Karlsruhe Institute of Technology

Test, Validation, Analysis

Enabling Cross-Domain Consistency Preservation for Delta-Oriented Product Line Development

Master's Thesis
of

Simon Bothe

at the Department of Informatics
Institute of Information Security and Dependability
Test, Validation and Analysis (TVA)

Reviewer:	Prof. Dr.-Ing. Ina Schaefer
Second Reviewer:	Prof. Dr. Ralf Reussner
Advisors:	M.Sc. Dirk Neumann
	Dr.-Ing. Erik Burger

Completion period: 15.08.2025 – 16.02.2026

Zusammenfassung

Die Entwicklung moderner cyber-physikalischer Systeme erfordert die parallele Entwicklung von Modellen aus verschiedenen Domänen. Um das endgültige System abzuleiten, müssen die implementierten Teile aus allen Modellen zusammenpassen, was nur möglich ist, wenn alle Modelle während der parallelen Bearbeitung über alle Domänen hinweg konsistent bleiben. Ansätze wie Vitruvius bieten zwar Möglichkeiten, die Konsistenz zwischen den Domänen durch Konsistenzerhaltungsregeln zu erhalten, sie wurden jedoch noch nicht auf variable Systeme wie Produktlinien angewendet. Die Variabilität in diesen Produktlinien kann mit delta-orientierter Modellierung realisiert werden. Die Behandlung der Variabilität delta-orientierter Produktlinien mit Vitruvius erfordert allerdings eine Erweiterung des Vitruvius Ansatzes, welche noch in keiner bestehenden Arbeit untersucht wurde. Wir erweitern den Vitruvius Ansatz, um die Konsistenzerhaltung bei delta-orientierten Produktlinien zu ermöglichen und implementieren alle erforderlichen Tools, um unseren erweiterten Ansatz auf eine Fallstudie anzuwenden. Darüber hinaus untersuchen wir die Änderungen an bestehenden Konsistenzerhaltungsregeln, wenn Variabilität neu in Modelle eingeführt wird, deren Konsistenz sie bewahren. Wir führen eine Definition für die Korrektheit von Konsistenzerhaltungsregeln für delta-orientierte Produktlinien ein und schlagen eine Methode vor, um Konsistenzerhaltungsregeln auf ihre Korrektheit zu überprüfen. Wir diskutieren Probleme mit Konsistenzerhaltungsregeln, die nur in variablen Modellen auftreten, zum Beispiel wo Änderungen platziert werden sollen, oder wie Abhängigkeiten behandelt werden sollen und kombinieren die vorgeschlagenen Lösungen zu einem Framework. Dieses Framework wird hinsichtlich Funktionalität und Skalierbarkeit bewertet, indem es auf die Body-Comfort-System Fallstudie angewendet wird. Die Ergebnisse zeigen, dass unser Ansatz eine skalierbare Möglichkeit bietet, Konsistenzerhaltungsregeln für delta-orientierte Produktlinien zu erstellen und dass deren Anwendung die Konsistenz in diesen Produktlinien bewahrt.

Abstract

Engineering modern cyber-physical systems requires the parallel development of models from multiple different domains. To derive the final system, implemented parts from all models must fit together, which is only feasible if all models stay consistent across domains throughout parallel editing. While approaches such as Vitruvius provide ways to preserve consistency between domains via consistency preservation rules, they have not yet been applied to variable systems, such as product lines. Variability in these product lines can be realized with delta-oriented modeling. Handling the variability of delta-oriented product lines with Vitruvius requires an expansion of the Vitruvius approach, which is not explored by any existing work. We expand the Vitruvius approach to enable consistency preservation on delta-oriented product lines and implement all required tools for applying our expanded approach to a case study. Additionally, we explore the changes to existing consistency preservation rules when variability is introduced to the models they preserve consistency on. We introduce a definition for the correctness of consistency preservation rules for delta-oriented product lines and propose a way to test consistency preservation rules for correctness. We discuss problems with consistency preservation rules that only emerge in variability models, such as where to locate changes or how to handle dependencies, and combine the proposed solutions into a framework. This framework is evaluated regarding functionality and scalability by applying it to the Body-Comfort-System case study. The results show that our approach provides a scalable way to create consistency preservation rules for delta-oriented product lines and that applying them preserves consistency in these product lines.

Contents

List of Figures

List of Tables

Acronyms

SPL	Software Product Line
VSUM	Virtual Single Underlying Model
Var-VSUM	Variable Virtual Single Underlying Model
CPR	Consistency Preservation Rule
Product-CPR	Product Consistency Preservation Rule
Delta-CPR	Delta Consistency Preservation Rule
BCS	Body Comfort System
E/E	Electrical/Electronic
ECU	Electronic Control Unit
MDSD	Model-Driven Software Development
UML	Unified Modeling Language
MOF	Meta Object Facility

1. Introduction

Contribution of this Thesis

Developing modern cyber-physical systems requires collaboration from people of different engineering domains. Engineers in such projects are concerned with their own domains and might not have the required knowledge for other domains. But some changes in one domain can lead to changes in an entirely different domain, since they represent the same cyber-physical system from different viewpoints. While this means communication between the engineers is important, tool support for collaborating is just as important and can simplify the development process significantly. In this context, we will focus on the modeling aspect of such cyber-physical systems and, in particular, on preserving consistency between models of different domains (cross-domain). A tool support approach for preserving cross-domain consistency is Vitruvius [Kla+21]. It utilizes CPRs to define consistency relations between (or within) models and automatically enforces them when changes occur. Vitruvius and the reference implementation Vitruv have already successfully been applied to a multitude of different case studies, as can be seen in the Vitruv case study repository [1]. However, none of these case studies include models with variability, such as Software Product Lines (SPLs), and only use Product Consistency Preservation Rules (Product-CPRs) for fully specified product models. While other approaches for preserving cross-domain consistency in SPLs exist [SDD15][TVS21][LE10][LE12][KS16], they either require a developer to manually repair the inconsistency, or they are hand-tailored for specific models, making them inapplicable for most projects. Such SPL models can be implemented in various ways. One way is to implement them with delta-oriented modeling techniques [Sch10][CHS15]. Delta-oriented modeling constructs products by incrementally applying model changes, bundled in deltas, which include the model change operations. This thesis was initiated to explore the application of delta-oriented modeling techniques in multi-domain projects by enabling automated consistency preservation between the models. Instead of building separate products (product-based development), the goal is to enable building whole product families (platform-based development [Ant+14]). Such product families allow for quicker introductions of new

[1] https://github.com/vitruv-tools/Vitruv-CaseStudies (v3.0.0)

products, as well as lower costs in comparison to developing all products individually [DKS00]. This makes the approach especially attractive for manufacturers of configurable cyber-physical systems, such as cars, which is the reason why the BCS case study[Lit+13][NLS18] for cars is used to implement and evaluate our approach. The BCS is a model case study, consisting of a cyber-physical system modeled as an SPL. As the main result of this thesis, we implement our approach as a framework to be used for automatic consistency preservation on EMOF-based [OMG14a] delta-oriented models. We apply the framework to the BCS and implement a delta-oriented case study as proof-of-concept. Achieving the research goal requires the successful framework implementation and application to this case study, as well as some further analysis on its applicability to other case studies.

The novel approach taken by this thesis will be the introduction of Delta-CPRs to allow automatically preserving cross-domain consistency for all EMOF-based delta-oriented models. Delta-CPRs are explicit rules between deltas for preserving cross-domain consistency between the delta-oriented models. We connect Delta-CPRs to already explored Product-CPRs by proposing a practical and useful definition for consistency between the delta-oriented models. They are consistent if consistency rules defined on the products are fulfilled whenever the products are constructed by applying the delta operations. The thesis will conceptualize Delta-CPRs, analyze how their structure and contents relate to those of Product-CPRs, and provide insights into translating given Product-CPRs to Delta-CPRs based on the findings. This also includes analyzing the prospect of automatically translating Product-CPRs to Delta-CPRs. We explicitly do not aim to explore consistency between the domain model and the feature model, which is often used in SPLs to describe the feature space. While the evaluation is performed with a specific delta implementation, most concepts can be generalized and applied to different delta-oriented implementations. To address possible scalability issues regarding model correspondences used by Vitruvius, we will analyze the number of correspondences that are maintained when using Delta-CPRs. Moreover, this thesis will also introduce a *Delta-CPR Testing Kit*. It compares cross-domain consistency-preserving changes made by a Delta-CPR and a Product-CPR to test if, for given models and modifications, a Delta-CPR preserves cross-domain consistency as intended.

Structure of the Thesis

The development of the framework is divided into multiple steps. We start with the planning and implementation of a baseline product case study. This implementation includes CPRs defined between the product model instances. Next, we conceptualize Delta-CPRs by applying CPRs to a delta case study and solve occurring problems. The delta case study is then implemented with the Delta-CPRs defined between the delta model instances. In the next step, we evaluate the CPRs properties in the delta case study by comparing them to the CPRs properties in the baseline product case study. Lastly, we determine the feasibility of automatically translating Product-CPRs to Delta-CPRs and evaluate it based on the translatability of the CPRs in our implemented case studies. The techniques used to manually translate Product-CPRs from products to Delta-CPRs, as well as the evaluation procedures for Delta-CPRs, represent our framework to enable consistency preservation for delta-oriented SPLs. All steps to arrive at that framework are divided into multiple smaller activities themselves.

To plan the baseline product case study, we modify the existing BCS case study in Section 5.2.1. The case study in the most recent form [Lit+13][NLS18] is completely delta-oriented. To build a product case study, we need to apply the deltas to retrieve products and translate changes on the deltas to changes on these products. The products get implemented as model instances. The implemented Product-CPRs are designed to capture all connections between (and within) the model instances and provide a meaningful consistency expression.

Chapter 3 introduces Delta-CPRs and defines correctness for them to achieve a logical definition based on the enforcement of consistency rules. This also allows us to introduce the Delta-CPRs testing kit as a way to test for the correctness of these Delta-CPRs, if a product caste study with equivalent products and Product-CPRs is present. Based on the constructs defined by the generic delta metamodel, we analyze how Delta-CPRs can be constructed from Product-CPRs, while preserving correctness as we define it.

Based on the implemented Product-CPRs for the products, we design Delta-CPRs for a delta case study in Section 5.2.2. The delta case study consists of two model instances, which contain deltas to build all available products, and the implemented Delta-CPRs between the two model instances.

We evaluate the Delta-CPRs based on the BCS case study in Chapter 5. The application of CPRs to delta-oriented SPL development is considered successful if all Product-CPRs are correctly and entirely translated to Delta-CPRs. We also investigate differences between Delta-CPRs and Product-CPRs regarding scalability. This is significant, since SPLs provide good properties for scalability, and these properties might or might not transfer to Delta-CPRs used in combination with them. Additionally, we analyze to what extent our implemented Product-CPRs can be automatically translated to Delta-CPRs by deconstructing Product-CPRs into constructs and assigning these constructs either to automatically translatable or not automatically translatable. We use the knowledge from the automatic translatability analysis to conceptualize an automatic Product-CPR-to-Delta-CPR translator and implement the foundation for it.

2. Basics

2.1 Body Comfort System

To visualize problems and apply solutions, we are using the BCS [Lit+13] [NLS18] case study for the rest of this thesis. The BCS describes an embedded software system, consisting of Electronic Control Units (ECUs), which themselves are part of the vehicle Electrical/Electronic (E/E) architecture. The ECUs control functionality, such as door systems, wiper systems, or automatic headlights, in a vehicle. Each one of these ECUs is represented in a model by exactly one **component**. The BCS includes two different types of component models:

- **Component statemachine**: This model represents the behavior of a component. It is realized by a statemachine, that includes regions, states, and transitions with triggers, conditions, and behaviors attached to them. Statemachine regions can be included in a state as a subregion. The transitions might include signal inputs as triggers and signal outputs as actions. We will refer to such a model as a **statemachine model**.
- **Component interface specification**: This model represents a component interface to the rest of the vehicle. It includes a system with components in them, sensor input signals, actuator output signals, the corresponding ports that are used to send and receive these signals, and connectors between the ports. We will refer to such a model as a **component model**.

Visualization for both types of models is provided by the BCS and also shown in Fig. 2.1 as an example for the *Manual_PW* component. We will always refer to components when talking about models, instead of the ECUs they represent.

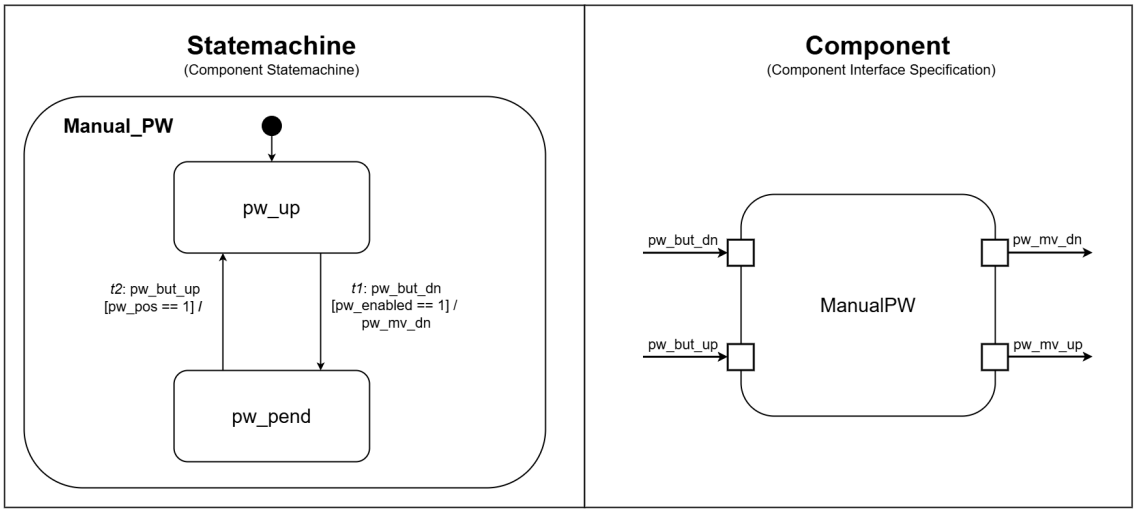

Figure 2.1: Partial statemachine model (left) and component model (right) for the *Manual_PW* component.

2.2 Model-Driven Software Development

2.2.1 MDSD Basics

Model-Driven Software Development (MDSD) is an approach to develop software systems that heavily relies on the usage of models [SVC06]. Models in our context follow the definition by Stachowiak [Sta73]. They represent an original but omit all but selected properties and attributes to simplify a system to the essential parts for a use case. This abstraction is useful, since reducing a system to the essentials allows for better understanding and platform-independent development. The structure and functionality of the system can be developed and represented in a model without specifying implementation details. In the context of MDSD, a complete model can be used to generate the bulk of the final implementation for multiple different platforms. This usually includes transformations, which transform the model structure and behavior onto selected platforms, saving time and resources in contrast to platform-dependent development. In the developing process, models are not just a documentation artifact but an integral part of the final software [SVC06]. Models are also used in testing and quality assurance, since they represent a digital version of the system with all essential parts contained, before the system is fully constructed.
To utilize MDSD, the models require enough expressiveness to capture all relevant properties and attributes. Plain Unified Modeling Language (UML) models are usually not expressive enough and either need to be extended or entirely new types of models have to be defined. Defining a new type of model always requires the definition of a **metamodel**. The metamodel describes what constructs and relations between these constructs exist in a model [BCW12]. Models, which only include constructs defined in a given metamodel, are called **model instances** of that metamodel. Model instances are a vital part of MDSD, since transformations can be defined on metamodels and then applied to any arbitrary model instance of that metamodel. Since metamodels are models themselves, they require a **meta-metamodel**, which defines which constructs exist in the metamodels. Such a meta-metamodel is provided by the **MOF** model [OMG14a] and its reference implementation, **Ecore**. Ecore

models are tree structures, where every node represents an EObject, which can hold attributes and references similar to objects in object-oriented programming languages. The MOF model is often used for the first of four modeling layers: the

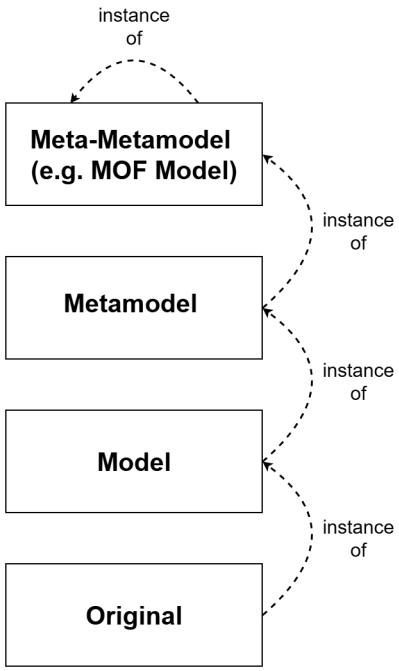

Figure 2.2: The four modeling layers, often used together with the MOF model. The MOF model is constructed in such a way that it can describe all of its own constructs. It is a model instance of itself.

meta-metamodel, the metamodel, the model, and the original (see Fig. 2.2). Applying this to the BCS yields the following allocation:

1. Since we implement the BCS in Ecore, the meta-metamodel will be the implementation of the MOF model.

2. The BCS includes two types of models and therefore two metamodels: one metamodel describing the constructs in a statemachine model (such as states and transitions) depicted in Fig. 2.3, and one metamodel describing the constructs in a component model (such as ports) depicted in Fig. 2.4. Both types of metamodels only contain constructs that were defined in the MOF model.

3. For the two metamodels, there are also two types of model instances: Statemachine models and component models. 2.1 already shows two such instances.

4. The original is a component (representing an ECU). The statemachine and component models both describe aspects of such a component.

The four-layer architecture will be the modeling basis for the remainder of this thesis. Since the original and the meta-metamodel are already given in our case, we will focus on the metamodel and model layers.

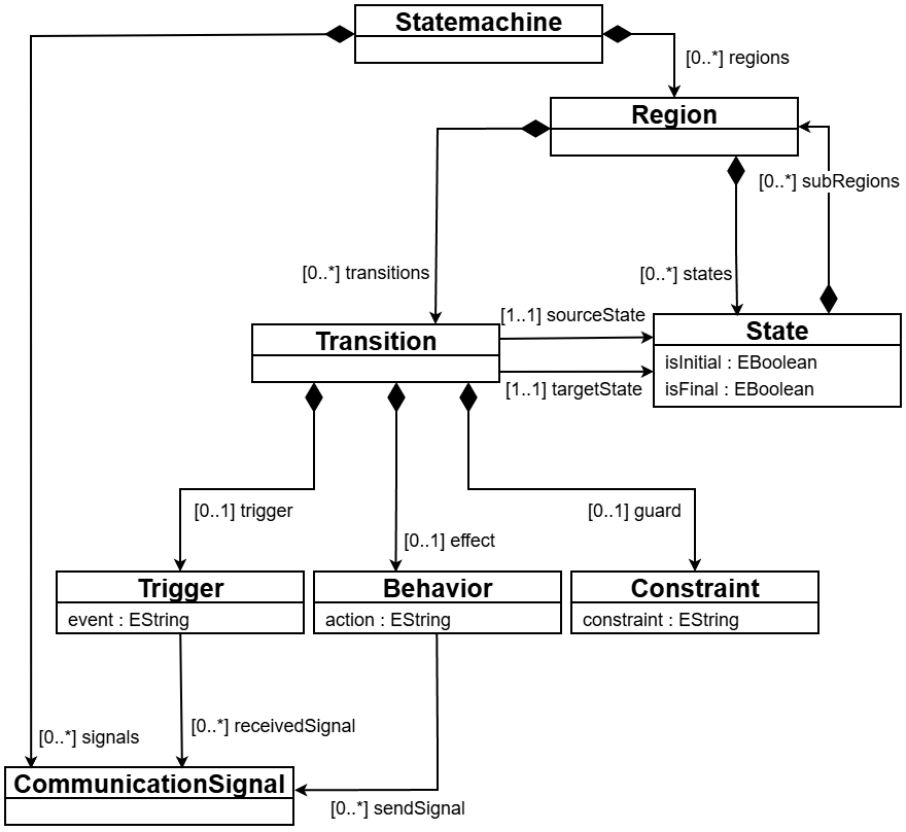

Figure 2.3: Metamodel for the statemachine domain. This visualization is missing an abstract `Identifier` type, which assigns every other class a unique identifier and name.

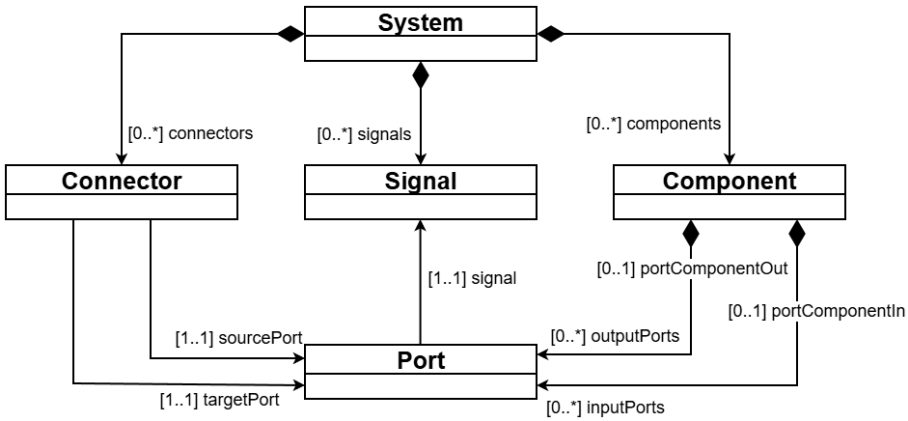

Figure 2.4: Metamodel for the component domain. This visualization is missing an abstract `Identifier` type, which assigns every other class a unique identifier and name, and the sub-component reference, which we did not use in our case study implementation.

2.2.2 Multi Domain Modeling

A **domain** in the MDSD context describes a bounded field of interest or knowledge [BCW12]. For our application, we represent a domain by a metamodel, since the metamodel determines which properties of the original can be contained in a model. So if a model is part of a certain domain, it translates to this model being an instance of the **domain metamodel**.
Engineering modern cyber-physical systems requires the parallel development of such model instances from different domains. The final system is derived by combining all these model instances. This way of modeling the overall system divided into different domains is called **multi-domain modeling**. The BCS is a multi-domain model, since both the statemachine models and component models are instances of different domain metamodels and include properties of the final system necessary to derive all components.

2.3 Software Product Lines

2.3.1 Software Product Line Basics

A prerequisite to the long-standing software engineering goal of software reusability is variability [Gur00]. To address variability in software-intensive systems, a **SPL** can be introduced. An SPL is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [CN01]. Particular **product variants** are generated by applying modifications based on a feature selection to this common set of core assets [CHS15]. This way, cloning the product gets completely avoided.
The set of managed features can be visualized by a **feature diagram**. Since the BCS is realized as an SPL, it serves as an example in this case as well. A subset of the BCS feature diagram is given in Fig. 2.5 For variability in software models, there are multiple ways to implement such an SPL[LKS16]: Annotative and compositional variability modeling approaches [Sch+12], as well as transformational variability modeling approaches such as **delta modeling** [Sch10][CHS15].

2.3.2 Delta-Oriented Modeling

Delta Modeling [Sch10][CHS15] for software models generates a product variant by incrementally applying model changes to a common core. These changes are encapsulated in **deltas**, which are collected in a **delta repository**. A delta gets applied to the model based on a selection or absence of one, or a combination of features. This feature selection in a delta is called an **application condition**. Deltas do not only include a list of changes called **delta operations** and an application condition but can also be partially ordered. Since this partial order determines which deltas get applied in which order, it is named the **application order** and gets realized by **after-clauses**. An after-clause in a delta is a list of deltas, which have to be applied before it can be applied itself. Fig. 2.6 shows an example for two deltas on the BCS statemachine domain. *Delta_Wiper* is part of the after-clause in *Delta_Clean*, since the addition of transitions and states required the existence of the *enable* and *disable* states, as well as the existence of the region itself. If the `Wiper` feature is selected, but no `Clean` feature, only *Delta_Wiper* will be applied. If both are selected,

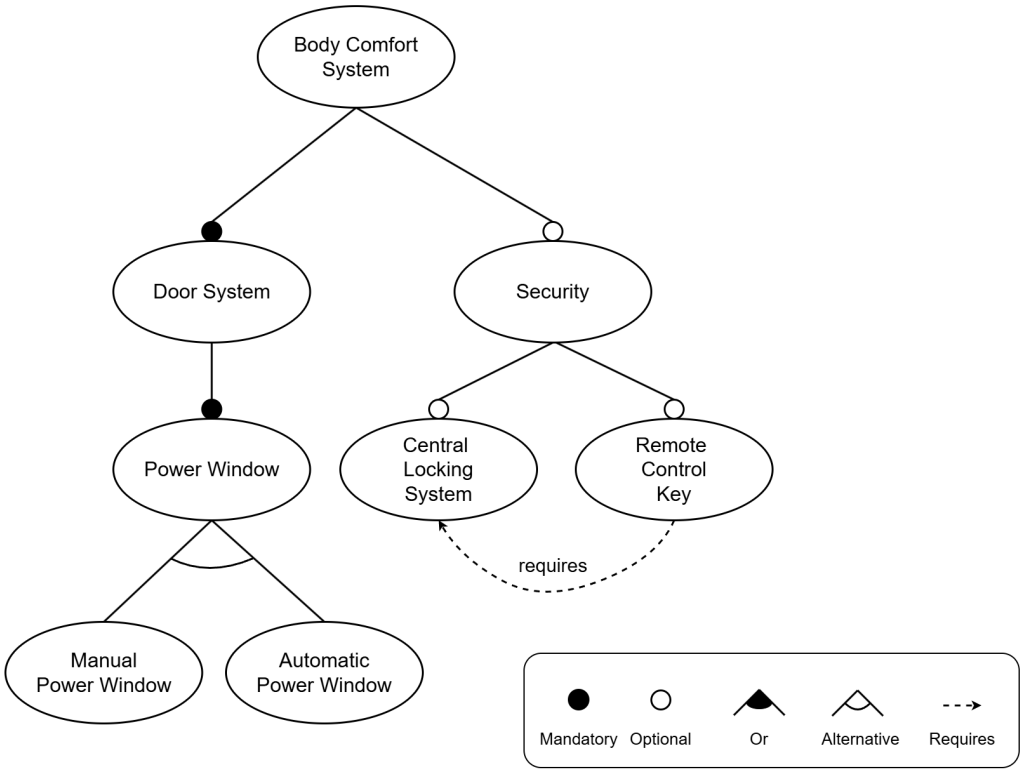

Figure 2.5: A subset of the BCS feature diagram. It includes mandatory and optional features. The selection of these features at construction time determines which product is constructed.

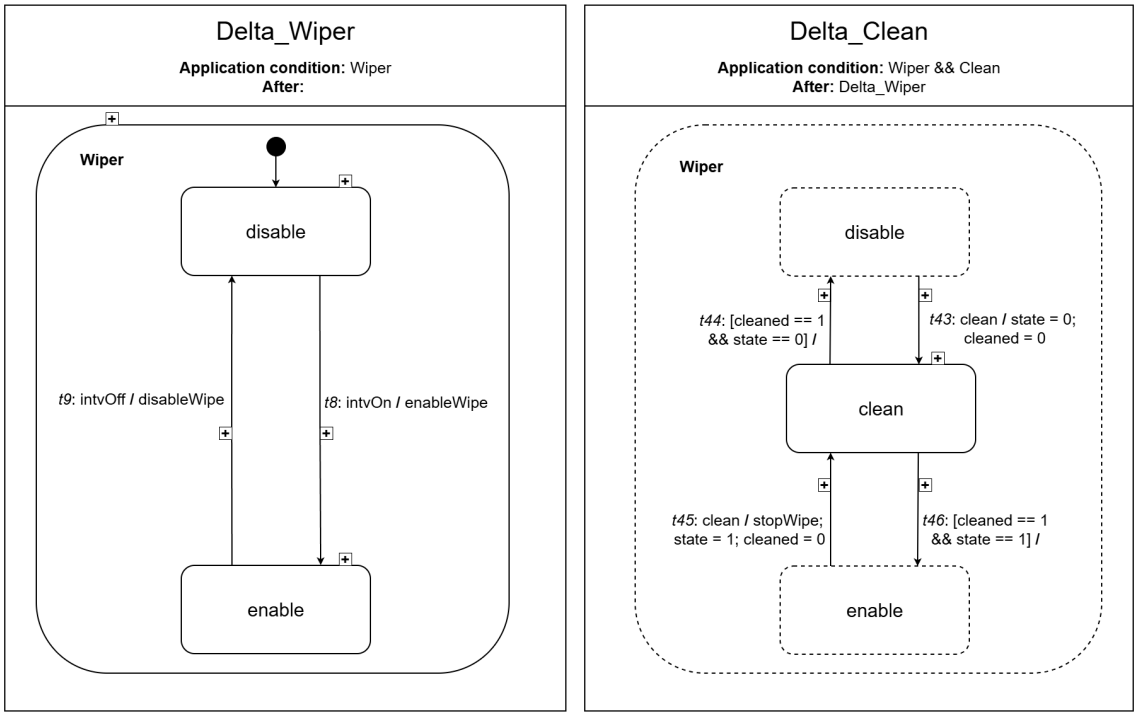

Figure 2.6: Two modified BCS deltas. *Delta_Wiper* adds a region, two states, and two transitions with triggers/behaviors into that region. *Delta_Clean* adds another state and four transitions into the region that was added by *Delta_Wiper*. This dependency can be seen in the after-clause.

Delta_Wiper will be applied first, and *Delta_Clean* after.
Software evolves to adapt to changing requirements and novel technologies [MSC14]. This also applies to SPLs [SV02]. By allowing an empty core model in Delta Modeling, evolution of the product variants translates solely to changes on deltas [SD10]. Changes on such deltas can themselves be represented as deltas again. Such deltas of deltas are called **higher-order deltas** [LKS16]. A higher-order delta encapsulates changes (such as addition, removal, or modifications) of deltas or their contained operations. The BCS provides higher-order deltas as change scenarios on an existing delta repository. Since delta-oriented modeling helps to build an extensible platform in the form of an SPL, we call development with the usage of deltas **platform-based development**[Ant+14]. This stands in contrast to **product-based development**, where individual products are built.
To implement delta-oriented modeling, we use the **Generic Delta Metamodel**. The generic delta metamodel models all constructs necessary for creating deltas, delta operations, and delta repositories. For delta operations, the generic delta metamodel provides possibilities to set/unset model root `EObjects`, `EObject` references, and `EObject` attributes. We can use a dialect creation mechanism similar to the one introduced by DeltaEcore [SSA14] to generate domain metamodels for the statemachine/component domain and create **delta statemachine model** instances and **delta component model** instances.

2.4 Consistency Preservation

2.4.1 Consistency in Models

When working with model instances, it is desirable to preserve certain model properties at all times. This includes when model changes occur. Unifying all these properties, which should be preserved at all times, yields a **consistency** concept. A model instance is consistent if, and only if, all these properties apply to it. Taking measures to ensure the consistency of a model instance is called **consistency preservation**. In case the changes in a model instance require consistency preservation in the same instance, the type of consistency preservation is called **intra-domain consistency preservation**. Intra-domain consistency detection can be implemented by formulating model constraints in Object Constraint Language (OCL)[OMG14b], but we will explore possibilities to implement intra-domain consistency preservation in chapter 3.3 of this thesis. An example of intra-domain consistency for the statemachine domain (Fig. 2.3) of the BCS could be formulated in natural language: *Every statemachine region in the model must have an initial state defined.*

When working with model instances in multi-domain modeling, consistency can refer to model properties that should be preserved over multiple model instances at once. In case the changes in a model instance require consistency preservation in a different instance (potentially of a different metamodel), the type of consistency preservation is called **cross-domain consistency preservation**. An example of cross-domain consistency preservation from the statemachine to the component domain of the BCS could be formulated in natural language: *Every input signal used as a transition trigger in the statemachine region of a component requires an input port at that component for that signal.*

2.4.2 Vitruvius Approach

The process of consistency preservation can be automated with **CPRs** [Kla16]. They are a part of the **Vitruvius** approach [Kla+21] and its reference implementation **Vitruv** [Kla+25]. Vitruvius provides a framework for view-based software development that can automate consistency preservation for model instances. To achieve this, Vitruvius combines multiple model instances with CPRs between them into a **VSUM**. For this thesis, we will confine ourselves to the **reaction** language [Kra17] implementation of CPRs and use reaction interchangeably with CPRs. Reactions are explicit rules for preserving cross-domain consistency between the models contained in the VSUM. Reactions trigger when an **original change** in the **reaction source** model instance occurs and reacts with **consequential changes** in a **reaction target** model instance. They are mainly designed toward cross-domain consistency preservation with different source and target model instances but can also be used for intra-domain consistency preservation with the same source and target model instance. In the context of the four-layer modeling layers seen in Fig. 2.2, reactions are always defined on the metamodel layer but applied to the model layer (to the model instances of these metamodels). An example VSUM with reactions can be seen in Fig. 2.7. Vitruvius uses **correspondences** to keep track of the changes in the

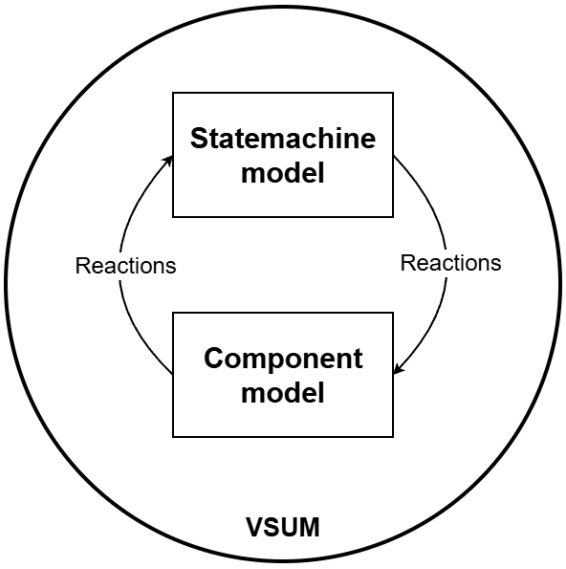

Figure 2.7: A VSUM combining the model instances of a statemachine model and a component model. The reactions between them ensure cross-domain consistency preservation. In this example, no intra-domain consistency-preserving reactions are defined.

source model instance and their consequential changes in the target model instance. They are important, since existing correspondences from earlier reaction applications might be necessary to locate a consequential change for a future reaction application. Since an implementation like Vitruv has to manage these correspondences in memory, they can become an important factor for scalability.

Pseudocode examples for reactions and the usage of correspondences for the BCS are given by Listing 2.1 and Listing 2.2.

```
1 reaction: if region added into statemachine {
2   add component into system
3   create correspondence between region and component
4 }
```

Listing 2.1: Region-To-Component reaction

```
1 reaction2: if signal set in trigger {
2   add inPort into component corresponding to trigger.region
3   create correspondence between trigger and inPort
4 }
```

Listing 2.2: Trigger-To-Input-Port reaction

The Region-To-Component reaction (Listing 2.1) is executed when a statemachine region is inserted into a statemachine (line 1). The consequential change is the insertion of a component in the system in the component domain (line 2). Additionally, a correspondence between the region and component is created for later usage in the Trigger-To-Input-Port reaction (line 3).
The Trigger-To-Input-Port reaction (Listing 2.2) is executed when a signal is set in a transition trigger in the statemachine domain (line 1). The consequential change is the insertion of an input port for that signal (line 2). To locate the component in which the port must be inserted, a correspondence lookup is performed for the region in which the trigger is present (line 2). Additionally, a correspondence between the trigger and input port is created for later usage (line 3). A visualization of the Trigger-To-Input-Port reaction application on the BCS is shown in Fig. 2.8. The addition of a trigger in the statemachine model instance leads to the addition of an input port in the component model instance via a CPR application. If the CPRs are defined for models that are not representing an SPL (like the ones in Listings 2.1 and 2.2), we call them a *Product-CPRs*, since they preserve consistency between fully specified products.

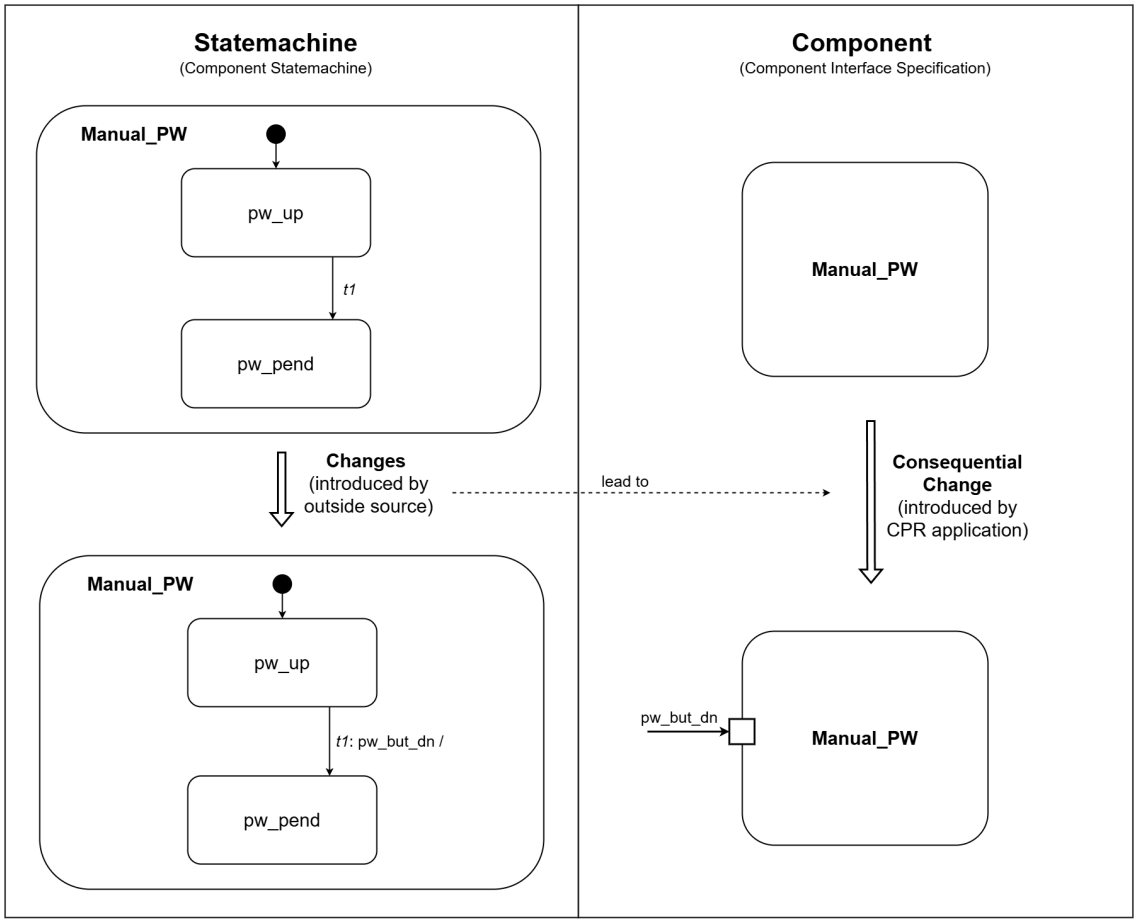

Figure 2.8: Example application of a Trigger-To-Input-Port reaction (Listing 2.2) on a BCS example. The top statemachine/component model instances are the original model instances before changes. The model instances on the bottom are the model instances after the changes.

3. Design

This chapter aims to introduce Delta-CPRs and solve problems connected to their usage. We will also discuss the translation from Product-CPRs to Delta-CPRs, and how much of that process can be automatically performed.

3.1 Delta Consistency Preservation Rules

In contrast to Product-CPRs, we introduce **Delta-CPRs**. A Delta-CPR is a CPR, defined for models that are representing an SPL with Delta Modeling. In this thesis, we will restrict ourselves to the case in which the source and target model both represent an SPL with Delta Modeling. VSUMs with only Delta-CPRs and no Product-CPRs defined in them are called **Variable Virtual Single Underlying Models (Var-VSUMs)** The Delta-CPRs will trigger when delta operations in the delta repository of the source model change and react with consequential changes of delta operations in the delta repository of the target model. This represents the application of a higher-order delta as a trigger and the application of another higher-order delta as the consequential change. An example reaction for the change performed similar to that in Fig. 2.8, but expressed as a higher-order delta application, can be seen in Fig. 3.1

3.2 Challenges for Delta Consistency Preservation Rule Application

Fundamentally, the change from Product-CPR to Delta-CPR only means changing the metamodels and model instances on which they operate. But this change alone brings a set of challenges that have to be addressed to arrive at a functional and useful concept of consistency preservation on delta-oriented SPLs. These challenges mainly stem from new constructs and the overall construction logic of a product (such as application conditions or application order) not present in fully specified products. The following subsections discuss one challenge each and how they can be solved. For implementation details on these solutions, refer to Chapter 4.

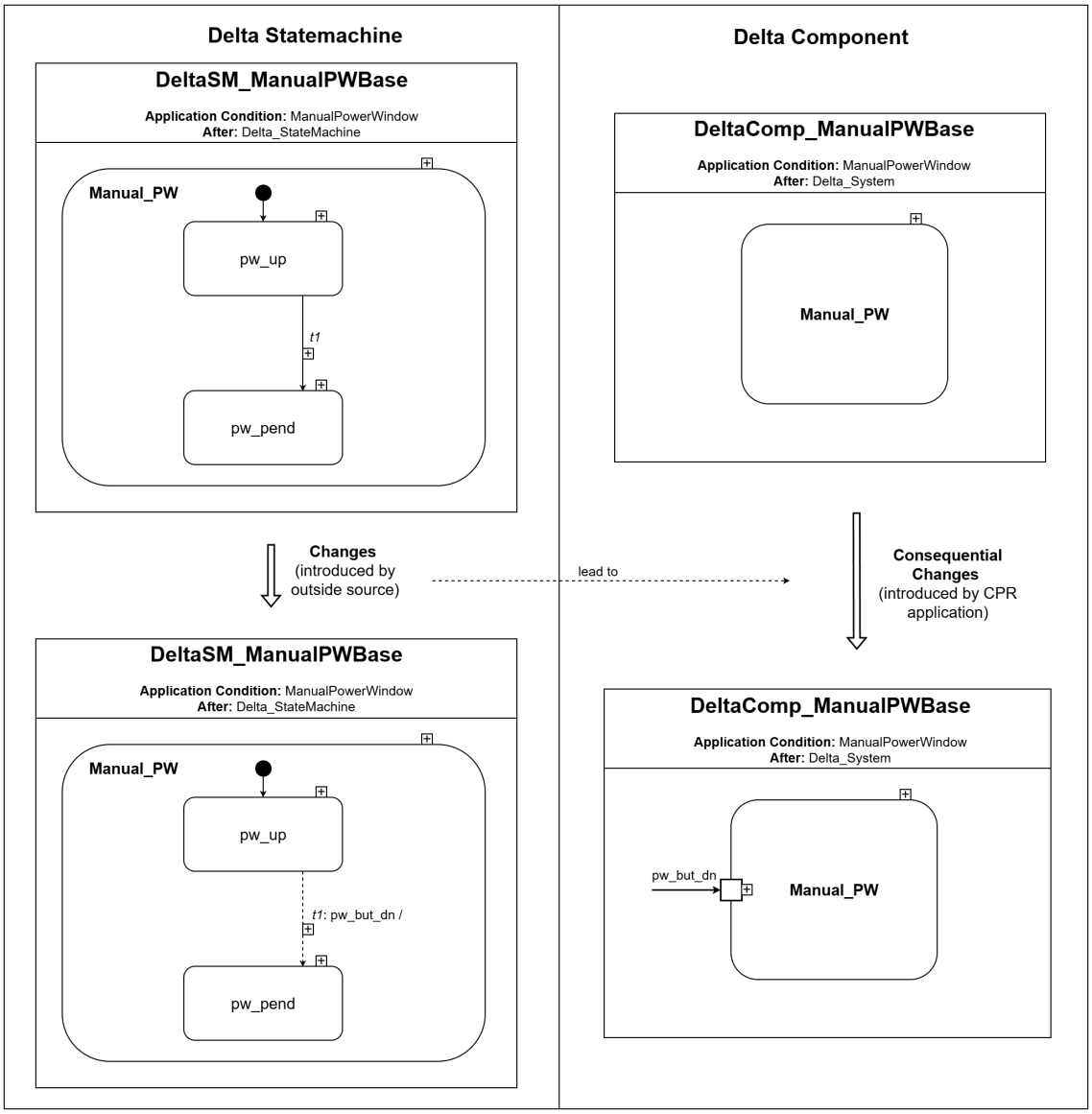

Figure 3.1: Example application of a Delta-CPR reacting to a change as in Fig. 2.8, but expressed as a higher-order delta application on a BCS example. The top delta statemachine/component model instances are the original model instances before changes. The delta model instances on the bottom are the model instances after the changes.

3.2.1 Operation Mapping Problem

Firstly, we have to take a look at how operations on products translate to operations on deltas. The generic delta metamodel already provides all necessary constructs for this (see Section 2.3.2), but there are different ways to interpret changes on a product. As an example for reference removal, consider a situation with an existing statemachine region in the statemachine domain of the BCS. For deltas, this translates to an existing `AddStatemachineRegionOperation`. If an outside change removes said statemachine region, it can either be interpreted as the removal of this `AddStatemachineRegionOperation` or as an addition of a `RemoveStatemachineRegionOperation`. Both delta interpretations lead to the same product when applied. The same logic applies when changing a reference: It can either be interpreted as a `ChangeOperation` or as a `RemoveOperation` followed by an `AddOperation`.
For the removal of an existing reference, the effects of either interpretation depend on the products in our SPL. If the majority of products still include that reference, adding explicit `RemoveOperations` requires a smaller amount of changes. If the majority of products do not include the reference, it is the other way around. In this case, removing the `AddOperation` and adding explicit new `AddOperations` to deltas for products that include them requires a smaller amount of changes. In general, the goal should be to minimize delta operations to keep the changes as small and understandable for developers as possible. This principle should also be applied to consequential changes of Delta-CPRs.
For the change of an existing reference, the generic delta metamodel is designed to only provide `SetOperations`/`UnsetOperations` for single-value references and `AddOperations`/`RemoveOperations` using a list for multi-value references. This means for single-value references, we always use a `ChangeOperation` that overrides the previous value. For multi-value references, we will use a `RemoveOperation` followed by an `AddOperation`, but in theory the generic delta metamodel could also be used with index referencing to change a multi-value reference directly.

3.2.2 Showing Functional Correctness

One of the most important questions is how to arrive at a functional and useful concept of consistency preservation for delta-oriented SPLs. Formally, the consistency and consistency preservation concepts from Vitruvius are valid but should be expanded to become useful for keeping consistency on the constructible products. For this purpose, we define the **functional correctness** of a Delta-CPR.

Let:

- C be the set of all possible changes on a product model instance of one domain.
- CPR^p be a Product-CPR that defines consistency preservation between two domains and is executed when a change $c \in C$ is applied.
- PL be the set of all possible products contained in a product line before changes.
- `apply(`cpr`,` c`,` m`)` be a function that applies the changes c, and the consequential changes cpr reacts with, to the model instance m.

- r^d be a delta repository to construct every product $p \epsilon PL$.
- `construct`(p, r) be the function to construct a product p from the delta repository r.
- c^d be the change c formulated on deltas as a higher-order delta (see Section 3.2.1).
- p' be the new iteration of the product p, after changes were applied and CPRs were executed.

Definition 3.1 *Then: A Delta-CPR CPR^d is functionally correct regarding CPR^p, iff:* $\forall c \in C \; \forall p \in PL$:
`apply`$(CPR^p,\ c,\ p) \equiv$ `construct`$(p',$ `apply`$(CPR^d,\ c^d,\ r^d))$

Definition 3.1 formalizes a rather intuitive idea. Whenever a consequential change is performed on a product by a Product-CPR, the result should be equivalent to the construction result of the SPL for that product, after a Delta-CPR performed consequential changes based on an equivalent original change. This results in a one-to-one mapping between Product-CPRs and Delta-CPRs. While the definition might be extendable to allow for one-to-many or even many-to-many mappings, we will limit ourselves to Definition 3.1 for the sake of simplicity.

Proving the functional correctness of a Delta-CPR requires the existence of a corresponding Product-CPR. But even in that case, it is unclear if proving the functional correctness is possible at all. Since reactions translate to arbitrary Java code [Kla+25], formally proving functional correctness in all cases would require solving the halting problem [Str65]. However, we can test for functional correctness in specific cases. Checking the functional correctness of a selected Delta-CPR for a specified change and product is useful too. Applying the same principle as with unit testing, we can check the functional correctness of a Delta-CPR for a set of inputs and gain a certain level of confidence in their correctness without formally proving it. To test the functional correctness of a Delta-CPR, we propose the **Delta-CPR Testing Kit**. The testing kit application is visualized on a BCS example in Fig. 3.2 The Delta-CPR Testing Kit applies changes to the product and delta domain (addition of wiper), triggers the CPRs, and constructs the product from the resulting deltas (Product 1*). After the construction process, it compares both products of the same type. If the comparison outputs that both are the same, we gain confidence in the Delta-CPR's functional correctness. Otherwise, we know the Delta-CPR is not functionally correct.

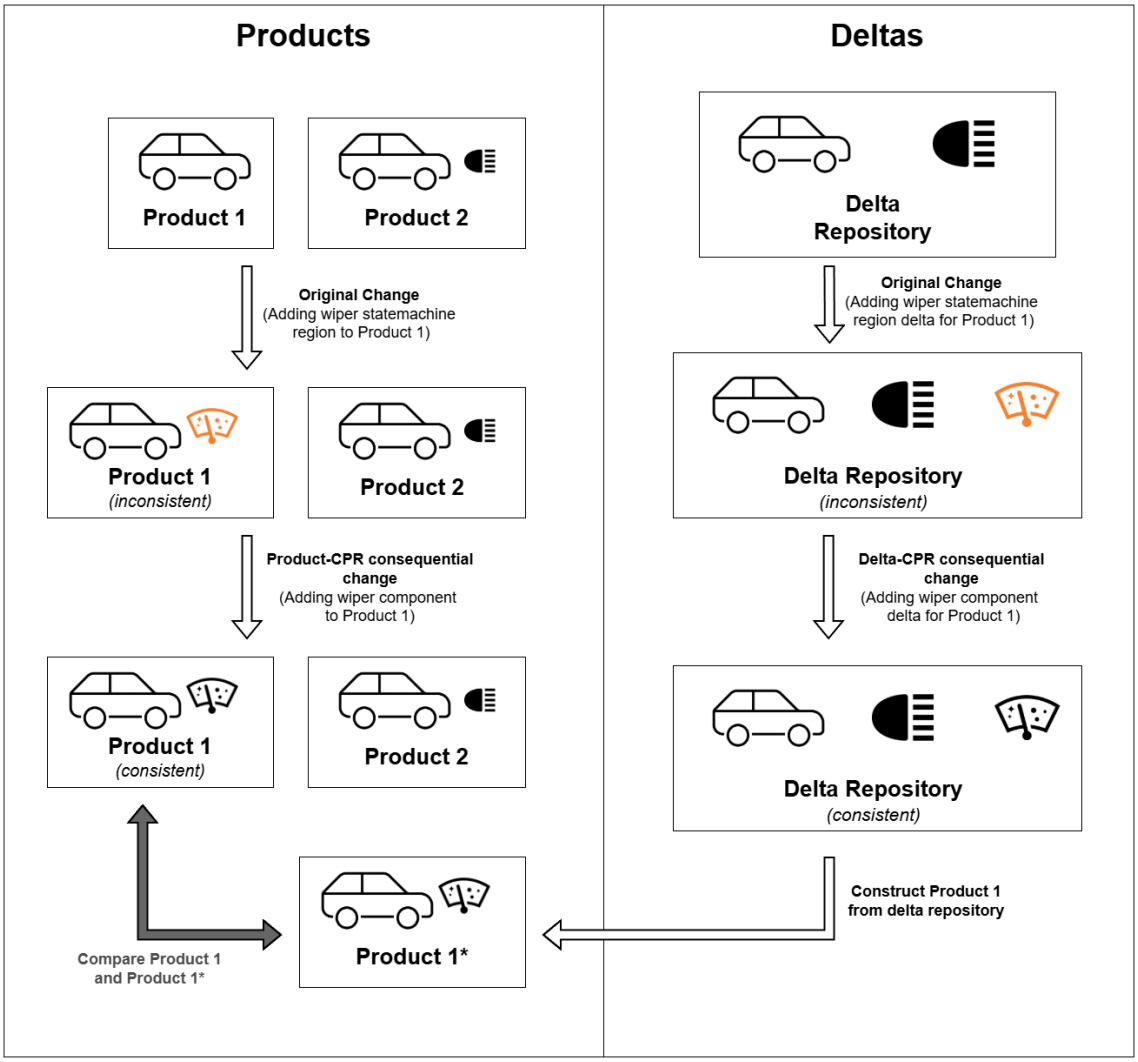

Figure 3.2: A Delta-CPR Testing Kit example for the addition of a wiper statemachine region in the BCS. The CPRs create components corresponding to statemachine regions that were added. The result of the comparison in gray is the final testing result.

3.2.3 Operation Dependency Problem

When defining consequential changes for reactions on deltas, delta operations often have a dependency to other delta operations. For example, with the BCS, consider the addition of an `AddPortOperation`. This operation needs to be inserted into a component, hence it has a dependency to an `AddComponentOperation`. If the component is not added, no port can be added either. We can see such a reaction for products in Listing 2.2. When working on products, Ecore allows us to navigate from a trigger to the region containing it by ascending the containment tree structure (see Fig. 3.3). This can be used with the correspondence model to

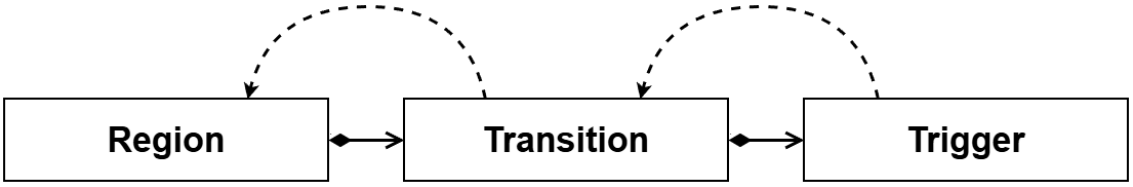

Figure 3.3: Visualization of the navigation from the trigger to the container region as used in Listing 2.2

retrieve the component and insert the port, as seen in line 2 of the listing. However, when working on deltas, this approach does not work anymore. Instead of being contained in each other, the responsible delta operations are contained in potentially different deltas (see Fig.3.4). The easiest but most inefficient way would be

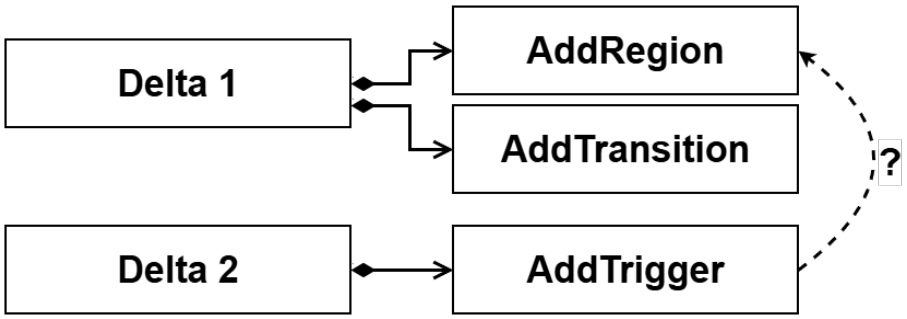

Figure 3.4: Visualization of the necessary navigation from an `AddTriggerOperation` to its containment `AddStatemachineRegionOperation`. This is needed to translate a reaction like Listing 2.2 to deltas.

searching the complete delta repository for the operation. This approach will always work. But if all deltas are **well-formed**, we can interpret the dependencies between delta operations as dependencies between deltas. Deltas inside a delta repository are well-formed if they can be applied in an arbitrary order, as long as a valid feature combination is provided and the application order of the deltas is not violated, and the resulting product is always the same. By having all deltas be well-formed, we can ensure that all delta operation dependencies are resolvable by applying just the delta application order alone.
Whenever we search for a delta operation, well-formed deltas allow finding the operation by searching only through the delta operations included in the search-starting delta or deltas, which have to be applied before the search-starting delta as specified by the application order. For example, we can look at the situation in Fig. 2.6. If

Delta_Wiper and *Delta_Clean* are well-formed, then *Delta_Clean* must be applied after *Delta_Wiper* per application order, since a transition is always added in an existing region, between existing states. This tells us that the searched `AddStatemachineRegionOperation` and `AddStateOperations` must either be in *Delta_Clean* (which they are not in this case) or in a delta included in the after-clause of *Delta_Clean*. In general, the operation could also exist in deltas in the after-clause of *Delta_Wiper* or in any other delta that gets applied before *Delta_Clean* per application order.

If all deltas are well-formed, we only have to search through a relevant subset of deltas in the delta repository to find operations and resolve dependencies. This can reasonably be assumed for deltas created by humans, but deltas created by Delta-CPRs must also be well-formed. We call this type of subset search through deltas **dependency search**. Section 3.2.4 will include strategies to set the after-clauses of deltas created by consequential changes to ensure they are well-formed.

3.2.4 Operation Localization Problem

The **operation localization problem** describes the problem of where the Delta-CPR is supposed to locate the consequential changes. Looking at the BCS and Listing 2.1 as an example, we can see that one clearly defined component is added as a consequential change. But when translating that same consequential change in Delta-CPRs, it is completely unclear in which delta the `AddComponentOperation` is supposed to be located . To determine the location of delta operation additions, we propose two possible solutions: **Functional Delta Assignment** and **Delta Mirroring**.

- *Functional Delta Assignment:* Always applicable and directly aims to ensure all deltas are well-formed. Changes in one delta in the source domain get translated into one or multiple deltas in the target domain. For every delta operation addition in the target domain, a delta with the application condition of the source delta gets created. The after-clause for this newly created delta then gets set on the deltas, which include delta operations the added delta operation depends on. This can be done efficiently by using the searching technique introduced by Section 3.2.3. By constructing deltas this way, we can ensure that functional consistency holds for the Delta-CPR (since the changes only get applied in parallel to the source delta application conditions), and they are always unambiguously applicable and well-formed (since all dependencies are resolved by the application order). The big problem with this approach is the fragmentation of deltas, because it produces deltas with only a single delta operation in them. To combat this, we can use **delta compression**. Delta compression simply merges deltas with the same application condition and application order to reduce the number of deltas to a minimum while still ensuring they are well-formed and get applied under the same conditions as before.

- *Delta Mirroring:* Applicable to cross-domain consistency preservation if both domains have **parallel dependencies**. Given the addition of a delta operation Op_s in the source domain and a Delta-CPR that adds a delta operation Op_t in the target domain in response: The source and target domain have parallel dependencies if the existing Delta-CPRs ensure all the operations that Op_t depends on, have been added to the target domain as consequential changes when the operations Op_s depends on, got added. This is obviously not given for every domain with every Delta-CPR, but can always be achieved by restricting the possible Delta-CPRs. For domains such as the BCS, this restriction still allows for translating all Product-CPRs while maintaining parallel dependencies in the statemachine and component domain.
  We will call a delta operation in the source domain, whose addition triggers a delta operation addition in the target domain, a **cross-domain operation**. We can create a **mirror delta** for every delta in the source domain that includes one or more cross-domain operations. All target delta operations, created by adding the cross-domain operations, can then be grouped in their corresponding mirror delta. The goal is copying the intent of the source delta grouping of delta operations to the mirror delta grouping of delta operations. Mirror deltas share the same application condition as the delta they originate from. Consider a delta d_x with cross-domain operations inside and the corresponding mirror delta d_x^m. The delta application order of d_x^m must ensure that it is applied after all mirror deltas d_{x-1}^m,...,d_0^m that were created from all deltas d_{x-1},...,d_0 that d_x has a dependency to. As long as parallel dependencies are maintained, this guarantees well-formed deltas, since all dependencies of delta operations contained in a newly added mirror delta have to be present in already existing mirror deltas.

It is important to note that a target model delta repository built with delta mirroring can be translated to one using functional delta assignment, but not vice versa. The reason for this is the grouping of delta operations in the source model deltas. Which was used to directly derive deltas for delta mirroring but completely omitted when using functional delta assignment. This user intent gets lost when applying the Delta-CPRs with functional delta assignment. For that reason, we choose delta mirroring for our reference implementation of the BCS. Consider the addition of three deltas in the delta statemachine domain as seen in Fig. 3.5 with the reaction from Listing 2.1 and 2.2 translated to deltas. We will visualize all proposed solutions that locate the consequential changes in the delta component domain.
Delta mirroring maintains the number of deltas and has the `AddPortOperations` separated, since every delta includes at least one cross-domain operation. The resulting deltas in the delta component domain can be seen in Fig. 3.6.
Functional delta assignment without compression adds the addition of the input ports into one delta each. The resulting deltas in the delta component domain can be seen in Fig. 3.7.
Functional delta assignment with compression compresses the addition of the input and output port into one single delta, since both only depend on the existence of the component and get applied under the same application conditions. The resulting deltas in the delta component domain can be seen in Fig. 3.8.

The three solutions can be categorized in *no groupings* for functional delta assignment without compression, *maximum amount of groupings* for functional delta assignment with compression, and *groupings per user intend* for delta mirroring.

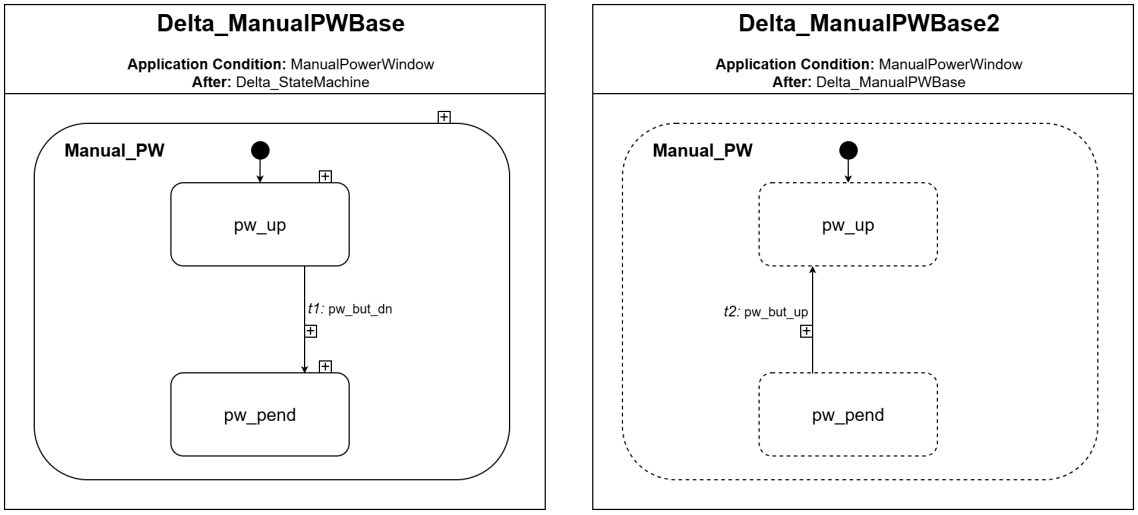

Figure 3.5: The left delta adds a new region, two states, and a transition with a trigger. The right delta is applied after and adds a transition with a trigger between the states added by the left delta.

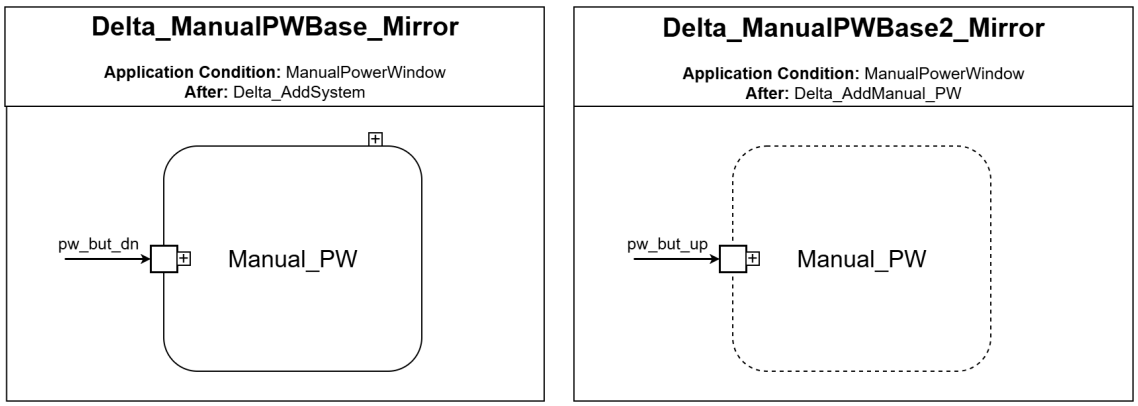

Figure 3.6: The resulting deltas in the delta component domain for adding the deltas depicted in Fig. 3.5 with delta mirroring.

3.2.5 Existing Element Dependency Problem

Sometimes Product-CPRs should have different behavior depending on which elements already exist in the source or target model. An example of this is Listing 2.2. If multiple transitions in a statemachine region have the same signal as a trigger, this reaction would create the same input port multiple times. But we only want the input port one single time, meaning the reaction should do nothing if the input port already exists for that component in the target domain. Listing 2.2 completely omits this behavior for simplicity reasons, but it could be included in the real-world reaction by simply checking the existing input ports of the component.

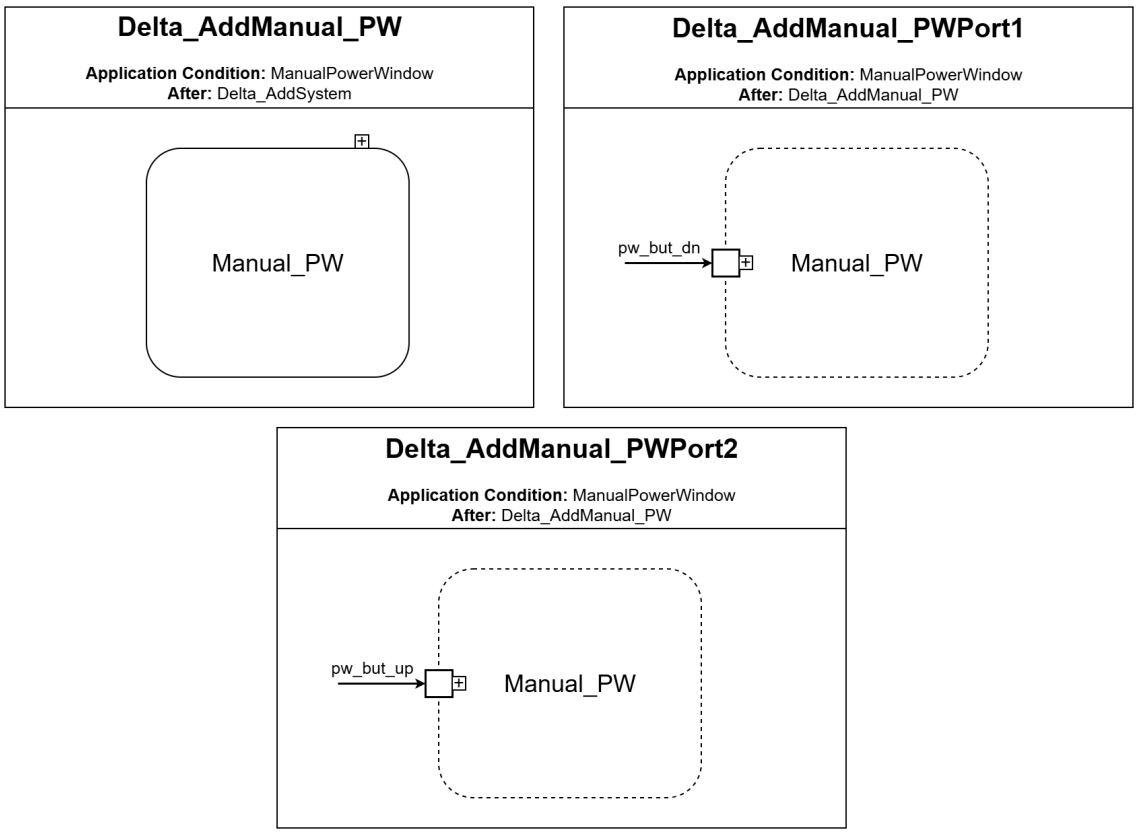

Figure 3.7: The resulting deltas in the delta component domain for adding the deltas depicted in Fig. 3.5 with functional delta assignment without compression.

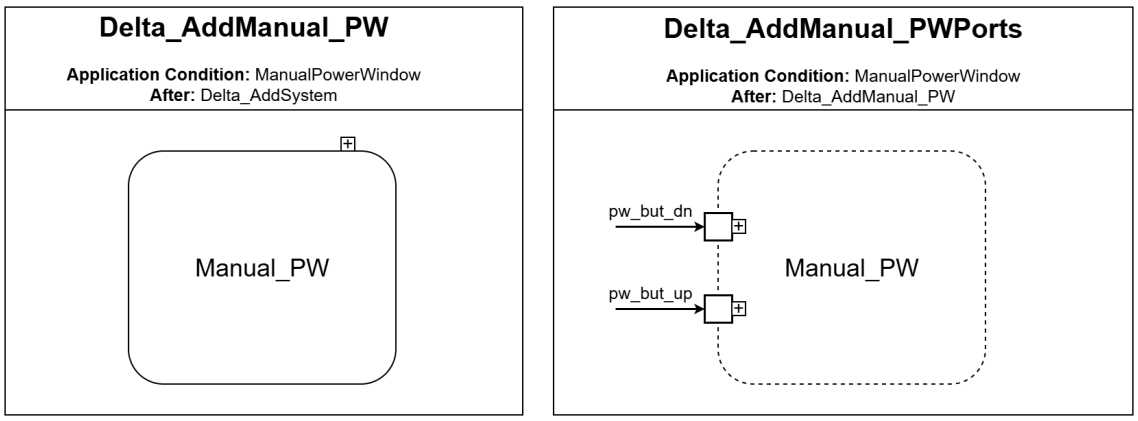

Figure 3.8: The resulting deltas in the delta component domain for adding the deltas depicted in Fig. 3.5 with functional delta assignment using compression.

Translating such a Product-CPR to a Delta-CPR makes the checking-for-port mechanism significantly more complicated. Consider the addition of the two deltas in Fig. 3.9 into the delta statemachine domain. We assume a reaction similar to Listing 2.2 would be executed, with the change of reacting to behavior additions with output ports. It is completely unclear if just one of them gets applied, both get

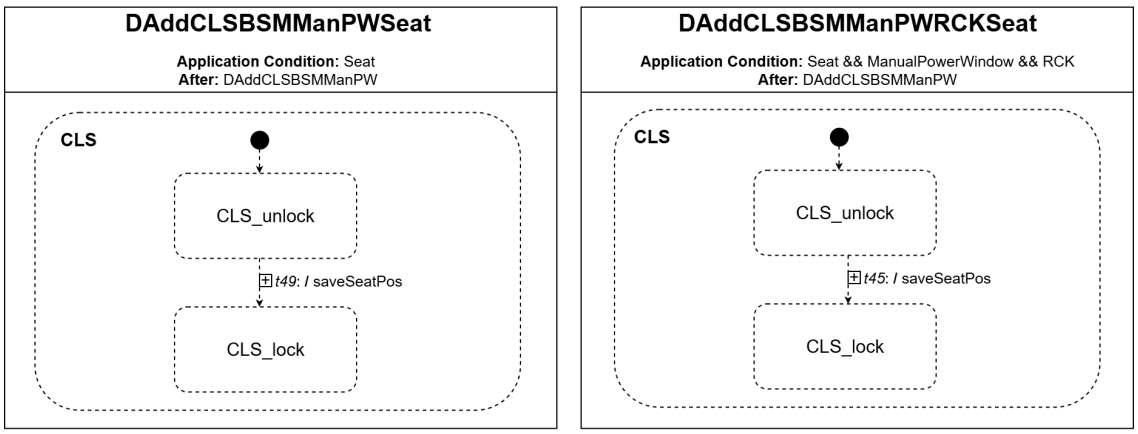

Figure 3.9: Two deltas adding the same behavior signal to two different transitions between the same states.

applied, or in which order the two get applied, since the features `Seat`, `ManualPowerWindow`, and `RCK` are completely independent of each other. If either one or both get applied, we want the resulting product to include one single output port in the `CLS` component for the signal `saveSeatPos` inside the component domain. But since the Delta-CPR is applied when the delta operations are added, and not when the delta operations are applied, the Delta-CPR is unable to check for existing ports.
To solve this problem, it is necessary to interpret the existing element dependency problem not as a problem the Delta-CPRs solve, but as a problem that is solved during construction time. The generic delta metamodel is designed to handle these existing element dependencies by uniquely identifying `EObjects` in the constructed model instances via unique identifiers and permitting duplicates of those `EObjects` that share the same unique identifier. During `AddOperation` applications, if an `EObject` with the same unique identifier is already contained in the model, the delta operation is ignored. This allows us to have the Delta-CPRs add multiple `AddOperations` for the same `EObject` and never consider if they could already exist during construction time (see Fig. 3.10). In this example, we just have to ensure that the output ports for `saveSeatPos` in the `CLS` component, added by applying either one of the mirror deltas, have the same unique identifier. If this is ensured, applying either one or both will result in just one single port in the constructed product component domain. Applying neither will result in no port. This is the expected and desired behavior for the translated Delta-CPR, to achieve functional correctness.

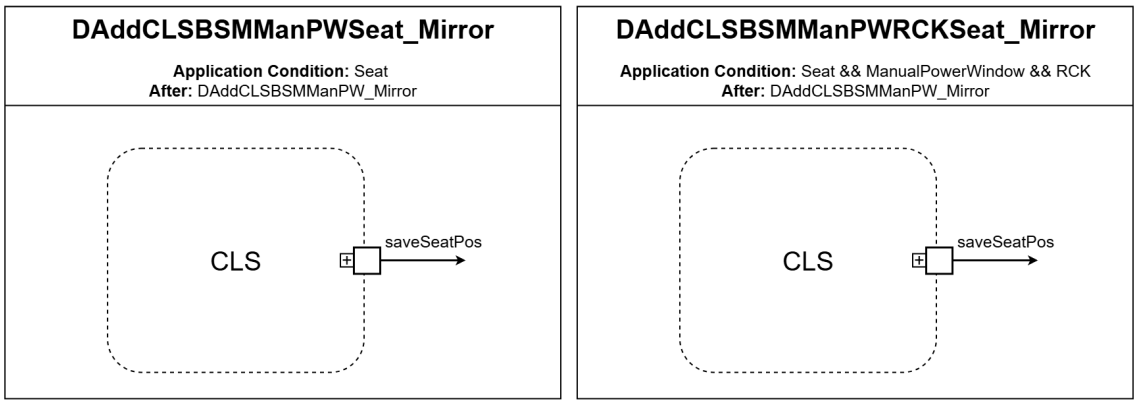

Figure 3.10: The created deltas for the addition of the deltas in Fig. 3.9 when applying the delta translated reaction for output port creation with delta mirroring.

3.3 Intra-Domain Delta Consistency Preservation Rules

While the focus of Vitruvius and CPRs is on cross-domain consistency preservation, they can also be applied to intra-domain consistency preservation. When setting the same source and target domain, Delta-CPRs can effectively ensure **intra-domain delta consistency preservation** in that domain. Most challenge solutions introduced in previous chapters still apply to intra-domain, with one exception: Delta mirroring is no longer possible, since there is just one single domain. In the case of intra-domain delta consistency preservation, the localization problem must be solved with functional delta assignment (with or without compression).

Intra-domain delta consistency preservation can be applied to the BCS for visualization. The Delta-CPRs are defined from and to the domains as shown in Fig. 3.11. In the delta component domain, one example of intra-domain Delta-CPR would be

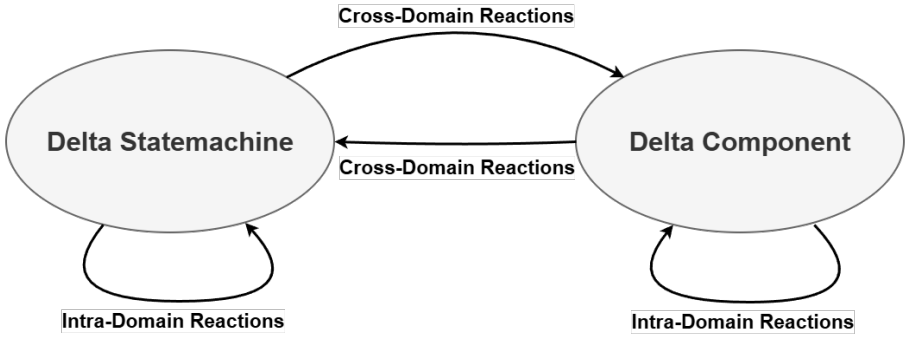

Figure 3.11: The BCS case study domains with the reaction defined between them. The case study includes cross-domain delta consistency preservation, but also intra-domain delta consistency preservation.

the addition of connectors. If the constructed product includes an input and an output port for the same signal in the component domain, it should also include a connector to connect these two ports in the component domain. The reaction has to keep track of `AddOperations` for ports in deltas and create a new delta (via functional delta assignment) for the `AddOperation` of a connector. In case the same ports are added in multiple deltas, we solve the existing element dependency

problem the same way as for cross-domain delta consistency preservation (see section 3.2.5) and add multiple deltas that add the connector with the same unique identifier. The only difference from the original idea of functional delta assignment is that now the added delta operation is dependent on two instead of one delta operation. This means the newly created delta is only applicable if all deltas it depends on get applied as well. We connect all application conditions of these dependency deltas with *AND* to achieve that outcome. Additionally, the newly created delta must be well-formed, meaning the application order must ensure it is applied after the deltas it depends on. An example addition of an `AddConnectorOperation` is provided in Fig. 3.12. The intra-domain Delta-CPR works in close collaboration with the cross-domain Delta-CPR, since consequential changes of a cross-domain Delta-CPR can trigger consequential changes of an intra-domain Delta-CPR. When using delta mirroring for cross-domain delta consistency preservation and functional delta assignment for intra-domain delta consistency preservation, it becomes easier for developers to trace which deltas in which domains were created by which type of CPR.

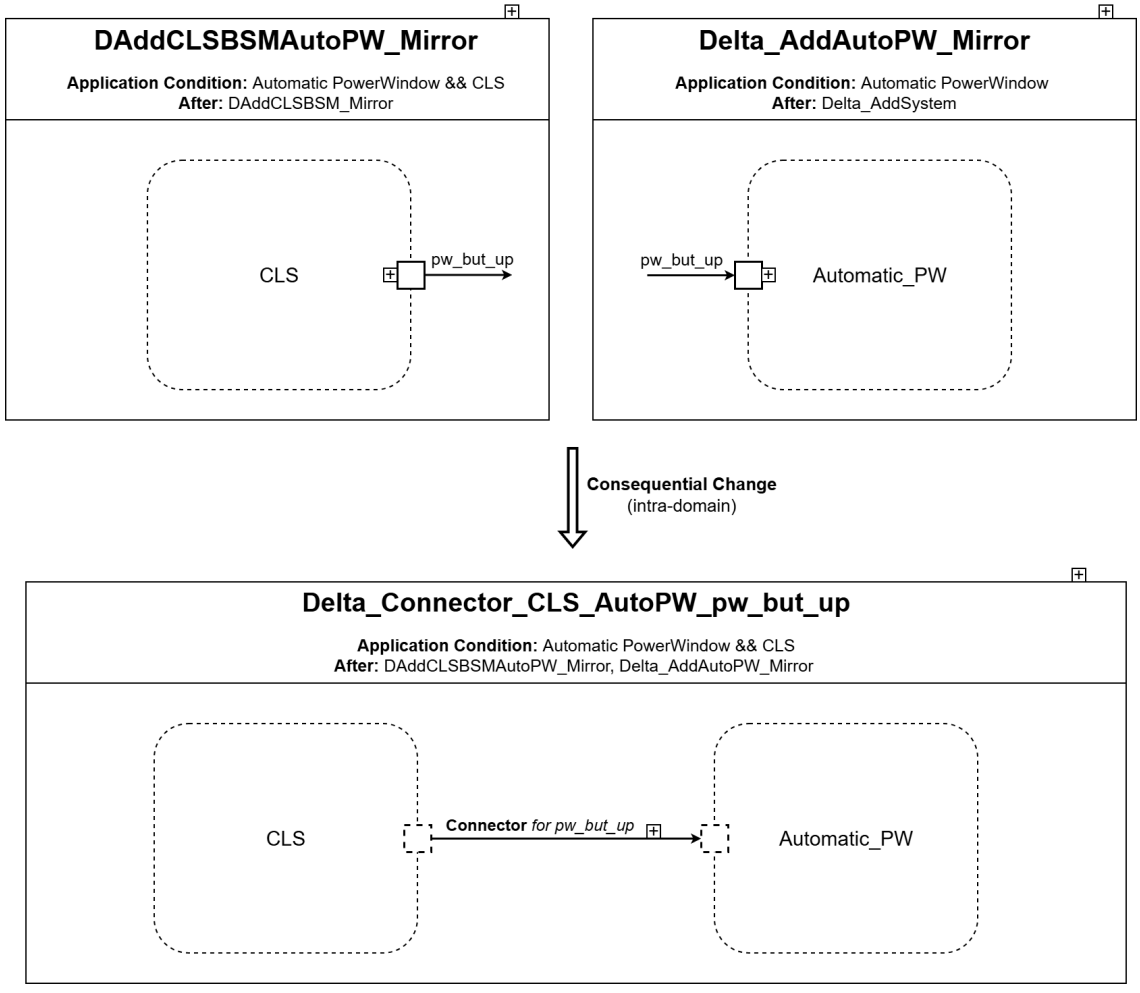

Figure 3.12: An intra-domain Delta-CPR reacting to the addition of the two deltas *DAddCLSBSMAutoPW_Mirror* and *Delta_AddAutoPW_Mirror* with the addition of the *Delta_Connector_CLS_AutoPW_pw_but_up* delta as a consequential change. The new delta is constructed with functional delta assignment, but with dependencies to two delta operations.

3.4 Automatically Translating Delta Consistency Preservation Rules

3.4.1 Translation Basics

As seen in section 3.2.2, the functional correctness of the Delta-CPRs we consider is always bound to the behavior of a Product-CPR. This makes automatically translating Product-CPRs to Delta-CPRs a promising approach. Correctly translating a Product-CPR to a Delta-CPR would mean the Delta-CPR is functionally correct regarding the source Product-CPR of the translation. Not only would this facilitate the migration from VSUMs to Var-VSUMs and therefore the migration from product-based development to platform-based development[Ant+14], but it would also guarantee functional correctness by construction for Delta-CPRs.
We aim to set a fundamental understanding for how such an automatic translation could work by discussing which constructs from a reaction on products are universally mappable to constructs of a reaction on deltas. The mappings are not strictly necessary to achieve functional correctness, since reactions get translated to arbitrary Java code. So creating two different reactions with the same behavior but different constructs is possible. However, the mappings provide a way to ease the migration process by providing an incomplete blueprint for the translation process.

3.4.2 Mappable Constructs

Certain constructs in reactions on products can always be translated to constructs in reactions on deltas. Some of these constructs are translated this way, based on solutions presented in this chapter. Listing 3.1 represents a translation of Listing 2.1 and will be used to show a few of the following translation constructs.

```
1 reaction: if createRegion added into delta {
2   add createComponent into getMirrorDelta(delta)
3   create correspondence between createRegion and createComponent
4 }
```

Listing 3.1: AddRegion-To-AddComponent delta reaction

- *Reaction triggers:* The reaction trigger can be fully translated when using the operations for the change. Instead of reacting to a change in an `EObject`, the translated reaction reacts to the addition of the corresponding change operation to a delta (Listing 3.1 line 1).
- *Adding `EObjects` in the target model:* Creating an `EObject` can always be translated to an `CreateOperation` for this exact `EObject` (Listing 3.1 line 2). Because of the unique identifier in the generic delta metamodel (see section 3.2.5), all `CreateOperations` for the same `EObject` belong uniquely to that `EObject`.
- *Creating correspondences to the correspondence model:* Creating a correspondence between `EObjects` can always be translated to creating a correspondence between the `CreateOperations` for this exact `EObject` (Listing 3.1 line 3).

- *Applying changes to the target model:* While applying changes to the target model of a product is done by just adding, removing, or changing existing elements, the localization problem (see Section 3.2.4) must be solved for a delta target model. Translated reaction must always implement and use delta mirroring, or functional delta assignment, to apply changes to the target domain without losing functional correctness or the well-formed property (Listing 3.1 line 2).

- *Changing the model root:* The change of a model root `EObject` can always simply be translated to a `InitializeRootObjectOperation`, provided by the generic delta metamodel.

- *Changing a reference in a model:* The change of a reference requires the dependency search for the `EObject` holding the reference in the target domain. This might include a dependency search in the source domain, combined with the usage of the correspondence model. Once the `AddOperation/SetOperation` for the reference holder is found as described in section 3.2.3, the change itself can be translated to a delta operation, as provided by the generic delta metamodel, and added to a delta. In the case of a containment reference, this operation is a `CreateOperation`. In case it is not a containment reference, we require the addition of the operation into the correspondence model if the reaction was triggered by the addition of an operation itself.

- *Changing an attribute in a model:* The change of an attribute gets translated similarly to the change of a reference. After a dependency search for the `EObject` holding the attribute, the change itself can be translated to a delta operation and added to a delta. The attribute `SetOperation` also needs to be added to the corresponding model if the reaction was triggered by the addition of an operation itself. The only difference is that changing an attribute, contrary to a containment reference, cannot result in the addition of a `CreateOperation`.

- *Fetching a reference in a model:* Fetching a reference is translatable to searching for the first `AddOperation/SetOperation` in deltas upwards of the application order tree, similar to dependency search. This is not possible if the search starting delta is independent of the delta that includes the searched `AddOperation/SetOperation`. But this is expected behavior, since we cannot use the result of the search in a delta if the existence of that result is not guaranteed when applying the delta.

- *Reading an attribute value in a model:*
  Reading an attribute value is translatable similarly to the fetching of a reference. The only difference is, instead of searching for an `AddOperation/SetOperation` for an `EObject`, we have to search for the first `AddOperation/SetOperation` of the attribute we can find. Because we assume all deltas are well-formed, we will not be able to find multiple of such operations without an explicit application order between them. The same failing criteria as for references also applies to attributes, but this is expected behavior for the same reason.

- *Arithmetic operations on primitive data in a model:*
  Arithmetic operations on primitive data (data contained in Ecore primitive data types) itself do not need to be translated at all. This also includes arithmetic operations on primitive data, which happens in external method calls. The only differences might occur for the reading of attributes to retrieve the primitive data used in the arithmetic operations.

- *Adding correspondences to the correspondence model:*
  The correspondences in the correspondence model must be set between the operations that add the `EObjects` into the model, instead of between the `EObjects` directly. The action of adding/removing correspondences itself stays the same otherwise.

- *Retrieving correspondences from the correspondence model:*
  Since the correspondences in the correspondence model are set between operations, instead of retrieving the `EObjects` in the containment hierarchy directly and manipulate their attributes/references, we can either retrieve the `CreateOperations` for directly changing the addition of an `EObject` or the `SetOperations` for the attributes/references to manipulate them.

- *Bidirectional attribute/reference manipulation:*
  Reactions themselves are unidirectional, but we can define one reaction per domain to achieve bidirectional reactions for a VSUM with two domains (such as the BCS). An example of such a bidirectional reaction is the synchronization of names between statemachine regions and components. For products, this is achieved by fetching the corresponding region/component and changing the name. But that also triggers the name synchronization reaction of the target domain, meaning the two reactions would constantly trigger each other. Vitruv includes a mechanism to prevent this. However, the mechanism requires the source attribute/reference of the first reaction to be the target attribute/reference of the second reaction. But when adding `SetNameOperations` to deltas, those are newly added `EObjects`, which leads the Vitruv prevention mechanism to fail. This problem can be completely avoided by adding correspondences between the `SetNameOperations` and checking their existence before executing a consequential change in the target domain.

3.4.3 Problematic Constructs

Section 3.4.2 lists mappable constructs and what they are mapped to when translating a Product-CPR to a Delta-CPR. This enables those constructs to be automatically translated when transitioning to delta-oriented models. In addition to these constructs, we identified further constructs that are very difficult to translate. Their automatic translation is not feasible because of complexity reasons or because they rely on semantics without syntax. The problematic constructs are:

- *External method calls with model `EObjects` as arguments:*
  The behavior of external method calls might be dependent on `EObjects` contained in the model instance. These dependencies must include the passing of `EObjects` contained in the model instance (or the whole model instance itself) to an external method call via an argument. In contrast to just passing primitive data, which does still exist in the delta metamodel, passing whole `EObjects` that were defined in the product metamodel requires changing the externally called code. Since the expected `EObject` data types are not defined in the delta metamodels, this change is always necessary. However, external code might not always be available in non-compiled form, and moreover, the code can be arbitrarily complex for Turing-complete languages. Even if the code is present in a non-compiled and modifiable state, analyzing and translating it to be applicable to deltas automatically is therefore unfeasible.

- *User interactions:*
  Vitruvius allows the definition of user interactions in order for the user to influence the outcome of a reaction application. The user is presented with a prompt in text format (usually natural language) and can answer the prompt with one or multiple values, depending on the respective prompt. When translating the concept of user interactions to deltas, the prompt itself is a significant problem. If the prompt is written in natural language, it doesn't follow a strict syntax. Additionally, the semantics of the prompt might only make sense to a user for products, but not for deltas. To solve this problem, we would have to interpret the semantics of the prompt correctly (without an existing syntax) and translate the semantics to a prompt that a user can reasonably understand for deltas, with the goal of achieving functional correctness for given user inputs. Since interpreting and translating semantics without a given syntax is required for this, translating user interactions automatically is unfeasible.

- *Multi-level existing element dependencies:*
  The solution discussed in Section 3.2.5 is only applicable if the existing element dependency element does not have a dependency itself to other elements. An example of a two-level dependency can be observed for connectors in the BCS. The existence of connectors is completely dependent on the existence of an input and an output port for a signal. But an input/output port does not get added if the component already has an input/output port for that signal. So the existence of the port the connector depends on is not certain. Such cases can be solved by searching through the whole delta repository, but we were unable to design a more efficient solving method, thus we will treat them as problematic.

3.5 Design Summary

Chapter 3 introduced Delta-CPRs and the challenges that arise when migrating the CPR concept to delta modeling. These challenges included the mapping of changes on products to changes on deltas, dealing with dependencies and localization of those changes, as well as showing their equivalence (functional correctness). We introduced concepts to overcome the challenges, explained them with examples, and laid out fundamentals on how to construct Delta-CPRs from existing Product-CPRs. The knowledge gained is used for two things:
First, we analyze how Delta-CPRs can not only be used for cross-domain consistency but also for intra-domain consistency. This is accompanied by an implemented example for intra-domain consistency preservation in the BCS case study.
Second, we directly apply the knowledge to discuss automatic translation, including constructs from Product-CPRs, which are feasible to automatically translate, but also such constructs, which are not feasible to automatically translate.
The main takeaway for the reader from this chapter should be that applying consistency preservation to delta-oriented software product lines is not simple but possible. Parts of the creation processes can be automated, and migrating from product-based to platform-based development is feasible. All the answers and knowledge provided in this chapter can be combined into a framework, guiding the process along the way and providing solutions to frequently arising problems in this domain. To prove the practicality of this framework, it was applied to the BCS case study and implemented. All details for this implementation and the corresponding evaluation can be found in chapters 4 and 5.

4. Implementation

The following sections give an overview of some implementation details for our BCS case study implementation and our framework implementation. We look at how problem solutions from Chapter 3 are implemented and which changes we have to apply to the BCS case study model instances to make them work with Vitruv in practice. Additionally, we describe tooling developed to help speed up the model instance creation and ensure said model instances are complete. The whole implementation is written in Java, since that is the only language with Ecore support and is used by Vitruv.

4.1 Case Study Implementation

The new BCS report[NLS18] implicitly changes the way transitions between states work, compared to the old BCS report[Lit+13]. Instead of just allowing one trigger or behavior per transition, it allows multiple triggers and behaviors per transition. At the same time, the new report does not include explicit deltas for the addition of the door systems (manual power window and automatic power window) or the remote control key system. It does, however, include explicit deltas for the addition of the status LED statemachine region and central locking system. These new deltas are mandatory for applying higher-order deltas later on. For the power windows and remote control key system, we simply assume the base behavior from the old report, since the higher-order delta application does not modify their statemachine regions. This base behavior for the door systems and remote control key system is modeled as deltas and forces us to apply modification deltas from the old report on them. The two deltas in this case are `DAddManPWCLS` and `DAddAutoPWCLS`. All other deltas only modify either the central locking system behavior, status LED statemachine region, or a behavior of a newly added component. This means all remaining deltas can be taken from the new BCS report.

The reports also have some smaller inconsistencies in the case study itself, which prevent a clear and functional Ecore-based implementation with Vitruv. Most of these are only name duplicates, which are optional to fix. We modify the following constructs to fit our implementation:

- The `Delta_Medium_HighSensor` delta incorrectly deletes the `t23` and `t24` transitions, which cannot exist during application, since they are added only if `low quality sensor` is a selected feature. The delta is only applicable if the `high quality sensor` feature is selected, but `high quality sensor` and `low quality sensor` are mutually exclusive features. We remove the incorrect deletion of those transitions in our implementation.
- The `DAddCLSBSMAutoPWSeatKey` delta needs to have the `automatic power window` features as part of the application condition instead of the `manual power window` feature to be applicable. This is also already implied by the delta name. We exchange the `manual power window` feature in the application condition with the `automatic power window` feature.
- The delta `DAddManPWCLS` has two transitions each, named `t22` and `t25`. Our implementation changes the duplicate names to `t26` and `t27` to fix this.
- All states with only subregions in them have the same name as the included subregion. We add a *_state* at the end of their names to avoid name duplicates.
- Transition names of the remote control key system are identical to transition names added by higher-order deltas. To avoid name duplication, we add a *_rck* to the end of every such transition.
- The higher order delta for the wiper system adds the addition of a state, as well as a signal with the name *clean*. To avoid name duplicates, we rename the state to `clean_state`.
- The higher-order delta for automatic headlights adds the addition of two states with the name *low_beam*. To avoid name duplicates, we rename the *low_beam* state in the `AutoLight` region to *low_beam_auto*.
- The higher order delta for automatic headlights adds the addition of a state and a signal, both with the name *lights_off*. To avoid name duplicates, we rename the state to *lights_off_state*.

4.2 Solving the Localization Problem: `DeltaCreator`

The `DeltaCreator` class is an important part of our framework and solves the localization problem described in Section 3.2.4 in various ways. It supports the creation of a mirror delta via a `createCorrespondingDeltaForDelta` method, functional delta assignment via a `createChildDelta` method, and merging deltas via a `mergeDeltas` method. When Delta-CPRs are only used to preserve cross-domain consistency, the implementation of delta mirroring is sufficient.

To implement delta mirroring, we developed an algorithm called **Delta Correspondence Search**. It is used to find all deltas that are required to be included in the mirror delta's after clause. The algorithm can be seen in Listing 4.1. Delta correspondence search represents a breadth-first search that finds all deltas that have to be applied before the start delta and have a corresponding delta in a second domain. It adds all the corresponding deltas in the second domain in a list and returns the list. The application order for deltas is transitive, meaning if *Delta A* has to be

applied before *Delta B*, and *Delta B* before *Delta C*, then it requires *Delta A* to be applied before *Delta C*. Because of this transitivity, delta correspondence search can stop the search for a path as soon as a delta with correspondences is found. When implementing delta correspondence search, we additionally added a set that functions as a cache to avoid adding a delta to the search queue multiple times. Since the delta correspondence search algorithm requires a view of the correspondence model, we implement the `DeltaCreator` as a singleton to avoid working on multiple different inconsistent views of the correspondence model.

```
deltaCorrespondenceSearch(startDelta) {
    List result
    Queue searchQueue
    searchQueue.add(startDelta)
    while (searchQueue not empty) {
        currentDelta = searchQueue.retrieveFirst()
        if (currentDelta has corresponding deltas) {
            result.addAll(corresponding deltas)
        }
        else {
            searchQueue.addAll(currentDelta.afterClause);
        }
        searchQueue.remove(currentDelta)
    }
    return result
}
```

Listing 4.1: Delta Correspondence Search Algorithm

To implement functional delta assignment, the `DeltaCreator` requires a list of deltas as an input. The list includes all deltas that the new delta should be dependent on. In the context of functional delta assignment, this means that the new delta should encapsulate one or multiple operations that are dependent on one or multiple operations inside the input deltas. The `DeltaCreator` creates a new delta, adds all input deltas in the after-clause, and sets the application condition to the application conditions of all input deltas combined in an AND-expression. This way, the newly constructed delta only gets applied after all input deltas get applied, and only if all input deltas get applied. Notably, this implementation of functional delta assignment does not include delta compression, since our implementation only uses it for intra-domain cleanup routines in the component and delta component domains.

The merging of deltas is required to work around the multi-level existing element dependencies, which are present when creating connectors in our implementation of the BCS (see Section 3.4.3). The `DeltaCreator` merges a source and a target delta by adding all entries in the source after-clause into the target after-clause. Additionally, the operation conditions get combined with an OR-expression, meaning the merged delta gets applied every time one of the included deltas would be applied, while still respecting their original application order. Since a merged delta can be merged a second time, this implementation allows for merging an arbitrary amount of deltas.

4.3 Solving the Dependency Problem: `DeltaOperationFinder`

The `DeltaOperationFinder` class is also part of the framework and responsible for finding delta operations in deltas to resolve the delta operation dependencies described in Section 3.2.3. It can resolve multiple different dependencies. To emulate the `.eContainer()` of `EObjects` on deltas, the `DeltaOperationFinder` provides the `getRefOperationForEObjectIDAndType` method, and to emulate `EcoreUtil.getRootContainer()`, it provides the `getContainmentRootOperationForEObjectID` method. The `getRefOperationForEObjectIDAndType` method does not exactly replicate the behavior of `.eContainer()`, since it can only return a `RefDeltaOp`, and not a `InitializeRootObjectDeltaOp`. A `RefDeltaOp` is a delta operation adding an `EObject` into a reference, while a `InitializeRootObjectDeltaOp` is a delta operation that sets a new model root when applied. This means when the container for a `CreateOperation` is a root, this method will not return the correct container. Instead, in this case, the reaction developer (or automatic translator) has to use the `getContainmentRootOperationForEObjectID` method. This issue stems from the `InitializeRootOperations` in the generic delta metamodel not being of type `RefDeltaOp`. Additionally, the `DeltaOperationFinder` provides the `getAtomicOperationForClassAndEObjectID` method. This method is used to resolve dependencies that are neither of type `RefDeltaOp` nor `InitializeRootObjectDeltaOp` and only searches for the affected `EObject`, instead of the one that is being added. The method is optional in an implementation and only used to generate meaningful names for ports and connectors from already existing names of components and signals.

For all three types of dependency resolution, we perform a very similar breadth-first search as for delta correspondence search (see Listing 4.1). The idea can be seen in Listing 4.2 for the `getRefOperationForEObjectIDAndType` method, which searches for a `RefDeltaOp` of a given type that sets an `EObject` with a given unique ID. The search is a breadth-first search and starts at a given start delta. The algorithm searches upward through the after-clauses to find the operation that adds an `EObject` with a given unique ID when it gets applied. The search looks almost identical for finding an `InitializeRootOperations`, with the only difference being there is no type argument, since only one type of root exists per model instance in the BCS. As long as the deltas are all well-formed, this algorithm will always find the target operations, if they are present, because all deltas that have to be applied before the start delta per application order get searched. The `DeltaOperationFinder` also has another, more niche use: It can perform a search for a `RefDeltaOp` on the complete repository. This is done by using a purpose-built method, which is only used in the delta component cleanup routines. It is required to resolve the two-level existing element dependencies when creating ports. A global search is inefficient, but it is the only known way for us to resolve such dependencies (see Section 3.4.3).

To achieve higher performance when applying Delta-CPRs, the `DeltaOperationFinder` is equipped with caches for every domain. Those caches are implemented by using mappings between the unique ID of the `EObject` that was added to a reference or set as a root and the operation itself. To ensure the caches are being used and held consistent, we also implement the `DeltaOperationFinder` as a singleton to avoid the existence of multiple cache structures side by side.

```
getRefOperationForEObjectIDAndType(startDelta, id, type) {
  List result
  Queue searchQueue
  searchQueue.add(startDelta)
  while (searchQueue not empty) {
    currentDelta = searchQueue.retrieveFirst()
    for (delta operations in currentDelta) {
      if (operation.type == type and operation.addedID == id) {
        return operation
      }
    }
    searchQueue.remove(currentDelta)
    searchQueue.addAll(currentDelta.afterClause)
  }
}
```

Listing 4.2: Finding Ref Operation of Type Algorithm

4.4 Delta Consistency Preservation Rules Testing Kit

Another part of our proposed framework is the Delta-CPR evaluation kit. The Delta-CPR evaluation kit application is split into three parts. First, the products need to be present, and external changes on them must be applied. Second, we need to have a way of constructing fully specified products by applying delta in the delta repository for a feature configuration. Third, we need to have a way to compare constructed products and find out if they are equivalent.

4.4.1 Model Instances and Changes

The Vitruv version we used (v3.2.2) does not support the application of higher-order deltas. This means we cannot apply a higher-order delta to a delta model instance and have reactions trigger according to it. The Delta-CPR testing kit, as seen in Fig. 3.2 can still be implemented by building the model instances from an empty model instance. We construct a model instance completely from an empty model instance by adding an input model instance of one of the two domains to the VSUM and applying all CPRs during this process to build an output model instance of the second domain. Since the component domain can be built completely from the statemachine domain, but not vice versa, the input and output model instances for the component domain are identical, but the input and output model instances for the statemachine domain are different. The same applies to the corresponding delta domains. Additionally, we also decided to implement the comparison mechanism to work domain-wise, instead of for a whole VSUM at once. Since we specifically want to analyze what changes the reactions apply when reacting to the application of the higher-order deltas, we split the comparison into products before the higher-order

deltas were applied (base) and after (complete), as later described in Section 5.2.1. This way we can determine if a failure in constructing the product from deltas stems from the higher-order delta application or if it was problematic even before that. In total, there are four different comparisons we have to compute. Figure 4.1 visualizes them. Firstly, we need to build the baseline products that are completely indepen-

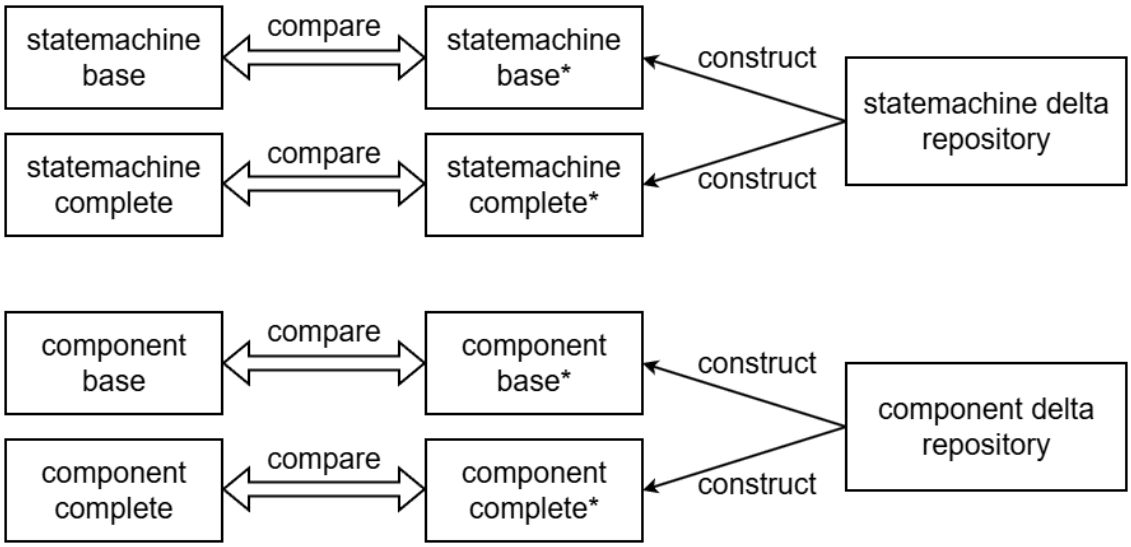

Figure 4.1: Model instances and comparisons needed for our implementation of the Delta-CPR testing kit.

dent of the deltas. As already stated, this is done by adding input model instances, letting the reactions trigger completely, and collecting the output model instances. This is done for both domains, once for the base version and once for the complete version. Those model instances are the ones seen on the left side of Figure 4.1. All output model instances in the component domain are missing constructs compared to the input model instances, since statemachine domain includes information not included in the component domain. The delta repositories seen on the right side are built with the same input-output mechanism. Lastly, the products to compare the baseline against are constructed by applying the deltas according to the base and complete version feature selection of the product we want to test. In summary, we have four input products, which are used to build four output products. The four output products are used as the baseline for the comparison. We have two input delta repositories, which are used to build two delta repositories. Each of those output delta repositories is used to construct two products, which are then used as the other side of the comparison. In addition to these output checks, we also added checks for the inputs. This can be achieved by using the four baseline product inputs on the left side and the two input delta repositories on the right side of Figure 4.1. The test is executed once per product variant. If this test succeeds, we know the input products can be constructed from the input delta repositories, meaning if the tests for the output model instances fail, the source has to be the reactions.

4.4.2 Product Construction: `ModelBuilder`

The `ModelBuilder` class provides a way to construct a model instance by applying deltas in a delta repository model instance for a given selection of features. It only applies deltas, with application conditions matching the selected features, and applies them in an order that does not violate the delta application orders.

Since the application condition of a delta can consist of multiple features connected with NOT-, AND-, and OR-expressions, we resolve the application condition in parts. Our implementation uses recursion to evaluate every part of the application condition individually by calling the `isValidPerFeatureSelection` method. It first checks if the application condition is empty. If this is the case, the delta should always be applied. Usually this is only the case if the delta is part of the common product core. In a second step, the method checks if the application condition is only a single feature. If this is the case, it just has to check if that single feature is included in the feature selection. Otherwise, the remaining application condition must be either a NOT-, OR-, or AND-expression. To compute if a NOT-expression is valid per feature selection, the method recursively calls itself with the negated remaining application condition, negates that result, and returns it. For an AND-expression, the method recursively calls itself with the remaining application conditions connected by that expression. If all results of these recursive calls are true, the method returns true. Otherwise it returns false. For an OR-expression, the method recursively calls itself with the remaining application conditions connected by that expression. If one or more results of these recursive calls are true, the method returns true. If all of them are false, it returns false.

After filtering out all deltas with valid application conditions per feature selection, the next step is to apply them without violating the application orders. The first step is to find an initialization operation for a root `EObject` that is still present after filtering. The delta containing this operation should have an empty after clause (since no other operation can be applied before a root was added), and there should only be one of these deltas (otherwise the deltas are not well-formed). After initializing the root `EObject` and thus creating a model instance, further delta operations can now be applied to this model instance. The product construction algorithm then cycles through all remaining deltas and checks if they are applicable. To decide this, it holds a list of already applied deltas and marks such deltas as valid, where all deltas in the after-clause are included in the applied delta list. After every cycle, it applies all marked deltas and adds them to the applied delta list. Applying a delta means applying all contained delta operations to the model instance created earlier. This cycling is repeated until either the number of remaining deltas is zero or no new deltas get marked in a cycle. If no new deltas get marked, but at least one delta is still present, then some remaining delta is most likely formulated incorrectly, since it should be applicable per application condition but depends on a delta that could not be applied itself.

4.4.3 Product Comparison: `BCSInstanceComparator`

We want to pairwise compare product model instances of our implemented BCS case study. However, since every `EObject` in these model instances has a unique identifier and all multi-value attributes/references do not have a strict order, the model instances should not be identical but only equivalent. Equivalency in this context means they should be the same, except for two properties: First, the unique identifiers of the included `EObjects` are allowed to be different from each other. Second, the multi-value attributes/references have to include the same values/references, but the order of these is not relevant.

To achieve this equivalence check, we implement the `BCSInstanceComparator` class. It directly extends the `EcoreUtil.EqualityHelper` class, but modifies a few methods to enable checking for equivalence. For the comparison of two `EObjects`, we modify the `haveEqualFeature` method to ignore the unique identifier attribute. Since all `EObjects` not only have unique identifiers but also names (which are identical between the BCS model instances), we can use them to enable the orderless comparison of multi-value references by modifying the `equals` method. Before comparing two multi-value references, we simply sort the references alphabetically by the names of the `EObjects` the reference points to. This is already sufficient, since our BCS model instances do not include multi-value attributes and no two `EObjects` with the same name in the same domain.

4.5 Automatic Consistency Preservation Rules Translator

The goal of an implemented automatic CPR translator is to enable automatic translation of input Product-CPRs into an output Delta-CPRs. This input/output mechanism is applicable per reaction file, meaning the translator reads an input reaction file and returns an output reaction file. Reaction files have some required constructs that are necessary to define a reaction file in the first place. To accommodate this, we split the translation into two parts: First, create a template that includes all the necessary constructs to define a valid reaction file. Second, use a decorator to extend and adjust the reaction file with optional constructs, with the goal to achieve functional correctness for all reactions in the reaction file and meaningful names for them.

4.5.1 Automatic Translation Templates

There are two translation templates: One for cross-domain reactions and one for intra-domain reactions on deltas. The translation templates include all the necessary constructs to define a valid reaction file and a few further constructs that we know are necessary to implement reactions on deltas. For defining a valid reaction file, the template includes a reaction segment header, consisting of a reaction segment name and a source and target domain for the included reactions. While the generic name is just the template name, the generic source and target domain are both set to the generic delta metamodel. All of them can later be adjusted with the usage of the decorator. Additionally, the template includes multiple constructs, which we know are necessary to implement cross-domain reactions on deltas. These include

imports for both templates and a routine to implement delta mirroring, as well as a reaction to mirror existing delta repositories for the cross-domain template. The imports include imports for `EcoreUtil`, which is used to generate unique identifiers, deltas, and delta repository constructs, and the `DeltaCreator` and `DeltaOperationFinder`. All of these classes are very likely to be used for defining reactions on deltas, and without them, only extremely rudimentary reactions are possible. Additionally, the templates include a generic delta metamodel model import since they include the delta and delta repository constructs. For a full overview of the templates, refer to the Appendix A.

4.5.2 Automatic Translation Decorator

The translation decorator is implemented in the `ReactionFileDecorator` class. It can be used to add and adjust multiple constructs to the reaction file created by applying the template. Since a full implementation of the automatic translator is out of scope for this thesis, we only implemented some fundamental functionality for the `ReactionFileDecorator`. As already stated in Section 4.5.1, the `ReactionFileDecorator` can change the name, source domain, and target domain of a reaction file. It can also add new model imports, which is necessary when changing the source or target domain to models different from the generic delta metamodel. The `ReactionFileDecorator` can insert reaction and routine stubs. They have no arguments and get a configurable modified name of the reaction they originate from. The reaction and routine stubs also come with a comment including the original reaction or routine. This enables a clear mapping for the developer and is a first step towards simplifying the manual translation. In a full implementation, the `ReactionFileDecorator` would fully generate all reactions and routines instead of stub whenever possible.

4.6 Additional Tooling for Delta Generation

The implementation activities for this thesis reach beyond the implementation of Delta-CPRs and tooling for their application. In this section, we present how the most important by-products of the implementation were constructed: the Ecore-based BCS model instances. This includes the product model instances, as well as the delta product instances. To keep it in scope for this thesis, we cut parts of the case study that are not of interest when applying the higher-order deltas. More information on these modifications can be found in Section 5.2. We implemented the product model instances by hand, according to the case study reports [Lit+13][NLS18]. To accelerate the implementation of the delta instances, we developed a migration helper to generate delta operations from product model instances.

4.6.1 Case Study Migration to Deltas

While the whole product case study with all model instances is created manually, the delta case study creation was created using automatic support. We implemented the `BCSToDeltaBCSMigrationHelper` class, which can generate delta operations based on given `EObjects`. This is accomplished by implementing `Transformers` for all `EObjects` contained in the model instances. A transformer takes an `EObject` as input and outputs a list of delta operations that can be applied to generate that exact `EObject`. As an example, consider a statemachine `EObject` with the name attribute value of *premium product*. The statemachine transformer will take this statemachine as input and output two distinct delta operations: an `InitializeRootStatemachineOperation` and a `SetNameInStatemachineOperation`, with the value *premium product*, and the initialized statemachine as the target. The `BCSToDeltaBCSMigrationHelper` receives the product model instance as input and performs a breadth-first search through all `EObjects` included in the containment hierarchy. For every `EObject`, the `BCSToDeltaBCSMigrationHelper` calls the corresponding transformer and transforms the `EObject` to delta operations. The delta operations can then manually be split into groups to form deltas according to the case study. For our implementation of the BCS case study, this migration step is performed once for the statemachine domain and once for the component domain. If multiple products are to be migrated this way, we have to ensure that the common `EObjects` have the same unique identifier. This is required since delta operations for different products need to be merged into one delta repository, and their common operation targets need to be consistent across products. The `ProductUUIDFixer` class is the implementation for this exact step. Notably, the whole approach only works for products that are built with additions of `EObjects`. This implementation approach has the clear limitation of not being able to produce any *RemoveOperations*. The few deltas in the BCS case study that require the removal of `EObjects` are implemented manually and cannot be migrated this way.

4.6.2 Delta Repository Plausibility Check

To practically test the `BCSToDeltaBCSMigrationHelper` and check if the resulting delta operations manually grouped in deltas are well-formed, we implement the `RepositoryChecker` class. It performs a few simple plausibility checks that should all be passed by a delta repository only including well-formed deltas. This is purely a quality assurance tool and proved very useful when implementing the `BCSToDeltaBCSMigrationHelper`. The checks performed by the `RepositoryChecker` include the following:

- If a delta operation adds an `EObject` as a reference of another `EObject`, ensure the `DeltaOperationFinder` can find the operation that added the `EObject` containing the reference.
- If a delta operation sets a source or target state for a transition, ensure the `DeltaOperationFinder` can find the operation that added the state.
- If a delta operation adds a communication signal in a transition trigger or behavior, ensure the `DeltaOperationFinder` can find the operation that added the communication signal.

- If a delta operation sets a signal for an input port or output port, ensure the `DeltaOperationFinder` can find the operation that added the signal.

If all checks are successful, the deltas included in the delta repository are all well-formed.

4.7 Workflow for Tool Usage

The multiple tools and case studies that were implemented for this thesis have dependencies with each other. This means there is a required workflow to use them properly. If this workflow is not followed, some necessary artifacts might not be available, and parts of the implementation are unable to run in that case. We explain the necessary workflow steps from obtaining a product instance to running the Delta-CPR testing kit, but also take a look at the optional steps that can be used in between or as substitutes.

We start by obtaining input product model instances for every domain. These can either be created manually, or you can use the already existing ones in the case study implementation. To generate consequential changes, we need to implement or use already provided Product-CPRs. They have to be implemented in the form of reactions so Vitruv can apply them. After all input product model instances and reactions on them have been implemented (or you used provided ones), the product case study `Runner` class can be executed. It adds all the input product model instances into VSUMs and triggers the reactions on them. The `Runner` will output VSUMs, including product model instances for both domains once per product. For the next step, we either use provided delta input model instances, migrate our product model instances, or manually create entirely new delta model instances. If new product model instances were created instead of using provided ones, a migration is necessary. In this case, the `ProductUUIDFixer` can be used to synchronize unique identifiers between the product model instances, and the `BCSToDeltaBCSMigrationHelper` can be used to automatically generate a significant part of the necessary delta operations. The delta operations have to be manually sorted into deltas of choice and can be checked to be well-formed with the usage of the `RepositoryChecker`. Similarly to the product case study, you also have to ensure there are reactions defined. You can either create your own or use the provided ones. After all input delta model instances and reactions on them have been implemented (or you use provided ones), the delta case study `Runner` class can be executed. The `Runner` will output a Var-VSUM, including one delta model instance for each domain. In the end, we can apply the evaluation tools, such as the Delta-CPR testing kit. This requires all input and output model instances for all products and deltas to be present. The evaluation results can be obtained by running the `TestingKit` class and the `EObjectCounter` class for RQ3.

5. Evaluation

This evaluation chapter is divided into multiple subsections. Firstly, we present the research questions, which we aim to answer. This includes explaining the methodology on how to answer them, as well as discussing the consequences of potential results. Secondly, we present the two subject systems for the evaluation: a product-based and a delta-based implementation of the BCS. The delta-based BCS implementation was implemented with the solutions discussed in Chapter 3 and the implementation details shown in Chapter 4. Next, we explain the experiments and present the results. This includes experiment setup details to enable recreation of our evaluation results. The results themselves are discussed, interpreted, and connected to the research questions they answer. After presenting and discussing the results, we address potential threats to validity. Constructs and external threats are discussed, and their severity gets assessed. Lastly, we conclude the evaluation by summarizing the results and judging the feasibility of Delta-CPRs and our framework based on them.

5.1 Research Questions and Methodology

We set out to answer four research questions in regard to Delta-CPRs. The four research questions are the following:

- **RQ1: Is it possible to apply the CPR concept to delta-oriented modeling?** This question is relevant, since the usage of CPRs in the context of delta-oriented modeling has not been explored yet. Modeling cyber-physical systems often include modeling an SPLs, which can be implemented with delta-oriented modeling. Enabling automatic consistency preservation, in the form of CPRs on those projects, could boost development productivity and reduce costly inconsistencies between model domains.
  The research question is answered by an implementation example. We manually implement Delta-CPRs for the BCS and introduce changes to the system. The consequential changes from Delta-CPR application are analyzed and checked for functional correctness. Checking a Delta-CPR for functional correctness is only possible after RQ2 has been answered. Hence, these two

research questions are closely interlinked and can only be answered together. If we do not succeed in achieving functional correctness for the implemented Delta-CPRs in the BCS case study, we find at least one delta-oriented model to which the CPR concept cannot be fully applied to. If we do succeed, we show that the CPR concept is applicable to at least a subset of possible delta-oriented models. The expected result is the successful implementation and establishment of functional correctness for all implemented Delta-CPRs.

- **RQ2: How can we test functional correctness of CPRs for delta-oriented modeling?** This question is relevant for answering RQ1. In our context, the successful application of a Delta-CPR is given if it is functionally correct. But proving functional correctness for general cases is very hard (see section 3.2.2), and testing for given changes on given models is currently the only feasible way to check for it.
  The research question is answered by implementing and applying the Delta-CPR testing kit from section 3.2.2 to a BCS implementation. The Delta-CPR testing kit tests for functional correctness per definition, so it is evaluated based on its capabilities and the ability to guide the Delta-CPR development process. The Delta-CPR testing kit gets applied repeatedly during the development process of the Delta-CPRs to help find errors that prevent functional correctness.
  Ideally, the testing kit should be practically applicable, as seen in Fig. 3.2 and help guide the Delta-CPR development process. If this is the case for the BCS case study, we find at least one delta-oriented modeling case study for which the Delta-CPR testing kit can be successfully applied and used to guide the Delta-CPR development. If not, the testing kit remains an important tool to check functional correctness. However, in this case it is less useful, since it provides little guidance for the development of Delta-CPRs. We expect the Delta-CPR testing kit to be practically applicable and provide noticeable guidance for implementing Delta-CPRs for the BCS case study.

- **RQ3: Does the CPR application on deltas result in significant changes to the number of correspondences between the models?** Vitruv can use correspondences to keep track of cross-domain dependencies. While the usage of correspondences is not strictly required, it helps significantly in achieving CPRs with good performance, which do not require repeatedly searching through model instances. Using correspondences can therefore be seen as a clean code principle for developing CPRs in Vitruv. To apply and use such CPRs, these correspondences must be available in the memory of the executing machine. But the introduction of Delta-CPRs also results in the introduction of new cross-domain dependencies between `EObjects` in the model instances, such as between deltas. This makes the answer to this question relevant, since the introduction of Delta-CPRs might result in a scalability issue and prevent the usage in real-world projects. The usage for real-world projects is especially critical, since they might be multiple times the size of the BCS case study and therefore have to set a special focus on scalability.
  The research question is answered by measuring the amount of correspondences needed by Product-CPRs, or Delta-CPRs, when the BCS case study is directly implemented as products, or a delta-oriented SPL. We compare the

combined measurements on all products with the measurements on the SPL and discuss the results. This is especially interesting, since some deltas can be used to construct multiple products, meaning the correspondence for the addition of a potential `EObject` only exists once in deltas, but the `EObjects` themselves exist multiple times, distributed in the products.
Scalability is given if the number of correspondences needed for the SPL implementation is in the same or smaller order of magnitude than the combined number of correspondences. In that case, the whole proposed framework with Delta-CPRs can be applied to case studies of arbitrary size, without scalability issues regarding correspondences. If the approach is not scalable, the proposed framework is only practically applicable to case studies of limited size. The expected comparison outcome is a similar number of correspondences for both products and deltas, since the implemented BCS includes a very limited amount of products.

- **RQ4: Is it possible to automatically translate Product-CPR to a Delta-CPR?**
When migrating from products (product-based development) to deltas (platform-based development), many artifacts have to be newly created. This research question is relevant, since the automatic translation of Product-CPRs to Delta-CPRs would significantly ease the migration process by allowing us to skip the manual creation of Delta-CPRs. For a complete translation, this would also supply the added benefit of having functional correctness per construction, since the automatic translation would be aimed at creating a functionally correct translation.
The research question is answered by applying the automatic translation blueprint (see section 3.4) to the implemented BCS case study and determining if, and what parts of, the Product-CPRs are translatable. This is done via analyzing which parts of the Product-CPRs are mappable constructs and which are not.
If all Product-CPRs are fully translatable, this would prove a complete automatic migration regarding the CPRs for a case study of relevant size is possible. If none, or only a small percentage of constructs are translatable, it shows the automatic translation approach is not feasible or helpful when migrating from product-based development to platform-based development. The expected result is the possibility to automatically translate a significant part of the Product-CPRs, but not a complete translation.

5.2 Subject System for the Evaluation

To evaluate and answer RQ1 - RQ4, we implement the BCS case study with CPRs in Vitruv. The case study is only partially implemented, compared to the BCS reports[Lit+13][NLS18]. We use a part of the first report [Lit+13] and interpret the higher-order deltas introduced in the second report [NLS18] as changes that our CPRs should react to with consequential changes. The feature diagram of the implemented parts before any changes can be committed was already provided in Fig. 2.5. A complete feature diagram of the implemented parts is provided in Fig. 5.1. To evaluate functional correctness and everything based on it, we have to implement the BCS as a delta consistency preservation case study with one model instance and a product consistency preservation case study with one instance per available product. This way, we can apply deltas from the delta case study according to a product feature selection and compare it to the corresponding product instance from the product case study. This also means changes to a product are changes to the corresponding product model instance, while changes to a delta are changes to the one existing delta model instance.

The BCS products, in the form we implement them, have a special property: All information included in the component domain can be derived from the statemachine domain, except for the connectors. Because of this property, we can construct the component model instance completely from the statemachine model instance via cross-domain consistency preservation and use intra-domain consistency preservation for the connector creations. However, the statemachine domain includes information not available in the component domain, such as states and transitions between them. Statemachine model instances can therefore only be partially constructed from component model instances.
The same principles can be applied to the delta consistency preservation case study. All delta operations of the delta component model instance can either be constructed via cross-domain consistency preservation based on the delta operations in the delta statemachine model instance or via intra-domain consistency preservation if connectors are involved.

5.2.1 Product Consistency Preservation Case Study

For the product-based consistency preservation case study, we model two products as model instances: The **standard product** and the **premium product**. This is a minimal selection of products for implementing them in an SPL, but it gives us the opportunity to discover if correspondence scalability is already given for a minimal number of products. When talking about a feature selection in the context of a product, we always refer to the feature selection, which can be used for delta application to construct the exact product. This comes up multiple times, since the original BCS from the reports[Lit+13][NLS18] is a delta case study and had to be migrated to products to construct the standard product and the premium product. The features selected for the **base variants** (before changes) of the standard product and the premium product are seen in Table 5.1. The changes introduced by the second BCS report[NLS18] also introduce additions to the feature diagram, as seen in Fig. 5.1. We call the product variants with these new changes **complete variants**. The features selected of the additionally introduced features to construct the complete variants of the standard product and the premium product are seen

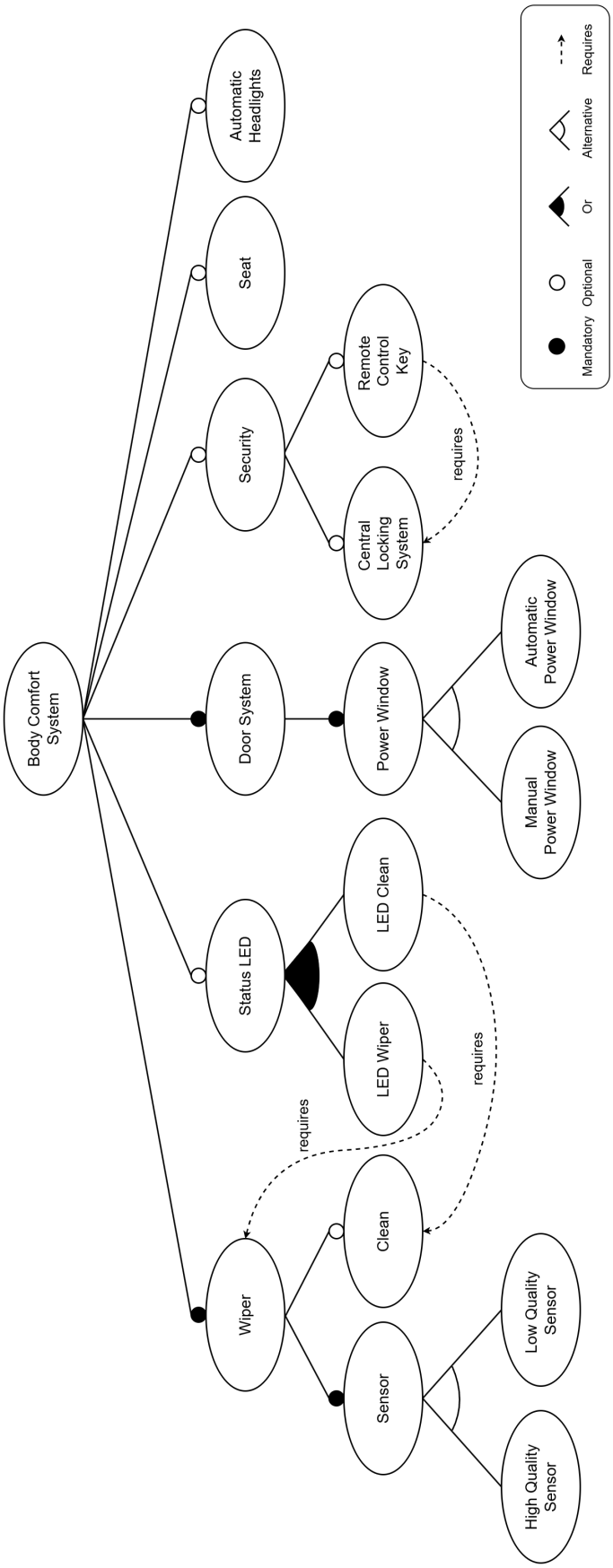

Figure 5.1: The full feature diagram for the BCS case study implementation.

Feature	standard product	premium product
Manual Power Window	x	
Automatic Power Window		x
Security	x	x
Remote Control Key (RCK)	x	x
Central Locking System (CLS)	x	x

Table 5.1: Feature selection of the BCS for the base variant feature diagram (Fig.2.5), to construct the standard and premium product.

Additional Feature	standard product	premium product
Clean	x	x
Low Quality Sensor	x	
High Quality Sensor		x
Status LED	x	x
LED Wiper	x	x
LED Clean	x	x
Seat	x	x
Automatic Headlights		x

Table 5.2: Additional feature selection of the BCS for the complete variant feature diagram (Fig.5.1), to construct the standard and premium product.

in Table 5.2. To construct the complete variants, features from both Table 5.1 and Table 5.2 have to be selected. The feature selection per product is chosen carefully to enable testing Delta-CPR application in three different scenarios:

- *Applying changes to all products, depending on new feature selection:* This happens for the addition of the `Wiper` system. Since the standard product has the `Low Quality Sensor` feature selected, but the premium product has the `High Quality Sensor` feature selected, the newly added changes differ in both variants.
- *Applying changes to all products, depending on existing feature selection:* This happens for the addition of the electric seat adjustments. The corresponding seat feature is selected for both variants, but the exact application of the deltas depends on the application of deltas for the central locking system, which itself depends on the selected power window. Since the standard product has the `Manual Power Window` feature selected and the premium product has the `Automatic Power Window` feature selected, the delta application differs and applies different changes to both instances, despite both having the `Seat` feature selected.
- *Applying changes to one product, while not changing the others:* This happens for the addition of the `Automatic Headlights`. Since the premium product has the `Automatic Headlights` feature selected, but the standard product has not, changes are applied to the premium product while leaving the standard product untouched.

Both the standard and the premium product get implemented with the same set of reactions between them. For the scope of this thesis, we implement such reactions that react to the addition of an `EObject` with the addition of an `EObject` as the consequential change, and such reactions that react to modifications of names. This is the minimal number of required CPRs to apply the Delta-CPR testing kit later on and answer the research questions. In a real-world example, the reactions would most likely also include reacting to the deletion and further modifications of `EObjects`. For the statemachine domain to the component domain, the implemented reactions are as follows:

- *Synchronizing roots:* Whenever a statemachine gets initialized as the root of the statemachine model instance, the reaction checks if the component model instance has a corresponding system initialized as the root. If this is not the case, it creates one and a correspondence between them.
- *Synchronizing root names:* Since the roots in both model instances are supposed to represent the same product, when setting the name in a statemachine, the reaction copies the same name to the corresponding system.
- *Synchronizing signals:* Whenever a communication signal is inserted into a statemachine, the reaction checks if a corresponding signal exists in the system. If not, it adds a signal into the system the statemachine corresponds to. The reaction also adds a correspondence between the communication signal in the statemachine and the signal in the system.
- *Synchronizing signal names:* Since the communication signal in the statemachine and the signal in the system represent the same signal, whenever a communication signal name is set, the reaction copies the same name to the corresponding signal.
- *Synchronizing components with regions:* Every component region is supposed to be represented by one statemachine region. But in the BCS, statemachine regions can also just be used for bundling multiple subregions. In that case, they only include subregion states without explicit transitions between them. If a region includes transitions between states, we can therefore deduce it represents the behavior of a component. Whenever a transition gets added to a statemachine region, the reaction checks if a corresponding component already exists. If that is not the case, it adds a component into the system, which has a correspondence to the statemachine the region is located in. Additionally, the reaction also adds a correspondence between the statemachine region and the component. This represents a more complicated version of Listing 2.1, since it also handles subregions bundled together in other regions. Since the setting of the region name might have happened before a transition was added, the reaction also has to check if a name for the region is set whenever the corresponding component was created. If that is the case, the reaction sets the component name to the statemachine region name.

- *Synchronizing component names with region names:* Since the behavior of a component should be represented by one statemachine region, whenever a region name is set, the reaction copies the same name to the corresponding component. This reaction is only used if the region name is set after a corresponding component was created for the region.

- *Synchronizing input ports with triggers:* When a signal appears as the trigger for a transition in a statemachine region, the signal needs to be receivable by the component with a correspondence to the region. Formulating this reaction is not trivial, since the needed port for the signal should only exist once per component. Whenever a communication signal is set as the trigger for a transition in a statemachine region, the reaction checks if an input port for the signal with a correspondence to the communication signal exists for the component with a correspondence to the statemachine region. If not, it creates the input port and correspondences between the input port and the region. This reaction represents a slight variation of Listing 2.2. The reaction skips the indirection step from the trigger to the statemachine region by directly adding a correspondence to said region.

- *Synchronizing output ports with behaviors:* When a signal appears as the behavior for a transition in a statemachine region, the signal needs to be sendable by the component with a correspondence to the region. Whenever a communication signal is set as the behavior for a transition in a statemachine region, the reaction checks if an output port for the signal with a correspondence to the communication signal exists for the component with a correspondence to the statemachine region. If not, it creates the input port and correspondences between the output port and the region. This represents a similar reaction as the one above, but instead of reacting to signals set in triggers, it reacts to signals sent in behaviors, and instead of creating input ports, it creates output ports.

The reactions from the statemachine domain to the component domain cannot simply be reversed to retrieve all the reactions from the component domain to the statemachine domain. Because of this, we list them separately:

- *Synchronizing roots:* Whenever a system gets initialized as the root of the component model instance, the reaction checks if the statemachine model instance has a corresponding statemachine initialized as the root. If this is not the case, it creates one and a correspondence between them.

- *Synchronizing root names:* Since the roots in both model instances are supposed to represent the same product, when setting the name in a system, the reaction copies the same name to the corresponding statemachine.

- *Synchronizing signals:* Whenever a signal is inserted into a system, the reaction checks if a corresponding communication signal exists in the statemachine. If not, it adds a communication signal into the statemachine, the system corresponds to. The reaction also adds a correspondence between the signal in the system and the communication signal in the statemachine.

- *Synchronizing signal names:* Since the signal in the system and the communication signal in the statemachine represent the same signal, whenever a signal name is set, the reaction copies the same name to the corresponding communication signal.

- *Synchronizing regions with components:* Because of the bundling mechanic in the BCS discussed in the statemachine domain to component domain reactions, statemachine regions added as a consequential change for the addition of a component could either be a region directly in the statemachine, or subregions in a state. To solve this issue, the reaction requires a user interaction. For the case study, we only allow new subregions to be created in a region contained in a statemachine, that is not a subregion itself. Whenever a component gets added, the reaction checks if a corresponding statemachine region already exists. If this is not the case, the reaction starts a user interaction and collects information about a potential subregion through it. If needed, the reaction creates a new region in the statemachine, and a new state to add a new subregion into. The reaction then adds a region into the statemachine, or a subregion into the new state, depending on the results of the user interaction. Additionally, the reaction also adds a correspondence between the component and this newly added region. A correspondence for the region, which includes a newly added subregion, does not get set.

- *Synchronizing region names with component names:* Since the behavior of a component should be represented by one statemachine region, whenever a component name is set, the reaction copies the same name to the corresponding statemachine region.

The list for the reaction from the component domain to the statemachine domain does notably not include reactions for the additions of input ports or output ports. This is by design, since the addition of a signal as a trigger or behavior requires the existence of that trigger or behavior. However, no other reaction guarantees the existence of transitions, triggers, or behaviors when input and output ports are getting added. This is the case, since the statemachine domain includes information not derivable from the component domain (as already discussed in the introduction of Section 5.2.1). Having a user interaction ask for those things is also not reasonable, since they are a big part of the statemachine domain and a potential component domain expert should not be required to need that knowledge to add input or output ports.

In addition to the cross-domain reactions, we also use intra-domain reactions for the component domain. They represent a domain **cleanup routine** that adds connectors and names for those connectors whenever ports that need connectors are added:

- *Adding connectors between input and output ports:* This cleanup routine aims at implementing auto-connect. A connector is automatically added between an output and an input port if they send/receive the same signal. We assume only one output port exists per signal in this case, since otherwise outputs become ambiguous. Whenever the signal is set in an input port or an output port, the reaction checks if input or output ports for this signal already exist. If that is the case, and a connector needs to be added between two ports, the

reaction checks if a connector corresponding to these ports already exists. If it does not exist, the reaction adds a connector and correspondences between the newly added connector and each port, respectively. This takes the task of manually creating connectors from the developer and completely automates it. We can now also ensure that every connector is always corresponding to the respective ports it connects.

- *Setting connector names:* Since a connector is always corresponding to two ports, the connector name should be descriptive of this fact. Whenever a connector gets added to the component model instance, the connector name is set to a name derived from the names of its corresponding ports.

A lot of our reactions (cross-domain and intra-domain) are strongly dependent on creating and looking up correspondences. The strong dependency to the correspondence model propagates to their delta counterparts, which is observable in Section 5.2.2. This solidifies the importance of RQ3 and the impact it can have on overall performance when using Delta-CPRs.

5.2.2 Delta Consistency Preservation Case Study

The delta consistency preservation case study includes a statemachine delta repository and a component delta repository, with cross-domain reactions between them and intra-domain reactions in the delta component domain. It represents the same standard and premium product and uses the same feature diagram, as seen in Section 5.2.1. The big difference is that the product model instances are not present directly but can be built via application of existing deltas. This also means, instead of two VSUM instances (one per product), only one instance of a Var-VSUM is present. We implement all deltas of the BCS that are required to construct the standard and premium products in such a way that applying the deltas according to the feature selection shown in Table 5.1 and Table 5.2 builds exactly the standard and the premium product, without derivation. This is crucial, since the application of the Delta-CPR testing kit relies on the equivalence of the constructed and original products.

To construct the Var-VSUM of this case study, we also require implementing the necessary Delta-CPRs in the form of reactions on deltas. These reactions are carefully implemented, since the goal is to apply all given reactions from the product case study (see Section 5.2.1) to deltas. The reactions are tested via the Delta-CPR testing kit with the goal of showing functional correctness. This is equivalent to a manual translation of the Product-CPRs to Delta-CPRs. All the listed reactions were designed and implemented by applying the knowledge from Sections 3.2 and 3.3. This includes the usage of delta mirroring for cross-domain consistency preservation and functional delta assignment for intra-domain consistency preservation (see Section 3.2.4). Additionally, the list only covers reactions that are triggered by the addition of a delta operation.

The manually constructed cross-domain reactions from the delta statemachine domain to the delta component domain are:

- *Synchronizing delta repository roots:* Both delta model instances represent the same SPL in different domains. We need to ensure that if the delta statemachine model instance can include deltas, the delta component model instance can also include deltas to enable cross-domain consistency preservation. Otherwise, applying delta mirroring or functional delta assignment fails when building a Var-VSUM from the ground up. Whenever a delta repository is added as the root of the delta statemachine model instance, the reaction checks if a corresponding delta repository exists. If not, it creates one, sets it as the root for the delta component model instance, and creates a correspondence between the two. This reaction is special since it cannot be tested for functional correctness but is always necessary when developing reactions for deltas in the context of this thesis. Alternatively, this reaction can be interpreted as an extra step in delta mirroring or functional delta assignment. Whenever one of the two is trying to add a delta to the target model instance, it checks if a delta repository exists. If this is not the case, it creates a delta repository and sets it as the root of the target model instance.

- *Synchronizing product roots:* Whenever an `InitializeRootStatemachineOperation` is added to the delta statemachine model instance, the reaction checks if the delta component model instance has a corresponding `InitializeRootSystemOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta component model of the source delta that the `InitializeRootStatemachineOperation` was added to.

- *Synchronizing product root names:* Whenever a `SetNameInStatemachineOperation` is added to the delta statemachine model instance, the reaction checks if the delta component model instance has a corresponding `SetNameInSystemOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta component model of the source delta that the `SetNameInStatemachineOperation` was added to. This reaction includes the first newly appearing correspondence check to properly translate bidirectional cross-domain attribute manipulation, and further *synchronizing names* reactions include them as well.

- *Synchronizing product signals:* Whenever an `AddCommunicationSignalOperation` is added to the delta statemachine model instance, the reaction checks if the delta component model instance has a corresponding `AddSignalOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta component model of the source delta that the `AddCommunicationSignalOperation` was added to.

- *Synchronizing product signal names:* Whenever a `SetNameInCommunicationSignalOperation` is added to the delta statemachine model instance, the reaction checks if the delta component model instance has a corresponding `SetNameInSignalOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta component model of the source delta that the `SetNameInCommunicationSignalOperation` was added to.

- *Synchronizing product components with regions:* This reaction is a bit more complicated and needs to be translated cautiously. The constructed products should only have a component in the component domain if a delta application added a statemachine region and the same, or a different, delta application added a transition into that statemachine region. However, we need to maintain parallel dependencies to apply delta mirroring. So the placement of the `AddComponentOperation` in the component domain is crucial. Whenever an `AddTransitionOperation` is added to the delta statemachine model, the reaction searches for the `AddRegionOperation` it depends on (see Section 3.2.3). Once the operation is found, the reaction checks if the delta component model instance has a corresponding `AddComponentOperation`. If this is not the case, it creates one and a correspondence between them. It adds the new operation to the mirror delta for the delta the `AddRegionOperation` is located in.

- *Synchronizing product component names with region names:* Whenever a `SetNameInRegionOperation` is added to the delta statemachine model instance, the reaction checks if the delta component model instance has a corresponding `SetNameInComponentOperation` and also checks if a potential `AddComponentOperation` for the component exists. If no corresponding operation was found and the `AddComponentOperation` exists, it creates a `SetNameInRegionOperation` and a correspondence between it and the `SetNameInRegionOperation`, then adds it to the mirror delta in the delta component model of the source delta that the `SetNameInRegionOperation` was added to.

- *Synchronizing product input ports with triggers:* Whenever a `SetTriggerOperation` is added to the delta statemachine model instance, the reaction searches for the `AddRegionOperations` and `AddCommunicationSignalOperations` it depends on. It then fetches the corresponding `AddComponentOperations` and `AddSignalOperations` from the delta component model instance. This cannot fail, since setting a trigger for a transition is only possible if the transition exists in a statemachine region, so an `AddComponentOperation` can always be found. After fetching all necessary operations, the reaction checks if an `AddInputPortOperation` for the component added by the `AddComponentOperation` and the signal added by the `AddSignalOperation` exists. If that is not the case, it creates a new `AddInputPortOperation` for the component and signal and adds a correspondence between the new operation and the originally found `AddRegionOperation`. This represents the deviation from Listing 2.2 already discussed in Section 5.2.1. If an existing `AddInputPortOperation` is found, the reaction just copies that one. Multiple copied ports only appear once on the final constructed product (see Section 3.9). The newly created/copied operation is then added to the mirror delta in the delta component model of the source delta that the `SetTriggerOperation` was added to. Additionally, the reaction also adds a `SetNameOperation` and a `SetSignalOperation` to the same target delta, depending on the `AddSignalOperation`.

- *Synchronizing product output ports with behavior:* Whenever a `SetBehaviorOperation` is added to the delta statemachine model instance, the reaction searches for the `AddRegionOperation` and `AddCommunicationSignalOperation` it depends on. It then fetches the corresponding `AddComponentOperation` and `AddSignalOperation` from the delta component model instance.

After fetching all necessary operations, the reaction checks if an `AddOutputPortOperation` for the component added by the `AddComponentOperation` and the signal added by the `AddSignalOperation` exists. If that is not the case, it creates a new `AddOutputPortOperation` for the component and signal and adds a correspondence between the new operation and the originally found `AddRegionOperation`. If an existing `AddOutputPortOperation` is found, the reaction just copies that one. The newly created/copied operation is then added to the mirror delta in the delta component model of the source delta that the `SetBehaviorOperation` was added to. Additionally, the reaction also adds a `SetNameOperation` and a *set signal* to the same target delta, depending on the `AddSignalOperation`.

The manually constructed cross-domain reactions from the delta component domain to the delta statemachine domain are:

- *Synchronizing delta repository roots:* Whenever a delta repository is added as the root of the delta component model instance, the reaction checks if a corresponding delta repository exists. If not, it creates one, sets it as the root for the delta statemachine model instance, and creates a correspondence between the two. The same distinct properties of this reaction remain the same as for the equivalent from the delta statemachine domain to the delta component domain.
- *Synchronizing product roots:* Whenever an `InitializeRootSystemOperation` is added to the delta component model instance, the reaction checks if the delta statemachine model instance has a corresponding `InitializeRootStatemachineOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta statemachine model of the source delta that the `InitializeRootSystemOperation` was added to.
- *Synchronizing product root names:* Whenever a `SetNameInSystemOperation` is added to the delta component model instance, the reaction checks if the delta statemachine model instance has a corresponding `SetNameInStatemachineOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta statemachine model of the source delta that the `SetNameInSystemOperation` was added to.
- *Synchronizing product signals:* Whenever an `AddSignalOperation` is added to the delta component model instance, the reaction checks if the delta statemachine model instance has a corresponding `AddCommunicationSignalOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta statemachine model of the source delta that the `AddSignalOperation` was added to.
- *Synchronizing product signal names:* Whenever a `SetNameInSignalOperation` is added to the delta component model instance, the reaction checks if the delta statemachine model instance has a corresponding `SetNameInCommunicationSignalOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta statemachine model of the source delta that the `SetNameInCommunicationSignalOperation` was added to.

- *Synchronizing product regions with components:* Just like the product variant of this reaction, the delta variant also requires a user interaction. Whenever an `AddComponentOperation` gets added to the delta component model instance, the reaction checks if the delta statemachine model instance has a corresponding `AddRegionOperation`. If this is not the case, the reaction starts a user interaction and collects information about a potential subregion through it. If needed, the reaction creates a new `AddRegionOperation` and `AddStateOperation`. The reaction then creates an `AddRegionOperation` for the new region or subregion, depending on the results of the user interaction. Additionally, the reaction also adds a correspondence between the `AddComponentOperation` and this newly created `AddRegionOperation`. All newly created operations are added to the mirror delta in the delta statemachine model of the source delta that the `AddComponentOperation` was added to.

- *Synchronizing product region names with component names:* Whenever a `SetNameInComponentOperation` is added to the delta component model instance, the reaction checks if the delta statemachine model instance has a corresponding `SetNameInRegionOperation`. If this is not the case, it creates one and a correspondence between them, then adds it to the mirror delta in the delta statemachine model of the source delta that the `SetNameInComponentOperation` was added to.

The cross-domain reactions from the delta component domain to the delta statemachine domain do not include reactions that trigger for the addition of an *add input port* or `AddOutputPortOperation`, for the same reason, the product reactions do not include the product equivalent.

The cleanup routine is also implemented with reactions for the delta consistency preservation case study. The two reactions work as follows:

- *Adding product connectors between input and output ports:* An `AddConnectorOperation` is automatically added if an `AddInputPortOperation` and an `AddOutputPortOperation` exist, with a `SetSignalInInputPortOperation` and a `SetSignalInOutputPortOperation` associating them with the same signal. Just as with the product variant, we assume only one `SetSignalInPortOperation` exists for an output port per signal, since otherwise outputs become ambiguous when applying the deltas. Realizing this reaction is especially problematic, since the existence of connectors depends on ports, which themselves only exist if no other port for that signal was added to the component via a `AddPortOperation` during construction time. The problem can be seen as a two-layer existing element dependency problem (see Section 3.4.3). Whenever a `SetSignalInInputPortOperation` or `SetSignalInOutputPortOperation` is added, the reaction checks if other `SetSignalInInputPortOperation` or `SetSignalInOutputPortOperation` already exist for this signal. Since solving the element dependency problem for ports includes the addition of multiple `AddPortOperations` for the same port, fetching an *add input port* and an `AddOutputPortOperation` and checking for any connectors corresponding to them is not sufficient. Instead, the reaction checks if an `AddConnectorOperation` corresponding to a fetched `AddSignalOperation` exists between the connector

source and target components. If this is the case, it modifies the delta in which the `AddConnectorOperation` was found in such a way that, in addition to the existing application conditions and order, it also gets applied whenever (and after) the `SetSignalInInputPortOperation` and `SetSignalInOutputPortOperation` are applied. If not, it creates a new delta that only gets applied whenever (and after) the `SetSignalInInputPortOperation` and `SetSignalInOutputPortOperation` are applied and adds a `AddConnectorOperation` to it. The reaction also adds a `SetSourcePortOperation` and a `SetTargetPortOperation` for the two ports into the new delta. Additionally, it adds a correspondence between the `AddConnectorOperation` and the `AddInputPortOperation`, the `AddOutputPortOperation`, and the `AddSignalOperation` for the two components (needed for duplicate detection as seen above).

- *Setting product connector names:* Whenever an `AddConnectorOperation` is added to the delta component model instance, the reaction fetches the `SetNameInPortOperations` for the two ports connected by the connector and creates a `SetNameInConnectorOperation` based on them. The new operation is then added into the same delta the `AddConnectorOperation` is located in, and the delta is renamed accordingly to have a meaningful name.

5.3 Evaluation Experiments and Results

In this section, we will describe how the subject system for evaluation was used to answer our research questions (Section 5.1), and list the answers themselves. For every research question, we elaborate on the setup, the procedure that was performed, the results we got from the procedure, and interpret the results in a discussion paragraph.

5.3.1 Manual Translation of Consistency Preservation Rules

In this experiment, we aim to answer RQ1: Is it possible to apply the CPR concept to delta-oriented modeling?

Setup

We implement the product case study of the BCS as described in Section 5.2.1. This includes both products in their base and complete variants, as well as all Product-CPRs between the statemachine and component domain and the cleanup routines in the component domain. We generate the delta statemachine and delta metamodels from the generic delta metamodel to allow for the creation and application of deltas in these domains. The delta statemachine and delta component metamodels are used to create model instances, which allow the exact construction of the products in their base and complete variants by applying the deltas included in them. Additionally, we implement the Delta-CPR testing kit to test delta model instances for functional correctness.

Procedure

We analyze the Product-CPRs and manually determine how we can generate functionally correct Delta-CPRs from them. Problems that arise during this process get

solved and are documented in Section 3.2. We design and implement the Delta-CPRs between the delta statemachine and delta component domains, as well as the cleanup routines in the delta component domain. The resulting Delta-CPRs represent the ones listed in Section 5.2.2. After fully implementing the deltas and Delta-CPRs for the BCS case study, we apply the Delta-CPR testing kit to it to test for functional correctness. The original change, which the Delta-CPRs has to react to, in this case represents the changes applied by higher-order deltas to derive the complete variants from the base variants of the products. In total, the Delta-CPR testing kit is applied four times: Twice to test the functional correctness of the Delta-CPRs from the delta statemachine domain to the delta component domain by applying the higher-order deltas for the standard and premium products, respectively, to the delta statemachine model instance. And two more times to test the functional correctness of the Delta-CPRs from the delta component domain to the delta statemachine domain by applying the higher-order deltas for the standard and premium products, respectively, to the delta component model instance. The delta component domain cleanup routines are also being tested in the first two testing kit applications, since the consequential changes by the cross-domain Delta-CPRs trigger them and their result is part of the final deltas applied by the Delta-CPR testing kit for the equivalence check.

Results

The Delta-CPR testing kit confirms that all constructed products are equivalent (identical except for unique IDs) to the ones from the product case study. This means the Delta-CPRs we designed and implemented were all functionally correct. We manage to completely apply the concept of consistency preservation from the product BCS case study to the delta BCS case study, while preserving consistency properties on the product level.

Discussion

We were successful in applying the CPR concept to deltas for the BCS case study. The tools we used to do so are entirely applicable to any delta metamodel created by using the generic delta metamodel. This makes the concept of Delta-CPRs also applicable to other delta-based case studies. It is important to note that the experiment for this research question requires the implementation and usage of the Delta-CPR testing kit, but the testing kit is part of the answer for RQ2. Tooling to find answers to these two research questions got developed in parallel, and the experiment for this research question is highly dependent on the answer of RQ2.

5.3.2 Using the Delta Consistency Preservation Rules Testing Kit

In this experiment, we aim to answer RQ2: How can we test functional correctness of CPRs for delta-oriented modeling?

Setup

The setup is almost identical to the one from Section 5.3.1, since both experiments are conducted in parallel. The only difference being the implementation status of the Delta-CPR testing kit. For this experiment, the testing kit implementation is not part of the setup, because the design and implementation are part of the experiment procedure itself.

Procedure

Before analyzing the Product-CPRs to determine how to generate functionally correct Delta-CPRs from them to answer RQ1, we already implemented the Delta-CPR testing kit. The testing kit is then applied iteratively to confirm the functional correctness for the implementation of each reaction. This means, in addition to the four applications at the end, the Delta-CPR testing kit is applied at least once per implemented reaction to answer RQ2. The Product-CPRs applied on the product side of the testing kit have to be reduced to the already translated ones for the tests to be meaningful. Whenever an implementation is faulty, the application of the Delta-CPR testing kit will notify the developers of it. This ensures no implemented reactions depend on consequential changes performed by faulty older reactions, since faulty reaction implementations can be fixed before starting with the implementation of new ones.

Results

The repeating application of the Delta-CPR testing kit significantly helps to translate the reactions. Even considering the size of the model instances for our BCS case study implementation, spotting faulty reactions without testing kit application is unfeasible. Some problems arising when implementing the *synchronizing product input ports with triggers*, *synchronizing product output ports with behaviors*, as well as the *adding product connectors between input and output ports* reaction, were not visible during the design and initial implementation phase. Only the application of the testing kit confirmed the existence of these problems, and the exact comparison results gave a starting point for fixing them. Additionally to the iterative application, we also use the Delta-CPR testing kit to test all reactions together in a final integration test. This final integration test passed once all the design problems and faulty implementation issues were solved.

Discussion

The Delta-CPR testing kit does not only turn out to be an effective tool to test for functional correctness of Delta-CPRs, but also to be very versatile. It can be applied to either a single or a group of Delta-CPRs for arbitrary changes on arbitrary model instances, as long as the Product-CPRs, the product model instances they operate on, and the product variant of the changes also exist. Because of this, the Delta-CPR testing kit serves the role of unit tests but also integration tests for Delta-CPRs. The comparison results of the constructed and the baseline product give the developer insight into why the testing kit application potentially did not pass and allow for quick fixing of the problems after discovery.

5.3.3 Measuring Scalability of Delta Consistency Preservation Rules

In this experiment, we aim to answer RQ3: Does the CPR application on deltas result in significant changes to the number of correspondences between the model instances?

Setup

The setup for this experiment consists of performing all setup and procedure steps described in the experiment from Section 5.3.2, to ensure all model instances and CPRs are correctly implemented both for products and for deltas. This includes the Delta-CPR testing kit applications.

Procedure

After applying the higher-order delta changes to the base variants to construct the complete variants, we count the number of `EObjects` for both the complete standard and premium products in both domains. Additionally, we also count the number of correspondences that Vitruv adds to the correspondence models for both the complete standard and premium products. This includes the correspondences, which are added by applying the higher-order changes, but also the ones from building the base variant. We also count the number of `EObjects` in the delta model instance in both domains. For deltas, we also count the number of correspondences that Vitruv adds to the correspondence model of the delta model instance when building the delta repository from the ground up. All these numbers can then be compared. Since the core principle of using SPLs with Ecore is the reuse of `EObjects` when enough valid products are part of it, we calculate the correspondences per `EObject` ratio. If the number of correspondences in both product model instances combined divided by the number of `EObjects` in both product model instances combined is bigger than the number of correspondences in the delta model instance divided by the number of `EObjects` in the delta model instance, correspondences scale when growing the SPL and adding more products. Additionally, we can also discover how significant the difference is for a minimal number of products in an SPL (two products).

Results

The counted number of `EObjects` in all instances and domains are listed in Table 5.3. To calculate a correspondence per `EObject` ratio, we also require the numbers of correspondences added by Vitruv for each model instance. The counted number of correspondences for every instance is listed in Table 5.4. To calculate a correspondence per `EObject` ratio, we also require the numbers of correspondences added by Vitruv for each model instance. The counted number of correspondences for every instance is listed in Table 5.4. We can perform some calculations to adequately answer RQ3. The correspondence per `EObject` ratio for both products combined is $\frac{548}{784+354} \approx 0.48$. Meanwhile, the correspondence per `EObject` ratio for the delta repository able to build both products is $\frac{666}{2056+1319} \approx 0.2$. A few things become apparent when looking at all the ratios: First, the delta-base case study uses a far smaller granularity for `EObjects` than the product-based one. This is to

	(Delta) statemachine domain	(Delta) component domain
Standard product	294	131
Premium product	490	223
Combined product	784	354
Delta repository	2056	1319

Table 5.3: Number of `EObjects` in the implemented BCS model instances, as well as the delta model instances. The number of `EObjects` in the model instances combined includes doubles.

	Count of correspondences
Standard product	200
Premium product	348
Combined product	548
Delta repository	666

Table 5.4: Number of correspondences added by Vitruv for the implemented BCS model instances, as well as the delta model instances.

be expected, since `EObjects` with multiple attributes get sliced into an `AddOperation` and multiple `SetAttributeInOperations`. Second, the delta-base case study actually uses more correspondences than the product-based case study. This is also expected, since many new correspondences (for example, between deltas) are newly introduced to implement concepts such as delta mirroring. But despite this fact, the correspondence per `EObject` ratio for the delta-based case study is less than half that of the product-based one. Every addition of a new product to this experiment would increase the number of duplicate elements between the product-case case studies. Even for the worst possible SPL case (two products), the product-based case study has only $(1 - \frac{548}{666}) \approx 17.7\%$ fewer correspondences, making an SPL usage feasible. In summary, the scaling properties of SPLs also apply to the number of correspondences added by Vitruv. Although the migration from product-based to delta-based case studies might increase the number of `EObjects` for SPLs with only a few products, the number of correspondences does not change significantly enough to make the application of Delta-CPRs on delta-oriented case studies unfeasible in that case.

Discussion

For SPLs with very few products, the usage of deltas and Delta-CPRs will actually increase the number of correspondences added and maintained by Vitruv. This is to be expected, since a lot more correspondences are required for Delta-CPRs application. For SPLs with many products, the amount of `EObject` duplication between the product model instances will be quite noticeable, and at some point, the number of total `EObjects` in the product-based variant surpasses the amount of total `EObjects` in the delta-based variant. This is a basic concept of SPL usage. The experiment indicates that this also transfers to correspondences. As we can see by the correspondences to `EObject` ratio, the duplication of `EObjects` also leads to duplication of correspondences between the product model instances. This leads to the delta-based variant having superior scaling properties regarding correspondences.

5.3.4 Automatic Translation of Consistency Preservation Rules

In this experiment, we aim to answer RQ4: Is it possible to automatically translate Product-CPR to a Delta-CPR?

Setup

The complete BCS case study on products and on deltas gets implemented. This includes both the standard and the premium product variant, as well as the cross-domain CPRs between the domains and the cleanup routines. Additionally, the Delta-CPR testing kit gets implemented and applied to ensure all Delta-CPRs in the case study are functionally correct, and the model instances are complete.

Procedure

This research question can be answered by implementing an automatic translator for the constructs listed in Section 3.4.2. In that case, the procedure would include applying the implemented translator to generate Delta-CPRs, testing their functional correctness with the Delta-CPR testing kit for the implemented case study, and evaluating how many of the reactions could be successfully translated automatically. But the complete implementation is out of scope for this thesis, and we only implemented some basic translation functionality (see Section 4.5 for details). Instead, we check the Product-CPR implementation for all mappable and problematic constructs listed in Section 3.4.2 and 3.4.3. This way, we can theoretically determine which reactions can be automatically translated without issues and which ones include problematic statements. Not only do we assign each reaction to one of these two categories, but we also analyze what parts of the reaction have to be manually implemented.

Results

Table 5.5 shows the result of the automatic translatability analysis. Only two of our reactions cannot be automatically translated: *Synchronize regions with components* and *adding connectors between input and output ports*. The reasons for each are very different, but both are already listed and explained in Section 3.4.3. The *synchronize regions with components* reaction requires a user interaction, since it is not possible for the system to automatically determine if a region corresponding to an added component is a subregion of another already existing region. This user interaction must be translated manually. The rest of the reaction can be automatically translated, meaning the translation effort is relatively small. The *adding connectors between input and output ports* reaction runs into a two-level existing element dependency problem. The details for this reaction are already discussed, since it served as the example for the problem introduction. To resolve this issue, the reaction either has to search for `AddPortOperations` in the whole delta repository, or a developer has to implement another solution to the problem. In our implementation, we chose the latter and added an extra correspondence per connector, as described in Section 5.2.2. All other reactions only included constructs, which are unambiguously mappable, making them automatically translatable.

Reaction	Auto-translatable?
Synchronizing roots (both directions)	Yes
Synchronizing root names (both directions)	Yes
Synchronizing signals (both directions)	Yes
Synchronizing signal names (both directions)	Yes
Synchronizing components with regions	Yes
Synchronizing regions with components	No
Synchronizing component names with region names	Yes
Synchronizing region names with component names	Yes
Synchronizing input ports with triggers	Yes
Synchronizing output ports with behaviors	Yes
Adding connectors between input and output ports	No
Setting product connector names	Yes

Table 5.5: Table of all reactions and their results for the automatic translatability analysis.

Discussion

The automatic translatability analysis for the BCS implementation shows that not every possible Product-CPR can be easily and automatically translated to a Delta-CPR. While it is possible to translate a subset of possible Product-CPR like that, it is not possible in general. It seems very unrealistic in a real-world project to only have such reactions that use easily mappable constructs. Because of this, an automatic CPR translator should be seen less as a complete migration tool, but more as a migration assistance for reaction developers. It can drastically reduce the amount of manual labor required but cannot perform complete product-based to delta-based CPR translations fully automatically.

5.4 Threats to Validity

5.4.1 Threats to Construct Validity

Correctness definition not always applicable: The correctness definition for CPRs as presented in Section 3.2.2 is only applicable to Delta-CPRs that originate from Product-CPRs and always requires a one-to-one mapping between Product-CPRs and Delta-CPRs. This questions the whole validity of the Delta-CPRs testing kit. While it is possible to translate a single Product-CPR to multiple Delta-CPRs and vice versa, our operation mappings show that it is always possible to translate in such a way that a one-to-one mapping exists. This is the case, since one trigger can always be translated to one trigger, and everything else can be merged from multiple reactions into one. In the case of the correctness definition not being applicable because the Delta-CPRs only exist in the delta domain, this approach cannot help. In our BCS case study implementation, one of these Delta-CPRs already appeared in the form of *synchronizing delta repository roots*, but others are also possible. Our correctness definition and the Delta-CPRs testing kit are specifically designed to represent and test a sensible idea of Delta-CPRs. The goal is for Delta-CPRs to preserve consistency for the constructed products the same way as Product-CPRs would, but on a delta level. Delta-CPRs that have no counterpart on products

are explicitly excepted from this idea, since they cannot enforce rules on products. For this kind of reaction, the only correctness criterion is the intent of the reaction developer.

No implementation for the automatic translation: The translatability analysis only works on a theoretical level. Since no formal proofs are provided for the translatability properties of the Product-CPR, the whole analysis represents an informal application of the mappings from Section 3.4.2. A full mapping analysis was out of scope for the thesis. While a translatability analysis is a good and important first step, a full implementation of a translator in the future would reinforce our claims.

5.4.2 Threats to External Validity

Constructing model instances from empty instances: We build our models from empty model instances instead of iteratively, as in a real-world scenario. Instead of applying changes to the already existing base variants of the products to create the complete variant, we build the complete variants by applying deltas to an empty model instance. We selected this procedure because it is practical to implement and allows for a quick and repeatable evaluation. However, as compensation, the testing kit also checks the successful construction of the base variants from deltas (see Section 4.4), removing this derivation as a factor that might influence the results.

Experiments and application only on the BCS case study: All of our experiments were only conducted on a single case study. Notably, the BCS only includes two domains. Cross-domain consistency preservation in more than two domains might uncover new problems that did not appear in our case study, especially when combined with intra-domain consistency preservation. Therefore, the proposed framework should be applied to more case studies in the future, including such case studies with more than two domains. For the scope of this thesis, we only applied it to the BCS case study as a proof of concept and to collect data for an evaluation of the approach. This enables us to answer some fundamental research questions and show trends regarding scalability.

No guarantee for completeness of mappable/problematic construct list: The list of mappable constructs and problematic constructs was created based on the constructs found in the BCS case study, since those were all the necessary ones for our application. However, covering all possible constructs is unfeasible for the scope of this thesis. The list has no guarantee for completeness and might not include constructs present in other case studies, although most of the common constructs in Ecore conform models are already explored. If the framework and Delta-CPRs are applied to further case studies and such constructs are found, the list has to be extended accordingly.

5.5 Feasibility of Delta Consistency Preservation Rules

Product-CPRs can always be manually translated to deltas by a reaction developer. The goal for such a translation is the retrieval of functionally correct Delta-CPRs, meaning the Delta-CPRs preserve consistency for products constructed by applying the deltas the same way the Product-CPRs they originate from do (see Definition 3.1). This functional correctness cannot easily be proven but can be tested with the application of the Delta-CPR testing kit. The testing kit can guide the developer to achieve functionally correct translations and help with cross-domain and intra-domain CPRs. The newly translated Delta-CPRs get applied to more fine-granular models with more correspondences, but the scaling properties of SPLs also apply to those correspondences. While a manual translation is always possible, a fully automatic translation is most likely unfeasible, but an automatic translation tool can be used to assist a reaction developer, who performs the translation task. These results make the migration from platform-based to delta-based development with the translation from Product-CPRs to Delta-CPRs feasible in practice, with some manual development effort still being required.

6. Related Work

Consistency Preservation without Variability

The literature provides many ways to mostly detect but also preserve some kind of consistency in models, with or without tool support. This includes intra-domain consistency, but also cross-domain consistency. For cross-domain consistency, there are complete literature reviews like the one by Torres et al.[TVS21], which cites a collection of different approaches to check for cross-domain consistency, most of which come with tool support. However, none of them are explicitly designed for application on SPLs.
While the application of most of these consistency-checking approaches to delta-oriented modeling could yield interesting results, the approach we chose is Vitruvius[Kla+21]. Vitruvius is an approach to consistency preservation that utilizes explicit consistency repair rules to preserve consistency when changes to the model occur. We use Vitruvius as a tool and build on top of it by introducing new concepts, such as functional correctness. This makes our work not only a pure application of Vitruvius to a new metamodel but also provides a meaningful addition in terms of consistency preservation on SPLs.

Consistency Preservation with Variability

The literature also provides many ways to detect and deal with inconsistencies for models with variability, most of which have tool support. Literature reviews like the one by Santos et al.[SDD15] help to give an overview but also reveal that many publications are about consistency between entirely different artifacts. They can either aim at checking or preserving consistency in the problem space, in the solution space, or in between them.
One of these publications is *Co-Evolution of Models and Feature Mapping in Software Product Lines*[SHA12]. It offers a solution for dealing with the coevolution of the domain model and corresponding feature mappings. Feature mappings are explicit mappings between selected features in the problem space and artifacts in the solution space. They employ strategies to maintain consistency between the domain model and feature mappings and develop the FeatureMapper tool to automatically enforce this consistency. While the FeatureMapper tool works with any Ecore model

(just like our approach), the goal is entirely different, and cross-domain consistency in the solution space is not considered at all.
Mapping feature models onto domain models: ensuring consistency of configured domain models[BW14] takes a similar approach. They aim to achieve a consistent mapping of the feature model onto the domain model by adding constraints for feature annotations. So there is no cross-domain consistency preservation involved. Additionally, this means they employ an annotational approach, which is entirely different from the delta-oriented approach we are using, making it inapplicable to our use case.
Automated Analysis of Dependent Feature Models[Sch+13] introduces a new way to structure the problem space: dependent feature models. The usage of dependent feature models leads to splitting the problem space into multiple model instances for multiple different stakeholders of the project. Between these model instances, they employ a consistency detection mechanism with tool support. This approach relates to our approach in the sense that it tries to detect inconsistencies between models in only one of the two spaces (problem space or solution space). But instead of preserving consistency, it only provides a consistency-checking mechanism and is only applicable in the problem space, while our approach is only applicable in the solution space.
Structuring the modeling space and supporting evolution in software product line engineering[Dhu+10] cuts the domain model into multiple partial models, called fragments. They present an approach to detect inconsistencies between product line artifacts and the models representing these artifacts after changes (such as these model fragments). Their approach heavily relies on utilizing the decision-based variability modeling approach. It is not directly applicable to our delta-oriented approach, only detects inconsistencies, and does not support actual cross-domain consistency.
Detecting Inconsistencies in Multi-View Models with Variability[LE10] proposes a technique called SafeComposition for structural cross-domain consistency. SafeComposition checks for type safety across domain model instances by using SAT-Solvers. The main differences to our approach are the inconsistency detection, instead of consistency preservation, as well as the limited consistency rule scope. While our approach can preserve arbitrary consistency rules, their approach can only check for type safety, meaning they check if structural elements in one domain are also present in another domain. While this consistency scope can be useful in some cases, it is severely limited.
Towards Fixing Inconsistencies in Models With Variability[LE12] takes a first step towards cross-domain consistency preservation. The goal of the publication is to determine where consequential changes to preserve consistency should be located. Their research question is very similar to the localization problem we discuss in Section 3.2.4. However, their solution is completely bound to feature-oriented SPL implementations instead of delta-oriented ones. Because of the change in granularity when using a delta-oriented SPL implementation, the problem changes and their solution is no longer applicable.
Another tool-assisted approach to consistency preservation for models with variability is *Variants and Versions Management for Models with Integrated Consistency Preservation*[Ana+18] (VaVe). VaVe is an approach for preserving consistency between variants and versions of a product. This means it is built to handle variability in space (variants) and variability in time (versions). VaVe utilizes Vitruvius, just

like our approach, allowing for arbitrary CPR definition. While VaVe is an extension and application of the Vitruvius approach just like our work, it is different because of the single-domain scope used by VaVe and because it does not utilize the delta-oriented modeling approach. Instead of preserving consistency on a multi-domain level, they aim to preserve consistency between variants and versions in a single domain.

Incremental Consistency Checking in Delta-oriented UML-Models for Automation Systems[KS16] introduced categories for consistency rules and applies incremental cross-domain consistency checking on specific UML models for each category. Three different cross-domain consistency approaches are presented, two being product-based and one being delta-based. All approaches are hand-tailored to an application on UML models and not applicable to a wide range of domains. Additionally, only consistency checking and no consistency preservation is discussed.

Towards Consistency Preservation for Multi-Domain Variability Modeling [NPS25] uses a lot of the same fundamentals as we do in this thesis. Their approach differs from ours in the level of change abstraction. Changes are being applied to a product model instance of one domain in the form of a delta. Vitruvius CPRs react to the delta application and create consequential changes in the product model instance of another domain that can be transformed back into a delta. This way, it is possible to retrieve a delta containing all consequential changes in a domain that must be applied when applying a delta from another domain. Our approach works differently. Instead of reacting to a delta application in one domain, we react to a higher-order delta application in a delta domain. This means we do not react to a change in the product but to a change in deltas. The consequential changes in another delta domain retrieved from a Delta-CPR application is also not a delta, but a higher-order delta, meaning it represents a change on deltas.

7. Conclusion and Outlook

Conclusion

The thesis introduces all the basics needed for constructing a meaningful cross-domain consistency preservation mechanism for delta-oriented modeling approaches. With all the basics introduced, we propose Delta-CPRs and formal correctness for them. While the focus is clearly on preserving cross-domain consistency with them, we also apply our findings to intra-domain consistency preservation. Our correctness definition explicitly enables preserving the consistency for all products contained in the product line, and we provide a tool to test for it in the form of the Delta-CPR testing kit. In contrast to previous literature, all Delta-CPR applications are reactions to changes on deltas and not changes to the products directly. This allows for our consistency preservation framework to be used for platform-based development, where no consistency preservation approach existed before. Since platform-based development has numerous advantages over product-based development, solving the non-existence of consistency preservation mechanisms for it, with the introduction of our framework, enables an even more advantageous way to develop product lines. We apply the framework to a cyber-physical system case study to test the practical applicability and evaluate it. During this evaluation, we find that the framework is not only practically applicable for cross-domain and intra-domain consistency preservation, but the Delta-CPR testing kit works as intended and provides a useful tool for consistency testing. Additionally, we discover that the scalability advantages of using a platform-based development approach in the form of delta-oriented modeling directly transfer to the application of Delta-CPRs. Based on these findings, one might want to implement a cyber-physical product line using platform-based development instead of product-based development and apply the proposed framework just like we did in our case study to keep the advantage of automatic consistency preservation. However, if the product line is already implemented with a product-based approach and Product-CPRs, a full migration is necessary. The thesis includes sections for this exact migration by laying the foundation for an automatic translation of Product-CPRs to Delta-CPRs. We identify directly translatable and problematic constructs in Product-CPRs and give translation mappings, if possible. This categorization and the mappings can be used in tool support development to fully implement a tool that can automatically translate most constructs

in a Product-CPRs. Such a tool could simplify the migration from platform-based development to product-based development for projects using Product-CPRs by requiring minimal manual labor from a developer to migrate these Product-CPRs.

Outlook

Future work should include applying the framework on larger case studies, with more than two domains. This would help to ensure the scalability of Delta-CPRs also applies for larger product lines and discover if more than two domains lead to new challenges for a Delta-CPR application. Another direction for future research is a refinement of the Delta-CPR correctness. On one hand, testing the correctness might not be sufficient for some projects, where incorrect models can lead to high damages. In these cases formally proving functional correctness of a Delta-CPR might be necessary. One approach for this could be a restriction of constructs that are allowed in a CPR. On the other hand, even if proving functional correctness of Delta-CPRs is not necessary for modeling a system, the correctness definition is not applicable to all possible Delta-CPRs. Some Delta-CPRs do not have a corresponding Product-CPR. An example of this would be a Delta-CPR reacting to the change of a delta application condition. There cannot be any mapping for these constructs, making further research towards an expansion of your functional correctness definition helpful. In the scope of this these, such Delta-CPRs did not appear and were not discussed further. Future work should also use and expand the foundation we build in this thesis. While we provide mappings for the most frequent CPR constructs, our list has no claim for completeness. Additionally, future research could also consider correctness by construction. If a Product-CPR can be fully translated to a Delta-CPR with the help of an implemented translator, the Delta-CPR should always be functionally correct if that translator works as intended.

This thesis performed the first steps necessary for the practical application of automatic consistency preservation on delta-oriented product lines by introducing Delta-CPRs, which enables reacting to the application of higher-order deltas. However, not all related questions have been fully answered yet. The research field allows for further analysis of correctness definitions and Delta-CPR applications and more tool support for automatic CPR migration. Additional research on these topics, combined with the findings in this thesis, can lead to a seamless and fully automated consistency preservation for multi-domain product line engineering projects.

Bibliography

[Ana+18] Sofia Ananieva et al. "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. Madrid, Spain: Association for Computing Machinery, 2018, pp. 3–10. ISBN: 9781450353984. DOI: 10.1145/3168365.3168377. URL: https://doi.org/10.1145/3168365.3168377.

[Ant+14] Michał Antkiewicz et al. "Flexible product line engineering with a virtual platform". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 532–535. ISBN: 9781450327688. DOI: 10.1145/2591062.2591126. URL: https://doi.org/10.1145/2591062.2591126.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Vol. 1. Sept. 2012. DOI: 10.2200/S00441ED1V01Y201208SWE001.

[BW14] Thomas Buchmann and Bernhard Westfechtel. "Mapping Feature Models onto Domain Models: Ensuring Consistency of Configured Domain Models". In: *Software & Systems Modeling* 13.4 (Oct. 2014), pp. 1495–1527. ISSN: 1619-1374. DOI: 10.1007/s10270-012-0305-5.

[CHS15] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. "Abstract Delta Modelling". In: *Mathematical Structures in Computer Science* 25.3 (Mar. 2015), pp. 482–527. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129512000941. (Visited on 04/20/2025).

[CN01] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201703327.

[Dhu+10] Deepak Dhungana et al. "Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering". In: *Journal of Systems and Software* 83.7 (2010), pp. 1108–1122. ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.02.018.

[DKS00] Jorge Díaz-Herrera, Peter Knauber, and Giancarlo Succi. "Issues and Models in Software Product Lines". In: *International Journal of Software Engineering and Knowledge Engineering* 10 (Aug. 2000), pp. 527–539. DOI: 10.1142/S0218194000000286.

[Gur00] Jilles van Gurp. *Variability in software systems: the key to software reuse*. Department of Software Engineering & Computer Science, Blekinge Institute of ..., 2000.

[Kla+21] Heiko Klare et al. "Enabling consistency in view-based system development — The Vitruvius approach". en. In: *Journal of Systems and Software* 171 (Jan. 2021), p. 110815. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110815. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121220302144 (visited on 05/26/2025).

[Kla+25] Heiko Klare et al. *vitruv-tools/Vitruv: 3.2.2*. Version v3.2.2. July 2025. DOI: 10.5281/zenodo.15781406. URL: https://doi.org/10.5281/zenodo.15781406.

[Kla16] Heiko Klare. *Designing a Change-Driven Language for Model Consistency Repair Routines*. de. 2016. DOI: 10.5445/IR/1000080138. URL: https://publikationen.bibliothek.kit.edu/1000080138 (visited on 05/28/2025).

[Kra17] Max Emanuel Kramer. "Specification Languages for Preserving Consistency between Models of Different Languages". PhD thesis. Karlsruher Institut für Technologie (KIT), 2017. 278 pp. DOI: 10.5445/IR/1000069284.

[KS16] Matthias Kowal and Ina Schaefer. "Incremental Consistency Checking in Delta-oriented UML-Models for Automation Systems". In: *Electronic Proceedings in Theoretical Computer Science* 206 (Mar. 2016). arXiv:1604.00348 [cs], pp. 32–45. ISSN: 2075-2180. DOI: 10.4204/EPTCS.206.4. URL: http://arxiv.org/abs/1604.00348 (visited on 07/15/2025).

[LE10] Roberto E. Lopez-Herrejon and Alexander Egyed. "Detecting Inconsistencies in Multi-View Models with Variability". In: *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 217–232. ISBN: 978-3-642-13594-1 978-3-642-13595-8. DOI: 10.1007/978-3-642-13595-8_18. (Visited on 07/16/2025).

[LE12] Roberto E. Lopez-Herrejon and Alexander Egyed. "Towards Fixing Inconsistencies in Models with Variability". In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. Leipzig Germany: ACM, Jan. 2012, pp. 93–100. DOI: 10.1145/2110147.2110158. (Visited on 07/16/2025).

[Lit+13] Sascha Lity et al. "Delta-oriented software product line test models-the body comfort system case study". In: *Technical report, TU Braunschweig* (2013).

[LKS16] Sascha Lity, Matthias Kowal, and Ina Schaefer. "Higher-order delta modeling for software product line evolution". en. In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. Amsterdam Netherlands: ACM, Oct. 2016, pp. 39–48. ISBN: 978-1-4503-4647-4. DOI: 10.1145/3001867.3001872. URL: https://dl.acm.org/doi/10.1145/3001867.3001872 (visited on 05/27/2025).

[MSC14] Tom Mens, Alexander Serebrenik, and Anthony Cleve, eds. *Evolving Software Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-45397-7 978-3-642-45398-4. DOI: 10.1007/978-3-642-45398-4. (Visited on 07/09/2025).

[NLS18] Sophia Nahrendorf, Sascha Lity, and Ina Schaefer. "Applying higher-order delta modeling for the evolution of delta-oriented software product lines". In: *TU Braunschweig-Institute of Software Engineering and Automotive Informatics; Technical Report; Institute for Software Engineering and Automotive Informatics: Braunschweig, Germany* (2018).

[NPS25] Dirk Neumann, Tobias Pett, and Ina Schaefer. "Towards Consistency Preservation for Multi-Domain Variability Modeling". In: *Proceedings of the 29th ACM International Systems and Software Product Line Conference - Volume A*. A Coruña Spain: ACM, Sept. 2025, pp. 88–93. ISBN: 979-8-4007-2024-6. DOI: 10.1145/3744915.3748466. (Visited on 11/24/2025).

[OMG14a] Object Managment Group - OMG. *Meta Object Facility (MOF) Core (ISO/IEC 19508:2014(E))*. Standard. International Organization for Standardization, 2014.

[OMG14b] Object Managment Group - OMG. *Object Constrained Language (OCL)*. Standard. 2014. URL: https://www.omg.org/spec/OCL/2.4.

[Sch+12] Ina Schaefer et al. "Software Diversity: State of the Art and Perspectives". In: *International Journal on Software Tools for Technology Transfer* 14.5 (Oct. 2012), pp. 477–495. ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-012-0253-y. (Visited on 06/18/2025).

[Sch+13] Reimar Schröter et al. "Automated analysis of dependent feature models". In: *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '13. Pisa, Italy: Association for Computing Machinery, 2013. ISBN: 9781450315418. DOI: 10.1145/2430502.2430515. URL: https://doi.org/10.1145/2430502.2430515.

[Sch10] Ina Schaefer. "Variability Modelling for Model-Driven Development of Software Product Lines". In: *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*. Ed. by David Benavides, Don S. Batory, and Paul Grünbacher. Vol. 37. ICB-Research Report. Universität Duisburg-Essen, 2010, pp. 85–92. URL: http://www.vamos-workshop.net/proceedings/VaMoS%5C_2010%5C_Proceedings.pdf.

[SD10] Ina Schaefer and Ferruccio Damiani. "Pure delta-oriented programming". In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. FOSD '10. Eindhoven, The Netherlands: Association for Computing Machinery, 2010, pp. 49–56. ISBN: 9781450302081. DOI: 10.1145/1868688.1868696. URL: https://doi.org/10.1145/1868688.1868696.

[SDD15] Alcemir Rodrigues Santos, Raphael Pereira De Oliveira, and Eduardo Santana De Almeida. "Strategies for Consistency Checking on Software Product Lines: A Mapping Study". In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. Nanjing China: ACM, Apr. 2015, pp. 1–14. DOI: 10.1145/2745802.2745806. (Visited on 07/10/2025).

[SHA12] Christoph Seidl, Florian Heidenreich, and Uwe Assmann. "Co-evolution of models and feature mapping in software product lines". In: *ACM International Conference Proceeding Series* 1 (Sept. 2012). DOI: 10.1145/2362536.2362550.

[SSA14] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "DeltaEcore-A Model-Based Delta Language Generation Framework". In: *Modellierung 2014*. Gesellschaft für Informatik e.V., 2014, pp. 81–96. ISBN: 978-3-88579-619-0. (Visited on 06/27/2024).

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[Str65] Christopher Strachey. "An impossible program". In: *The Computer Journal* 7.4 (1965), pp. 313–313.

[SV02] K. Schmid and M. Verlage. "The economic impact of product line adoption and evolution". In: *IEEE Software* 19.4 (2002), pp. 50–57. DOI: 10.1109/MS.2002.1020287.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2006. ISBN: 0470025700.

[TVS21] Weslley Torres, Mark G. J. Van Den Brand, and Alexander Serebrenik. "A Systematic Literature Review of Cross-Domain Model Consistency Checking by Model Management Tools". In: *Software and Systems Modeling* 20.3 (June 2021), pp. 897–916. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-020-00834-1. (Visited on 07/10/2025).

A. Appendix

The Appendix includes both complete translation templates, for cross-domain and intra-domain consistency preservation. Each template already represents a valid reactions file and can be loaded into Vitruv. They are supposed to only provide the minimum required constructs and should be automatically expanded by the translator via the `ReactionFileDecorator`, or manually by a reactions developer.

```
import org.eclipse.emf.ecore.util.EcoreUtil
import edu.kit.kastel.tva.bcs.delta.consistency.util.DeltaCreator
import edu.kit.kastel.tva.bcs.delta.consistency.util.
    DeltaOperationFinder
import edu.kit.kastel.tva.deltavar.core.genericdeltametamodel.
    interfaces.Delta
import edu.kit.kastel.tva.deltavar.core.genericdeltametamodel.
    interfaces.DeltaRepository

import "https://tva.kastel.kit.edu/deltavar/core/
    genericdeltametamodel" as genericdeltametamodel
import "https://tva.kastel.kit.edu/deltavar/core/
    genericdeltametamodel/applicability" as applicability


reactions: crosstemplate
in reaction to changes in genericdeltametamodel
execute actions in genericdeltametamodel

routine addDeltaOperationToRepository(genericdeltametamodel::
    DeltaOp operation, genericdeltametamodel::Delta delta) {
     match {
         val targetDelta = retrieve optional genericdeltametamodel::
    Delta corresponding to delta
         val correspondingRepository = retrieve
    genericdeltametamodel::DeltaRepository corresponding to
    EcoreUtil.getRootContainer(delta)
     }
     update {
         if (targetDelta.isPresent()) {
```

```
22              targetDelta.get().getDeltaOperations().add(operation)
23          }
24          else {
25              var newDelta = DeltaCreator.getInstance().
    createCorrespondingDeltaForDelta(delta)
26              newDelta.id = EcoreUtil.generateUUID()
27              newDelta.name = delta.name
28              newDelta.getDeltaOperations().add(operation)
29              correspondingRepository.getDeltas().add(newDelta)
30              addCorrespondenceBetween(delta, newDelta)
31          }
32      }
33  }
34
35  reaction DeltaRepositoryInsertedAsRoot {
36      after element genericdeltametamodel::DeltaRepository inserted
    as root
37      call createAndRegisterRootDeltaRepository(newValue)
38  }
39
40  routine createAndRegisterRootDeltaRepository(genericdeltametamodel
    ::DeltaRepository repository) {
41      match {
42          require absence of genericdeltametamodel::DeltaRepository
    corresponding to repository
43      }
44      create {
45          val newRepository = new genericdeltametamodel::
    DeltaRepository
46      }
47      update {
48          newRepository.id = EcoreUtil.generateUUID()
49          newRepository.name = repository.name
50          persistProjectRelative(repository, newRepository, "
    generated.genericdeltametamodel")
51          addCorrespondenceBetween(repository, newRepository)
52      }
53  }
54  }
```

Listing A.1: Cross-Domain Translation Template:
Lines 1 - 5 are the imports for all external helper classes.
Lines 7 and 8 are the metamodel imports for the generic delta metamodel.
Lines 15 - 33 are the routine used for delta mirroring.
Lines 35 - 53 are the reaction and routine pair for handling the addition of delta repositories.

```
1  import org.eclipse.emf.ecore.util.EcoreUtil
2  import edu.kit.kastel.tva.bcs.delta.consistency.util.DeltaCreator
3  import edu.kit.kastel.tva.bcs.delta.consistency.util.
       DeltaOperationFinder
4  import edu.kit.kastel.tva.deltavar.core.genericdeltametamodel.
       interfaces.Delta
5  import edu.kit.kastel.tva.deltavar.core.genericdeltametamodel.
       interfaces.DeltaRepository
6
7  import "https://tva.kastel.kit.edu/deltavar/core/
       genericdeltametamodel" as genericdeltametamodel
8  import "https://tva.kastel.kit.edu/deltavar/core/
       genericdeltametamodel/applicability" as applicability
9
10
11 reactions: intratemplate
12 in reaction to changes in genericdeltametamodel
13 execute actions in genericdeltametamodel
14 }
```

Listing A.2: Intra-Domain Translation Template:
Lines 1 - 5 are the imports for all external helper classes.
Lines 7 and 8 are the metamodel imports for the generic delta metamodel.