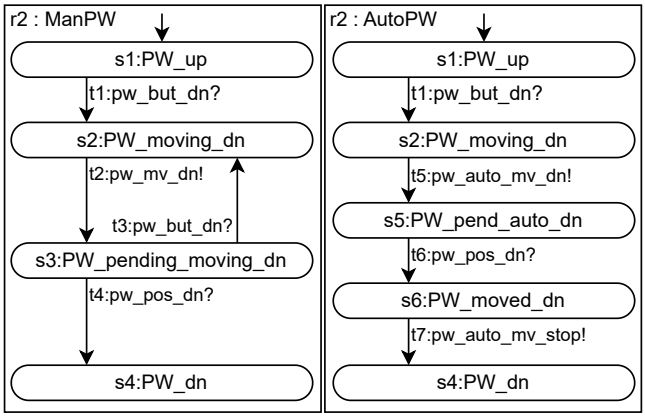
automatically resolved to the newly added `Transition` and the containing `Region` by the Vitruvius framework. A routine starts with a `match` block which ensures preconditions for the actual repair. In line 9, it is ensured that the new transition requires a signal. In line 10, the component is retrieved whose behavior is described in `region`, using a so-called *correspondence model* in which relations between model elements from different domains are stored. In line 11, the absence of a fitting input port for the signal of the transition is checked. If all preconditions are met, the target model is updated in the `update` block. In the example, a new `Port` is created and added to the input ports of the retrieved component.

3.2 Deriving Multi-Domain Variability

After preparing the consistency preserving context, a product line can be developed while exploiting the consistency preservation to propagate variability between the domains. The variability propagation is depicted in the lower part of Fig. 1.

Develop Core Models. To start multi-domain variability propagation, a consistent multi-domain core model has to be defined. We develop a core model of the component domain and call it P0. It consists of 4 different components: an `Exterior Mirror (EM)` which tilts the exterior mirrors in different directions, a `Manual Power Window (ManPW)` which can be opened/closed by continuously pressing the corresponding button, a `Finger Protection (FP)` which ensures that the power window does not pinch the user's fingers and a `Human Machine Interface (HMI)` which combines the different buttons into a single user interface. The communication between these components is realized using signals which are sent via ports. The composition of components is realized using connectors which connect two different ports. For example, the HMI can send the signals `pw_but_dn` and `pw_but_up` to the ManPW to give the command for lowering/ raising the window.

In the second domain, for each of the four components, a state machine is defined. These state machines consist of the different states that a component can be in and the transitions between them. The transitions either use a signal as *trigger* (marked as `in` signal or "?") or send a signal as *action* (marked as `out` signal or "!"). Figure 2a shows an excerpt from the state machine of the `ManPW` component that is responsible for lowering the window. This state machine



(a) Manual Power Window (P0) (b) Automatic Power Window (P7)

Figure 2: Excerpts of the Power Window State Machines.

has state `s1` as initial state in which the power window is up. The transition `t1` is triggered by receiving the signal `pw_but_dn?`. The states `s2` and `s3` represent a loop which lowers the window, i.e., sending the signal `pw_mv_dn!` (`t2`) as long as the button in the HMI is still pressed (`t3`). If the lower position of the window is reached, i.e., the signal `pw_pos_dn` is received, the state machine leaves the loop via transition `t4` and goes into the state `s4`. In our running example, we choose P0 as starting point meaning that `P0.statemachine` **5** (Fig. 2a) and `P0.component` **6** are the consistent core models.

Develop and Apply Source Domain Deltas. The BCS realizes the variability in both of its domains. In the first part of our running example, we develop the state machine delta **7** as source domain variability. Listing 2 illustrates an example delta for transforming P0 to P7. By applying deltas in the source domain, a new product variant within that domain is derived. This derivation is executed within the consistency preserving environment. In the running example, the state machine of P7 **8** (Fig. 2b) is derived by the state machine delta (Listing 2) to P0. First, in lines 2 to 4, transitions are deleted that are no longer needed, like the transition `t2`, which pointed from `s2` to `s3`. Then, in line 5, the no longer needed state `s3` is deleted, which realized the loop of lowering the window as long as the window-down button is still pressed. Afterwards, in line 6, the signal `pW_mv_dn` is deleted which is no longer used by any transition in the state machine. After the deletion, new elements are added: In lines 7 and 8 two new signals are added, followed by two new states in the lines 9 and 10. Adding states requires a reference to the state machine region in which they should be inserted, here `r2`. The regions `r1`, `r3` and `r4` refer to the state machine regions corresponding to the components `EM`, `FP` and `HMI` respectively. In lines 11 to 13, new transitions are added. A source and a target state have to be defined, and a signal has to be referenced which can either be marked as trigger signal with "`in`" or as action signal with "`out`". For example, the new transition `t5` directs from `s2` to the new state `s5` sends the signal `pw_auto_mv_dn`. Finally, the state machine region which now represents a different behavior is renamed in line 14 setting the `name` attribute to "`AutoPW`". If all delta operations

```
1  StateMachineDelta: State_PW_Man2Auto {
2    rem_transition(t2)
3    rem_transition(t3)
4    rem_transition(t4)
5    rem_state(s3)
6    rem_signal(pw_mv_dn)
7    add_signal(pw_auto_mv_dn)
8    add_signal(pw_auto_mv_stop)
9    add_state(s5,r2)
10   add_state(s6,r2)
11   add_transition(t5,s2,s5,pw_auto_mv_dn,out)
12   add_transition(t6,s5,s6,pw_pos_dn,in)
13   add_transition(t7,s6,s4,pw_auto_mv_stop,out)
14   mod_region(r2,name,"AutoPW")
15 }
```

Listing 2: Excerpt of the State Machine Delta ManPW → AutoPW.

```
1  ComponentDelta: Component_PW_Man2Auto {
2    rem_connector(cnt1)
3    rem_connector(cnt2)
4    rem_port(p1)
5    rem_port(p2)
6    rem_signal(pw_mv_up)
7    rem_signal(pw_mv_dn)
8    add_signal(pw_auto_mv_up)
9    add_signal(pw_auto_mv_dn)
10   add_signal(pw_auto_mv_stop)
11   add_port(p3,comp1,pw_auto_mv_up,out)
12   add_port(p4,comp1,pw_auto_mv_dn,out)
13   add_port(p5,comp1,pw_auto_mv_stop,out)
14   add_connector(cnt3,p3,ENV)
15   add_connector(cnt4,p4,ENV)
16   add_connector(cnt5,p5,ENV)
17   mod_comp(comp1,name,"AutoPW")
18 }
```

Listing 3: Component Delta ManPW → AutoPW.

have been executed, the state machine of the ManPW component, shown in Fig. 2a, has been transformed into the state machine of the AutoPW component, shown in Fig. 2b.

Retrieve and Execute Consequential Change. When a delta is applied in a source domain, CPRs **3** are checked if any of their triggers are activated. If so, the specified repair routine is called. This repair routine applies a consequential change in the target domain and derives a variant in the target domain that is consistent to the variant in the source domain. In the running example, in Listing 2, the delta operation in line 12 adds a new transition. This matches the trigger of the CPR defined in Listing 1, line 4. Therefore, the repair routine addPort is called, which applies the consequential change of adding a new output port to the PowerWindow component. These consequential changes lead to the component structure of P7 **9**.

Complete Target Domain Delta. The consequential changes can be captured as a delta by translating them from Vitruv's internal representation to instances of the target's domain delta language. Target domain deltas may need adaptation, as they are not necessarily complete. This may happen if model elements of the target domain are not connected to the source domain by CPRs.

Listing 3 shows the full delta for the component model that has to be applied to variant P0 to derive variant P7. First, the no longer needed model elements are deleted: the connectors in lines 2 and 3, the ports in lines 4 and 5, and the signals in lines 6 to 7. Afterwards, three new signals are added (lines 8 to 10). Then, the ports are added which use these signals. Adding a port, as in line 11, has the following parameters: identifier of the new port, component to which the port is added, signal that this port uses and the type of the port which is either "in" or "out". The identifier "comp1" refers to the component which is transformed from the ManPW component into the AutoPW component. The identifier does not change just as the actual ECU in the car does not change. All ports are output ports, since the AutoPW component uses the same input information as the ManPW component. After the ports, the connectors are added in lines 14 to 16. These delta operations are parameterized with a connector identifier, the identifier of the source port and the identifier of the target port. The target component in the example is, "ENV" which symbolizes the environment, which is not modeled in the component architecture. This can be a mechanical part of a vehicle or the user pressing a button. Last, a modify operation is used to change the name of the component in line 17. Since P0 and P7 only differ regarding the power window, the delta in Listing 3 derives the component structure of P7 from that of P0. Signals pw_mv_up and pw_mv_dn in ManPW for stepwise movement are replaced by signals pw_auto_mv_up and pw_auto_mv_dn in AutoPW for continuous movement and complemented by pw_auto_mv_stop to stop.

However, the derived component delta is incomplete. After adding the new output ports (lines 11 to 13), the delta operations in lines 14 to 16 must be completed by specifying the desired target of the connector: a) as highlighted in Listing 3, a default value (here the environment (ENV) can be specified in the CPRs and the developer is notified, b) during the retrieval of the target domain delta, the user is asked for a default value which is applied to all similar cases in this translation, c) the user is asked for each case individually. Ultimately, the component delta **10** in Listing 3 is completed.

3.3 Discussion

Limitations of the approach. Retrieving the complete target domain delta is only possible if the source metamodel dominates the target metamodel, i.e., contains all information needed in the target domain through application of CPRs. However, this is both, unlikely and undesirable, as the target metamodel would then be redundant as it was entirely determined by the source domain. In our example, we can consider the opposite direction's delta derivation and try to retrieve the statemachine delta from the component delta. The removals of the connectors, ports, and signals in the component delta (see Listing 3) in the lines 2 to 7 can be propagated as the removals of transitions using these signals in the state machine delta (see Listing 2), lines 2 to 4. Depending on the CPRs, making a component unreachable for a signal (removing connectors or ports) or deleting a used signal might result in the deletion of a corresponding transition. After the transitions t2, t3 and t4 are deleted, the states of the affected statemachine regions can be checked if any of them is now unreachable. Although it looks like this is the case for s3, it is not, as Fig. 2a only shows an excerpt of the statemachine. However, s2 has now no more transition to be left and s4 cannot be entered anymore. In summary, deleting states based on changes in the component model is not straightforward, as adding states and transitions. Signals can be deleted and added as they are represented in both metamodels, identifier modifications are also easily

transferrable as component name and state machine names correspond to each other. The resulting delta is as in Listing 2 without the orange delta operations as not automatically derivable.

Delta Completion Strategies. Several strategies exist to complete a delta when information is missing that is not part of the source domain. One option is to encode a default behavior in the CPR. For example, the developer of the CPRs can set them up such that the deletion of a connector results in the deletion of all transitions using the connector's signal. Alternatively, the CPR could also require the deletion of the signal itself as the current component might be reconnected to other components later. A second strategy is to use user interactions during the derivation of consequential changes in the consistency-preserving context. User interactions can be used to either set a default bahavior for multiple similar situations or to ask for manual resolution in individual situations. In the running example, the user may be asked whether a state should be deleted which became unreachable. A third strategy is to tolerate incomplete deltas to be completed by the developer manually.

Technical Realization. For a prototypical realization of the presented approach, we reuse and adopt existing tools. We base our implementation on three frameworks to set up a consistency preserving environment, as described in Section 3.1. The *Eclipse Modelling Framework (EMF)* [23] provides the *Ecore* metamodel standard. It grants access to many EMF utilities, and also the other two used frameworks build up on it. One of them is DeltaEcore [22] which is a framework for delta-oriented variability modeling [20, 21]. We reuse it to define delta languages based on Ecore metamodels. The other EMF-based tool is *Vitruv*[2] which implements the Vitruvius approach [11] and therefore can be used to define consistency between different domains. By integrating EMF, DeltaEcore, and Vitruv, we will establish a framework to realize consistency preserving variability propagation for multi-domain systems.

4 Related Work

Non-Variable Multi-Domain Consistency Checking and Preservation. Instant Consistency Checking [9] enables immediate identification of potential inconsistencies by precomputing and caching dependencies between model elements, allowing for real-time feedback during single-domain product development. *Orthographic Software Modeling* (OSM) [5] introduces the Single Underlying Model (SUM) which allows for the consistent development of a multi-domain product. By working only on projections of a SUM, consistency can be ensured when committing changes back into the SUM. However, the SUM relies on a specifically developed model and does not support integration of domains by reusing their metamodels. The *Vitruvius* approach [11] extends the SUM concept by introducing the V-SUM, in which existing metamodels are connected using CPRs. While Vitruvius ensures consistency through the application of predefined CPRs, *DesignSpace* [8, 17] relies on a more exploratory approach. By generating and evaluating alternative repair paths, a preselection of consistent alternatives is determined from which a developer can choose. While these approaches focus on the consistent development of a single product in multiple domains, they have not been extended to variable systems.

[2]https://github.com/vitruv-tools

Variable Multi-Domain Consistency Checking and Preservation. Adding variability to multi-domain systems introduces new types of inconsistencies. *Incremental Consistency Checking* for variable systems [12] divides inconsistencies in variable multi-domain systems into different categories. This categorization is also applied in cyber-physical production systems [10]. These categories differentiate between intra-domain and inter-domain inconsistencies, as well as between problem space inconsistencies (e.g., errors in modeling a product line's variability) and solution space inconsistencies (i.e., inconsistencies arising during product variant derivation) [18]. This categorization is not necessarily complete and so far does not address restoring or preserving consistency in variable systems. *VaVe* [3] focuses on maintaining consistency between variants and versions of artifacts within one domain. Our approach is the first to maintain consistency between multiple domains in variable systems and simultaneously enhances the development of these variable systems, paving the way towards platform-oriented development of variable multi-domain systems [4].

5 Conclusion and Future Vision

We understand our approach as a stepping stone towards a larger concept of combining multi-domain variability and consistency which contributes to different development paradigms for multi-domain product lines. Those paradigms range from product-oriented to platform-oriented development, following the idea of the *Virtual Platform* [4, 15, 16]. *Product-oriented development* [1, 2] focuses on the consistency preserving development of a single product variant and propagation of its variability back to the product line. In our approach, we utilize the consistency preservation rules to propagate variability between domains. First, this eases the development process since variability has only to be developed in the source domains. Second, the propagation of variability always results in consistent target domain variability. *Platform-oriented development* [4] addresses the development of the whole product line at the same time, which allows for joint development on core aspects included in multiple product variants. Our approach supports platform-oriented development for delta-oriented variability [20, 21] by retrieving the propagated variability as target domain deltas for further development and future reuse. To fully realize platform-oriented development, we intend to lift the current consistency preservation between model instances to consistency preservation between delta instances. This can be achieved by defining consistency preservation rules between deltas instead of between metamodels. This eventually allows not only for consistent propagation of variability, but also for consistent modification of variability. To realize this vision, advanced concepts of delta modeling, such as *higher-order deltas*, i.e., deltas modifying deltas [13], have to be incorporated and combined with consistency preservation. Continuing our work, we will develop a concept for an always-consistent multi-domain delta repository from which consistent product variants can be derived.

Acknowledgments

Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263.

References

[1] Sofia Ananieva. 2022. *Consistent View-Based Management of Variability in Space and Time.* Ph. D. Dissertation. Karlsruhe Institute of Technology. doi:10.5445/IR/1000148819

[2] Sofia Ananieva, Sandra Greiner, Jacob Krueger, Lukas Linsbauer, Sten Gruener, Timo Kehrer, Thomas Kuehn, Christoph Seidl, and Ralf Reussner. 2022. Unified Operations for Variability in Space and Time. In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '22)*. 1–10. doi:10.1145/3510466.3510483

[3] Sofia Ananieva, Heiko Klare, Erik Burger, and Ralf Reussner. 2018. Variants and Versions Management for Models with Integrated Consistency Preservation. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS '18)*. 3–10. doi:10.1145/3168365.3168377

[4] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Ștefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. 532–535. doi:10.1145/2591062.2591126

[5] 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*. 206–219. doi:10.1007/978-3-642-14819-4_15

[6] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '13)*. 1–8. doi:10.1145/2430502.2430513

[7] K. Czarnecki and S. Helsen. 2006. Feature-Based Survey of Model Transformation Approaches. 45, 3 (2006), 621–645. doi:10.1147/sj.453.0621

[8] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. 2015. DesignSpace: An Infrastructure for Multi-User/Multi-Tool Engineering. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*. 1486–1491. doi:10.1145/2695664.2695697

[9] Alexander Egyed. 2006. Instant Consistency Checking for the UML. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. 381–390. doi:10.1145/1134285.1134339

[10] Hafiyyan Sayyid Fadhlillah, Sandra Greiner, Kevin Feichtinger, Rick Rabiser, and Alois Zoitl. 2024. Managing Variability of Cyber-Physical Production Systems: Towards Consistency Management. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24)*. 945–949. doi:10.1145/3652620.3688216

[11] 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 (2021), 110815. doi:10.1016/j.jss.2020.110815

[12] Matthias Kowal and Ina Schaefer. 2016. Incremental Consistency Checking in Delta-oriented UML-Models for Automation Systems. *Electronic Proceedings in Theoretical Computer Science* 206 (2016), 32–45. doi:10.4204/EPTCS.206.4 arXiv:1604.00348 [cs]

[13] Sascha Lity, Matthias Kowal, and Ina Schaefer. 2016. Higher-Order Delta Modeling for Software Product Line Evolution. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD 2016)*. 39–48. doi:10.1145/3001867.3001872

[14] Sascha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. 2013. Delta-Oriented Software Product Line Test Models - The Body Comfort System Case Study. *Technical report, TU Braunschweig* (2013).

[15] Wardah Mahmood, Daniel Strüber, Thorsten Berger, Ralf Lämmel, and Mukelabai Mukelabai. 2021. Seamless Variability Management with the Virtual Platform. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1658–1670. doi:10.1109/ICSE43902.2021.00147

[16] Wardah Mahmood, Gül Çalıklı, Daniel Strüber, Ralf Lämmel, Mukelabai Mukelabai, and Thorsten Berger. 2024. Virtual Platform: Effective and Seamless Variability Management for Software Systems. 50, 11 (2024), 2753–2785. doi:10.1109/TSE.2024.3406224

[17] Luciano Marchezan, Wesley K. G. Assuncao, Roland Kretschmer, and Alexander Egyed. 2022. Change-Oriented Repair Propagation. In *Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering (ICSSP '22)*. 82–92. doi:10.1145/3529320.3529330

[18] Klaus Pohl, Günter Böckle, and Frank Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Vol. 1. doi:10.1007/3-540-28901-1

[19] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. 2007. Software Engineering for Automotive Systems: A Roadmap. In *Future of Software Engineering (FOSE '07)*. 55–71. doi:10.1109/FOSE.2007.22

[20] Ina Schaefer. 2010. Variability Modelling for Model-Driven Development of Software Product Lines. In *Fourth International Workshop on Variability Modelling of Software-Intensive Systems (ICB-Research Report, Vol. 37)*. 85–92. http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf

[21] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond - 14th International Conference (Lecture Notes in Computer Science, Vol. 6287)*. 77–91. doi:10.1007/978-3-642-15579-6_6

[22] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore-A Model-Based Delta Language Generation Framework, Vol. P-225. 81–96. https://dl.gi.de/handle/20.500.12116/20956

[23] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework.* Pearson Education.