Karlsruhe Institute of Technology

Test, Validation, Analysis

Incremental Consistency Analysis in Cyber-Physical Product Lines

Bachelor's Thesis
of

Kevin Boehnke

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. Philip Ochs
	M.Sc. Tobias Pett

Completion period: 07. January 2025 – 07. May 2025

Zusammenfassung

Eine Produktlinie bezieht sich auf eine Sammlung verwandter Produkte, die eine Reihe von Gemeinsamkeiten aufweisen, aber auch Variabilität zulassen, um spezifischen Kundenanforderungen gerecht zu werden. Im Zusammenhang mit cyber-physischen Systemen umfasst eine Produktlinie sowohl einen Problemraum, der die für den Benutzer sichtbaren Merkmale darstellt, als auch einen Lösungsraum, der die technischen Implementierungen darstellt. Die Aufrechterhaltung der Konsistenz zwischen diesen beiden Räumen ist von entscheidender Bedeutung, da sie sicherstellt, dass jede gültige Konfiguration im Problemraum auch erfolgreich im Lösungsraum realisiert werden kann. Da sich cyber-physische Produktlinien weiterentwickeln, wird die wiederholte Analyse aller Konfigurationen, wie sie bei einem produktbasierten Ansatz durchgeführt wird, aufgrund des kombinatorischen Wachstums des Konfigurationsraums unpraktisch. Um dieses Problem der Skalierbarkeit zu lösen, wird in dieser Arbeit ein inkrementeller Ansatz zur Konsistenzanalyse konzipiert und evaluiert. Dieser Ansatz nutzt die Deltamodellierung, um systematisch evolutionäre Änderungen darzustellen, so dass sich die Reanalyse ausschließlich auf die geänderten Teile der Produktlinie konzentrieren kann. Eine Optimierung, der sogenannte *Dominates Check*, reduziert den Rechenaufwand weiter, indem er eine Reihe von dominierenden Konfigurationen für nachfolgende Analysen identifiziert. Die Evaluierung des inkrementellen Ansatzes anhand einer Fallstudie, die die Evolution einer Produktlinie abbildet, zeigt deutliche Leistungsverbesserungen mit erheblichen Laufzeitverkürzungen und Geschwindigkeitssteigerungen im Vergleich zu einem produktbasierten Ansatz. Diese Ergebnisse verdeutlichen, dass der inkrementelle Ansatz nicht nur das Laufzeitverhalten verbessert, sondern vor allem die notwendige Skalierbarkeit bietet, um die Konsistenzanalyse während der Weiterentwicklung cyber-physischer Produktlinien zu unterstützen.

Abstract

A product line refers to a collection of related products that share a set of commonalities but also allow for variability to accommodate specific customer requirements. In the context of cyber-physical systems, a product line comprises both a problem space, representing user-visible features, and a solution space, representing technical implementations. Maintaining consistency between these two spaces is crucial, as it ensures that every valid configuration in the problem space can be successfully realized in the solution space. As cyber-physical product lines evolve, repeatedly analyzing all configurations, as done by a product-based approach, becomes impractical due to combinatorial growth of the configuration space. To address this scalability issue, this thesis conceptualizes and evaluates an incremental consistency analysis approach. This approach employs delta modeling to systematically represent evolutionary changes, allowing reanalysis to focus solely on modified parts of the product line. An optimization called the *dominates check* further reduces computational effort by identifying a set of dominating configurations for subsequent analyses. Evaluating the incremental approach using a case study representing the evolution of a product line demonstrates substantial performance improvements, achieving notable runtime reductions and speedups compared to a product-based approach. These results underscore that the incremental approach not only enhances runtime performance but, more importantly, provides the scalability required to support consistency analysis across the evolution of cyber-physical product lines.

Contents

List of Figures

List of Tables

Acronyms

BCS	Body Comfort System
CSP	Constraint Satisfaction Problem
CSV	Comma-Separated Values
EMF	Eclipse Modeling Framework
FM	Feature Model
PL	Product Line
PLE	Product Line Engineering
RAM	Random Access Memory
SAT	Boolean Satisfiability Problem
SG	Subgoal
SMT	Satisfiability Modulo Theorie
UCM	Unified Conceptual Model
XML	Extensible Markup Language

1. Introduction

In an increasingly digitalized and competitive world, manufacturers face the challenge of modeling, building and selling cyber-physical systems, where hardware and software interact in a complex way. As customer expectations shift towards more personalized and variable experiences, the need for product customization has become essential. Product Line Engineering (PLE) offers an approach by enabling manufacturers to systematically manage and reuse core artifacts for developing variable products [PBL05].

A Product Line (PL) consists of a problem space and a solution space [Ape+13]. The problem space represents the functionalities that users can directly experience, so called features, and their dependencies on an abstract, conceptual level. To model the features along with their dependencies, a FM can be used. Figure 1.1 shows a running example from the automotive industry - a car's infotainment system. The example PL consists of the two features `navigation-system` and `multimedia-system`. Both of the features are optional, which means that a customer can either select none, one or both features. A combination of features constitutes a configuration and is considered valid if the combination satisfies all the dependencies of the FM.

Besides the problem space, the solution space of a cyber-physical PL addresses the technical realization of product configurations. It incorporates realization artifacts and associated conditions, such as software and hardware components, as well as their dependencies. In the solution space of the running example there are two software components and one hardware component (see Figure 1.1). The software component `navigation-system-software` implements the feature `navigation-system` and the software component `multimedia-system-software` implements the feature `multimedia-system`. The hardware component in this example is the processing unit `pu-1`.

While a configuration may be valid in the problem space, it must not be necessarily functioning in the solution space [Hen+22] because of possibly incompatible realization artifacts. In the running example, the hardware component `pu-1` may provide resources to satisfy either the demands of the software component `multimedia-system-software` or the demands of the software component `navigation-system-`

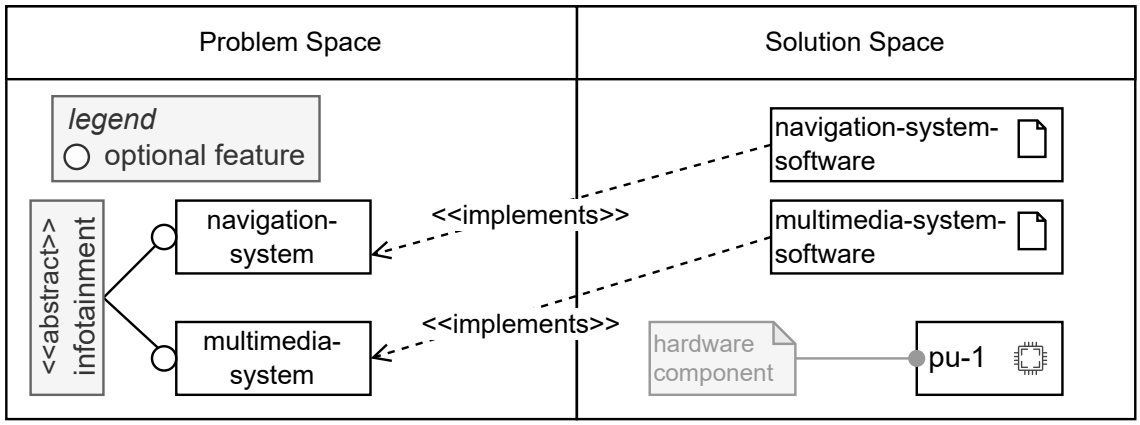

Figure 1.1: Problem and solution space for the PL of the running example (adopted from Ochs et al. [OPS24]).

`software` but not both together. Thus the configuration selecting both features is valid but the realized product variant is non-functioning. This causes an inconsistency between modeled (problem space) and realizable (solution space) variability. Ensuring consistency between the problem space and the solution space in PLs is a highly complex task for manufacturers, as the number of configurations within a PL grows combinatorially. This makes manual execution impractical, requiring computer-based formalization and analysis instead.

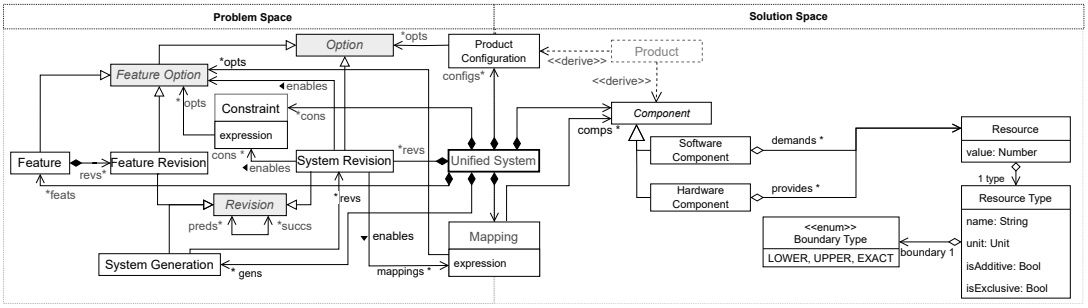

Figure 1.2: The Unified Conceptual Model [OPS24].

An approach to formally describe cyber-physical PLs is the unified conceptual model (UCM). It enables an integrated view of both the problem and solution spaces, creating a structured representation of the connection between artifacts of both spaces [WKR22]. Based on this model (see Figure 1.2), Ochs et al. [OPS24] presented an approach to decide the consistency between problem and solution space of a cyber-physical PL, based on realizability analysis. However, this product-based approach does not scale well for large cyber-physical PLs as it requires analyzing all possible variants individually for their realizability.

In addition, cyber-physical PLs continuously evolve over time where structural changes and altered interdependencies lead to multiple versions. Due to these changes, consistency may be compromised in any new version. Therefore, it is essential to analyze each new version with regard to its consistency. Performing a full reanalysis for each new version amplifies the scalability problems as the number of configurations increases with both variability in space (variants) and time (ver-

sions). As a result, applying a product-based approach in the context of evolving PLs becomes increasingly impractical due to the added complexity introduced by changes over time, indicating the need for a more scalable and version-aware analysis strategy.

Goals of this Thesis

This bachelor thesis aims to conceptualize and assess an incremental analysis approach to improve the scalability of consistency analysis between problem and solution space in evolving cyber-physical PLs. An incremental approach recomputes only parts of a PL affected by changes, instead of reanalyzing all possible configurations, as done in the existing, product-based approach [OPS24]. This reduces redundant computations, particularly for configurations unaffected by any change. To achieve this goal, we define three Subgoals (SGs) as follows.

SG1 Modeling Change Operations in Cyber-Physical PLs

We first identify and model relevant change operations in a cyber-physical PL, e.g. adding a feature (problem space) or removing a software component (solution space). To represent these changes systematically, we build on the concept of Delta Modeling, described in Section 2.4, which supports a modular and structured expression of changes. Based on this, we specify a delta dialect that serves as the basis for our incremental analysis.

SG2 Concept for Incremental Consistency Analysis in Cyber-Physical PLs

Informed by related work, we propose an incremental analysis approach for defined change operations in problem and solution space, and integrate it into the existing consistency analysis approach [OPS24]. An already analyzed consistent base version forms the foundation for our subsequent incremental analyses. We sequentially analyze successive versions of a PL as long as they remain consistent. The analysis of each successive version is divided into two stages: First, we perform certain preprocessing steps, based on the examination of the effect of each delta operation defined in SG1 on consistency analysis. In the second stage, the main consistency analysis is performed, focusing solely on the parts affected by the applied changes. In the context of this work, each delta operation is analyzed independently. Potential combinations or dependencies between multiple operations are not considered.

SG3 Performance Comparison of Incremental Consistency Analysis in Cyber-Physical PLs

To evaluate our concept, we measure the performance regarding run-time for incrementally analyzing consistency between problem and solution space along an adaptation of the existing BCS case study [Lit+13], a cyber-physical PL from the automotive domain, and its evolution [Rak24]. Starting from a version with an initially consistent problem and solution space, we iteratively apply sets of change operations describing each consecutive version, and recompute the PL's consistency using our incremental analysis concepts outlined in SG2. Applying the existing product-based consistency analysis approach to it as well enables us to compare the performances of both the existing product-based and our incremental approach.

Structure of the Thesis

The subsequent chapters of this thesis are arranged as follows: Chapter 2 introduces the basic concepts and fundamentals relevant to this thesis. Chapter 3 gives a detailed description of the design decisions of the solution approach. In Chapter 4 the implementation of our design is presented. Chapter 5 comprises the evaluation and discussion of our work. An overview of related work is presented in Chapter 6. The thesis concludes with Chapter 7, which summarizes the results and presents an outlook on future work.

2. Basics

In this chapter, we introduce foundational concepts relevant to the modeling and analysis of variability in cyber-physical PLs. In Section 2.1 we describe the concepts of PLE with regard to problem and solution space, as well as variability in space and time. In Section 2.2, we present the UCM, which meta-models a PL's evolution with its problem and solution space. In Section 2.3 we introduce the concept of consistency between problem and solution space of a PL, which we refer to simply as *consistency* throughout this work. In Section 2.4 we introduce Delta Modeling as an approach to manage variability in time and to enable incremental analysis. In Section 2.5 we describe the concept of syntactic consistency for modeling artifacts. Finally, we present our running example in Section 2.6.

2.1 Product Line Engineering

The concept of PLE allows manufacturers to use the advantages of mass production while still addressing specific customer needs. Instead of developing a single product, various related products are developed, sharing a common set of core artifacts. The products can differ in variable artifacts that complement the common set. Together, these related products constitute a PL [PBL05; CN01].

A PL can be distinguished into a problem space and a solution space [Ape+13]. The problem space captures user-visible functionalities, so-called features, and describes their interdependencies at a conceptual level. Features serve as a central abstraction for expressing both commonalities and variabilities between products. The solution space, on the other hand, represents the technical implementation of these features. It contains the actual implementation artifacts, such as software components, which realize the user-visible functionality described in the problem space, and their dependencies. Using these artifacts, concrete products can be derived.

To model and manage the variability created by features and their dependencies, FMs are often used. First of all, an FM defines parent-child relations [Ape+13], which can be expressed as as hierarchical trees. A child feature can only be selected when the parent feature is also selected, indicating an implication from the child feature to the parent feature. These parent-child relations can have four different

types: while an optional child feature may be selected, a mandatory child feature is always selected when the parent feature is selected, building a biconditional relation between child and parent feature. The relation between a parent feature and a group of child features is defined by or-groups, in which at least one child feature has to be selected when the parent feature is selected, and alternative-groups, in which exactly one child feature has to be selected. These relations can also be expressed as propositional logical formulas using Boolean variables for the features. In addition to parent-child relations, expressions composed of features combined using logical connectives can express further relations between features (e.g. requires or excludes), known as cross-tree constraints. Together, these relations define the conditions under which the features can be composed.

A concrete selection and deselection of features is referred to as a configuration, which represents a potential product in the solution space. Different products in the solution space are called variants. A configuration is considered valid if all constraints associated with the features are satisfied. For example, selecting two features from an alternative-group violates its exclusivity constraint and results in an invalid configuration.

Besides this variability in space, variability in time refers to the evolution of a PL over time [PBL05]. A PL may change, introducing new features or additional constraints in the problem space, or removing them, causing a change of the configuration space. The solution space changes as well, as new realization artifacts may arise leading to new variants, or existing artifacts can change through updates, resulting in different versions of variants. For instance, a software component can be modified to meet changing requirements (e.g., demanding more Random Access Memory (RAM)), reflecting the updated implementation. Also, dependencies in the solution space can change through time.

2.2 Unified Conceptual Model

The UCM is a metamodel that unifies problem and solution space of a PL while integrating concepts for variability in space and time. It was introduced by Annanieva et al. [Ana+22] and extended by Wittler et al. [WKR22], Ochs et al. [OPS24], and Rak [Rak24]. The UCM by Ochs et al. [OPS24] is shown in Figure 1.2.

While the problem space of a UCM can be represented by an FM, containing features and their dependencies, modeled as constraints, the solution space consists of components, which in the case of a software component demand resources, and in the case of a hardware component provide resources.

Resources consist of a `value` and are described using resource types. A resource type consists, besides a `name` and a `unit`, of the Boolean attributes `isAdditive`, `isExclusive`, and a boundary type, which can be `LOWER`, `UPPER` or `EXACT`. If `isAdditive` is `True` the `value` of different resources of the same resource type can be summed. `IsExclusive` indicates whether a resource can be assigned to multiple components simultaneously or is restricted to at most one. The `value` of a provided resource of a resource type with boundary type `LOWER` must not be less than the `value` of a demanded resource of this resource type to satisfy it. `EXACT` specifies that the demand must be met by a provision with the same `value`, while `UPPER` restricts the provision to not exceed the amount demanded to satisfy the demand.

Artifacts of both problem and solution space are linked through mappings, which connect the conceptual functionalities of the FM and the concrete realizations from solution space artifacts. For instance, it can be specified which software component implements which feature. Furthermore, a feature can be associated to a resource, enabling feature-defined resource demands, a concept that will be further explained in the following section.

Rak [Rak24] extended the UCM in the problem space, concretizing the model representation of a constraint into tree constraints, which can be of type `MANDATORY`, `OPTIONAL`, `OR` or `ALTERNATIVE`, and cross-tree constraints, which are expressions composed of features as literals combined using logical connectives.

2.3 Consistency of a PL

The problem and solution space of a PL are consistent if the modeled variability in the problem space, i.e., all valid configurations, matches the realizable variability in the solution space. As a feature may introduce demands (feature-defined resource demands), the resources demanded by a software component can vary, depending on which features are selected. Including or excluding a specific feature determines whether the introduced resource demand is considered.

To decide whether a configuration is realizable, the specific resource demands for the software components are derived, as described by Ochs et al. [OPS24]. If the demanded resources of the software components can be satisfied by given hardware components in compliance with all solution space constraints, the variant is considered realizable. A configuration valid in the problem space but not realizable leads to a non-functioning product, and thus to an inconsistency between problem and solution space. Ensuring consistency plays a crucial role, especially in evolving PLs, as a previously functioning variant may be non-functioning after an update, due to increased resource demands, that can no longer be satisfied by the provisions.

To determine the realizability of a configuration, Ochs et al. [OPS24] proposed an approach based on Constraint Satisfaction Problems (CSPs), which are defined as a set of variables with associated domains and constraints that specify allowable combinations of values [RND10], that operates directly on an instance of the UCM. The derived resource demands of the software components are assigned to resource provisions of the hardware components, allowing for a comparison of demands and provisions. The additional assignment of software components to hardware components guarantees that each software component is deployed to only one hardware component. If the CSP has at least one solution, the configuration is considered realizable.

2.4 Delta Modeling

Delta modeling (Δ-modeling) is a variability modeling approach for systematically managing changes in PLs. Originally intended to support variability in space [Sch10], delta modeling can also be used for variability in time, describing the evolution of a PL. It captures differences through delta models, each describing a set of changes, such as additions, removals, or modifications, relative to a core model. In a PL these changes can include, for example, adding a feature or modifying a resource.

A defined set of such change operations constitutes a delta dialect. Since delta modeling is not tied to any specific modeling or implementation language, it can be applied across different levels of abstraction, enabling its application to the UCM.

2.5 Syntactic Consistency

Another form of consistency relevant to this thesis is syntactic consistency, which refers to the structural correctness of software models with respect to the rules of the modeling language in which they are expressed [SZ00]. This includes general well-formedness criteria that ensure the technical validity of a model. Syntactic consistency is a fundamental prerequisite for any further analysis or integration of models. Without it, models cannot be reliably interpreted or composed, as violations prevent tools from processing models correctly. For example, an FM forbids contradictory cross-tree constraints, as there could not be a solution satisfying the propositional logical formulas which represent the FM, indicating that the FM is syntactically inconsistent [Ape+13].

Within a delta, temporary inconsistencies can arise as a result of applying a delta operation. For instance, removing a tree constraint in a FM can leave a feature unattached, leading to a syntactic inconsistency. To restore consistency, model repair techniques are used to suggest or apply repair operations that resolve violations and restore the model to a syntactically valid state. Continuing the example, removing the now unattached feature along with the constraint would reestablish consistency. These approaches range from fully automatic to user-guided and are a key part of consistency management in model-driven engineering [MJC17].

2.6 Running Example

To illustrate our proposed concepts, we create a running example used throughout this work. We therefore extend the example from the introduction, by creating a UCM instance. The example already defines a FM in the problem space including the two optional features `navigation-system` and `multimedia-system`. In the solution space, the two software components `navigation-system-software` and `multimedia-system-software` are presented, implementing the corresponding features. Lastly, the hardware component `pu-1` is described. We further add the resource types RAM, specifying it as additive, but not exclusive, and with a lower boundary, and maximum response time, which is neither additive nor exclusive, and has a upper boundary. While the feature `navigation-system` introduces a resource demand for 3 GB of RAM and a maximum response time of 50 ms, the feature `multimedia-system` introduces a resource demand for 1 GB of RAM and a maximum response time of 100 ms. As the software component `navigation-system-software` implements the feature `navigation-system` and the software component `multimedia-system-software` implements the feature `multimedia-system`, each component either has no demands or adopts the demand of its corresponding feature, depending on whether the feature is selected in a given configuration. Since the hardware component `pu-1` provides 5 GB of RAM and a maximum response time of 50 ms, it covers the demands of all possible configurations. Hence, all valid configurations are realizable, thereby establishing the consistency of our running example.

As we propose an incremental approach, we extend this base version of the running example to a second one. Specifying a delta dialect containing the change operations add tree constraint, add feature, add hardware component, add resource, and modify tree constraint allows us to create a second version of the running example as follows:

The second version also consists of the optional feature `navigation-system`, the feature `multimedia-system` is now marked as mandatory, and beneath that feature, there is a new alternative-group containing the features `radio` and `cd-player`. The feature `radio` introduces an additional resource demand for 1 GB of RAM and a maximum response time of 70 ms, and the feature `cd-player` introduces a resource demand for 1 GB of RAM and a maximum response time of 60 ms. Both features are also implemented by the software component `multimedia-system-software`. Thus, `multimedia-system-software` either demands 2 GB of RAM and a maximum response time of 70 ms or 2 GB of RAM and a maximum response time of 60 ms depending on which feature from the alternative-group is selected. Furthermore, a newly added hardware component `pu-2` provides 2 GB of RAM and a maximum response time of 60 ms. Since the two features `radio` and `cd-player` cannot be selected simultaneously, as it would violate the mutual exclusivity constraint of the alternative-group, the hardware component `pu-2` provides enough resources to satisfy all possible resource demands of the software component `multimedia-system-software`. The hardware component `pu-1` has not changed at all and still satisfies the demands of the software component `navigation-system-software`. Therefore, the second version of our running example is also consistent. Both versions are illustrated in Figure 2.1, with the base version shown on the left, and the evolved version on the right.

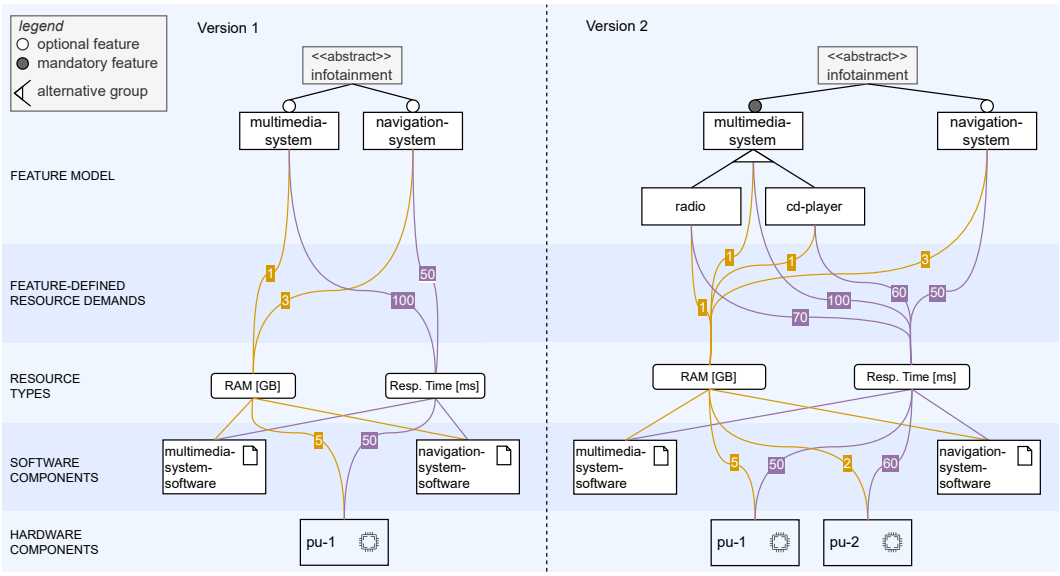

Figure 2.1: UCM instances of version 1 (adopted from Ochs et al. [OPS24]) and 2 of our running example, presenting the FMs, feature-defined resource demands, resource types, software components, and hardware components.

3. Design

In this chapter, a detailed description of the solution approach is given. Section 3.1 describes the specified delta dialect, which is necessary to formally represent possible modifications in a cyber-physical PL. In Section 3.2, we introduce an optimization concept called the dominates check, used to compute subsets of configurations for subsequent realization analysis, thus reducing computational effort. Finally, we represent our incremental analysis approach in Section 3.3.

3.1 Specifying the Delta Dialect

To model the evolution of a cyber-physical PL, we use the approach of Delta Modeling [Sch10], where the evolution of a base version is represented by a series of deltas, each consisting of a sequence of delta operations. A delta operation describes a specific modification to the PL, such as an addition, removal, or modification of structural elements. Single delta operations can introduce temporary syntactic inconsistencies, as described in Section 2.5 following Spanoudakis et al. [SZ00]. However, within the context of this thesis, we require all inconsistencies to be resolved by applying appropriate repair operations [MJC17] in the form of other delta operations, before the completion of a delta. Consequently, each delta results in a syntactically consistent cyber-physical PL once fully applied. This ensures that our incremental consistency analysis can operate reliably on valid intermediate states.

Regarding SG1, we first specify our delta dialect, based on the UCM by Ochs et al. [OPS24]. Additionally, we integrate the changes in the problem space by Rak which differentiate between tree constraints and cross-tree constraints [Rak24]. Figure 3.1 shows the UCM, highlighting configurable parts. In the problem space, features, tree constraints, and cross-tree constraints can either be added or removed from the Unified System. Additionally, the constraint Type of a tree constraint can be modified. In the solution space, resource types, software components, hardware components, and resources can be added or removed. Resource types and resources can also be modified. While resources can change their `value` attribute, resource types can change their `isAdditive` and `isExclusive` attributes, as well as the boundary type. An overview of all supported delta operations is provided in Table 3.1.

In our running example, the delta representing the changes from the base version to the second version, as described in Section 2.6, specifies the following delta operations:

```
add tree constraint (alternative-group)
add feature (radio)
add feature (cd-player)
add hardware component (pu-2)
add resource (1 GB of RAM, introduced by radio)
add resource (maximum response time of 70 ms, introduced by radio)
add resource (1 GB of RAM, introduced by cd-player)
add resource (maximum response time of 60 ms, introduced by cd-player)
add resource (2 GB of RAM, provided by pu-2)
add resource (maximum response time of 60 ms, provided by pu-2)
modify tree constraint (optional to mandatory)
```

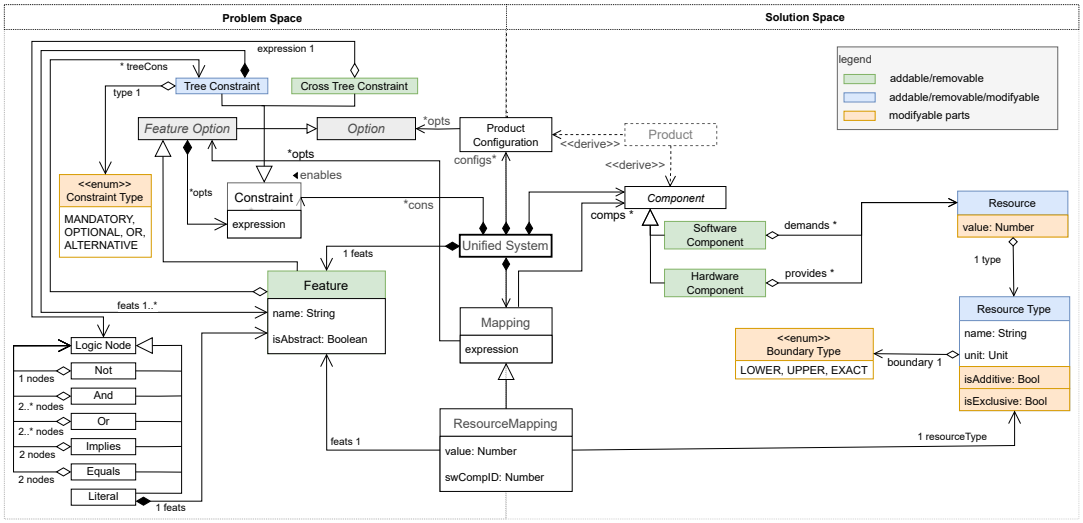

Figure 3.1: The considered parts of the extended UCM by Rak [Rak24] showing the configurable parts in our design.

Change Type	Add	Remove	Modify
Feature	X	X	
Tree Constraint (TC)	X	X	X
Cross-Tree Constraint (CTC)	X	X	
Resource Type (RT)	X	X	X
Software Component (SWC)	X	X	
Hardware Component (HWC)	X	X	
Resource	X	X	X

Table 3.1: Overview of delta operations.

3.2 Dominates Check

The dominates check identifies dominating configurations that include maximum features and a maximum of overall resource demands. Such a configuration is unambiguous, as alternative-groups and cross-tree constraints typically prevent the simultaneous selection of all features in most FMs. We use the dominates check to compute a subset of configurations for subsequent realization analysis, decreasing computational effort.

The key assumptions are that a feature can introduce resource demands and that the addition of a resource demand always results in an overall resource demand that is at least as strict as before.

Consider two valid configurations, A and B from our running example. Configuration A selects the features `multimedia-system` and `navigation-system`, while configuration B selects only `multimedia-system`. Thus, configuration A contains all features of configuration B plus the additional feature `navigation-system`.

Following our assumptions, we expect configuration A to have overall resource demands that are at least as strict as those of configuration B. In Section 2.6 we already described the feature-defined resource demands of our running example, resulting in the software component `multimedia-system-software` demanding 1 GB of RAM and a maximum response time of 100 ms for both configurations. For configuration A the software component `navigation-system-software` demands 3 GB of RAM and a maximum response time of 50 ms, for configuration B the software component does not have any demands, as the feature `navigation-system` is not selected.

To satisfy the resource demands of configuration A a provision of at least 4 GB of RAM and a maximum response time of 50 ms are required, while for configuration B 1 GB of RAM and a maximum response time of 100 ms would be enough. Thus, the overall demands of configuration A are stricter than those of configuration B. Even if the additional feature `navigation-system` did not introduce any resource demands, both configuration would have the exact same overall resource demands.

If configuration A is realizable, then configuration B is also realizable, since the provision of 4 GB of RAM and a maximum response time of 50 ms would satisfy the demands of 1 GB of RAM and a maximum response time of 100 ms by far. Therefore, in terms of consistency analysis, it is sufficient to analyze configuration A, and configuration B can be omitted.

The dominates check is formally defined as follows: **A configuration *A* dominates a configuration *B* if *A* contains at least all features that *B* contains.** This criterion enables the comparison of a set of configurations, retaining only the dominating configurations, i.e. configurations that are not dominated by any other configuration.

The dominates check can be applied to the base version, as well as to deltas. In the base version for instance, the dominates check avoids analyzing the realizability of every valid configuration. The base version of our running example has four valid configurations, as shown in Figure 3.2. The first configuration selects none of the features, the second configuration selects only the feature `navigation-system`, the third configuration selects the feature `multimedia-system`, and the fourth configuration selects both features.

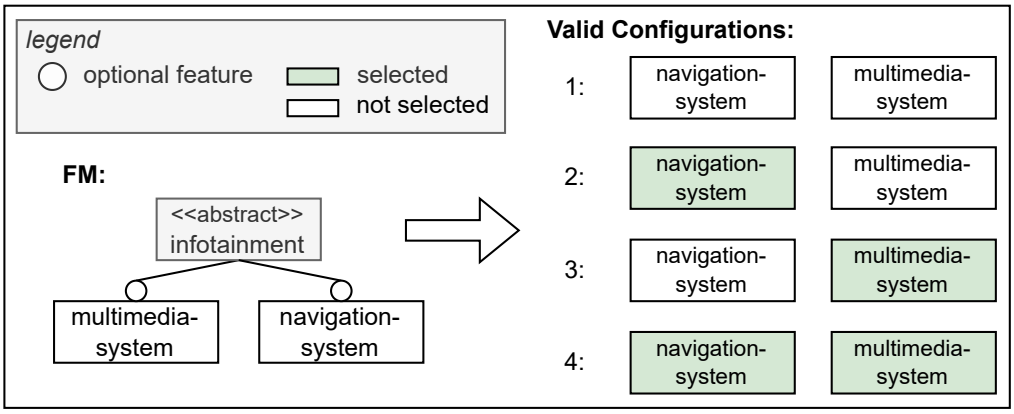

Figure 3.2: The four valid configurations of the base version of our running example.

To compute the subset sufficient for analyzing the realizability of each configuration, and thus determining the consistency of the base version, we process the valid configurations sequentially. In the example presented, the order follows the sequence presented in Figure 3.2; however, in practice, the configurations are considered in an arbitrary order.

Initially, the first valid configuration is considered dominant, as no other configurations have been encountered yet. The second configuration is then compared to the first: If it is dominated by the first, we continue with the next configuration. If it is not dominated, we check whether it dominates the first configuration. If it does, we discard the first configuration and retain the second; otherwise, we keep both. The second configuration from our running example dominates the first configuration as the first does not contain any feature. Hence, the first configuration is no longer considered. The second and third configurations do not dominate each other, as each configuration contains a feature that the other does not. Last but not least, the fourth configuration is compared to the second and third configurations. It is not dominated by either of them, but it dominates both configurations two and three, as it contains all features of the FM. As a result, only the fourth configuration remains dominant.

In the worst case, the dominates check compares each configuration with all previously retained configurations, resulting in a time complexity of $\mathcal{O}(n^2)$ for n configurations. However, in practice, many comparisons can be avoided, as a configuration is discarded as soon as it is found to be dominated by another. Since it reduces the number of valid configurations that need to be reanalyzed for their realizability, and realizability analysis being NP-complete [Coo71], as constraints related to resources and their allocation are solved, its integration into the analysis workflow is well justified.

3.3 Incremental Analysis Approach

After specifying the delta dialect for the UCM and introducing the dominates check, we now propose an incremental approach to analyze the consistency in evolving PLs. An already analyzed consistent base version builds the basis for our approach

to analyze subsequent versions. To improve performance, we divide the consistency analysis into two stages: (1) preprocessing of a delta (see Section 3.3.1) to determine if and how a reanalysis is required and (2) optimized conditional reanalysis of the consistency of the PL (see Section 3.3.2). A visualization of the precise workflow is shown in Figure A.5 and is elaborated in the following subsections.

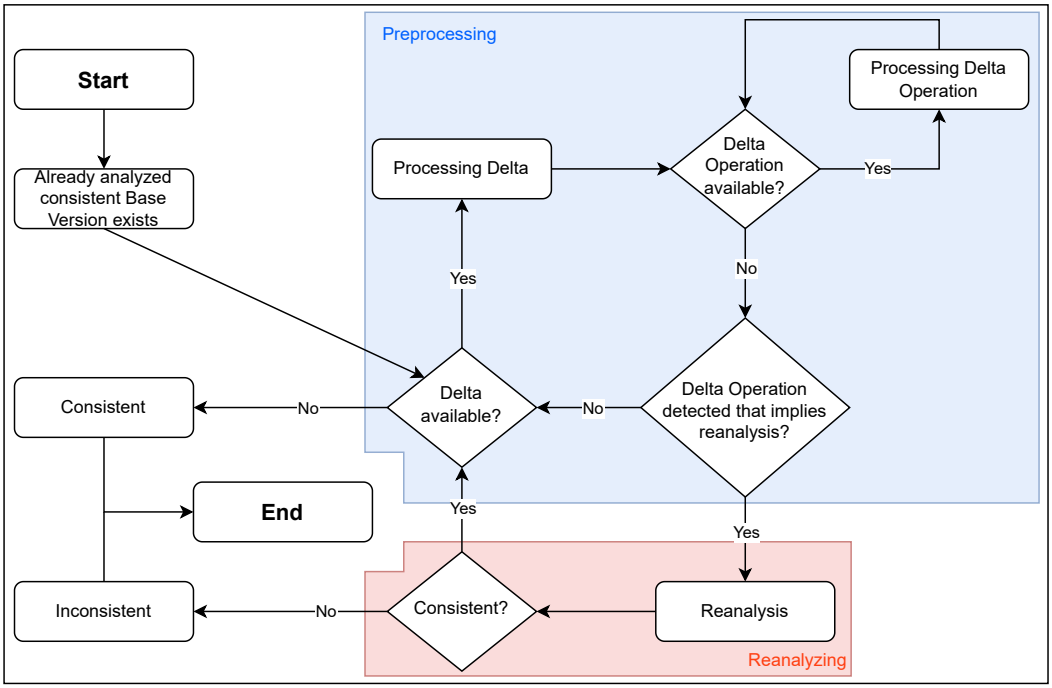

Figure 3.3: Overview of the workflow.

3.3.1 Preprocessing Deltas

Before starting the main consistency analysis, we perform certain preprocessing steps. For each delta, we preprocess all included delta operations sequentially and investigate how a delta operation affects the subsequent analysis of consistency between problem and solution space of a cyber-physical PL. Figure 3.4 depicts a decision tree that visualizes all possible cases.

A new feature added by the delta operation `add feature` does not introduce demands by default and thus has no impact on the realizability of configurations and subsequently the consistency between problem and solution space. Similarly, the delta operation `remove feature` removes a demand-free feature, based on the assumption of syntactic consistency and required repair operations [SZ00; MJC17] and therefore has no impact on consistency. The delta operation `add CTC` merely restricts valid configurations further by introducing problem space constraints, and hence does not lead to inconsistency. A tree constraint defines structural relations between features. Since the addition and removal of a feature do not have an impact on consistency, adding or removing the corresponding tree constraints does not impact consistency either. Consequently, both the delta operations `add TC` and `remove TC` do not impact consistency.

In the solution space, a resource type defines characteristics, such as whether the resource type is additive, which means that in a configuration the values of demanded

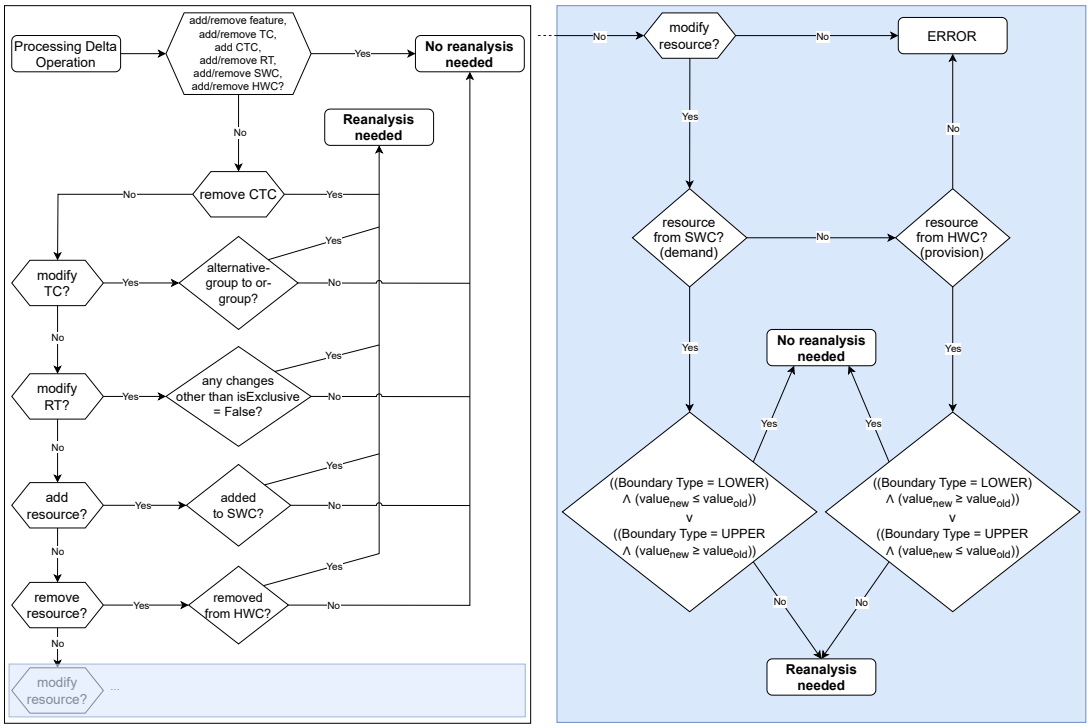

Figure 3.4: The decision tree illustrating whether a delta operation leads to reanalysis or not.

resources of this type should be added. Adding a new resource type does not impact the realization of existing configurations. Therefore, starting from a consistent state, applying the delta operation `add RT` itself does not introduce an inconsistency. When removing a resource type, all related demands and provisions must be removed first [SZ00; MJC17]. Thus, the delta operation `remove RT` does not impact consistency. Software and hardware components are initially added without demands, similar to features, and therefore have no influence on consistency. Their removal likewise requires first removing all connected resources[SZ00; MJC17].

The delta operation `modify TC` initiates a reanalysis iff an alternative-group is changed to an or-group, as previously only one feature from the group could be selected, whereas now several can be selected, allowing new combinations of features that have not been analyzed yet and can therefore possibly lead to inconsistency. Consequently, only these newly enabled configurations, i.e., those that contain at least two features of the group, need to be analyzed.

The delta operation `modify RT` does not necessitate a reanalysis if only the `isExclusive` attribute is changed and set to False. Otherwise, all configurations inducing resource demands of the changed resource type need to be reanalyzed.

Since software components only demand resources, adding a resource to a software component results in additional demands. Consequently, the overall resource demand becomes at least as strict as before. In contrast, removing a resource from a software component reduces its demands, resulting in an overall resource demand that is either less strict or equally strict, but never stricter than before. Similarly, as hardware components only provide resources, adding a resource to a hardware component increases its provisions, while removing a resource decreases them. Hence,

the delta operation `add resource` induces a reanalysis, if a resource is added to a software component, reanalyzing all configurations demanding the newly added resource. Conversely, the delta operation `remove resource` induces a reanalysis, if a resource is removed from a hardware component, reanalyzing all configurations demanding a resource of the removed resource's resource type.

If the value of a resource changes due to the delta operation `modify resource`, the necessity of a reanalysis depends both on whether the resource is associated with a software component (demand) or a hardware component (provision) and on the boundary type. Specifically, if the boundary type is `LOWER` and the modified demand decreases, or if the boundary type is `UPPER` and the modified demand increases, a reanalysis can be avoided. Formally, this condition can be expressed as

$$((b = \text{LOWER}) \wedge (v' \leq v)) \vee ((b = \text{UPPER}) \wedge (v' \geq v))$$

where v is the old value, v' is the new value, and b is the boundary type. The same reasoning applies to provisions, but with the direction of the inequalities reversed:

$$((b = \text{LOWER}) \wedge (v' \geq v)) \vee ((b = \text{UPPER}) \wedge (v' \leq v))$$

The delta operation `remove CTC` always requires reanalysis, as it can enable previously invalid configurations that have not been analyzed for realizability yet. In this case, only configurations involving the newly allowed feature combinations need to be reanalyzed.

In addition to investigating the effects of delta operations on subsequent analyses, we also want to update information about valid configurations in the preprocessing stage, modifying the configurations, adding new configurations, and removing obsolete configurations. Since storing and updating all valid configurations of a previous version would not scale in large PLs, we only store and update the previously computed dominating configurations. However, due to the use of the dominates check, a substantial amount of information about the configuration space is lost, which limits the ability to track changes and to continuously update the dominating configurations. While delta operations such as `add feature` or `remove feature` do not pose a problem, the delta operation `add CTC` prevents updating if the newly introduced cross-tree constraint is not satisfied by all currently dominating configurations. In this case, there is insufficient information to determine the new set of dominating configurations. Similarly, the delta operations `modify TC` and `remove CTC` also prevent updating the set of dominating configurations, as shown in the decision tree in Figure 3.5. Once we are unable to update the dominating configurations, we discard them.

In our running example, updating the dominating configuration from the base version to the second version is unproblematic. As discussed in Section 3.2, the dominating configuration of the base version selects both features `multimedia-system` and `navigation-system`. In the second version only two features of an alternative-group are added. Consequently, the previously dominating configuration is extended separately with each of the new features, resulting in two updated dominating configurations: (1) selecting the features `navigation-system`, `multimedia-system`, and `radio`, and (2) selecting the features `navigation-system`, `multimedia-system`, and `cd-player`.

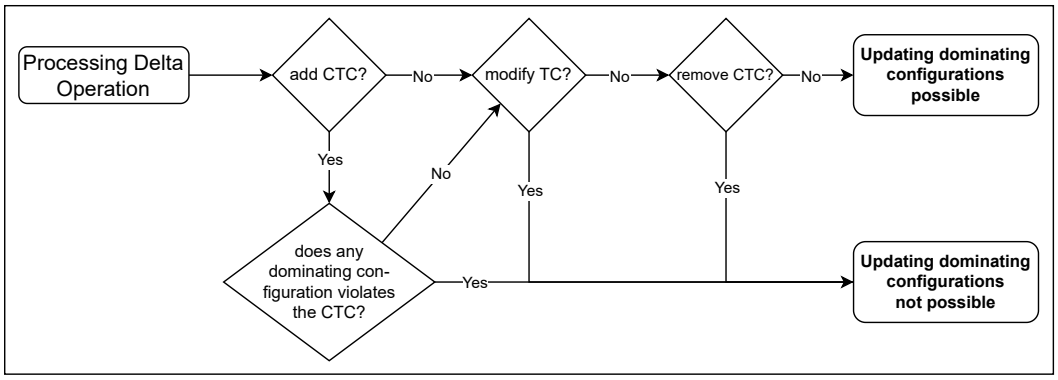

Figure 3.5: A decision tree determining whether updating the dominating configurations is possible or not.

3.3.2 Reanalysing consistency between problem and solution space

Once preprocessing is complete and at least one delta operation requiring reanalysis is identified, we reanalyze the consistency between problem and solution space of the updated PL. Whether or not we were able to update the dominating configurations, depending on the specific delta operations applied within a delta, determines which of two reanalysis approaches is used.

(1) If we were able to update the dominating configurations, we only need to reanalyze their realizability to assess the consistency of the updated PL.

(2) If we were unable to update the dominating configurations, we need to reanalyze the updated PL for its valid configurations. However, preprocessing usually restricts the set of valid configurations which we must reanalyze regarding realizability. Hence, we only reanalyze affected parts of the PL, decreasing computational effort.

In cases where a delta operation affects a globally relevant resource or component, or when the accumulated effect of multiple delta operations within a delta impact a substantial part of the PL, it may still be necessary to reanalyze (almost) the entire PL. In such a worst case, our reanalysis would be equivalent to that of the product-based approach proposed by Ochs et al. [OPS24], which evaluates the realizability of all valid configurations. Although our preprocessing introduces additional overhead, which would result in even worse performance than the product-based approach, we address this issue by applying the dominates check once more before the final realizability analysis, thereby reducing redundant computations.

If any reanalyzed configuration is not realizable, the problem and solution space of the updated PL is inconsistent, and the analysis ends.

4. Implementation

In this chapter we present the code implementation of our approach which we use later in the evaluation. First we describe the reuse of existing code artifacts (see Section 4.1), then we present our adjustments and extensions (see Section 4.2).

4.1 Reuse of Existing Code Artifacts

The program[1] developed by Ochs et al. [OPS24] serves as a codebase to analyze the consistency of the base version of a PL. It takes a DIMACS file describing the FM of the PL, an Extensible Markup Language (XML) file for the resource demands, a Comma-Separated Values (CSV) file for the resource types, and another CSV file for the resource provisions as input. After reading the files, a Z3 Theorem Prover[2] [Bjø12], which is able to solve Boolean Satisfiability Problem (SAT) problems as well as Satisfiability Modulo Theorie (SMT) problems, is initialized with all the constraints representing the FM. The solver then calculates all valid configurations using blocking clauses to not calculate a configuration twice. For each configuration, the realizability is calculated. For this purpose, a separate Z3 instance is created that considers all resource provisions and demands of the specific configuration, as described in Section 2.3. If a configuration is not realizable, the program terminates. The realizability decision implemented by Ochs et al. [OPS24] is also used in our reanalysis to decide consistency between problem and solution space.

Rak [Rak24] expanded the program by creating one XML file, which contains all the information that was previously distributed across four separate files, along with a parser that reads the file. We integrated both the XML file and the parser into the program by Ochs et al. [OPS24]. Rak [Rak24] also created XML files for each version of the evolving Body Comfort System [Lit+13] case study, which we later adapt and use for our purposes. These XML serializations originate from the Eclipse Modeling Framework (EMF)[3] based model of the UCM, and ensure that all relevant data is structurally contained, enabling the reliable storage and reuse of complete system instances.

[1] https://github.com/KIT-TVA/solution-space-realisation-checker
[2] https://github.com/Z3Prover/z3
[3] https://eclipse.dev/modeling/emf/

4.2 Adjustments and Extensions to existing Code Artifacts

In this section, we describe and explain necessary adjustments and extensions. The explanation is structured into three parts: (1) the implementation of the dominates check (see Section 4.2.1), which allows us to restrict the realizability analysis to only a small set of dominating configurations, (2) the modifications necessary for handling and preprocessing deltas (see Section 4.2.2), and (3) the reanalysis of consistency between problem and solution space (see Section 4.2.3).

4.2.1 Implementation of the Dominates Check

To decrease the number of configurations that we need to reanalyze using the dominates check, as described in Section 3.2, we proceed as follows: A Z3 instance is initialized with the constraints of the FM. An empty list is created, and the first valid configuration is generated. This configuration is then passed to a filtering function, which applies the dominates check. Specifically, it is checked whether the currently computed configuration is dominated by any configuration already in the list. If this is not the case, all configurations in the list that are dominated by the new one are removed, and the new configuration is added. If the new configuration is dominated by an existing one, it is discarded, and the next valid configuration is computed using the Z3 solver, followed by another invocation of the filtering function with the new configuration.

Each configuration contains information about every feature and consists of n tuples (n = number of features). Each tuple compromises a Z3 variable corresponding to the respective feature name and its Boolean truth value in that configuration. A predefined list of features is used to impose a consistent order. The order does not matter, as long as it is applied consistently across all configuration representations. Each configuration is then transformed into an n-tuple, a sequence of ones and zeros, allowing configurations to be compared bitwise. For each feature at index i in the predefined ordered list ($0 \leq i < n$), its truth value (0 = `False`, 1 = `True`) is stored at the i-th position of the n-tuple. Listing 4.1 shows an example converting a configuration from the base version of our running example, selecting the feature `navigation-system`, to the n-tuple representation. The comparison criterion, as defined in Section 3.2, can then be restated in terms of these tuples: **Configuration A dominates configuration B, if for every index i, the condition $A[i] = 1 \vee B[i] = 0$ holds.**

```
# predefined feature order
features = ["navigation-system", "multimedia-system"]

# example configuration
selected_features = {"navigation-system"}

# build binary tuple
binary_tuple = tuple(1 if feature in selected_features else 0
    for feature in features)
# output: (1,0)
```

Listing 4.1: Convert configuration to binary tuple

For the base version of our running example, Figure Figure 3.2 shows the four valid configurations. With the exemplary feature order `"navigation-system, multimedia-system"`, the configurations are transformed into tuples as shown in Listing 4.1 resulting in the following n-tuple representations:

1: 00 2: 10 3: 01 4: 11

Starting with an empty list and the configurations from above, we proceed as follows: Configuration 1 is added to the empty list, as there are no other configurations present yet that could dominate or be dominated. Configuration 2 is not dominated by Configuration 1, but dominates Configuration 1, and thus, Configuration 1 is removed from the list while Configuration 2 is added to the list. This is because configuration 2 has a 1 at index 0 and a 0 at index 1, while configuration 1 has zeros at both indexes. The comparison is also illustrated in Figure 4.1. Configurations 2 and 3 do not dominate each other, and are therefore both included in the list, as both configurations have a 1 at an index where the other configuration has a 0. Configuration 4 is neither dominated by Configuration 2 nor by Configuration 3, but it dominates both, resulting in Configuration 4 being the only remaining configuration in the list. This is due to the fact, that Configuration 4 only consists of ones. The current states of the list after each iteration is depicted in Listing 4.2.

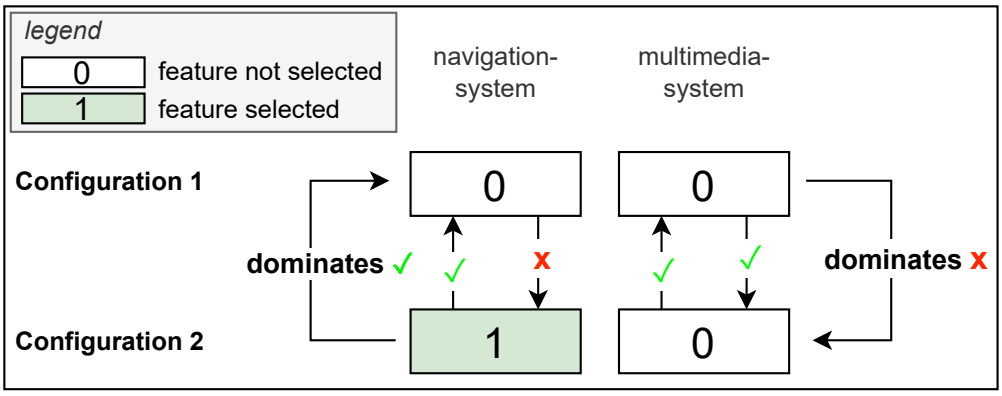

Figure 4.1: Comparing Configurations 1 and 2 with the dominates check.

```
# list after first iteration
dominating_configurations = [(0,0)]

# list after second iteration
dominating_configurations = [(1,0)]

# list after third iteration
dominating_configurations = [(1,0), (0,1)]

# list after fourth iteration
dominating_configurations = [(1,1)]
```

Listing 4.2: Convert configuration to binary tuple

4.2.2 Adjustments and Extensions for Preprocessing Deltas

In order to sequentially analyze the consistency of the deltas, they are incorporated into the same XML file in which the base version is represented. The structure is oriented towards the delta representation proposed by Rak [Rak24]. In the XML file, a new tag named <versions> is introduced after the definition of the resource types, which previously marked the end of the file. Within this tag, the various deltas can be specified using <delta> blocks. Each delta consists of individual delta operations, described in Section 3.1, which define the modifications relative to the previous version. A delta operation is encapsulated within a <delta_operation> tag.

These deltas are read and stored accordingly. For each delta, an `IncrementalAnalyzer` object is instantiated and initialized with all necessary information from the previous version, such as resource types, demands, provisions or dominating configurations, passed as parameters. The method `incremental_analysis` is then invoked on this object. This method iterates through each <delta_operation> tag and executes the corresponding preprocessing methods. This includes storing constraints representing the restrictions of valid configurations that must be reanalyzed, as described in Section 3.3.1. In addition, deciding whether a reanalysis is necessary by updating the Boolean variable variable `reanalysis_needed`, which is initially set to `False`, is also part of the preprocessing methods. Furthermore, the passed information from the previous version has to be updated, including both the list of dominating configurations and the dominating configurations themselves. Each configuration is stored as a dictionary to enable constant-time lookups, i.e. in $\mathcal{O}(1)$, as fast access to the truth value of individual features is often required. Updates are for instance adding the name and truth value of a newly introduced feature to each dictionary or appending a dictionary representing a new dominating configuration to the list.

To determine whether such updates are feasible, the Boolean variable `updating_dom_possible` is introduced, which is initially set to `True` when the `incremental_analysis` method is invoked. Within the method `_add_cross_tree_constraint(self, expression)`, it is verified whether every dominating configuration satisfies the provided `expression`. If this is not the case, `updating_dom_possible` is set to `False`. The Boolean variable is also set to `False` if either the method `_modify_tree_constraint` or `_remove_cross_tree_constraint` is called.

Once all delta_operations of a delta have been processed, the value of `reanalysis_needed` is checked. If it is `True`, a reanalysis of the PL's consistency is initiated. If it is `False`, we can proceed with the next delta, as the recently analyzed delta did not impact the PL's consistency.

4.2.3 Reanalyzing Consistency between Problem and Solution Space

As described in Section 3.3.2 the reanalysis depends on whether we were able to update the dominating configurations, checking the Boolean variable `updating_dom_possible`. If it is `True`, we can directly compute the realizability of each dominating configuration stored in the list.

If `updating_dom_possible` is `False`, the dominating configurations have to be recomputed. Thus, an additional XML file is required, containing information about

the PL after applying the delta. The file follows the same structure as the XML file used to analyze the consistency of the base version and is essentially an adapted version of the one originally created by Rak [Rak24]. After the XML file is read and processed, the Z3 solver is provided with the constraints that describe the feature model and its dependencies. The constraints stored in preprocessing are converted into Z3-processable constraints, which are combined into a single clause using disjunctions and added to the solver as an additional constraint. Using the Z3 solver in combination with the dominates check, we compute the dominating configurations. Each of these configurations is subsequently evaluated for realizability.

If any configuration is found to be unrealizable during this process, the Boolean variable `still_consistent` is set to `False`, and the entire analysis ends, including all subsequent deltas. If the PL remains consistent, the analysis proceeds with the successive delta if one is available.

5. Evaluation

In this chapter we evaluate our incremental approach for analysing the consistency between problem and solution space of a cyber-physical PL in terms of scalability. Concretely, we want to answer the following research question:

RQ: How does the incremental consistency analysis approach affect performance regarding runtime compared to a product-based analysis approach in the context of evolving cyber-physical PLs?

To address this question, we measure the time required to assess consistencies of consecutive versions of a cyber-physical PL using our incremental approach. We then measure the corresponding time required to assess the consistencies with the baseline, the product-based approach [OPS24], and compare the runtimes of both approaches. In addition to overall runtime, we measure the time required by individual analysis components, such as the time needed to perform the dominates check, enabling a more detailed examination of the strengths and potential drawbacks of the incremental analysis. This allows us to evaluate performance improvements and calculate the resulting speedup.

We start by introducing our setup in Section 5.1, then we present our results in Section 5.2, before discussing them in Section 5.3. Lastly, Section 5.4 describes the threats to validity.

5.1 Setup

The setup is divided into three parts, (1) introducing our subject system in Section 5.1.1, (2) describing the time measurement procedure in Section 5.1.2, and (3) presenting the execution environment in Section 5.1.3.

5.1.1 Subject System: The Body Comfort System Case Study and its Evolution

The BCS Case Study introduced by Lity et al. [Lit+13] is a case study from the automotive domain, modeling a selected subset of a car PL. We use the evolution

described by Rak [Rak24] as a basis and further extended and adapted it to fit our use case, as the versions are originally in an inconsistent state. Thus, we initially adjust each version to maintain consistency between problem and solution space to ensure a meaningful evaluation under high computational demand. Inconsistent versions often result in lower runtimes, because the analysis may terminate early, once an unrealizable configuration is found, and thus the remaining valid configurations no longer need to be analyzed [OPS24]. In contrast, consistent versions represent a worst-case scenario in terms of analysis effort and thus provide a more robust basis for assessing runtime improvements and speedup compared to the product-based baseline.

Version 2.0, which included the Wiper System, was sliced out following the approach proposed by Acher et al. [Ach+11], because it required excessively high computational effort due to the addition of many features, leading to a combinatorial explosion of the configuration space. This made it infeasible to compute this version using the product-based approach in a reasonable time. Therefore, all features, including associated tree constraints and cross-tree constraints, as well as corresponding realization artifact, such as software components and resources, both demands and provisions, were removed. Only the introduced resource types were retained to preserve their influence on subsequent versions. Moreover, all types of delta operations used to describe the changes in Version 2.0 are covered by the rest of the evolution. Version 5.1 from Rak had to be split into two separate versions caused by non-conformity to delta operation ordering, e.g., additions first, then modifications, then deletions [Sch10]. Furthermore, an additional version was added to cover change operations that had not been represented in the existing evolution. A description of the changes to each version is given in the following subsections. As Version 2.0 was sliced out, the sliced FMs are provided in the Appendix A.1.

Version 0

The base version of our adapted BCS Case Study evolution is almost the same as Version 1.0 from Rak [Rak24]. The FM, presented in Figure 5.1, also consists of 27 Features and six cross-tree constraints, resulting in a total of 11616 valid configurations. In the solution space, there are eight resource types, four software components, and two hardware components. To create a consistent base version, the boundary type of the resource type with ID 5 changed from `EXACT` to `LOWER`. In addition, the resources are adapted, as the second hardware component now provides a `value` of 5 instead of 4 of the resource type with ID 5, which results in all demands being fully covered by the provisions, thus a consistent problem and solution space of the described PL.

Version 1

The changes from Version 0 to Version 1 correspond to the changes from Version 1.0 to Version 1.1 from Rak [Rak24]: An additional safety hardware component is added to provide all resources associated with passenger safety.

Version 2

Version 2 adds a new feature, `Seat`, along with new demands, and a new software component. The number of valid configurations increases to a total of 21984. Since

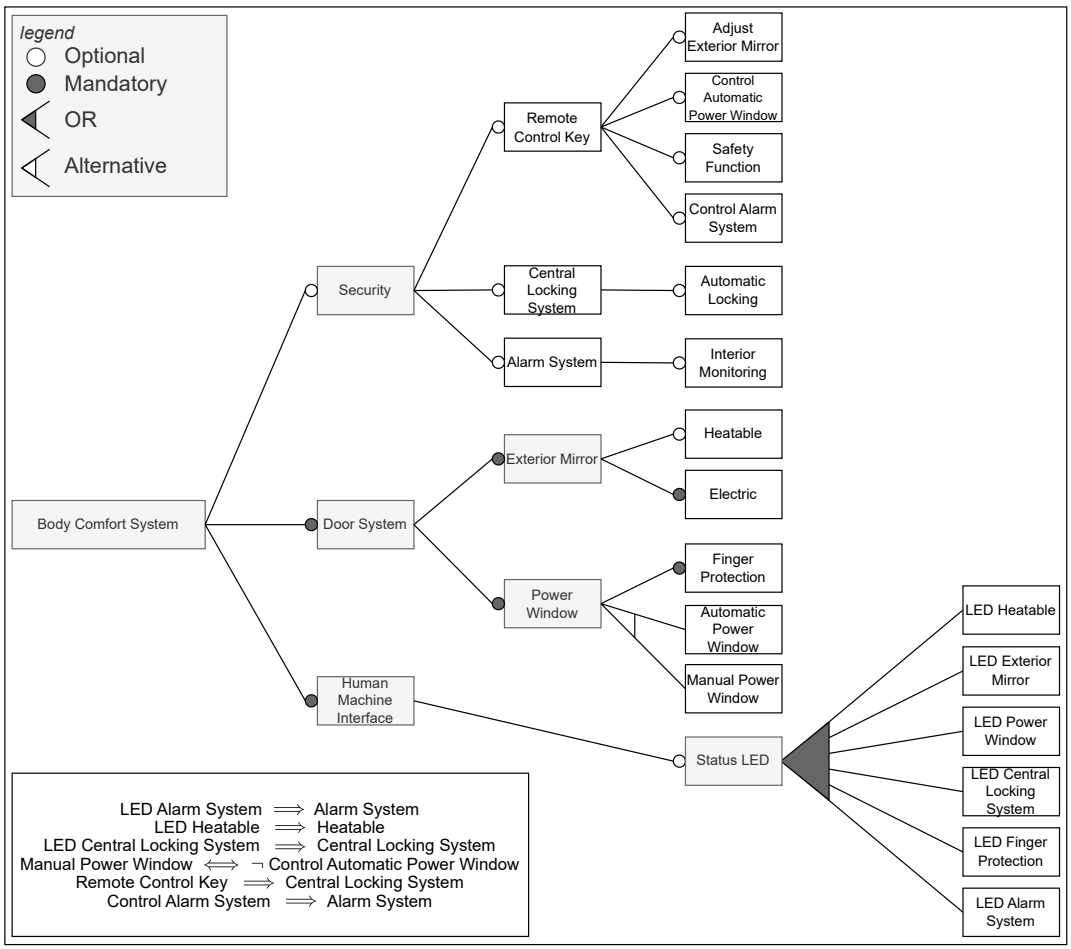

Figure 5.1: The FM of the BCS [Lit+13].

Version 2.0 from Rak [Rak24] was sliced out [Ach+11], the resource types originally added in that version are also included in our Version 2, as they are required in later stages. The corresponding provisions are adjusted accordingly to ensure consistency in the updated PL version.

Version 3

Version 3 only adds a new resource type `Response Time`, as well as a single demand and a corresponding provision for this resource type.

Version 4

The changes introduced in Version 4 largely correspond to those made in Version 4.0 by Rak [Rak24]. The feature `Window Heatable` and its associated `StatusLED LED Window Heatable` are added. Additionally, a cross-tree constraint is introduced, establishing an implies relation from `LED_Window_Heatable` to `Window_Heatable`. This leads to a total of 65952 valid configurations. Finally, the version includes associated demands and provisions, along with the addition of a new software component.

Version 5

In Version 5, the new feature `Automatic Headlights` is introduced, which doubles the number of valid configurations to 131904. In addition, six new resource types are added, each with a corresponding resource demand, as well as a new software component, and a new hardware component that satisfies all resource demands.

Version 6

Version 5.1 violates the invariant for maintaining a consistent order of add, modify, and remove operations [Sch10]. As a result, we split the changes into two versions. Version 6 begins by adding two resource provisions, followed by the removal of all six resource types introduced in Version 5, along with their associated demands and provisions.

Version 7

To complete the changes of Version 5.1, three optional features `Beam`, `Daytime Running Lights`, and `Parking Lights` are added, together with two new resource types and their corresponding demands and provisions. With a total of 593568, this Version has the highest count of valid configurations.

Version 8

With Version 8 a new version is added to the existing evolution, covering delta operations such as `modify TC`, `modify RT`, `remove Feature`, and `remove CTC`, which did not occur in previous deltas. Due to low customer demand, the features `Window Heatable` and `LED Window Heatable` are removed including their associated resource demands and the cross-tree constraint between them. To enhance security, the features `Security` and `Interior Monitoring` are now marked as mandatory. Lastly, the boundary type of the resource type `no. security cameras` is changed from `LOWER` to `EXACT`, ensuring consistency. The changes in this Version decrease the number of valid configurations to a total of 110160.

The delta operations `remove SWC` and `remove HWC` are omitted, as they do not affect consistency and would therefore not contribute to the evaluation. For reference, Table 5.1 provides an overview of the number of valid configurations across all versions discussed above.

Version	Number of Valid Configurations
Version 0	11616
Version 1	11616
Version 2	21984
Version 3	21984
Version 4	65952
Version 5	131904
Version 6	131904
Version 7	593568
Version 8	110160

Table 5.1: Number of valid configurations for each version

5.1.2 Time Measurement Procedure

For the naive approach, we use the time measurement procedure proposed by Ochs et al. [OPS24] as a baseline. In the incremental analysis approach, we measure the runtime for each version and additionally the accumulated times of the dominates check, allowing us to evaluate whether the additional overhead of the dominates check yields a favorable trade-off in terms of overall runtime performance. Listing 5.1 illustrates the individual time components measured during our incremental analysis. These will be explained in more detail in the following, supported by selected and simplified code excerpts. Colored highlights in the listings provide visual guidance, indicating where and when the measurements were taken.

Time Measurements for the Base Version in the Incremental Analysis Approach: Since no previous version exists from which information could be reused, it is necessary to perform a full consistency analysis on the base version. In order to avoid evaluating the realizability of every valid configuration, as done by the product-based approach, we apply the dominates check as an optimization. Accordingly, we perform the same time measurement procedure as for the naive approach, additionally measuring the overhead introduced by the dominates check. This allows us to assess its performance impact and to quantify the trade-off between reduced realizability analysis and increased overhead. The overhead includes the time required to convert a configuration into a binary sequence, to perform the dominates check itself, and to transform the binary sequence back into a dictionary format (see Listing 5.2).

```
begin base version
measure total-time, accumulated validation-/realization-/
   domination check-times
end base version
while delta available:
  begin delta
    start delta-time
    begin preprocessing
      measure accumulated validation-/domination-times
    end preprocessing
    if reanalysis:
      begin reanalysis
        measure accumulated validation-/realization-/
           domination-times
      end reanalysis
    finish delta-time
  end delta
resume all (accumulated) times for base version for each
   delta
```

Listing 5.1: Convert configuration to binary tuple

Time Measurements for the Main Incremental Analysis: For each delta, the total runtime is measured as shown in Listing 5.3, including preprocessing and reanalysis. During preprocessing and reanalysis the accumulated times for determining valid configurations, performing the dominates check as well as the realization analysis follow the corresponding time measurement procedures described previously, allowing us to calculate their share of overall runtime.

```
valid_configs = []
result = []

while True:
    if [...]:
        break
    [...]
    dominates_check_timings.start()
    config = tuple(1 if model.eval(feat, model_completion=True)
       else 0 for feat in features)
    if not any(_is_dominated(config, c) for c in valid_configs):
        valid_configs = _remove_dominated(valid_configs, config)
        valid_configs.append(config)
    dominates_check_timings.finish()
    [...]
dominates_check_timings.start()
result = [{str(feat): bool(value) for feat, value in zip(features,
   config)} for config in valid_configs]
dominates_check_timings.finish()
```

Listing 5.2: Excerpt of Python code showing the dominates_check_timings measurement for the base version in the incremental analysis.

```
def incremental_analysis(self, delta, count):
    self.delta_processing_timings.start()
    [...]

    for delta_operation in delta_operations:
        [...(Preprocessing)]

    if self.calculation_needed:
        [...(Reanalysis)]

    self.delta_processing_timings.finish()
```

Listing 5.3: Simplified Python code showing the delta_processing_timings and calculation_timings in the incremental analysis

5.1.3 Execution Environment

The program was executed on a high-performance workstation equipped with an AMD Ryzen Threadripper PRO 5955WX processor featuring 16 cores, and 128 GB of DDR4 RAM. The system ran on Ubuntu Server 24.04.2 LTS and utilized Python version 3.12.8[1] for all implementations.

We executed the product-based approach eleven times for each Version 0 to 6 of our BCS case study evolution, and ten times for Version 8. In contrast, our incremental approach was executed analyzing Versions 0 to 7 a total of 500 times, and Versions 0 to 8 an additional 100 times, as we expect the product-based approach to have a higher runtime. Table 5.2 summarizes the number of executions per version for both the product-based and incremental approaches. Since most versions were executed

[1] https://www.python.org/downloads/release/python-3128/

multiple times, the arithmetic mean is calculated for each case and presented in Section 5.2.

Version	Product-Based Approach	Incremental Analysis Approach
Version 0	600	11
Version 1	600	11
Version 2	600	11
Version 3	600	11
Version 4	600	11
Version 5	600	11
Version 6	600	11
Version 7	600	0
Version 8	100	10

Table 5.2: Number of program runs per version and analysis approach

5.2 Results

The measured runtimes for each version and analysis approach are based on multiple executions. Since runtimes exhibited very similar distributions across all versions, only Version 0 is depicted as a representative example in Figure 5.2 for brevity. This figure presents the runtime distribution of the consistency analysis for both the product-based and incremental approaches for Version 0, using violin plots. The product-based approach is shown on the left, the incremental analysis on the right. Each violin illustrates the distribution shape: wider areas represent more frequent values, while narrower regions indicate fewer data points. The black box within each violin shows the interquartile range (the middle 50 % of the data), with a horizontal line indicating the mean. Dashed lines mark the minimum and maximum values.

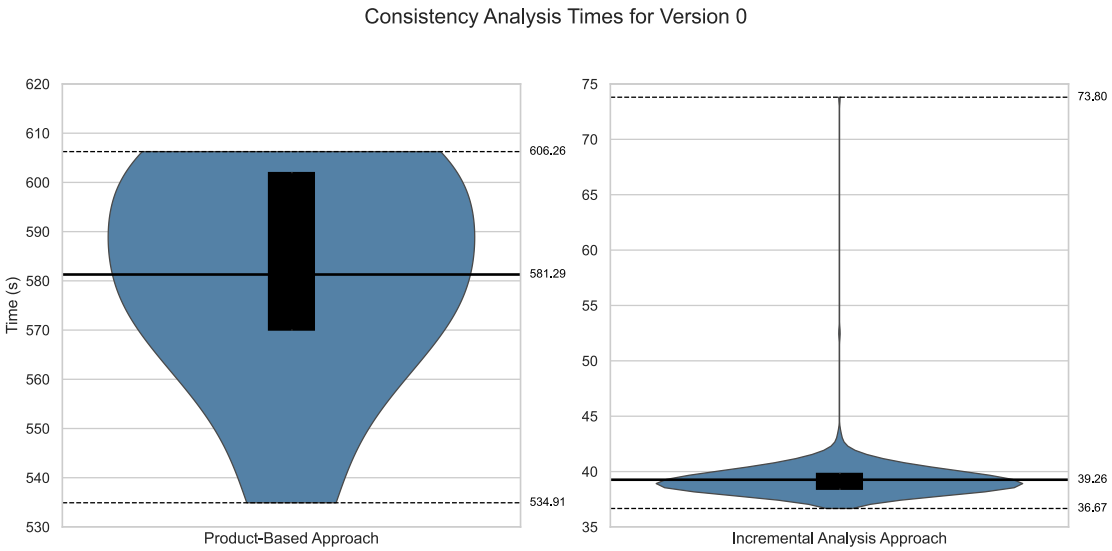

Figure 5.2: Violin plots of the consistency analysis runtimes for Version 0 using the product-based (left) and the incremental analysis approach (right).

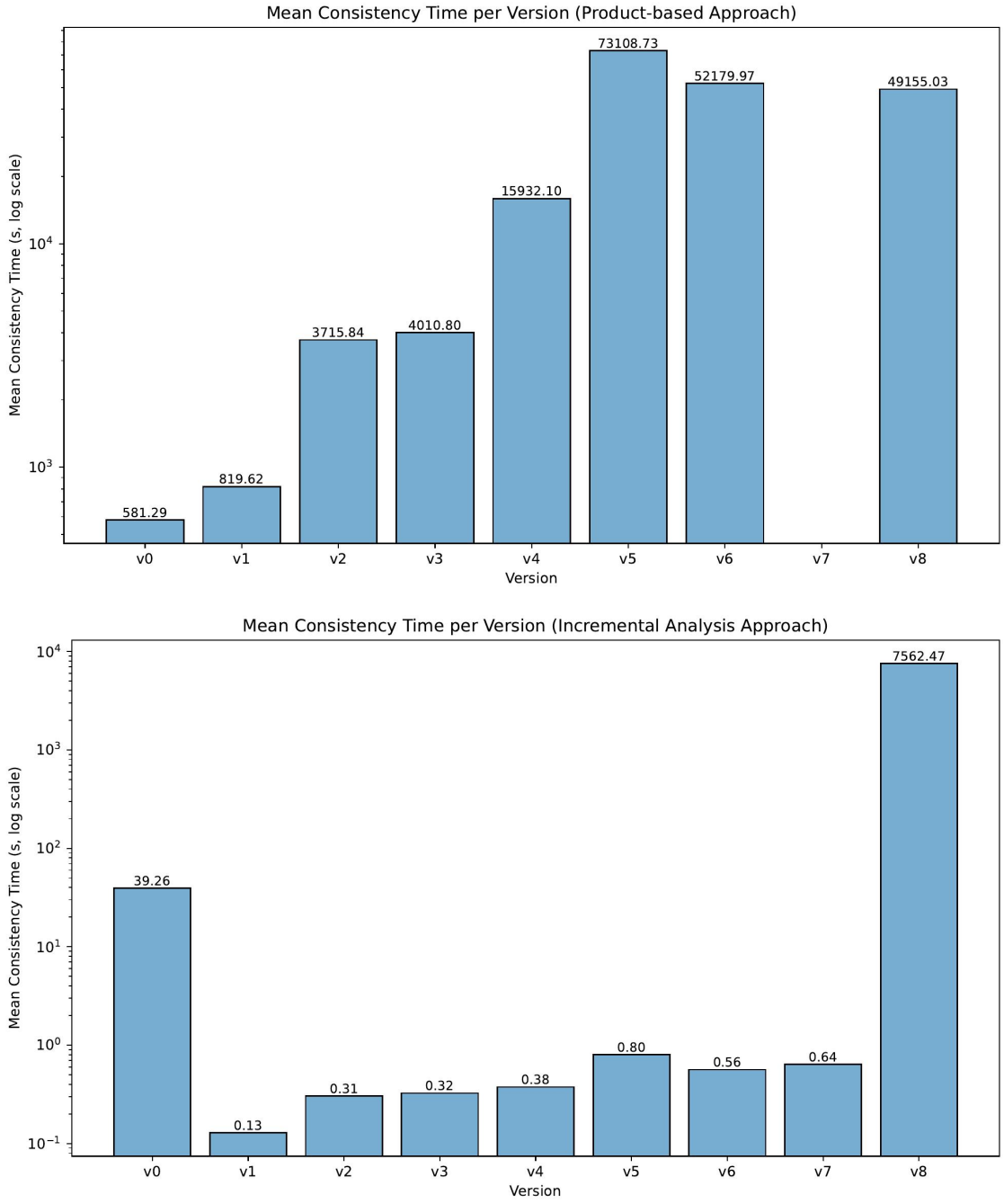

Figure 5.3: Mean runtimes of each version for both product-based (top) and incremental approach(bottom).

The product-based approach exhibits runtimes ranging from approximately 535 s to 606 s, with a mean of 581.29 s. The distribution is relatively broad and evenly spread. In contrast, the incremental approach shows runtimes between approximately 36.7 s and 73.8 s, with most values concentrated around the mean of 39.26 s, and a few data points lying well above the main distribution. The mean runtime for each versions for both product-based and incremental approaches are presented in Figure 5.3. To reduce complexity, all subsequent evaluations are based on the mean runtimes given in seconds.

Figure 5.3 presents the results for the product-based approach on the top, the results for the incremental approach on the bottom. In the product-based approach, Version 0 exhibits the lowest mean runtime with approximately 581 s. The runtimes increase with each subsequent version, until peaking in Version 5, which reaches the highest mean runtime of approximately 73109 s. As the product-based approach was not executed on Version 7, no data is available for that version.

In the incremental approach for Version 0, the mean runtime is approximately 39.26 s. Versions 1 through 7 show mean runtimes ranging between approximately 0.13 s and 0.80 s. In contrast, Version 8 shows an increase, with a mean runtime of approximately 7562.47 s.

Runtime breakdown	v0	v1	v2
Consistency Analysis Time	39.26347683	0.12881319	0.30526969
Valid Configurations Time	35.81865625	0.12874877	0.30512431
Realizability Analysis Time	0.00258295	0.00000000	0.00000028
Dominates Check Time	3.33763868	0.00000000	0.00000143
	v3	v4	v5
Consistency Analysis Time	0.32489365	0.37509637	0.80225042
Valid Configurations Time	0.32484506	0.37495223	0.80207663
Realizability Analysis Time	0.00000000	0.00000045	0.00000027
Dominates Check Time	0.00000000	0.00000265	0.00000151
	v6	v7	v8
Consistency Analysis Time	0.56418538	0.63997198	7562.47425747
Valid Configurations Time	0.56396119	0.63976865	-
Realizability Analysis Time	0.00000000	0.00000059	-
Dominates Check Time	0.00000000	0.00000305	-

Table 5.3: Mean overall runtimes and their breakdown into three components (Valid Configurations Time, Realizability Analysis Time, and Dominates Check Time) across all versions using the incremental approach.

Table 5.3 presents the mean runtimes for different analysis components across all versions using the incremental approach. The table presents the overall runtime and distinguishes between the time spent identifying and modifying valid configurations, performing the dominates check, and the time for the realizability analysis. The valid configurations time is consistently close to the overall consistency analysis time, while the other two components remain near zero, except in Version 0, where the dominates check takes approximately 3.34 s. For Version 8 only the consistency analysis time is presented.

5.3 Discussion

In this section, we interpret and discuss the results, presented in the previous section, providing insights into the strengths and limitations of our incremental analysis approach. We first examine performance trends regarding run time across different versions within the same approach (see Section 5.3.1). We then compare our approach to the naive product-based approach (see Section 5.3.2), enabling us to address the main research question posed in this thesis.

5.3.1 Comparing Versions within an Approach

Comparing the runtimes of the product-based approach with the number of valid configurations for each version, we can see a general trend of increasing runtimes depending on an increasing number of valid configurations, as shown in Figure 5.4, a direct consequence of the combinatorial explosion of valid configurations. For Version 7, which had a total of 593568 valid configurations, we obtained no result, after ending the analysis after approximately 5 days (432000 s). This reinforces our initial hypothesis that the product-based approach struggles to scale efficiently for large evolving cyber-physical PLs.

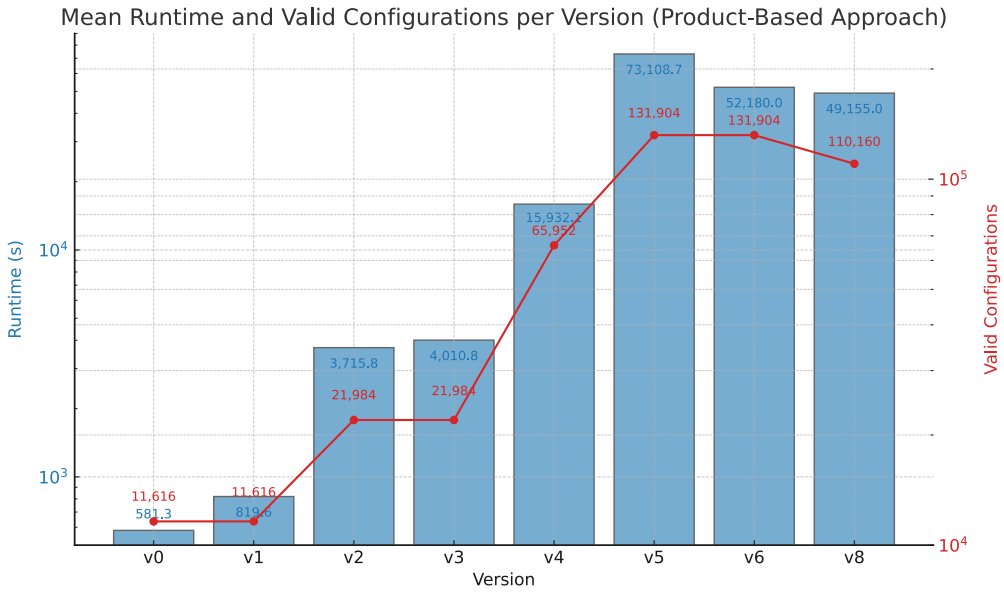

Figure 5.4: Mean runtimes regarding valid configurations for each version for the product-based approach.

Although Versions 0 and 1 both contain 11616 valid configurations, Version 1 exhibits a higher runtime (819.62 s) compared to Version 0 (581.29 s). This increase can be attributed to the changes introduced in Version 1: the addition of a hardware component with associated resource provisions introduces further constraints for the realizability analysis, increasing its complexity and computational overhead (as discussed in the following section). In contrast, Version 6 runs faster (52179.97 s) than Version 5 (73108.73 s) despite both comprising 131904 valid configurations. While Version 6 introduces two additional resource provisions, the concurrent removal of six resource types along with their associated demands and provisions noticeably

reduces the number of constraints in the realizability analysis, leading to lower complexity and shorter runtime.

Looking at the runtimes of the incremental approach (Figure 5.3), Version 1 exhibits a comparatively high runtime. This is expected, as it requires a full initial consistency analysis, as no prior version exists from which information can be reused. From Versions 1 through 7 the runtime generally increases, with a slight deviation in Version 5, which peaks at around 0.8 s, but still remains very low overall. These consistently low runtimes can be explained by the fact that the set of dominating configurations could be successfully updated in all of these versions. None of the applied deltas contained delta operations that, accordingly to Section 3.3.1, would prevent this. As a result, the realizability analysis could be performed directly on the updated dominating configurations without the need for additional computation.

Version 8, however, represents a worst-case scenario for the incremental approach. Due to the removal of a cross-tree constraint and a change in the boundary type of a resource type from `LOWER` to `EXACT`, the corresponding delta includes the delta operations `remove CTC` and `modify RT`, both of which trigger reanalysis (see Figure 3.4). Moreover, the delta operation `remove CTC` prevents the existing set of dominating configurations from being updated (see Figure 3.5), requiring the dominating configurations to be recomputed for reanalysis. The restrictions determined during preprocessing offer little reduction in the number of configurations to analyze, since the modified resource type is involved in many configurations, as the feature `Alarm System` introduces demands of this resource type. The feature `Alarm System` appears in at least half of all valid configurations, as no cross-tree constraint restricts its selection, and it is located beneath the mandatory feature `Security`, which is directly under the root feature `BCS`. Additionally, cross-tree constraints from other features implying `Alarm System` likely result in it being contained in an even greater share of valid configurations.

The high computational effort required for analyzing the consistency of Version 8 highlights a limitation of our approach: dependencies between delta operations are not taken into account (see SG2). In this case, the `remove CTC` operation prevents the update of the previously computed dominating configurations, which necessitates recomputing the set of dominating configurations for reanalysis. This computational effort could have been avoided, as the removed cross-tree constraint was tied to two features that were also deleted within the same delta. A similar case occurs in Version 6, where reanalysis is triggered by detecting a delta operation `remove resource` affecting a resource provision. However, the reanalysis could have also been avoided, since the removal of the corresponding resource type implicitly removes all related resource demands and provisions in accordance with syntactic consistency [SZ00], and no other delta operation in that version requires reanalysis.

5.3.2 Comparison between both Approaches

The results clearly demonstrate that our incremental approach offers substantial optimization over the product-based baseline. In the product-based approach, only about 16 % of the total runtime in Version 0 is spent generating valid configurations and performing realizability analysis, while the remaining 84 % constitutes overhead, primarily due to loading and processing data, as well as constructing CSPs [OPS24]. In contrast, our incremental approach reduces this overhead in Version 0 to just

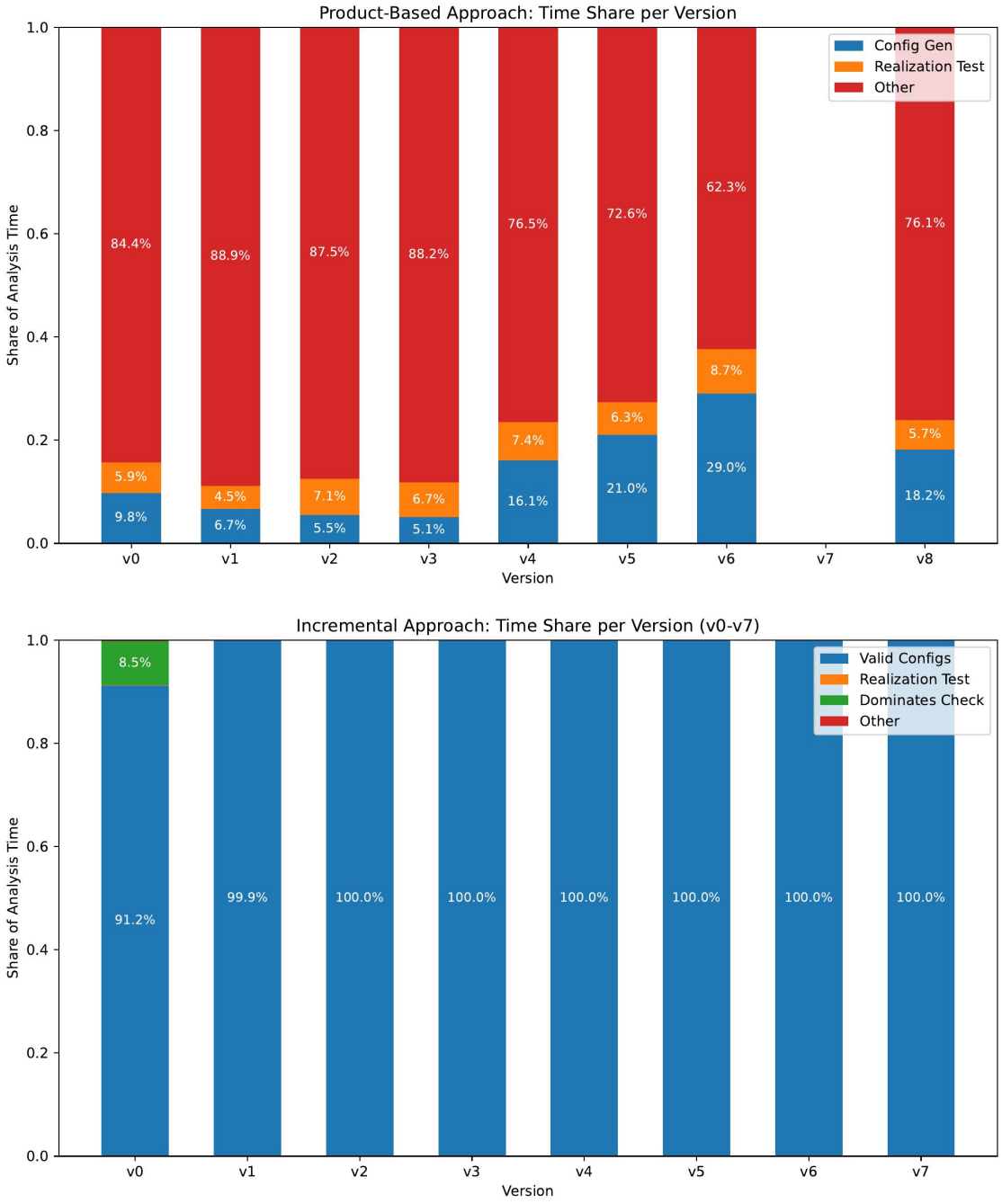

Figure 5.5: The normalized distribution of analysis time across individual components for Versions 0 to 8 for the product-based approach and for Versions 0 to 7 for the incremental approach.

8.77 %, the majority of which, about 8.5 %, is due to the dominates check. This suggests that the dominates check is highly effective, as it noticeably limits the number of configurations that require realizability analysis, resulting in realizability runtimes close to zero, and thereby additionally avoids the overhead associated with constructing necessary CSPs.

In Versions 0 to 3 of the product-based approach, the relative shares of time spent on generating valid configurations and on realizability analysis remain largely stable. From Versions 4 to 6, the portion of time dedicated to generating valid configurations

increases steadily, while the share for realizability analysis stays nearly unchanged. This can be explained by the fact that more valid configurations have to be generated, while the complexity of the CSPs necessary for realizability analysis decrease, especially in Version 6, as mentioned before. In contrast, the incremental approach exhibits overhead close to 0 %, in Versions 1 through 7. This is due to efficient storage and updating of the few dominating configurations. The breakdown of runtime shares for both the product-based approach (top) and the incremental approach (bottom) is presented in Figure 5.5, allowing for a direct visual comparison.

These optimizations explain the reduced runtimes for all versions, including the base version, where we reached a reduction of runtime from approximately 581.29 s to 39.26 s. Especially, when the first reanalysis procedure is applicable, i.e., storing and updating the dominant configurations across the end of a delta, the incremental approach provides fast results even in cases involving a dramatic increase in configuration space. This is evident in Version 7, where the product-based analysis was terminated after approximately five days without producing a result, whereas the incremental approach completed the consistency analysis in just 0.64 s.

A key factor behind this performance gain is whether dominating configurations can be stored and updated, which depends directly on the delta operations present in an applied delta. Accordingly, the speedup ($Speedup = \frac{T_{old}}{T_{new}}$) achieved by the incremental approach varies depending on whether the dominating configurations can be reused or must be recomputed. Versions 0 and 8, which require the computation of dominating configurations, achieve an average speedup of around 10.65. In contrast, Versions 1 through 6, where updating dominating configurations is possible, show a mean speedup of approximately 42830, with Version 6 peaking at nearly 92500. A complete overview of the speedup values is shown in Table 5.4.

Despite the differences between the two reanalysis procedures, the overall results clearly show that the incremental approach offers substantial performance advantages over the product-based approach. Across the entire evolution of the cyber-physical PL, we achieved an average speedup of approximately 32124. This underlines the scalability of incremental consistency analysis, making it a promising solution for handling consistency in evolving cyber-physical PLs.

Version	Speedup ($\frac{T_{Naive}}{T_{Incremental}}$)
v0	14.80
v1	6362.86
v2	12172.33
v3	12344.95
v4	42474.68
v5	91129.56
v6	92487.27
v7	-
v8	6.50

Table 5.4: Speedups across all versions comparing the product-based approach to the incremental approach.

5.4 Threads to Validity

In this section, we discuss internal threats to validity addressing the setup and evaluation procedure (see Section 5.4.1), as well as external threads to validity addressing generalizability (see Section 5.4.2).

5.4.1 Internal Threads to Validity

An internal threat arises from the randomness involved in generating valid configurations and executing the dominates check. Since configurations are compared to each other when performing the dominates check, as described in Section 4.2.1, the number of comparisons varies with the order in which configurations are processed. For instance, in the base version of our running example, generating the configuration which selects both possible features first, leads to 3 comparisons as all other configuration only need to be compared once, while in the example procedure described in Section 4.2.1, 6 comparisons are needed. Thus, runtime depends on the order in which the configurations are generated. To reduce the impact of varying runtimes, each analysis was executed multiple times and the average runtime was used to provide a more stable and representative basis for evaluating performance, counteracting the influence of outliers caused by randomness, as seen in Figure 5.2.

Another concern is the low number of evaluation executions across different versions of the PL regarding the product-based approach. However, the measured times show that additional runs are deemed impractical, given that the analyses already require an excessive amount of time to complete; as mentioned earlier, Version 7 did not deliver a result, even after approximately 5 days.

A further internal threat lies in the limited validation of intermediate analysis results. Although the overall implementation was tested during development, we did not verify the correctness of each subcomponent, such as the dominates check, during evaluation. However, this does not necessarily distort the results in a way that would benefit our findings. Since the number of dominating configurations is typically very small, an incorrect implementation would more likely result in longer runtimes due to excessive realizability checks. Therefore, any potential implementation error in the dominates check would likely reduce, rather than increase, the observed speedup.

5.4.2 External Threads to Validity

A central external threat concerns that the analysis of inconsistency was not a subject of the evaluation. However, since the focus was on measuring performance under high computational demand, only consistent versions were considered. As discussed in Section 5.1.1, this choice ensures that the results reflect the worst-case analysis effort and therefore offers a meaningful and robust basis for comparing runtime improvements against the product-based baseline.

Another external threat relates to the generalizability of our findings. The incremental approach was evaluated using only a single case study, the BCS [Lit+13]. Although this limits the scope of generalization, it is important to note that the BCS was developed in collaboration with domain experts and specifically designed to mirror the structure and complexity of a realistic cyber-physical PL [Mül+09]. Although further case studies are necessary to confirm the general applicability of our findings, the BCS case study provides a solid foundation for the initial evaluation.

6. Related Work

This chapter outlines foundational research and methods that inform and contextualize this thesis. The related work is grouped into three areas: consistency analysis between problem and solution space, sampling and filtering techniques, and delta modeling with UCM extensions. Each of these areas is discussed in relation to the challenges addressed by our incremental consistency analysis approach.

6.1 Consistency Between Problem and Solution Space

Several works address the gap between configurable options and their technical realizability. Hentze et al. [Hen+22] propose deriving a dedicated feature model for the solution space to quantify the mismatch with the problem space. Similarly, Thüm et al. [Thu+11] demonstrate how abstract features can inflate perceived variability without affecting realizability. Both contributions focus on structural causes for inconsistencies but remain confined to a static view of a PL.

Ochs [OPS24] introduces a CSP-based method to identify valid and realizable configurations using the UCM. His work extends earlier approaches by formally analyzing consistency across both domains. Like our work, it quantifies mismatches through configuration sampling. However, his method recomputes the full configuration space after each change. In contrast, our analysis incrementally reanalyzes only those parts affected by deltas, enabling version-aware analysis over evolving PLs.

6.2 Sampling and Filtering Techniques

Sampling techniques help to manage the combinatorial growth of configuration spaces. Medeiros et al. [Med+16] investigate multiple strategies for t-wise sampling, where all feature combinations of size t, such as pairs or triples, are covered by at least one configuration. These methods trade off completeness against feasibility and are frequently used to support scalable testing. Another widely used heuristic is the most-enabled-disabled approach, which selects one configuration with the maximum number of features enabled and another with the minimum [Med+16].

These approaches share the goal of reducing the number of configurations while maintaining analysis quality. However, they differ from the dominance-based filtering introduced in this thesis. T-wise sampling focuses on covering interaction combinations, whereas the dominance approach identifies configurations that are subsets of others and can thus be omitted without losing consistency information. The most-enabled-disabled strategy overlaps conceptually, but its all-features-enabled configuration is often invalid in domains with tight constraints and cannot serve as a reliable consistency representative.

6.3 Delta Modeling and UCM Extensions

The UCM introduced by Ananieva et al. [Ana+22] serves as a basis for integrating variability from both problem and solution spaces. Subsequent extensions by Wittler et al. [WKR22] and Ochs et al. [OPS24] enhance the representation of solution artifacts such as software components and resources, laying the groundwork for consistency analysis.

Building on this foundation, Rak [Rak24] applies delta modeling to represent variability in time within the UCM. However, his approach does not formalize a dedicated delta dialect; instead, changes are defined through modifications to relations between elements, rather than modifications to entities. In contrast, our work specifies a concrete and well-formed delta dialect and ensures that all structural inconsistencies can be identified and resolved, in accordance with established practices to ensure syntactic correctness [SZ00; MJC17].

7. Conclusion and Outlook

In this thesis, an incremental analysis approach was developed to improve the scalability of consistency analysis between problem and solution space in evolving cyber-physical PLs. To reduce redundant computations inherent in the product-based approach, we first specified a delta dialect to model the evolution of a cyber-physical PL and examined the effect of each delta operation on consistency analysis. In a preprocessing stage, we investigated the delta operations within a delta to determine if and how a reanalysis is required. Taking preprocessing into account, we reanalyzed only the parts of a cyber-physical PL that are affected by changes. This process was further optimized by the dominates check, which identifies dominant configurations and thus reduces the number of configurations for subsequent realization analysis. To answer the main research question "How does the incremental consistency analysis approach affect performance regarding runtime compared to a product-based analysis approach in the context of evolving cyber-physical PLs?", we used an adaption of the evolution of the BCS case study, and applied both product-based and incremental approaches to it, measuring the runtimes to decide consistency of each version. We then compared the runtimes to calculate the speedup. We came to the result that our incremental consistency analysis approach improves performance regarding runtime depending on the delta operations within a delta. The results demonstrate notable speedups, ranging from 6.5 to approximately 92,500, clearly highlighting the benefit of storing and updating dominating configurations. Moreover, the incremental approach consistently outperformed the product-based approach, achieving performance gains in all evaluated cases.

Future research could explore ways to store more information about dominating configurations so they can be updated in every case, always enabling the fast reanalysis. Furthermore, the combined effects of multiple delta operations and their interdependencies could be explored, as these further reduce computational effort, as seen in the evaluation. Another promising direction is the incremental analysis of a unified CSP that includes both problem and solution space constraints, as we analyzed them separately. However, treating them as a single combined model could offer additional optimization opportunities. Moreover, incremental SMT solving is an active area of research, making such an integrated approach practically relevant.

Bibliography

[Ach+11] Mathieu Acher et al. "Slicing feature models". In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. USA: IEEE Computer Society, 2011, pp. 424–427. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100089. URL: https://doi.org/10.1109/ASE.2011.6100089.

[Ana+22] Sofia Ananieva et al. "A conceptual model for unifying variability in space and time: Rationale, validation, and illustrative applications". In: *Empirical Software Engineering* 27.5 (May 30, 2022), p. 101. ISSN: 1573-7616. DOI: 10.1007/s10664-021-10097-z. URL: https://doi.org/10.1007/s10664-021-10097-z.

[Ape+13] Sven Apel et al. *Feature-Oriented Software Product Lines*. Jan. 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7.

[Bjø12] Nikolaj Bjørner. "Taking Satisfiability to the Next Level with Z3". In: *Automated Reasoning*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–8. ISBN: 978-3-642-31365-3.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Accessed: 2025-May-7. Aug. 2001. URL: https://insights.sei.cmu.edu/library/software-product-lines-practices-and-patterns/.

[Coo71] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. event-place: Shaker Heights, Ohio, USA. New York, NY, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 978-1-4503-7464-4. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[Hen+22] Marc Hentze et al. "Quantifying the variability mismatch between problem and solution space". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. MODELS '22. event-place: Montreal, Quebec, Canada. New York, NY, USA: Association for Computing Machinery, 2022, pp. 322–333. ISBN: 978-1-4503-9466-6. DOI: 10.1145/3550355.3552411. URL: https://doi.org/10.1145/3550355.3552411.

[Lit+13] Sascha Lity et al. "Delta-oriented Software Product Line Test Models – The Body Comfort System Case Study". In: (2013).

[Med+16] Flávio Medeiros et al. “A Comparison of 10 Sampling Algorithms for Configurable Systems”. In: Feb. 2016, pp. 643–654. DOI: 10.1145/2884781.2884793.

[MJC17] Nuno Macedo, Tiago Jorge, and Alcino Cunha. “A Feature-Based Classification of Model Repair Approaches”. In: *IEEE Transactions on Software Engineering* 43.7 (2017), pp. 615–640. DOI: 10.1109/TSE.2016.2620145.

[Mül+09] Tamara Müller et al. “A Comprehensive Description of a Model-based, Continuous Development Process for AUTOSAR Systems with Integrated Quality Assurance”. In: 2009. URL: https://api.semanticscholar.org/CorpusID:60559222.

[OPS24] Philip Ochs, Tobias Pett, and Ina Schaefer. “Consistency Is Key: Can Your Product Line Realise What It Models?” In: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. MODELS Companion ’24. event-place: Linz, Austria. New York, NY, USA: Association for Computing Machinery, 2024, pp. 690–699. ISBN: 9798400706226. DOI: 10.1145/3652620.3687812. URL: https://doi.org/10.1145/3652620.3687812.

[PBL05] Klaus Pohl, Günter Böckle, and Frank Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Publication Title: Software Product Line Engineering: Foundations, Principles, and Techniques. Jan. 2005. ISBN: 978-3-540-24372-4. DOI: 10.1007/3-540-28901-1.

[Rak24] Lennart Rak. “Implementing a System Generation Aware Unified Conceptual Model”. Abschlussarbeit - Bachelor. Karlsruher Institut für Technologie (KIT), 2024. DOI: 10.5445/IR/1000174678.

[RND10] Stuart J. (Stuart Jonathan) Russell 1962-, Peter Norvig, and Ernest Davis. *Artificial intelligence : a modern approach*. New Jersey, United States: Upper Saddle River : Prentice Hall, 2010. ISBN: 978-0-13-604259-4. URL: https://searchworks.stanford.edu/view/9317690.

[Sch10] Ina Schaefer. “Variability Modelling for Model-Driven Development of Software Product Lines.” In: Jan. 2010, pp. 85–92.

[SZ00] GEORGE SPANOUDAKIS and ANDREA ZISMAN. “INCONSISTENCY MANAGEMENT IN SOFTWARE ENGINEERING: SURVEY AND OPEN RESEARCH ISSUES”. In: *Handbook of Software Engineering and Knowledge Engineering*. _eprint: https://www.worldscientific.com/doi/pdf/10.1142/9 2000, pp. 329–380. DOI: 10.1142/9789812389718_0015. URL: https://www.worldscientific.com/doi/abs/10.1142/9789812389718_0015.

[Thu+11] Thomas Thum et al. “Abstract Features in Feature Modeling”. In: *2011 15th International Software Product Line Conference*. 2011, pp. 191–200. DOI: 10.1109/SPLC.2011.53.

[WKR22] Jan Willem Wittler, Thomas Kühn, and Ralf Reussner. "Towards an integrated approach for managing the variability and evolution of both software and hardware components". In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B*. SPLC '22. event-place: Graz, Austria. New York, NY, USA: Association for Computing Machinery, 2022, pp. 94–98. ISBN: 978-1-4503-9206-8. DOI: 10.1145/3503229.3547059. URL: https://doi.org/10.1145/3503229.3547059.

A. Appendix

A.1 Sliced Feature Models for the Evolution of the Body Comfort System

FM different than the initial one:

FM of Version 8:

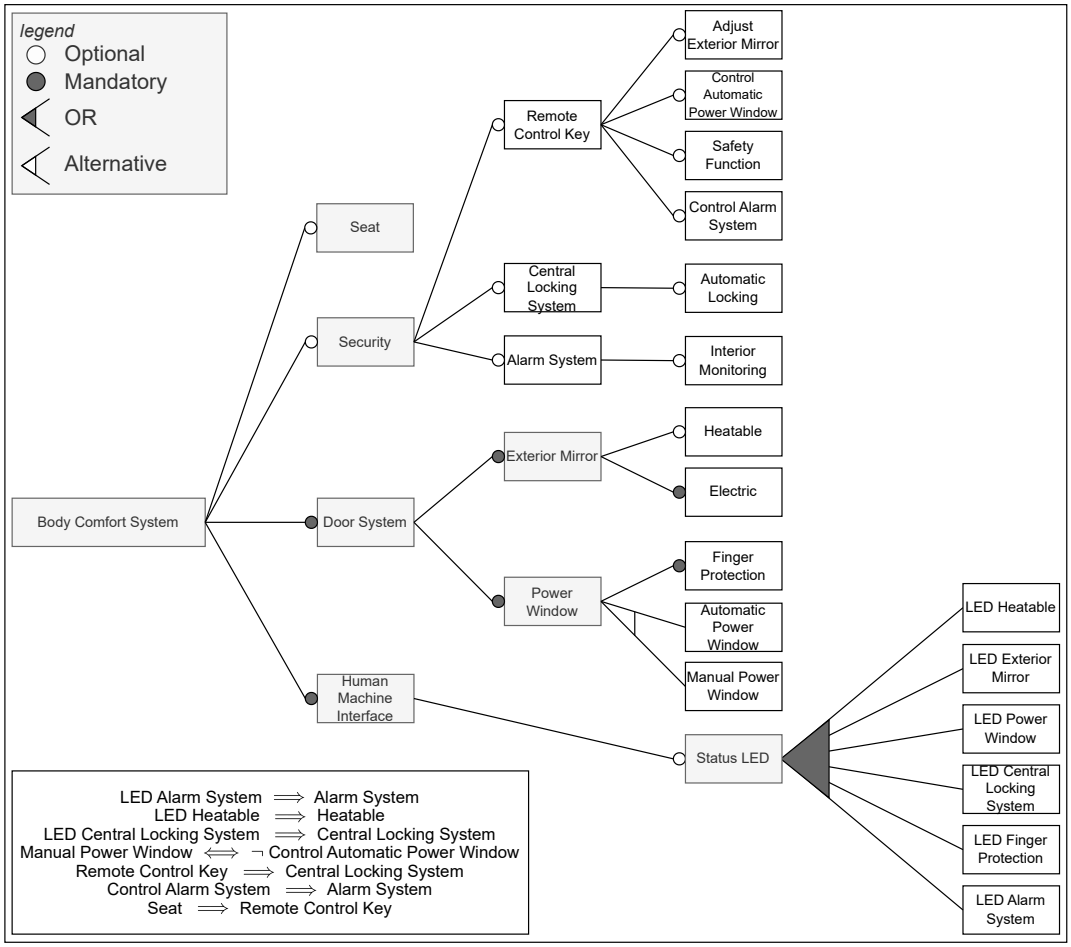

Figure A.1: FM of Version 2

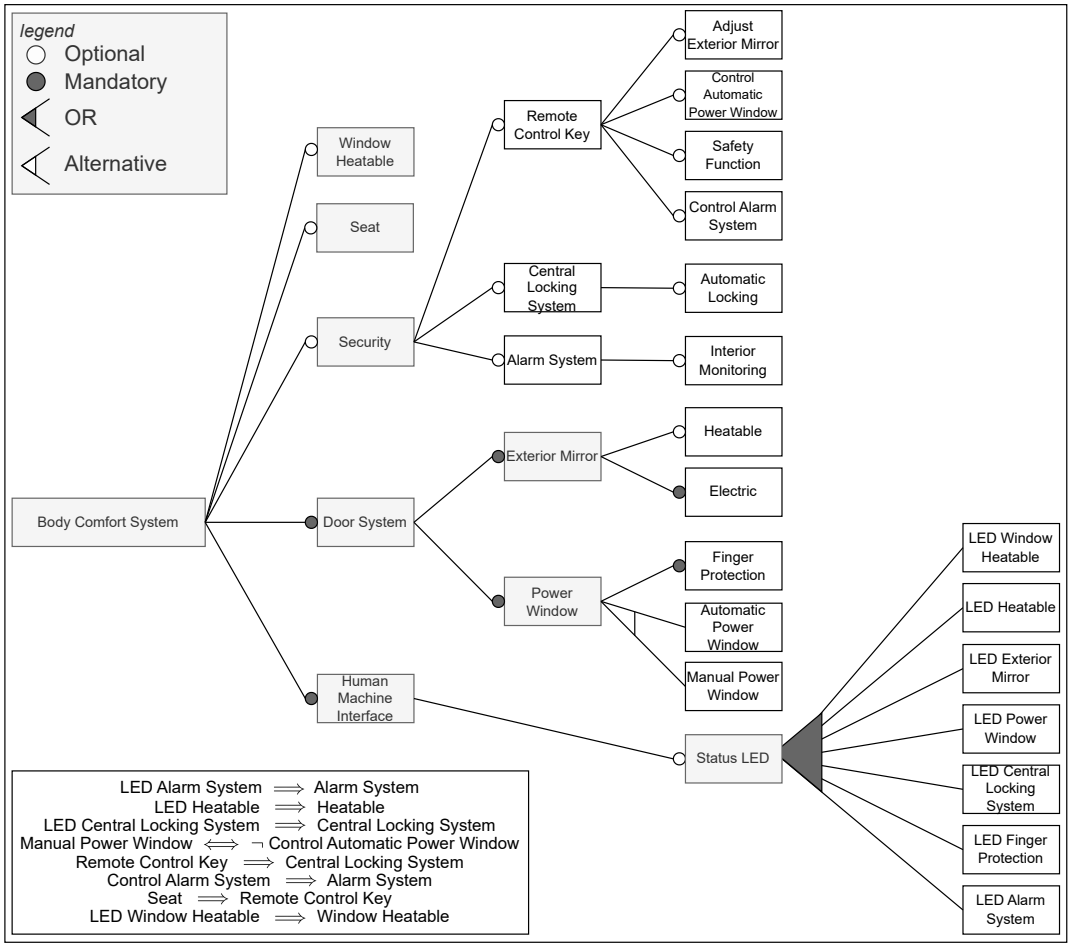

Figure A.2: FM of Version 4

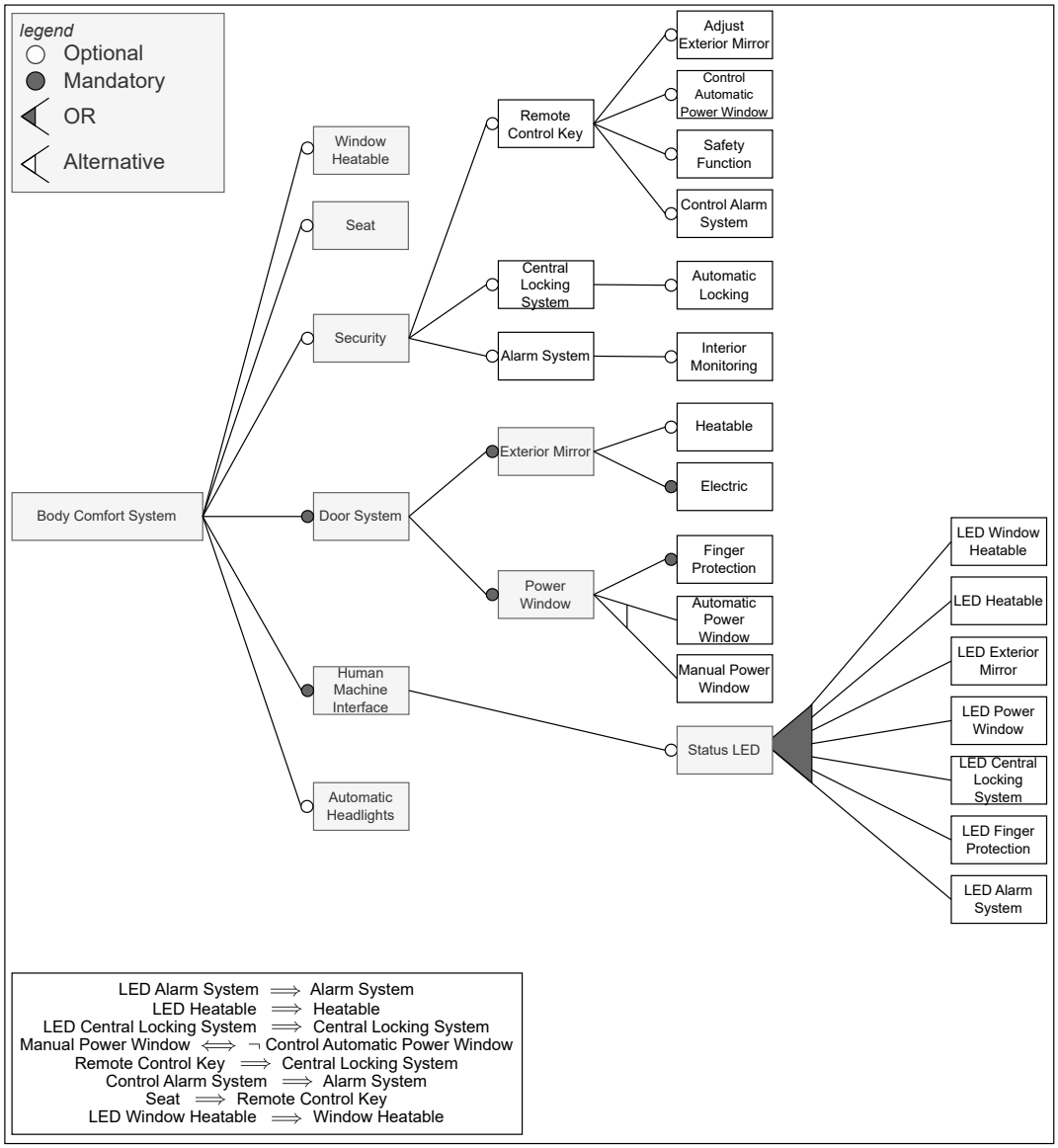

Figure A.3: FM of Version 5

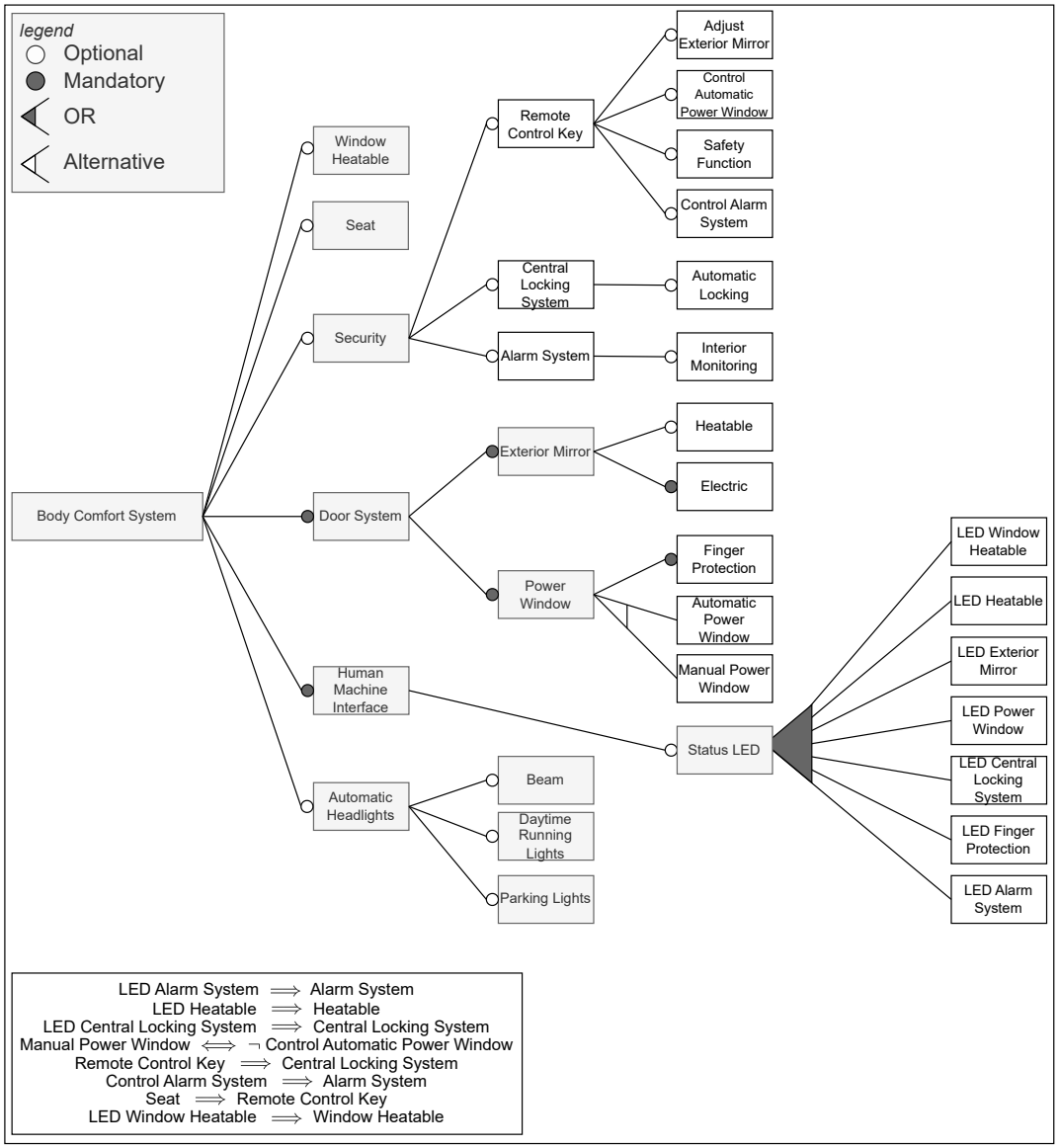

Figure A.4: FM of Version 7

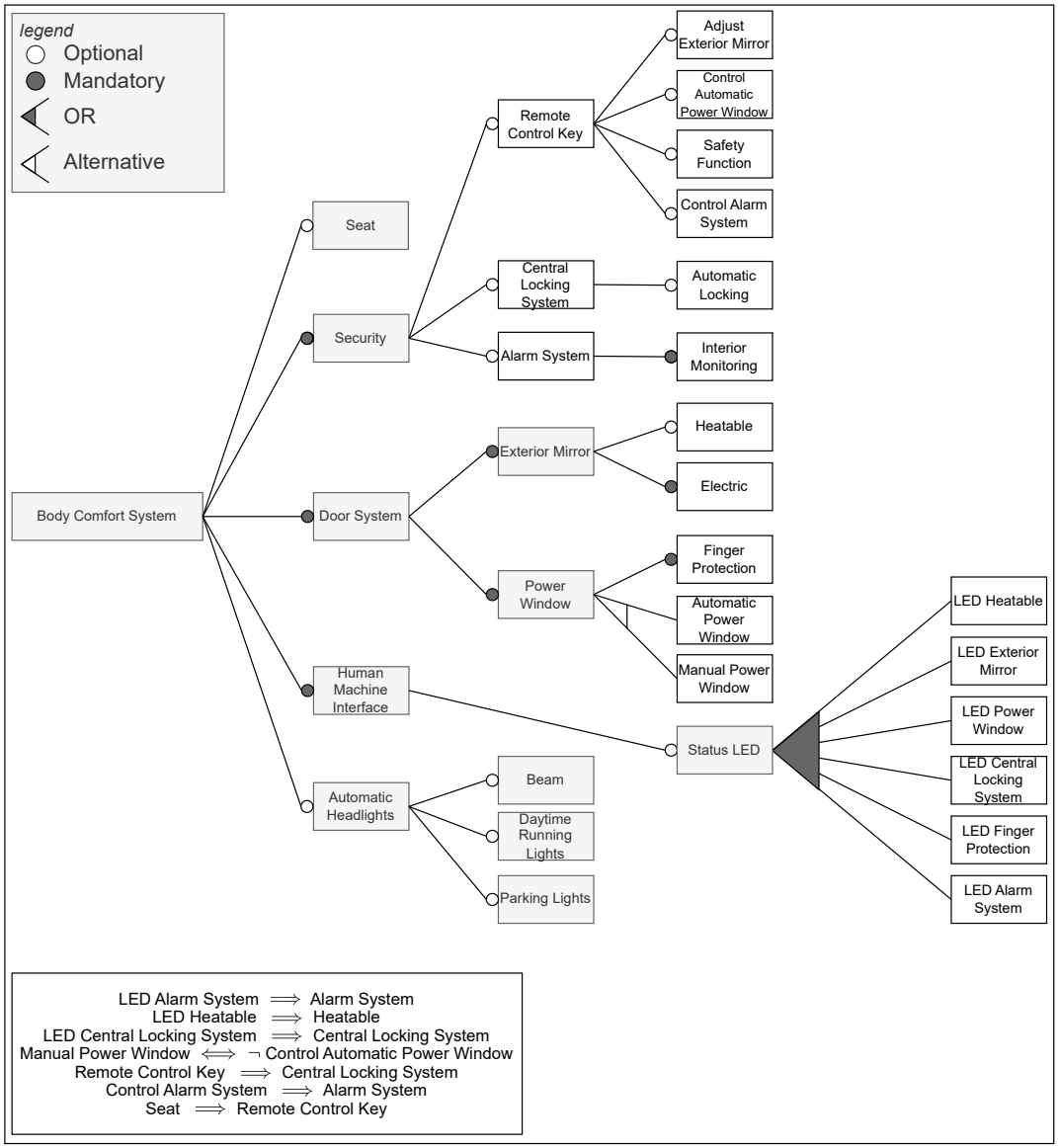

Figure A.5: FM of Version 8