Karlsruhe Institute of Technology

Test, Validation, Analysis

Scalable Product-Based Consistency Analysis of Cyber-Physical Product Lines

Bachelor's Thesis
of

Cornelius Niese

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

Reviewer:	Prof. Dr.-Ing. Ina Schaefer
Advisors:	M.Sc. Philip Ochs
	M.Sc. Tobias Pett

Completion period: 23. January 2025 – 23. May 2025

Zusammenfassung

Konfigurierbare cyber-physische Systeme können als cyber-physische Produktlinien verwaltet werden, um der wachsenden Komplexität und Variabilität moderner industrieller Systeme gerecht zu werden. Produktlinien werden dabei normalerweise in einen Problem- und einen Lösungsraum unterteilt. Der Problemraum umfasst die theoretische Variabilität der Produktlinie in Form von Konfigurationen, während der Lösungsraum Hardware- und Software-Artefakte umfasst, die diese Konfigurationen in Produkte umsetzen. Die Sicherstellung der Konsistenz zwischen dem Problem- und dem Lösungsraum einer Produktlinie ist von wesentlicher Bedeutung, da bestimmte gültige Konfigurationen zu nicht realisierbaren Produkten führen können, weil die modellierte Variabilität im Problemraum und die realisierbare Variabilität im Lösungsraum nicht übereinstimmen. Der aktuelle produktbasierte Ansatz zur Analyse der Konsistenz von cyber-physischen Produktlinien hat Schwierigkeiten, mit zunehmender Systemgröße zu skalieren, da er jede Konfiguration naiv analysiert. Diese Arbeit zielt darauf ab, die Skalierbarkeit des produktbasierten Ansatzes durch drei erweiterte Analysestrategien zu verbessern: Eine Repository-basierte Analysestrategie, die Ergebnisse von zuvor analysierten Konfigurationen wiederverwendet, eine parallele Analysestrategie, die die Konfigurationsanalyse auf Recheneinheiten verteilt, und eine parallele Repository-basierte Analysestrategie, die die Wiederverwendung früherer Analyseergebnisse und Parallelität kombiniert. Wir evaluieren die drei Analysestrategien anhand der Entwicklungsgeschichte einer bekannten Fallstudie aus der Automobilindustrie, wobei der produktbasierte Ansatz als Vergleichsgrundlage dient. Die Ergebnisse zeigen, dass jede Analysestrategie in Bezug auf die Laufzeit in jeder Version der Fallstudie mindestens 4,74 Mal schneller ist als der produktbasierte Ansatz. Die Ergebnisse zeigen jedoch auch, dass keine der Strategien linear mit zunehmender Anzahl von Konfigurationen in einer Produktlinie skaliert.

Abstract

Configurable cyber-physical systems can be managed as cyber-physical product lines to deal with the growing complexity and variability of modern industrial systems. Product lines are usually divided into a problem and a solution space. The problem space comprises the theoretical variability of the product line in the form of configurations while the solution space comprises hardware and software artifacts that realize these configurations into products. Ensuring the consistency between the problem and solution space of a product lines is essential, as certain valid configurations may lead to non-realizable products, because of a mismatch between the modeled variability in the problem space and the realizable variability in the solution space. The state-of-the-art product-based approach for analyzing the consistency of cyber-physical product lines struggles to scale with increasing system size as it analyzes each configuration naively. This thesis aims to improve the scalability of the product-based approach through three enhanced analysis strategies: A repository-based analysis strategy that reuses results from previously analyzed configurations, a parallel analysis strategy that distributes configuration analysis across computation units, and a parallel repository-based analysis strategy that combines reuse of previous analysis results and parallelism. We evaluate the three analysis strategies using the evolution history of a well-known case study from the automotive industry using the product-based approach as a baseline for comparison. The results show that, in terms of run-time, each analysis strategy is at least 4.74 times faster than the product-based approach in each version of the case study. However, the results also show that none of the strategies scales linearly with increasing numbers of configurations in a product line.

Contents

List of Figures

List of Tables

Acronyms

AS	Alarm System
BCS	Body Comfort System
CLS	Central Locking System
CV	Coefficient of Variation
CPPL	Cyber-Physical Product Line
CPS	Cyber-Physical System
CPU	Central Processing Unit
CSV	Comma-Separated Values
EM	Exterior Mirror
FD	Feature Diagram
FIFO	First-in-first-out
FM	Feature Model
FP	Finger Protection
KB	Kilobyte
PCP	Producer-Consumer Problem
PL	Product Line
PW	Power Window
RAM	Random-Access Memory
RAP	Resource Allocation Problem
RCK	Remote Control Key
RQ	Research Question
SD	Standard Deviation
UCM	Unified Conceptual Model
XML	Extensible Markup Language

1. Introduction

Cyber-Physical Systems (CPSs) describe the combination of software and hardware components that have become indispensable for modern industry [7]. The interaction of these components, however, leads to increasing complexity due to high product variability. To address this problem, CPSs can be managed as Cyber-Physical Product Lines (CPPLs) where products share a set of commonalities while customizable with variable functionality [3] [14]. For example, an infotainment system in a car could be customizable through the functionality of a navigation system and a multimedia system.

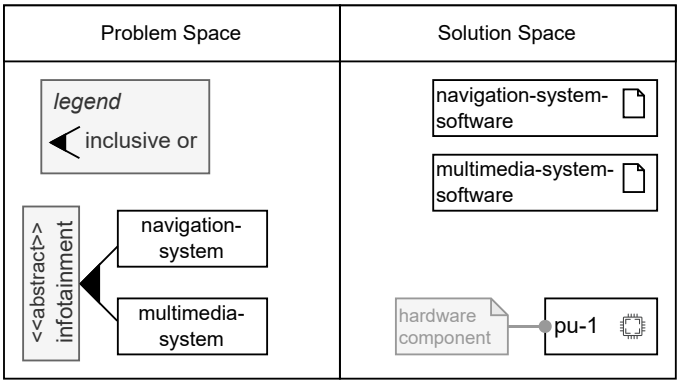

Figure 1.1: Problem and Solution Space of the Infotainment System of a Car Used as a Running Example (adapted from Ochs et al.[15]).

CPPLs are usually divided into a problem space and a solution space [3]. The problem space consists of features and their dependencies, where a feature represents user-visible functionality [3]. The features and their dependencies are usually modeled and managed in a Feature Model (FM) [3]. A set of selected and deselected features is called a configuration which becomes valid when all dependencies of the FM are fulfilled [13]. In our running example of a car product line illustrated in Figure 1.1 an example FM modeling an infotainment system is shown in the problem space. If a customer selects the feature *infotainment* in this FM, they must also

select either the feature *navigation-system* or the feature *multimedia-system* or both features at the same time, because they are modeled in an or-group.

The solution space of a CPPL consists of reusable realization artifacts, such as software or hardware components implementing the modeled functionality (i.e., features) from the problem space [24]. With these a valid configuration can be realized into a product by a selection of the corresponding solution space artifacts. In the solution space of the infotainment system in Figure 1.1 the realization artifacts are given through the software components *navigation-system-software*, *multimedia-system-software* and the hardware component *processing unit 1 (pu-1)*.

It is, however, often unclear whether a valid configuration from the problem space functions after realization in the solution space due to incompatible solution space artifacts [15]. For example, the hardware component pu-1 of the infotainment system in Figure 1.1 could meet the individual resource demands with either navigation-system or the multimedia-system selected but not when both are selected at the same time. With such a mismatch between the modeled variability in the problem space and the realizable variability in the solution space, the CPPL becomes inconsistent. To assess whether a CPPL is inconsistent, problem and solution space must therefore be analyzed together.

The Unified Conceptual Model (UCM) of Wittler et al. [24] offers a generic metamodel of the problem and solution space and is also able to describe the interplay between software and hardware. It does so by modeling solution space artifacts as components where software components demand resources and hardware components provide them. Based on the UCM, Ochs et al. [15] present a method for analyzing the realizability of valid configurations and the consistency of CPPLs. However, as their product-based approach analyzes each valid configuration naively, it does not scale well in larger CPPLs with a large number of configurations [23].

Goal of this Thesis

In this bachelor thesis, we take on the challenge of scaling *product-based* consistency analysis between problem and solution space for large CPPLs by investigating three analysis strategies: (1) *Repository-Based* consistency analysis, (2) *Parallel* consistency analysis and (3) *Parallel Repository-Based* consistency analysis. All analysis strategies are based on the product-based analysis of a configuration's realizability as proposed in the existing approach of Ochs et al. [15], but aim to improve the scalability of the overall consistency analysis in detail as follows.

1. *Repository-Based*: We reuse the results of previously analyzed valid configurations. Therefore, we hold and update a repository consisting of configurations already analyzed with their analysis results while iteratively analyzing the consistency of the CPPL. For example, if configuration $\mathcal{C}_\mathcal{A}$ has the same solution space artifacts as the configuration previously analyzed $\mathcal{C}_\mathcal{B}$, we reuse the analysis result of $\mathcal{C}_\mathcal{A}$ so that redundant computations are avoided.

2. *Parallel*: Instead of sequentially analyzing the realizability of each configuration as done by Ochs et al. [15] we want to parallelize this approach. We will therefore parallelize over the set of valid configurations so that each computation unit can iteratively analyze valid configurations for realizability.

3. *Parallel Repository-Based*: In addition to strategies (1) and (2), we envision the combination of them to further improve the scalability of the consistency analysis. For this, we parallelize our repository-based approach by keeping and updating a repository in each computation unit so that redundant computations in each unit are reduced.

Structure of the Thesis

The structure of this thesis is organized as follows. In Chapter 2, we introduce the basic concepts necessary for our design. The design of our three analysis strategies (1) repository-based, (2) parallel, and (3) parallel repository-based is introduced in Chapter 3 and their implementation in Chapter 4. In Chapter 5, we evaluate and discuss our strategies. In Chapter 6 an overview of related work is given. Chapter 7 presents a conclusion and an outlook on future work and summarizes the results of this thesis.

2. Basics

In this chapter, the foundational concepts of this thesis are presented. Section 2.1 describes Product Lines (PLs) as they are the core concept on which this thesis is based. In this section, we further describe concepts such as PL engineering, the Unified Conceptual Model (UCM) and consistency analysis, as they are essential for our strategies. In Section 2.2 the producer consumer problem is introduced that is needed to describe the concept of our parallel-based analysis strategies. Finally, Section 2.3 presents a running example.

2.1 Software Product Lines

For software PLs Apel et al. [3] explains that they allow for both individual solutions and mass customization of (software) systems. This is achieved by grouping together a set of products (i.e. software systems) that share a set of commonalities and a set of variable functionality [17].

According to Pohl et al. [17] *commonalities* describe the functionality that is part of every software of a software PL. In contrast to that, *variable* functionalities can be part of different software systems in variable combinations. Reusable artifacts are used to realize these functionalities into actual software systems.

The user-visible functionality of a software PL is called *feature* [3]. The modeling of features and their dependencies is usually achieved using a Feature Model (FM) described in the following Section 2.1.1.

2.1.1 Feature Model

Features and their dependencies are usually described in a FM [3]. FMs define a hierarchical tree-like structure where the nodes represent features. Tree-like because sometimes the tree structure is broken by cross-tree constraints which define dependencies between features located in different parts of the hierarchy. These cross-tree constraints can be added with additional edges between features. Hierarchical dependencies between features can be defined by grouping them into alternative or

or-groups, or by designating them as mandatory or optional. A selection and de-selection of features in a FM is called a configuration. If a selection and de-selection of features meet the rules defined by the FM the configuration becomes valid.

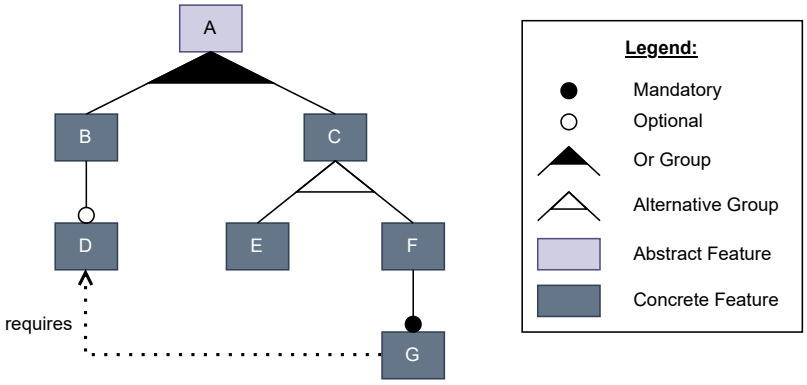

Figure 2.1: Example Feature Diagram.

Figure 2.1 illustrates a FM as a graphical Feature Diagram (FD) in which all hierarchical relationships are displayed. An or-group is illustrated below feature `A` where feature `B` or feature `C` or both features must be selected if their parent feature is selected. Below feature `C` the features `E` and `F` define an alternative-group where exactly one of the features must be selected if their parent feature is selected. Optional features such as `D` can be selected if their parent feature (e.g., `B`) is selected. In contrast, mandatory features such as `G` must be selected if their parent feature (e.g., `F`) is selected. Not fitting into the hierarchical structure, a cross-tree constraint is defined between features `D` and `G`.

Slicing of Feature Models

Slicing describes a technique used to extract a subset of a FM that contains fewer features, but preserves their dependencies [1]. To slice a FM, the features of interest have to be selected and form the *slicing criterion*. In the next step, all features related to the slicing criterion are identified - these are, for example, parent and child features in the hierarchy. In addition, all cross-tree constraints that involve features of the slicing criterion are identified. With these selected features, a *slice* is created that contains all selected features and their (hierarchical) dependencies. The slice is a new FM that contains only valid configurations that are also valid in the original model.

2.1.2 Software Product Line Engineering

For software PLs Apel et al. [3] describe that PL engineering has to focus on handling variability and allowing reuse of implementation artifacts. Addressing this, they define two dimensions that are crucial for PL engineering. These dimensions are called *problem* and *solution space*.

The problem space comprises the variability at the feature level that characterizes different configuration options within a PL. The problem space is usually formalized using FMs to model valid feature selections as configurations.

In contrast, the solution space refers to the architecture and implementation artifacts used to realize valid configurations defined in the problem space as product variants. Thus, it defines how the required variability is implemented and allows reuse of implementation artifacts by mapping features to corresponding implementation strategies.

2.1.3 Unified Conceptual Model

The UCM was first developed by Ananieva et al. [2] and represents a meta-model that unifies the problem and solution space of a software PL. Wittler et al. [24] and Ochs et al. [15] further refined the solution space of the UCM as shown in Figure 2.2, making it capable of describing Cyber-Physical Product Lines (CPPLs) at the resource level.

In the problem space, the UCM models variability through a FM. In the solution space, implementation artifacts are modeled through reusable software and hardware components which map to the defined feature options.

In detail, software components in the solution space demand resources, and hardware components provide them. A resource has a resource type that, in turn, has a unit and is specified by the three properties *isAdditive*, *isExclusive* and *boundary*. *isAdditive* defines whether multiple resources of the same resource type can be summed up. *isExclusive* defines whether a resource can be shared among multiple software components or must be assigned to only one. *boundary* is divided into the types `LOWER`, `UPPER` and `EXACT`. In case of a `LOWER` boundary type, the provided resource must match at least the value of the demanded resource. In case of an `UPPER` boundary type, the provided resource must match at most the value of the demanded resource. Finally, in case of an `EXACT` boundary type, the provided resource must match the exact value of the demanded resource.

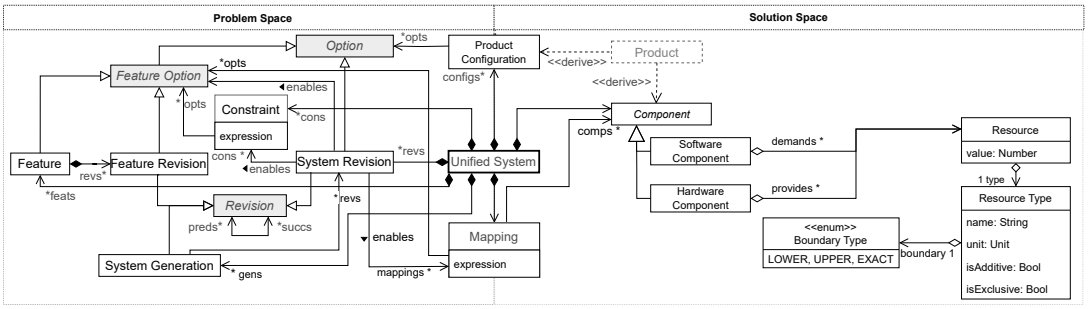

Figure 2.2: UCM by Ochs et al. [15].

2.1.4 Consistency between Problem and Solution Space of CPPLs

Product-based Analysis

With increased usage of PLs in general arose the need for analysis methods that are capable not only of analyzing a single product in terms of correctness and reliability, but also of analyzing a whole PL [23]. A common technique used for such needs is a *product-based* analysis.

A product-based analysis focuses on analyzing each possible product of a PL with standard analysis techniques. In the problem space, this could be analyzing each possible configuration modeled in a FM for validity. In the solution space, this could be analyzing if each valid configuration from the problem space is also realizable in the solution space.

Product-based Consistency Analysis

Ochs et al. [15] use a product-based *consistency analysis* to analyze whether there is a mismatch between the variability modeled in the problem space and the variability realizable in the solution space of CPPLs. If such a mismatch exists, they call a CPPL *inconsistent*. The whole consistency analysis is based on using the UCM as a meta-model to describe CPPLs.

To assess whether a valid configuration derived from the problem space of a CPPL is realizable in the solution space, a Resource Allocation Problem (RAP) must be solved between all defined software components $sw_i \in \mathcal{SW}$ and all defined hardware components $hw_j \in \mathcal{HW}$ of the PL. The reason is that each software component sw_i defines resource demands rd_k^i in the context of each resource type $r_k \in \mathcal{R}$. These resource demands must be satisfied by the hardware components hw_j that in turn define resource provisionings rp_k^j for each resource type r_k. To solve the RAP, each software component must be assigned to a single hardware component. If the resource demands of the software components cannot be satisfied by the resource provisionings of the hardware components, meaning that the RAP is not solvable, the derived configuration from the problem space is not realizable and the CPPL becomes inconsistent.

The resource demands of the software components are derived from the features selected in a configuration - this process is called *mapping* of resource demands. Therefore, each feature defines a resource demand in the context of a resource type and a software component. If in a configuration, multiple features define resource demands in context of the same resource type r_k and the same software component sw_i the properties *isAdditive* and *boundary* define how these demands are reduced to a single resource demand rd_k^i of the software component sw_i in context of the resource type r_k:

1. *isAdditive*: A sum is build over all resource demands of the features.

2. *boundary*: If the type is `LOWER`, the maximum resource demand of the features is taken. If the type is `HIGHER`, the minimum resource demand of the features is taken. In any other case, a random value is selected from the resource demands of the features given.

For our repository-based analysis strategy, we try to avoid analyzing the realizability of each configuration in a CPPL by reusing the resource demands of the software components defined for a configuration.

2.2 Producer-Consumer Problem

In parallel programming, communication between multiple processes introduces many challenges, one of them being the Producer-Consumer Problem (PCP) that was first introduced by Dijkstra [5]. The problem is based on two cyclic processes that communicate with each other through a First-in-first-out (FIFO) memory (i.e., buffer). Going through their cycles, the producer produces information and adds it to the buffer, while the consumer process takes information from the buffer and processes it. Access to the buffer should not be allowed to a producer and a consumer at the same time. In addition, a producer should only be able to add information to the buffer if empty positions are available, while the consumer should only be able to take information from the buffer if it is not empty. To solve the problem, Dijkstra introduced synchronization primitives that secure the buffer with the described access restrictions. We provide a description of the problem, as we use the concept of producers and consumers for the design of our parallel-based analysis strategies.

2.3 Running Example

In this section, we introduce the running example used in the remainder of the thesis to illustrate the functionality of our strategies. The running example is adapted from Ochs et al. [15], illustrated in Figure 2.3 and describes a CPPL for the infotainment system of a car. Figure 2.3 shows the division into problem and solution space, where a feature model is shown in the problem space and the artifacts `multimedia-system-software` (sw_0), `navigation-system-software` (sw_1), and `pu-1` (processing-unit-1) (hw_0) in the solution space. The feature model consists of the abstract feature *infotainment* and the features *multimedia-system* and *navigation-system* that can be selected via an or-group.

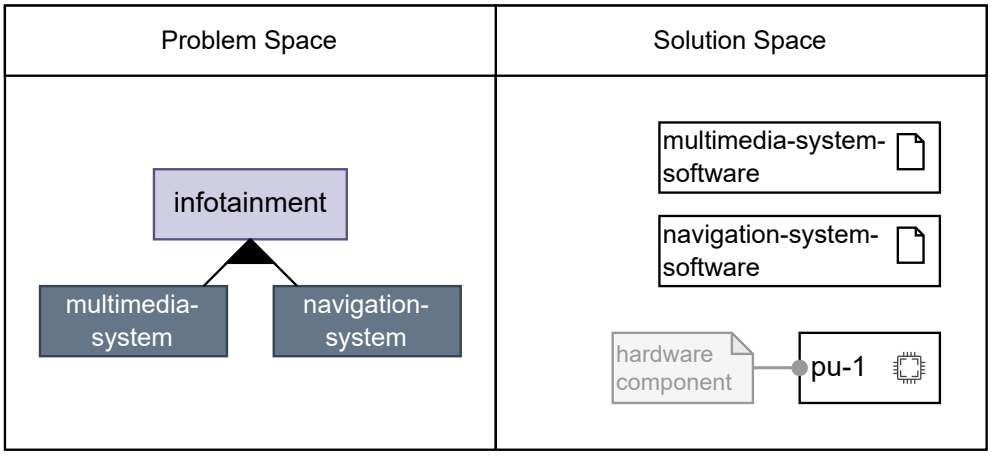

Figure 2.3: Problem and Solution Space of the Infotainment System of a Car Used as a Running Example (adapted from Ochs et al. [15]).

With this feature model, we have three possible valid configurations:
$\mathcal{C}_{\mathcal{A}} = \{$infotainment, multimedia-system$\}$,

$\mathcal{C}_\mathcal{B}$ = {infotainment, navigation-system} and
$\mathcal{C}_\mathcal{C}$ = {infotainment, multimedia-system, navigation-system}.

We define the two resource types Random-Access Memory (RAM) (r_0) and Response Time (r_1) in Table 2.1. RAM is additive, not exclusive, and has a lower boundary, which means that the provided resource must match at least the value of the demanded resource. Response Time is not additive, not exclusive and has an upper boundary, which means that the provided resource must match at most the value of the resource demanded.

k	name	unit	isAdditive	isExclusive	boundary
0	RAM	GB	True	False	LOWER
1	Response Time	ms	False	False	UPPER

Table 2.1: Resource Types of the Running Example

The resource types are provided by `pu-1` as illustrated in Table 2.2. `pu-1` provides 2 GBs of RAM and a response time of 10 ms.

Hardware Component (hw_j)	Provisions (rp_k^j)
`processing-unit-1` (hw_0)	$rp_0^0 = 2$ $rp_1^0 = 10$

Table 2.2: Resource Provisions of the Running Example.

The resource demands are illustrated in Table 2.3. The multimedia-system feature demands 2 GB RAM and a response time of at most 10 ms, which is addressed by the `multimedia-system-software`. The navigation-system feature has the same resource demands but addresses the `navigation-system-software` instead.

Feature	Software Components sw_i	Resource Demands (rd_k^i)
multimedia-system	sw_0	$rd_0^0 = 2$ $rd_1^0 = 10$
navigation-system	sw_1	$rd_0^1 = 2$ $rd_1^1 = 10$

Table 2.3: Resource Demands of the Running Example.

With this setting, only configurations $\mathcal{C}_\mathcal{A}$ and $\mathcal{C}_\mathcal{B}$, but not configuration $\mathcal{C}_\mathcal{C}$ are realizable. With configuration $\mathcal{C}_\mathcal{A}$ the resource demands of `multimedia-system-software` can be fulfilled by `pu-1` as can the same resource demands of configuration $\mathcal{C}_\mathcal{B}$ with the `navigation-system-software`. With configuration $\mathcal{C}_\mathcal{C}$ the resource demands of `multimedia-system-software` and `navigation-system-software` cannot be fulfilled by `pu-1`. The reason is the additivity of the RAM. The resource demands $rd_0^0 = 2$ and $rd_0^1 = 2$ are summed up to 4 which is higher than the RAM provided by $rp_0^0 = 2$. Thus, our CPPL consists of two valid and realizable configurations and one valid and non-realizable.

3. Design

In this chapter, the analysis strategies of the thesis are introduced. In Section 3.1, we define our repository-based analysis strategy trying to avoid redundant computations by reusing existing consistency analysis results. Section 3.2 introduces our parallel analysis strategy that is able to simultaneously analyze the realizability of multiple configurations. The combination of both analysis strategies, with the aim of avoiding both redundant computations by reusing existing consistency analysis results and performing configuration analysis in parallel, is introduced as a parallel repository-based analysis strategy in Section 3.3.

3.1 Repository-Based Analysis Strategy

Before we begin with the technical explanation, we clarify two key terms:

- *Resource demands of a software component*: The demand that this software component has on each resource type of a CPPL in a configuration.
- *Resource demands of a configuration*: The collection of resource demands assigned to each software component of a CPPL in a configuration.

The main idea behind a repository-based analysis is to reduce the number of realizability analyses that must be performed during the consistency analysis of a CPPL. We want to reduce the number of these realizability analyses as they consist of setting up and solving a RAP between software and hardware components in the solution space, which is very expensive in terms of run-time. To reduce the number of realizability analyses, we store the resource demands of analyzed, realizable configurations. When analyzing a new configuration, we check whether the same or a subset of its resource demands has already been successfully analyzed in the past. If so, we can reuse that result and skip the realizability analysis. We stop the whole consistency analysis as soon as an inconsistency is found.

For the repository-based strategy to work, we design a *configuration repository*, which is a data structure that enables fast access to the resource demands of the

analyzed configurations. Fast access to stored resource demands is necessary so that lookup time does not outweigh the benefit of skipping realizability analysis. We introduce this repository in Section 3.1.1 and describe how it is used in a repository-based consistency analysis in Section 3.1.2. In Section 3.1.3, we illustrate the repository-based consistency analysis using our running example.

3.1.1 Configuration Repository

The configuration repository is a *key-value store*. Each *key* represents the resource demands of a software component. The associated *value* is a collection of sets, where each set represents the resource demands of a realizable configuration that includes the component represented by the key.

Key Construction

To construct a key, we represent the resource demands of a software component as a string concatenation, where:

- Each resource type is represented in fixed order; this order could be ensured by assigning indices to the resource types and concatenating the resource demands according to those indices.
- A missing demand for a resource type is marked by a placeholder symbol (e.g., $\perp$).
- Values for different resource types are separated by a special delimiter (e.g., $*$) to prevent ambiguity.

This ensures that each key has the same structure for different software components and configurations.

Values: Sets of Configuration Demands

Each value associated with a key is a set that contains *sets of configuration demands*. Each set of configuration demands represents a complete configuration and consists of tuples:

- The first element of a tuple is a string of the resource demands of a software component constructed in the same way as keys.
- The second element is the number of software components in that configuration with exactly those demands.

This representation avoids duplication: If multiple software components have the same resource demands in a configuration, they are represented in a single tuple.

The whole structure lets us quickly check whether a new configuration includes only resource demands that have already been analyzed for realizability for another configuration.

3.1.2 Repository-Based Consistency Analysis

This section describes the procedure on how to use the configuration repository defined in Section 3.1.1 to perform a repository-based consistency analysis. We denote the configuration to be analyzed as $\mathcal{C}_{\mathcal{A}}$.

1. *Construct Keys and Set of Configuration Demands*
   - For each software component in the CPPL, construct a key based on its resource demands in $\mathcal{C}_{\mathcal{A}}$.
   - Build a set of configuration demands by grouping identical resource demands of software components.
2. *Check Key Presence in Configuration Repository*
   - If any key does **not** exist in the repository, then the resource demands of at least one software component in the resource demands of $\mathcal{C}_{\mathcal{A}}$ was never analyzed in any configuration before. We can draw this conclusion because if a key does not exist in the repository then the resource demands that were used to construct this key cannot be part of any configuration demand set stored in the repository. In this case, we have to perform a realizability analysis for $\mathcal{C}_{\mathcal{A}}$.
   - If all keys exist in the repository, retrieve the repository values referenced by the keys.
3. *Shared Set Identification*: Search for at least one set of configuration demands that appears in all the referenced repository values.
   - If we do **not** find at least one set that appears in each of the referenced repository values, we can draw the conclusion that we have never analyzed a configuration before in which there were software components with the same resource demands as in $\mathcal{C}_{\mathcal{A}}$. In this case, we have to perform a realizability analysis for $\mathcal{C}_{\mathcal{A}}$.
   - If we do find at least one set that appears in each of the referenced repository values, we can draw the conclusion that we have analyzed at least one configuration in which there were software components with the same resource demands as in configuration $\mathcal{C}_{\mathcal{A}}$.
4. *Check for Demand Reuse*
   - **Subset case**: The set of configuration demands of $\mathcal{C}_{\mathcal{A}}$ is a subset of at least one of the shared sets. In this case the resource demands of $\mathcal{C}_{\mathcal{A}}$ were already analyzed for a previous configuration and we can skip the realizability analysis for $\mathcal{C}_{\mathcal{A}}$.
   - **Count comparison case**: The set of configuration demands for $\mathcal{C}_{\mathcal{A}}$ is **not** a subset of at least one the shared sets. In this case, the counts (second entries) of each tuple in the set of configuration demands of $\mathcal{C}_{\mathcal{A}}$ are compared with those of the shared sets. If the resource demands of each software component in $\mathcal{C}_{\mathcal{A}}$ appear at least as often in any of the shared sets, then $\mathcal{C}_{\mathcal{A}}$ is covered, and we can skip the realizability analysis.

5. *Update Configuration Repository*: These steps must only be executed if a realizability analysis must be performed for $\mathcal{C}_{\mathcal{A}}$.

   - Create a new entry in the configuration repository for any new key.
   - Add the set of configuration demands of $\mathcal{C}_{\mathcal{A}}$ to each repository value associated with a key that is part of a tuple in $\mathcal{C}_{\mathcal{A}}$.

Run-Time Estimation

We define the following variables:

- Let $|\mathcal{SW}|$ be the total number of software components in a CPPL.
- Let $|\mathcal{R}|$ be the total number of resource types in a CPPL.
- Let M be the average number of analyzed configurations stored per entry in the configuration repository.

We orient the run-time estimation along the steps of the repository-based consistency analysis procedure and assume the case that no realizability analysis has to be performed.

1. *Construct Keys and Set of Configuration Demands*
   - Each key is constructed in $\mathrm{O}(|\mathcal{R}|)$, which results in $\mathrm{O}(|\mathcal{SW}| \cdot |\mathcal{R}|)$. The construction of the set of configuration demands takes $\mathrm{O}(|\mathcal{SW}|)$.
2. *Check Key Presence in Configuration Repository*
   - Checking the existence of a key in the repository takes $\mathrm{O}(1)$, which results in $\mathrm{O}(|\mathcal{SW}|)$ for all keys.
3. *Shared Set Identification*
   - For each of the $|\mathcal{SW}|$ referenced repository values, compare up to M sets. This results in $\mathrm{O}(M \cdot |\mathcal{SW}|)$.
4. *Check for Demand Reuse*
   - Each comparison checks tuples between two sets of size at most $|\mathcal{SW}|$. This results in $\mathrm{O}(M \cdot |\mathcal{SW}|)$.

The dominant cost is in comparing demand sets, so the total runtime is: $\mathrm{O}(M \cdot |\mathcal{SW}|)$. As $|\mathcal{SW}|$ is constant in a CPPL this can be further reduced to $\mathrm{O}(M)$. Thus, the repository-based analysis strategy has linear complexity in M, which is the number of stored configurations per repository entry, making it significantly faster compared to the exponential cost of a realizability analysis [15].

3.1.3 Example

This section aims to give an example of repository-based consistency analysis along our running example described in Section 2.3. We assume that we look at the configurations in order $\mathcal{C}_\mathcal{A}$, $\mathcal{C}_\mathcal{B}$ and $\mathcal{C}_\mathcal{C}$.

1. Starting with an empty configuration repository, the first configuration we examine is $\mathcal{C}_\mathcal{A}$ = {infotainment, multimedia-system}.

   In configuration $\mathcal{C}_\mathcal{A}$ the software component sw_0 has the resource demands $rd_0^0 = 2$ and $rd_1^0 = 10$. As the software component sw_1 does not have resource demands in this configuration, we denote them by $rd_0^1 = \perp$ and $rd_1^1 = \perp$. With these demands, we construct the two keys "$2 * 10$" and " $\perp * \perp$ " and the set of configuration demands {("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}.

   As the configuration repository is empty, we find no entries for both keys. Thus, we create entries for both keys and add the set of configuration demands to both of them. Finally, we analyze configuration $\mathcal{C}_\mathcal{A}$ for realizability and find that it is.

   The configuration repository now consists of two entries. One with "$2 * 10$" as a key and {{("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}} stored as a value and the other with " $\perp * \perp$ " as a key and {{("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}} stored as a value.

2. The second configuration we examine is $\mathcal{C}_\mathcal{B}$ = {infotainment, multimedia-system}.

   The resource demands of the software components sw_0 and sw_1 in $\mathcal{C}_\mathcal{B}$ are: $rd_0^0 = \perp$, $rd_1^0 = \perp$ and $rd_0^1 = 2$, $rd_1^1 = 10$. With this, we construct the two keys keys "$2 * 10$" and " $\perp * \perp$ " and the set of configuration demands {("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}.

   In the repository, we find that there is an entry for each of our keys and retrieve the two identical sets {{("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}} and {{("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}}. We find a shared set in {("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}, which appears in both values. With this we are now in the **subset case** as our current set of configuration demands {("$2 * 10$", 1), (" $\perp * \perp$ ", 1)} is equal to the shared set. Thus, we do not have to analyze the configuration $\mathcal{C}_\mathcal{B}$ for realizability and do not have to update the configuration repository.

3. The third and last configurations to examine is $\mathcal{C}_\mathcal{C}$ = {infotainment, infotainment-system, multimedia-system}.

   The resource demands of the software components sw_0 and sw_1 in $\mathcal{C}_\mathcal{C}$ are: $rd_0^0 = 2$, $rd_1^0 = 10$ and $rd_0^1 = 2$, $rd_1^1 = 10$. With these demands, we construct the two identical keys "$2*10$" and "$2*10$" and the set of configuration demands {("$2 * 10$", 2)}

   In the configuration repository, we find that there is an entry for the key "$2 * 10$" and retrieve the set {{("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}}. Our shared set is {("$2 * 10$", 1), (" $\perp * \perp$ ", 1)}, which is the only set we found in this entry. We are now in the **count comparison case** as our current set of configuration demands {("$2 * 10$", 2)} is not a subset of the shared set. When comparing the counts, we find that "$2 * 10$" appears only once in the shared set but twice

in our set of configuration demands. Thus, we add our set of configuration demands to the entry referred to by the key "$2 * 10$". Finally, we analyze configuration $\mathcal{C}$ for realizability and find that it is not. As we have found an inconsistency, we stop the consistency analysis.

In the end, the configuration repository consists of two entries. One with "$2 * 10$" as a key and $\{\{(\text{"}2 * 10\text{"}, 1), (\text{" } \bot * \bot \text{ "}, 1)\}, \{(\text{"}2 * 10\text{"}, 2)\}\}$ stored as a value and the other with " $\bot * \bot$ " as a key and $\{\{(\text{"}2 * 10\text{"}, 1), (\text{" } \bot * \bot \text{ "}, 1)\}\}$ stored as a value.

3.2 Parallel Analysis Strategy

In this analysis strategy, we use the naive consistency analysis of Ochs et al. [15] but parallelize it over the set of valid configurations. We do so using a producer and multiple consumers, where the producer is responsible for producing valid configurations $\mathcal{C}$ and the consumers analyze these configurations for realizability in parallel. A detailed description of this producer-consumer concept and its origin is given in Section 3.2.1.

3.2.1 Producer-Consumer Concept

The main challenge in parallelizing the naive consistency analysis of Ochs et al. [15] lies in distributing the valid configurations of a CPPL across multiple processes. If each process independently draws valid configurations from a given FM and analyses them for realizability, we must ensure that each configuration is only processed once. This requires that each process knows which valid configurations have already been drawn from the FM by other processes. As a result, all the configurations analyzed would need to be stored in a shared memory, and each new configuration would have to be checked against this shared memory to prevent duplicates. Such a mechanism could introduce redundancy and overhead as each configuration would require a lookup before analysis.

To avoid this, we introduce a producer process that is dedicated to drawing valid configurations from the FM, as in the naive consistency analysis of Ochs et al. [15]. The producer shares these configurations via a shared buffer with consumer processes that analyze these configurations for realizability in parallel. If no more valid configurations are available or if one consumer finds a configuration that is not realizable, we stop the consistency analysis. With this concept, the maximum number of consumer processes that can be initialized is set to $t - 1$, where t is the number of threads available for a machine. To protect the buffer against synchronous access from producer and consumer processes, we assume the synchronization primitives of Dijkstra [5] described in Section 2.2.

3.2.2 Parallel Consistency Analysis

This subsection describes the procedure on how we integrate the producer-consumer concept in the naive consistency analysis of Ochs et al. [15]. The whole procedure is illustrated in an activity diagram in Figure 3.1.

In addition to the shared buffer, the producer and consumers communicate via a shared Boolean value *consistent*. This value indicates the consistency of a CPPL

as long as it is set to **True**. When a consumer process finds a configuration that is not realizable, it sets the shared value to **False** stopping the consistency analysis for all processes.

The following paragraphs describe the behavior of the producer and consumer in detail, aligned with the activities shown in Figure 3.1.

Producer

As long as *consistent* remains **True**, the producer starts by drawing a valid configuration from the FM of the current CPPL to analyze (activity 2.1). If a configuration is given, it is added to the buffer (activity 2.2). The producer terminates either when no more valid configurations are available or when *consistent* is set to **False** by a consumer.

Consumers

As long as *consistent* remains **True**, each consumer repeatedly takes a configuration from the buffer (activity 1.1) and analyzes it for realizability (activity 1.2). If the configuration is realizable, the analysis continues. If not, an incosistency has been found and *consistent* is set to **False** which stops the analysis for all processes (activity 1.3). A consumer also terminates when no more configurations are available from the producer.

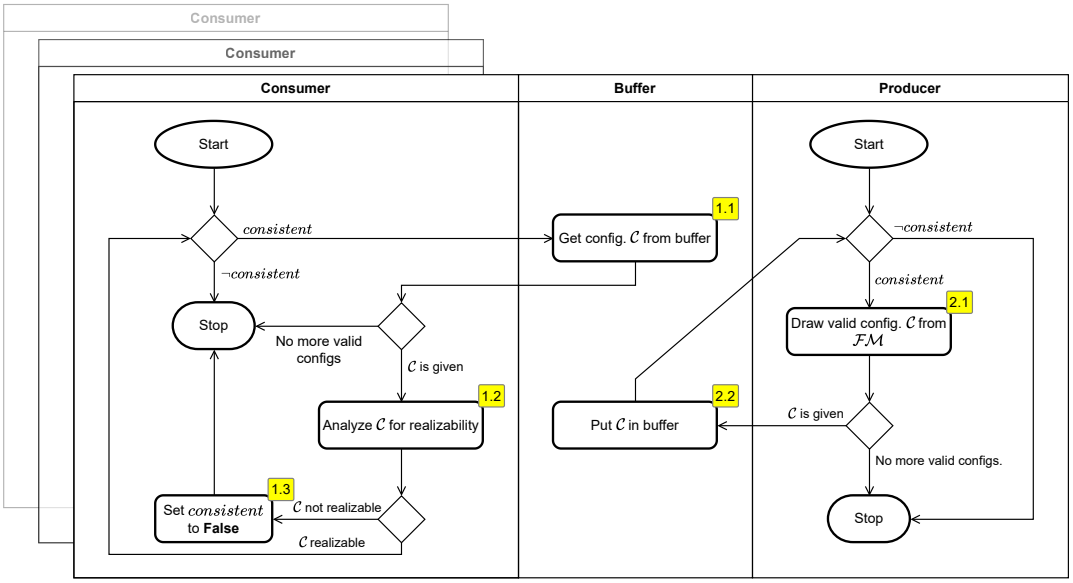

Figure 3.1: Activity Diagram of a Parallel Consistency Analysis

With the producer-consumer concept, we enable a parallel realizability analysis and distribute the workload across the available processes. Together with the repository-based analysis strategy described in Section 3.1, it forms the foundation for the combined analysis strategy described in the following Section 3.3.

3.3 Parallel Repository-Based Analysis Strategy

This analysis strategy unifies the concepts developed in Section 3.1 and Section 3.2. In this analysis strategy, each consumer process has its own configuration repository in order to avoid redundant realizability analyses. As a result, each consumer process can only reuse configurations it has previously analyzed itself. By combining these two analysis strategies, we aim to further increase the scalability of product-based consistency analysis beyond what either strategy could achieve on its own. The concept of using configuration repositories in a parallel analysis strategy is described in Section 3.3.1.

3.3.1 Usage of Configuration Repositories

Providing each consumer process with its own configuration repository may initially seem counterintuitive, as a shared repository would allow greater reuse of analyzed configurations. However, maintaining a single shared repository would have required synchronization mechanisms such as locking whenever a consumer accessed or modified the repository. This would have introduced significant overhead, increasing with the number of consumer processes involved in the analysis. The chosen design avoids this bottleneck by allowing each consumer to maintain a local repository, which reduces reuse potential but ensures that the consistency analysis remains scalable and lock-free.

In the following Section 3.3.2, we describe how we incorporate configuration repositories in the parallel consistency analysis.

3.3.2 Parallel Repository-Based Consistency Analysis

The procedure of a parallel repository-based consistency analysis is almost identical to the procedure of a parallel consistency analysis defined in Section 3.2.2. The procedure in the producer remains the same, and only the procedure defined for the consumers changes due to the usage of a configuration repository.

We describe these changes in the following paragraph along the activities illustrated in Figure 3.2. Changes compared to the parallel analysis strategy defined in Section 3.2 are shown in blue.

Consumers

The key change is that before analyzing the realizability of a configuration (activity 1.3), the consumer first consults its local configuration repository to determine whether this analysis is necessary (activity 1.2). The decision is made using the repository-based analysis strategy defined in Section 3.1. Based on the result, the consumer either proceeds with the realizability analysis (activity 1.3) or continues the consistency analysis with the next configuration.

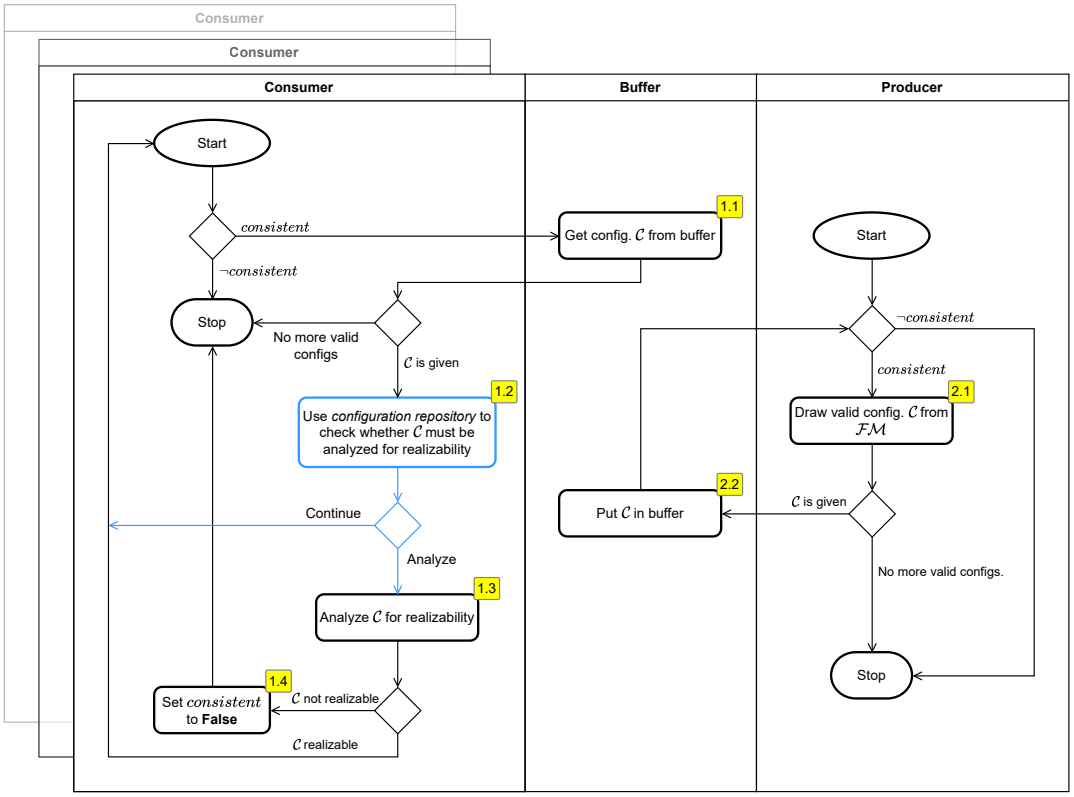

Figure 3.2: Activity Diagram of a Parallel Repository-Based Consistency Analysis.

4. Implementation

In this chapter, we present the code artifacts and tool support used to implement and evaluate the repository-based, parallel and parallel repository-based analysis strategies introduced in Chapter 3.

4.1 Reuse of Existing Code Artifacts

For the implementation of our repository-based, parallel and parallel repository-based strategies, we reuse the *combined problem solver tool* that was developed in the Python language by Ochs [16]. The combined problem solver tool provides an implementation of a product-based consistency analysis and thus serves as a foundation for our analysis strategies.

We extended the combined problem solver tool so that it is now possible to choose between the naive strategy of Ochs et al. [15] and each of our proposed analysis strategies to run the consistency analysis in.

Listing 4.1 provides an illustration of the general procedure for a naive consistency analysis as designed by Ochs et al. [15]. We use this procedure to demonstrate what changes were made to integrate the three analysis strategies repository-based, parallel, and parallel repository-based into the combined problem solver tool.

```
1  function consistency_decision():
2      FM, FD, RP, R ← load from files
3
4      C ← draw random valid configuration from FM
5      while C is given:
6          map resource demands for configuration C
7          construct RAP for C
8          realizable ← analyze realizability of C by solving RAP
9
10         if not realizable:
11             break
12         else:
13             C ← draw random valid configuration from FM
```

Listing 4.1: Naive Consistency Decision.

The procedure starts by initializing the FM, feature demands, resource provisions of the hardware components, and resource types from given files (line 2). The FM and the feature demands are stored in an Extensible Markup Language (XML) file, while the resource provisions and resource types are stored in two Comma-Separated Values (CSV) files. Through the consistency analysis, valid configurations are drawn from the FM as long as they are available (lines 4 and 5). The resource demands of the features of a configuration are then mapped to the defined software components (line 6). With these resource demands, a RAP is constructed and the configuration is analyzed for realizability (lines 7 and 8). If the configuration is not realizable, a mismatch was found and the consistency analysis is stopped (lines 10 and 11); otherwise, it continues with the next configuration (line 13).

4.2 Adapting and Extending Existing Code Artifacts

4.2.1 Repository-Based Analysis Strategy Implementation

Listing 4.2 illustrates the incorporation of a repository-based analysis strategy into the naive consistency analysis described in Section 4.1. Changes compared to the naive consistency analysis are visible in blue. The first change is the initialization of an empty configuration repository (line 5). Then after mapping the resource demands of the configuration, we construct the set of configuration demands and check if it is stored in the configuration repository (lines 8 and 9). A detailed illustration of this procedure is provided in the Appendix in Listing A.1. If the set of configuration demands is not stored in the configuration repository, the configuration is analyzed for realizability (lines 10, 11 and 12). Otherwise, we know that the configuration is realizable and continue with the next configuration (line 14).

```
1  function consistency_decision_repo():
2      FM, FD, RP, R ← load from files
3
4      C ← draw random valid configuration
5      Repo ← empty configuration repository
6      while C is given:
7          map resource demands for configuration C
8          construct the set of configuration demands CD
9          check if CD is stored in Repo
10         if not CD in Repo:
11             construct RAP for C
12             realizable ← analyze realizability of C by solving RAP
13         else:
14             realizable ← True
15
16         if not realizable:
17             break
18         else:
19             C ← draw random valid configuration
```

Listing 4.2: Repository-based Consistency Decision.

4.2.2 Parallel Analysis Strategy Implementation

For the implementation of the parallel and parallel repository-based analysis strategies, we used the Python multiprocessing library [18]. The library allows for the

creation and management of processes and assigns each process a target function on which to work. In addition, it allows for the creation of buffers (i.e., FIFO queues) that incorporate synchronization primitives that restrict the access of multiple producers and consumers as described in Section 2.2.

Listing 4.3 illustrates the adaptions in blue made to the naive consistency analysis described in Section 4.1. We start by initializing an empty FIFO queue $\mathcal{Q}$ as our buffer (line 2). Afterwards, we create the variable *consistent*, initially set to **true**, which is shared between the producer and consumer processes and indicates the occurrence of a mismatch when it is set to **false**. Then we initialize and start one *producer* (lines 4 to 5). Finally, we initialize and start as many *consumer* processes as defined by *num_consumers* (lines 11 to 15). In the end, we wait for the *producer* and *consumer* processes to stop either because there were no more valid configurations available or because a mismatch was found (lines 17 to 18).

```
1  num_consumers number of consumer processes to start (input constant)
2
3  function consistency_decision_parallel(num_consumers):
4      FM, FD, RP, R ← load from files
5
6      Q ← FIFO queue
7      consistent ← shared variable set to true
8      producer ← initialize process with produce_configs(FM)
9      start producer
10
11     n ← 0
12     while n < num_consumers:
13         consumer ← initialize process with analyze_configs(FD, RP, R)
14         start consumer
15         n ← n + 1
16
17     wait for producer and consumer processes to finish
18     get results
```

Listing 4.3: Parallel Consistency Decision.

Listing 4.4 illustrates the procedure used for the *producer* process in a parallel analysis strategy. Here, valid configurations are drawn at random from the current FM and placed in the FIFO queue (lines 6 to 10). Configurations are drawn as long as no consumer has found an inconsistency (line 6) or if no more valid configurations are available (lines 8 to 9). If no more valid configuration is available, **null** values are placed in the queue for every consumer as a termination condition (line 12, activity 2.3).

In contrast, Listing 4.5 illustrates the procedure used for the *consumer* processes. Here, the configurations are analyzed for realizability as long as the shared variable *consistent* is set to **true** (line 8) or the FIFO queue does not contain any more valid configurations (lines 10 to 12). For the analysis, configurations are taken from the queue in line 9 and analyzed for realizability in lines 14 to 16. If a configuration is not realizable, the value of *consistent* is set to **false**, and the analysis is complete (lines 18 to 19).

```
1  FM feature model (input constant)
2  Q FIFO queue (input)
3  consistent shared variable (input)
4
5  function produce_configs(FM, Q, consistent):
6      while consistent:
7          C ← draw random valid configuration
8          if C is not given:
9              break
10         add C to Q
11
12     add null to Q for every initialized consumer
```

Listing 4.4: Producing Valid Configurations.

```
1  FD feature demands (input constant)
2  RP resource provisionings (input constant)
3  R resource types (input constant)
4  Q FIFO queue (input)
5  consistent shared variable (input)
6
7  function analyze_configs(FD, RP, R, Q, consistent):
8      while consistent:
9          C ← get C from Q
10         if C is set to null:
11             consistent ← false
12             break
13
14         map resource demands for configuration C
15         construct RAP for C
16         realizable ← analyze realizability of C by solving RAP
17
18         if not realizable:
19             consistent ← false
```

Listing 4.5: Analyzing Valid Configurations for Realizability.

4.2.3 Parallel Repository-Based Analysis Strategy Implementation

For the implementation of the parallel repository-based analysis strategy, we reuse the procedures described for the parallel analysis strategy in Section 4.2.2, but adapt this for the consumer processes to include the implementation of the repository-based analysis strategy described in Section 4.2.1.

Listing 4.6 illustrates the adapted procedure for the consumer processes. We initialize an empty configuration repository (line 8) and use it to avoid redundant realizability analysis in the same way as for the repository-based analysis strategy (lines 16 to 22).

```
1  FD feature demands (input constant)
2  RP resource provisionings (input constant)
3  R resource types (input constant)
4  Q FIFO queue (input)
5  consistent shared variable (input)
6
7  function analyze_configs_repo(FD, RP, R, Q, consistent):
8      Repo ← empty configuration repository
9      while consistent:
10         C ← get C from Q
11         if C is not given:
12             consistent ← false
13             break
14
15         map resource demands for configuration C
16         construct the set of configuration demands CD
17         check if CD is stored in Repo
18         if not CD in Repo:
19             construct RAP for C
20             realizable ← analyze realizability of C by solving RAP
21         else:
22             realizable ← True
23
24         if not realizable:
25             consistent ← false
```

Listing 4.6: Analyzing Valid Configurations for Realizability Using a Configuration Repository.

5. Evaluation

In this chapter, we evaluate our three repository-based, parallel, and parallel repository-based analysis strategies defined in Chapter 3 through their implementation described in Chapter 4. For evaluation, we derive the Research Questions (RQs) defined in Section 5.1. We evaluate the strategies along a modification of the existing automotive Body Comfort System (BCS) case study [12] [20] and using the setup and methodology described in Section 5.2. We present the results of the evaluation in Section 5.3 and their discussion in Section 5.4. In the end, the threats to validity regarding the internal and external threats are discussed in Section 5.5.

5.1 Research Questions

We evaluate our three analysis strategies repository-based, parallel, and parallel repository-based regarding performance in terms of total run-time and potential trade-offs in terms of overhead through parallelization and additional memory usage through a repository-based analysis. Specifically, our objective is to answer the RQs defined below.

RQ 1: How is the performance of a product-based consistency analysis for CPPLs improved by a repository-based, parallel and parallel repository-based analysis strategy?

With RQ 1 we evaluate the impacts of our repository-based, parallel, and parallel repository-based analysis strategies on the performance of the consistency analysis. For this we will compare each of the analysis strategies with the naive analysis strategy of Ochs et al. [15], which serves as a baseline in this comparison because it performs consistency analysis without using any additional repository or parallelization techniques. Therefore, we individually measure the run-times of our three analysis strategies and the naive analysis strategy for analyzing consistent versions of the adapted BCS case study as a subject system.

RQ 2: Does the parallelization of a naive and repository-based consistency analysis lead to additional run-time overhead?

To evaluate whether parallelization introduces run-time overhead compared to the naive consistency analysis, we will measure the time needed to initialize all producer and consumer processes, and the time needed by these processes to put configurations in the buffer and get them out of it. We answer this RQ along the evaluation of RQ 1.

RQ 3: Does the use of a configuration repository in a consistency analysis lead to additional memory usage?

To evaluate whether the use of a configuration repository introduces additional memory usage compared to the naive consistency analysis, we will measure the memory usage of the repository-based, parallel repository-based and naive analysis strategies. We will compare the memory usage of the repository-based and parallel repository-based to the memory usage of the naive analysis strategy. We answer this RQ along the evaluation of RQ 1.

5.2 Setup and Methodology

5.2.1 Subject System: Body Comfort System (BCS) Case Study

The BCS is a case study in the automotive domain that was developed with industry partners. It represents parts of a car PL and was initially published by Lity et al. [12]. The case study was extended with an evolution scenario by Nahrendorf et al. [21] which describes six additional versions. From now on, we reference the initial version of Lity et al. [12] with version 1.0 and the additional versions by Nahrendorf et al. with versions 1.1, 2.0, 3.0, and 3.1.

The initial version 1.0 was extended with solution space artifacts and modeled as an instance of the UCM by Ochs et al. [15]. Rak [20] proceeded with the extension of the solution space to versions 1.1, 2.0, 3.0, and 3.1.

Versions 1.0, 1.1, 3.0, and 3.1 are initially inconsistent, but throughout our evaluation, we use consistent adaptions of these versions. Through these adaptions, we create a worst-case scenario where each valid configuration is also realizable, and thus has to be analyzed for realizability. With this increasing of configurations that have to be analyzed for realizability, we also increase the potential for total run-time differences between our strategies and the naive consistency analysis.

We do not evaluate version 2.0 as its introduction of new features leads to an explosion of possible valid configurations, making the computation of a consistent version infeasible. Thus, we sliced out all the features with their resource demands, nulled the resource provisionings, and kept the resource types introduced in version 2.0. The sliced features, resource demands and resource provisionings of version 2.0 are illustrated in Appendix Section A.1.2.

The changes made to each version used in the evaluation are described in Section 5.2.1.1, Section 5.2.1.2, Section 5.2.1.4 and Section 5.2.1.4.

5.2.1.1 Version 1.0

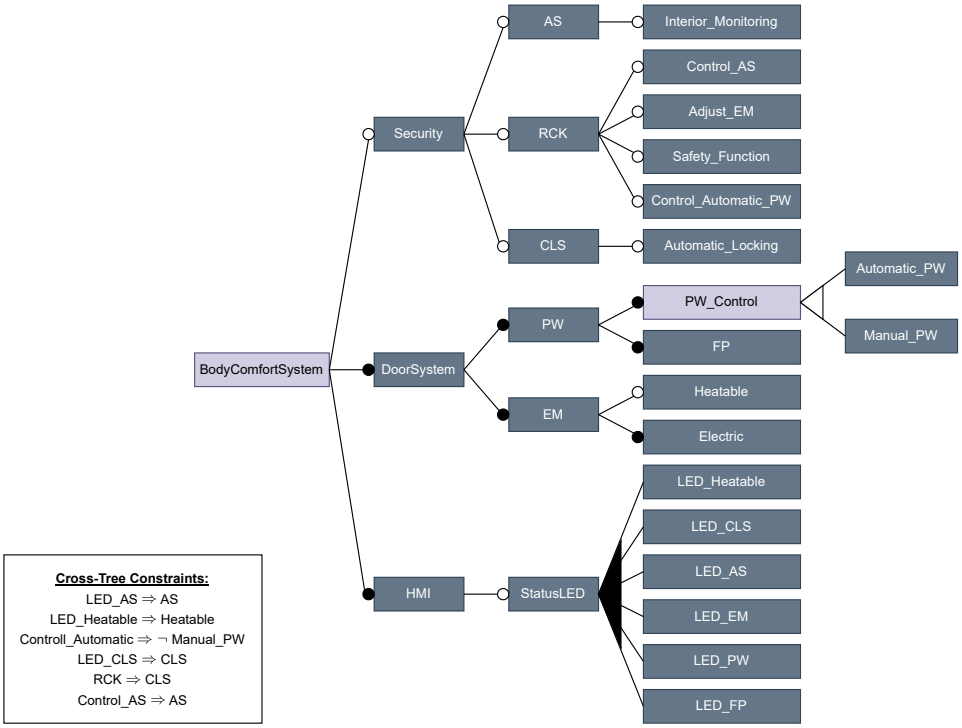

Figure 5.1: FM of Version 1.0.

The initial version 1.0 consists of 27 features that split into the tree subsystems `Security`, `DoorSystem` and `HMI`, visible in Figure 5.1. `HMI` has the optional feature `StatusLED` which in turn consists of multiple StatusLEDs that can be chosen individually through an or-group. `DoorSystem` has the features Power Window (PW) and Exterior Mirror (EM). PW is controlled automatically or manually and has a Finger Protection (FP). EM has the optional feature `Heatable` and must select the feature `Electric`. The optional feature `Security` consists of optional functionality for an Alarm System (AS), a Remote Control Key (RCK) and a Central Locking System (CLS).

The solution space consists of the software components `hmi-controller` (sw_0), `power-window-controller` (sw_1), `exterior-window-controller` (sw_2) and `security-controller` (sw_3). sw_0 controls the `HMI` and all `StatusLED`s, sw_1 controls the PW, sw_2 controls the EM and sw_3 controls the `Security` sub-tree.

The resource demands of the software components are illustrated in Table 5.1 while Table 5.2 introduces the hardware components `infotainment-hardware` (hw_0) and `security-hardware` (hw_1) with their resource provisionings. The resource types used for the resource demands of the software components and the resource provisionings of the hardware components are defined in Table 5.3.

Feature	Software Components sw_i	Resource Demands
LED_AS	sw_0	$rd_0^0 = 2$
LED_FP	sw_0	$rd_0^0 = 1$
LED_CL	sw_0	$rd_0^0 = 1$
LED_PW	sw_0	$rd_0^0 = 1$
LED_EM	sw_0	$rd_0^0 = 1$
LED_Heatable	sw_0	$rd_0^0 = 1$
Electric	sw_1	$rd_1^7 = 10$ $rd_1^1 = 5$
Heatable	sw_1	$rd_1^1 = 20$
FP	sw_2	$rd_6^2 = 1$
Manual_PW	sw_2	$rd_6^2 = 2$
Automatic_PW	sw_2	$rd_7^2 = 5$ $rd_6^2 = 3$
RCK	sw_3	$rd_2^3 = 2$ $rd_3^3 = 10$
Control_Automatic_PW	sw_1	$rd_7^1 = 5$
	sw_3	$rd_2^3 = 2$
Adjust_EM	sw_2	$rd_7^2 = 10$
	sw_3	$rd_2^3 = 2$
Control_AS	sw_3	$rd_2^3 = 2$ $rd_2^3 = 2$
Safety_Function	sw_3	$rd_2^3 = 2$
AS	sw_3	$rd_3^3 = 100$ $rd_5^3 = 4$
Interior_Monitoring	sw_3	$rd_3^3 = 700$ $rd_5^3 = 1$
CLS	sw_3	$rd_3^3 = 10$
Automatic_Locking	sw_3	$rd_4^3 = 1$

Table 5.1: Resource Demands of Version 1.0.

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware`(hw_0)	$rp_0^0 = 8$ $rp_1^0 = 25$ $rp_6^0 = 4$ $rp_7^0 = 128$
`security-hardware`(hw_1)	$rp_1^2 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$

Table 5.2: Resource Provisions of Version.

k	isAdditive	isExclusive	boundary	Unit	Description
0	True	False	LOWER		no. Interface Slots
1	True	False	LOWER	W	Power Specification
2	True	True	LOWER		no. Security Comm. Channels
3	False	True	LOWER	MHz	Security Processing Core Clock
4	False	True	UPPER	min	Security Automatic Relock Time
5	True	True	EXACT		no. Security Video Cameras
6	True	False	LOWER		no. Window Movement Sensors
7	True	False	LOWER	kbit/s	Bandwidth

Table 5.3: Resource Types of Version 1.0.

The resource provisions of version 1.0 are designed to not satisfy all resource demands, making the PL inconsistent. In particular, the resource demands of the feature `Interior_Monitoring`, $rd_5^3 = 1$ can not be satisfied. Resource type 5 (*no. Security Video Cameras*) defines an `EXACT` boundary but the only hardware component that provides the given type is `security-hardware` (hw_1) with $rp_5^1 = 4$ which does not match the demanded value of 1.

To make version 1.0 consistent, we first changed the boundary of resource type 5 to `LOWER`. In addition, we changed the provision of `security-hardware` (hw_1) to $rp_5^1 = 5$ so that it is always greater than any demand for resource type 5. With these adaptions, version 1.0 comprises 11616 valid and realizable configurations.

Changes are visible in blue in Table 5.4 and Table 5.5.

k	isAdditive	isExclusive	boundary	Unit	Description
...	...	...	...	...	...
5	True	True	LOWER		no. Security Video Cameras
...	...	...	...	...	...

Table 5.4: Resource Types of the Consistent Adaption of Version 1.0.

Hardware Component (hw_j)	Provisions (rp_k^j)
`security-hardware`(hw_1)	... $rp_5^1 = 5$

Table 5.5: Resource Provisions of the Consistent Adaption of Version 1.0.

5.2.1.2 Version 1.1

Version 1.1 introduces a new hardware component `safety-hardware` (hw_2) that takes over all resource provisions related to passenger safety.

Appendix A.1.1 shows the tables in detail.

With the new hardware component `safety-hardware` (hw_2) the resource demands $rd_7^2 = 5$ and $rd_6^2 = 3$ of the feature `Automatic_PW` can not be met by a single hardware component. hw_0 fulfills the resource demands $rd_7^2 = 5$ with the resource provisions of $rp_7^0 = 128$ while hw_2 fulfills the resource demands $rd_6^2 = 3$ with the resource provisions $rp_6^2 = 4$. The problem is that it is not possible to divide the demands of `Automatic_PW` between hw_0 and hw_2 as a software component must be assigned to exactly one hardware component.

To make version 1.1 consistent, we added the resource provision $rd_7^2 = 128$ to `safety-hardware` (hw_2). With that `safety-hardware` can meet both demands of the feature `Automatic_PW`. The number of valid and realizable configurations is the same as for the consistent adaption of version 1.0.

Changes are visible in blue in Table 5.6 and Table 5.7.

k	isAdditive	isExclusive	boundary	Unit	Description
...	...	...	...	...	...
5	True	True	LOWER		no. Security Video Cameras
...	...	...	...	...	...

Table 5.6: Resource Types of the Consistent Adaption of Version 1.1.

Hardware Component (hw_j)	Provisions (rp_k^j)
`security-hardware`(hw_1)	... $rp_5^1 = 5$
`safety-hardware`(hw_2)	... $rp_7^2 = 128$

Table 5.7: Resource Provisions of the Consistent Adaption of Version 1.1.

5.2.1.3 Version 3.0

In version 3.0 a new feature `Seat` is added as an optional child of the `BodyComfort-System` feature. In addition, the resource types *no. Motors* and *Long Term Memory* are added, which are demanded by a new software component `seat-controller` (sw_5) and are provided by the `infotainment-hardware` (hw_0).

Appendix A.1.3 shows the tables in detail.

The provisions in `infotainment-hardware` (hw_0) are not designed to fulfill the resource demand $rd^5_{12} = 2$ of the new `Seat` feature. This is because resource type 12 (*Long Term Memory*) defines a `LOWER` boundary but `infotainment-hardware` only provides the resource type with $rp^0_{12} = 1$ which is lower than the demanded value.

To make version 3.0 consistent, we applied the same changes as in versions 1.0 and 1.1. In addition, we changed the resource provision $rp^0_{12} = 1$ to $rp^0_{12} = 2$ in the hardware component `infotainment-hardware` (hw_0) to match the resource demand $rd^5_{12} = 2$ of the `Seat` feature. Due to these changes, all 21984 valid configurations of the adapted version 3.0 are also realizable.

Changes are visible in blue in Table 5.8 and Table 5.9.

k	isAdditive	isExclusive	boundary	Unit	Description
...	...	...	...	...	...
5	True	True	LOWER		no. Security Video Cameras
...	...	...	...	...	...

Table 5.8: Resource Types of the Sliced Consistent Adaption of Version 3.0.

Hardware Component (hw_j)	Provisions (rp^j_k)
`infotainment-hardware`(hw_0)	... $rp^0_{12} = 2$
`security-hardware`(hw_1)	... $rp^1_5 = 5$
`safety-hardware`(hw_2)	... $rp^2_7 = 128$ ...

Table 5.9: Resource Provisions of the Sliced Consistent Adaption of Version 3.0.

5.2.1.4 Version 3.1

In version 3.1 the resource type *response time* is added. The finger protection feature now has demands for this resource type and window movement sensors. These demands are fulfilled by the `safety-hardware` hardware component.

Appendix A.1.4 shows the tables in detail.

To make version 3.1 consistent, we applied the same changes as for version 3.0 and have the same number of valid and realizable configurations.

5.2.2 Evaluation Procedure

This section describes the procedures we followed to answer our three RQs. Section 5.2.2.1 introduces the evaluation procedure used to measure and compare the total run-times of each analysis strategy for each version of the BCS case study addressing RQ 1. In Section 5.2.2.2 we define the evaluation procedure to measure

the potential overhead created by parallelizing a repository-based and naive consistency analysis for each version of the adapted BCS case study addressing RQ 2. Section 5.2.2.3 defines the evaluation procedure used to measure the memory consumption of a naive, repository-based and parallel repository-based consistency analysis for each version of the BCS case study addressing RQ 3.

5.2.2.1 Measuring and Comparing Total Run-Times

This subsection is structured into two parts. The first part explains how we measure the total run-time of each analysis strategy for every version of the BCS case study. The second part describes how we use these measurements to compare run-times across analysis strategies.

Measurement of Total Run-Times

Listing 5.1 illustrates the procedure we use to measure the total run-times for the naive product-based consistency analysis by Ochs et al. [15] and the improved versions with our repository-based, parallel and parallel repository-based analysis strategies. As the run-times of the analysis strategies repository-based and parallel repository-based depend on the order in which the configurations are reused in the configuration repository and on the times the producer and the consumers need to place configurations in the buffer and take them out of it for the parallel and parallel repository-based analysis strategies, we repeated the consistency analysis five times per strategy for each BCS version to minimize these influences. This relatively small number of repetitions is discussed in the threats to validity in Section 5.5.

```
1  for each bcs_version in BCS versions:
2      for repetition = 1 to 5:
3          measure total_time_naive:
4              consistency_decision()
5
6          measure total_time_repo:
7              consistency_decision_repo()
8
9          for thread = 1 to max_threads - 1:
10             measure total_time_parallel_threads:
11                 consistency_decision_parallel(threads)
12
13             measure total_time_parallel_repo_threads:
14                 consistency_decision_parallel_repo(threads)
15
16     store timing results for current bcs_version and all strategies
```

Listing 5.1: Procedure to Measure the Total Run-Time of a Consistency Analysis.

For the naive and repository-based analysis strategies, we measure total run-times (total_time_naive and total_time_repo) for each version of the BCS case study.

For the parallel and parallel repository-based analysis strategies, we additionally iterate over the number of available threads, and measure the corresponding run-times (total_time_parallel_*threads* and total_time_parallel_repo_*threads*) in each iteration.

For each version of the BCS case study, we store the timing results of the five repetitions for each analysis strategy and each used number of threads.

Comparing of Total Run-Times

To compare total run-times, we directly compare the measurements for the naive and repository-based analysis strategies for each version of the BCS case study. For parallel and parallel repository-based analysis strategies, an example of the process of obtaining measurements for comparison is illustrated in Figure 5.2. The procedure is performed independently for both analysis strategies. For each version of the BCS case study, we first identify the number of threads with the lowest total run-time in median. We then calculate the median of these thread numbers. The measurements associated with this median number of threads are then used for run-time comparisons with other analysis strategies.

We use the number of threads for this aggregation because it gives us a control variable that has a direct impact on the performance of the analysis strategies. With this fixed number of threads for each version of the BCS case study, we prevent bias by individually optimizing the analysis strategies for individual versions. By selecting the median number of threads, we use a representative number of threads that improves both fairness and reproducibility in the evaluation.

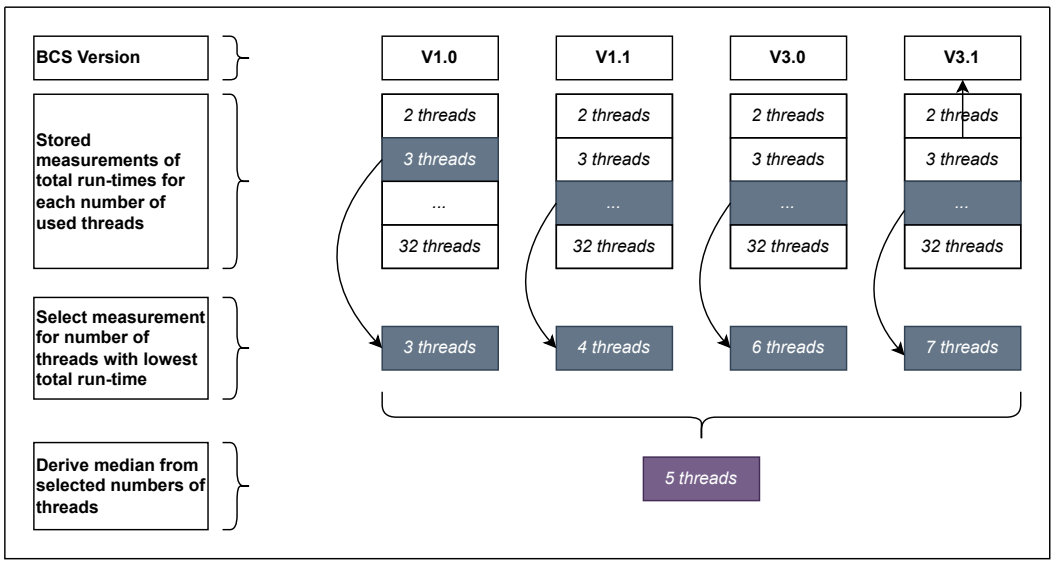

Figure 5.2: Deriving Optimal Number of Threads in Median.

5.2.2.2 Measurement of Where the Overhead for Parallelization Arises

We measure where parallel and parallel repository-based analysis strategies introduce additional run-time overhead, within the procedure used to measure total run-times illustrated in Listing 5.1.

For both parallel-based analysis strategies, we measure the total time needed to initialize the buffer, the producer, and all consumers (initialization_time) as illustrated in Listing 5.2. In the producer, we measure the total time that the producer needs to place configurations in the buffer (producer_time) as illustrated in Listing 5.3. In each consumer initialized, we measure the total time it takes them to get configurations out of the buffer (consumer_time) as shown in Listing 5.4. After the producer

and every consumer have finished working, we add the derived initialization_time, producer_time, and consumer_times and store this time (total_overhead_time) for the current version of the BCS case study and analysis strategy.

```
function consistency_decision_parallel(_repo)(threads):
    measure initialization_time:
        initialize buffer
        initialize produce_configs()
        initialize threads * analyze_configs()

total_overhead_time = initialization_time
                      + producer_time
                      + (threads * consumer_time)

store total_overhead_time result for current bcs_version
```

Listing 5.2: Procedure to Measure the Total Time for Initializing Producer and Consumer Processes.

```
function produce_configs():
    measure producer_time needed to put a config. in the buffer

store sum of producer_time results for current bcs_version
```

Listing 5.3: Procedure to Measure the Total Time for Putting Elements in the Buffer.

```
function analyze_configs():
    measure consumer_time needed to get a config. from the buffer

store sum of consumer_time results for current bcs_version
```

Listing 5.4: Procedure to Measure the Total Time for Getting Elements out of the Buffer.

5.2.2.3 Measurement of Memory Usage

To measure the memory usage of a consistency analysis performed with the naive, repository-based, and parallel repository-based analysis strategies, we use the procedure illustrated in Listing 5.5. We repeat the consistency analysis five times per analysis strategy for each BCS version to ensure the robustness of the results.

For naive and repository-based analysis strategies, we measure the memory usage (memory_naive and memory_repo) needed to perform a consistency analysis for each version of the BCS case study.

For the parallel repository-based analysis strategy, we perform memory measurement using the median optimal threads for each version of the BCS case study derived during the total run-time analysis in Section 5.2.2.1. This ensures that memory and run-time measurements are aligned under identical execution conditions, enabling a fair and meaningful cross-strategy comparison. Measurement of memory with a different number of threads as used for the total run-time comparison would reduce the comparability of the results across strategies.

To measure the total memory usage of the parallel repository-based analysis strategy (memory_parallel_repo), we have to individually measure and add the memory of the initialization function and the memory of the producer and consumer processes started by this function, since each of them spawns their own memory space.

```
for each bcs_version in BCS versions:
    for repetition = 1 to 5
        measure memory_naive:
            consistency_decision()

        measure memory_repo:
            consistency_decision_repo()

        measure memory_parallel_repo:
            measure memory_init (consistency_decision_parallel_repo())
            measure memory_producer
            measure memory_consumers
            memory_parallel_repo = memory_init
                                 + memory_producer
                                 + memory_consumers

    store memory results for current bcs_version and all strategies
```

Listing 5.5: Procedure to Measure the Memory Usage of a Consistency Analysis.

5.2.3 Software and Hardware Execution Environment

As tool support for our evaluation, we used the Python language in Version 3.12.8. For memory tracing, we used the Python library *tracemalloc* [19] that enables tracing of memory blocks allocated by Python. In addition, we used the *timings* module implemented by Ochs [16] to track the total run-time of a program.

We ran our evaluation using a Linux Server running Ubuntu 24.04.2 LTS as an operating system. The hardware used consists of 128 GB DDR4 RAM and the AMD Ryzen Threadripper PRO 5955WX Central Processing Unit (CPU) with 16-cores and the possibility of running 32 threads in parallel.

5.3 Results

The presentation and description of our evaluation results is divided into Section 5.3.1 illustrating the total run-times of each of our analysis strategies, Section 5.3.2 illustrating the additional run-time overhead of the parallel-based analysis strategies, and Section 5.3.3 illustrating the memory consumption of the naive and both repository-based analysis strategies.

5.3.1 Total Run-Times

This section presents the results of the total run-time measurements for the naive, repository-based, parallel, and parallel repository-based analysis strategies across all versions of the BCS case study. We begin by presenting the results of these measurements for each of the iterated threads for the parallel and parallel repository-based analysis strategies. From these results, we then infer, individually for each parallel-based analysis strategy, the number of threads that we use for direct comparisons of

the analysis strategies with each other. We then present such a comparison in terms of total runt-times between all analysis strategies and for each version of the BCS case study. In addition, we provide the speed-up each analysis strategy achieves compared to the naive one.

Overview of the Results for the Parallel and Parallel Repository-Based Analysis Strategies

Figure 5.3 and Figure 5.4 visualize the measurement results for the parallel and parallel repository-based analysis strategies, respectively. Each plot shows the median total run-time from these measurements in seconds for each version of the BCS case study, plotted against the number of threads used:

- The x-axis represents the number of threads.
- The y-axis represents the median total run-time in seconds corresponding to each thread count.
- Red markers indicate the minimum run-time for each case study version.
- A vertical blue line highlights the median thread count used for strategy comparisons.

For both analysis strategies the plots reveal a common trend: total run-times decrease up to a minimum as the number of threads increases. After reaching this minimum, the total run-times increase again. This pattern is especially noticeable for the parallel repository-based analysis strategy.

Detailed Results for the Parallel Analysis Strategy

For the parallel analysis strategy, the red markers in Figure 5.3 indicate the lowest median run-times achieved for each version of the BCS case study:

- $\sim$ 49.84 s with 15 threads for version 1.0.
- $\sim$ 55.86 s with 15 threads for version 1.1.
- $\sim$ 224.23 s with 26 threads for version 3.0.
- $\sim$ 237.89 s with 27 threads for version 3.1.

For cross-strategy comparisons, we select a fixed thread count of 21 (marked by a vertical blue line), resulting in the following total run-times:

- $\sim$ 55.06 s for version 1.0.
- $\sim$ 60.89 s for version 1.1.
- $\sim$ 227.03 s for version 3.0.
- $\sim$ 247.34 s for version 3.1.

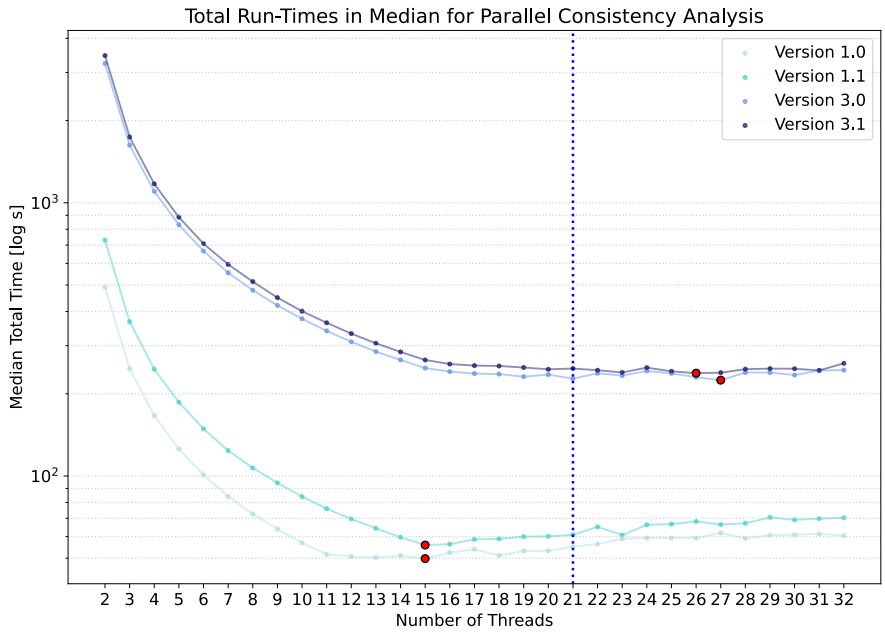

Figure 5.3: Total Run-Times in Median for Parallel Consistency Analysis.

Detailed Results for the Parallel Repository-Based Analysis Strategy

As for the parallel analysis strategy, the red markers in Figure 5.4 indicate the lowest median total run-times achieved with the parallel repository-based analysis strategy for each version of the BCS case study:

- $\sim$ 47.2 s with 8 threads for version 1.0.
- $\sim$ 48.08 s with 12 threads for version 1.1.
- $\sim$ 171.05 s with 15 threads for version 3.0.
- $\sim$ 182.65 s with 15 threads for version 3.1.

Here, the total run-times in median used for comparison between strategies are derived with 12 threads and are again marked by a vertical blue line in the plot:

- $\sim$ 50.06 s for version 1.0.
- $\sim$ 48.08 s for version 1.1.
- $\sim$ 196.26 s for version 3.0.
- $\sim$ 210.48 s for version 3.1.

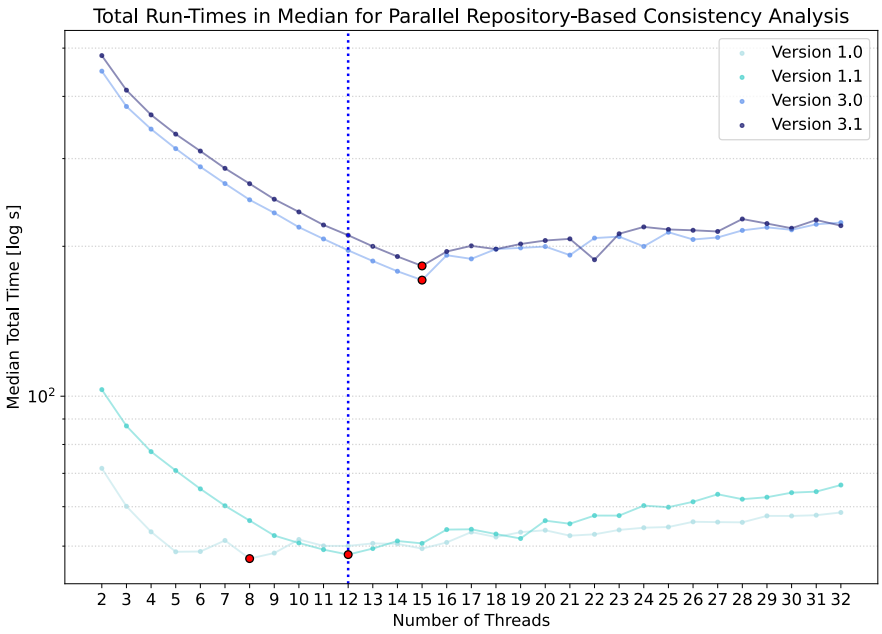

Figure 5.4: Total Run-Times in Median for Parallel Repository-Based Consistency Analysis.

Strategy Comparison and Speed-Up

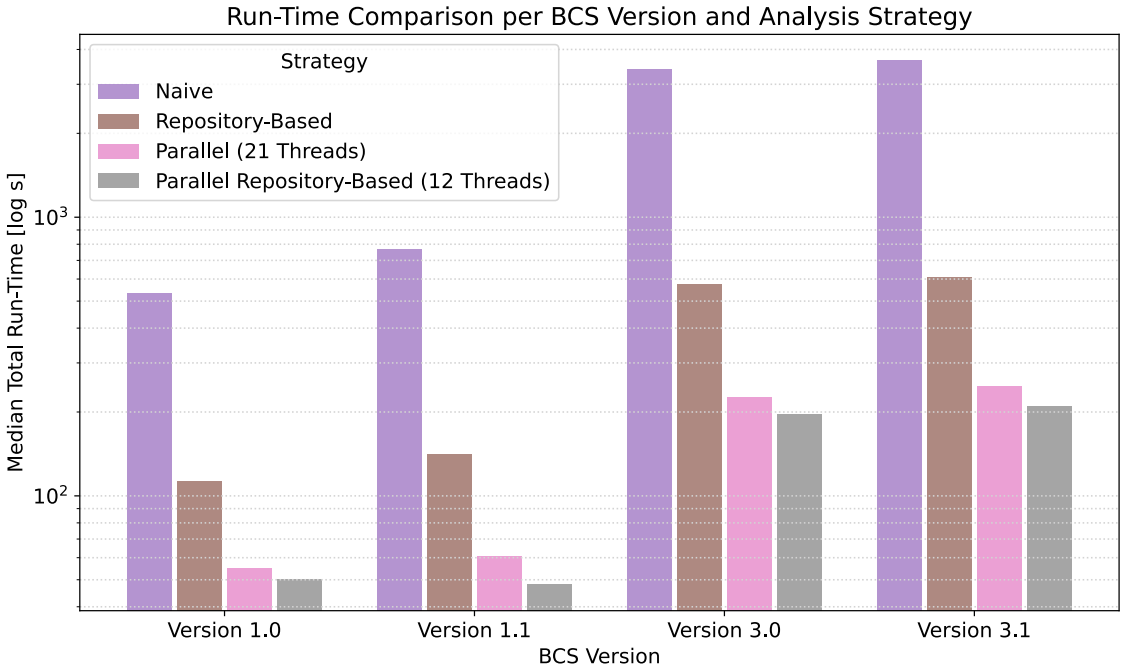

Figure 5.5: Runtime Comparison per BCS Version and Analysis Strategy.

Figure 5.5 presents a direct visual comparison of the total run-times in median for the naive, repository-based, parallel and parallel repository-based analysis strategies. The comparison is based across each version of the BCS case study. Each group of bars corresponds to one version of the case study, with four color-coded bars per

group representing the analysis strategies. The height of each bar denotes the median total run-time in seconds, derived from five independent runs. The bar chart format clearly illustrates how the total run-time decreases from the naive analysis strategy to the most optimized analysis strategy (e.g., parallel repository-based). For the parallel and parallel repository-based analysis strategies, we use the total run-times in median for the derived number of threads for cross-strategy comparison. These are 21 threads for the parallel strategy and 12 threads for the parallel repository-based strategy.

The total times in median for the naive analysis strategy are:

- $\sim$ 533.21 s for version 1.0.
- $\sim$ 767.57 s for version 1.1.
- $\sim$ 3406.76 s for version 3.0.
- $\sim$ 3648.66 s for version 3.1.

For the parallel repository-based analysis strategy these timings are:

- $\sim$ 112.51 s for version 1.0.
- $\sim$ 141.31 s for version 1.1.
- $\sim$ 575.75 s for version 3.0.
- $\sim$ 608.83 s for version 3.1.

To calculate the speed-up in terms of total run-time between the naive and the repository-based, parallel, and parallel repository-based analysis strategies, we use Equation 5.1 [8]. In the equation, t_{naive} represents the total run-time of the naive analysis strategy and t_{opt} represents the total run-time of one of our improvements. This metric provides a quantitative measure of the improvements in total run-time achieved by the optimized strategies. A value of $S_{run-time} > 1$ indicates that the run-time of the optimized analysis strategy is $S_{run-time}$ times shorter than the run-time of the naive analysis strategy. A value close to 1 suggests minimal improvement in run-time and a value less than 1 would imply that the run-time of the improved analysis strategy is higher then the run-time of the naive one. The calculated speed-ups compared to the naive analysis strategy are illustrated in Table 5.10.

$$S_{run-time} = \frac{t_{naive}}{t_{opt}} \tag{5.1}$$

The results illustrated in Figure 5.5 and Table 5.10 show a clear trend: each analysis strategy is at least 4.7 times faster than the naive analysis strategy. In addition, for all analysis strategies except the parallel one, speed-up increases through the analyzed versions of the BCS case study. For the parallel repository-based analysis, the speed-up decreases from 15.01 to 14.75 between versions 3.0 and 3.1. The repository-based analysis strategy achieves the lowest speed-up through all versions of the BCS case study, while the parallel repository-based analysis strategy achieves the highest speed-up and is more than 10.6 times faster than the naive analysis strategy for each version.

Version	Repository-Based	Parallel	Parallel Repository-Based
Version 1.0	4.74	9.68	10.65
Version 1.1	5.43	12.61	15.96
Version 3.0	5.92	15.01	17.31
Version 3.1	5.99	14.75	17.33

Table 5.10: Speed-ups Compared to the Naive Strategy.

5.3.2 Parallel Overhead Times

Figure 5.6 and Figure 5.7 present additional run-time overhead in the median measured for the parallel and parallel repository-based strategies.

Each plot shows the additional run-time overhead in the median over five runs for each version of the BCS case study, plotted against the number of threads used. It should be noted that this additional run-time overhead cannot be directly compared to the total run-times visualized in Figure 5.3 and Figure 5.4. This is because the additional run-time overhead was measured across multiple processes in each run and then summed up. As a result, these plots only illustrate how the waiting times of the producer and consumer processes increase with an increasing number of threads used.

All of the plots show that for some time the additional run-time overhead stays around the same level for additional numbers of threads but has a spike when a certain number is reached. After this spike, the additional run-time overhead increases with each additional thread used for both strategies and each illustrated version of the BCS case study.

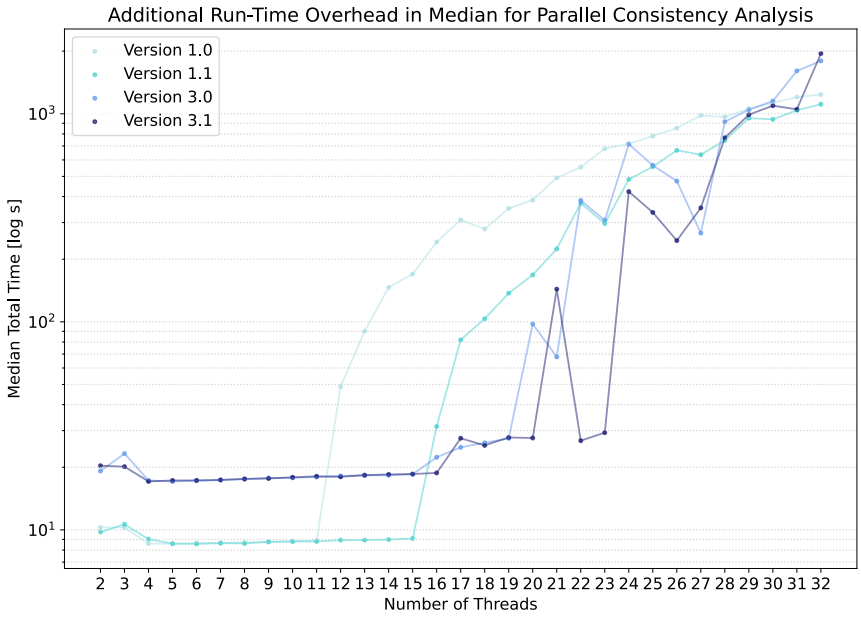

Figure 5.6: Additional Run-Time Overhead in Median for Parallel Consistency Analysis.

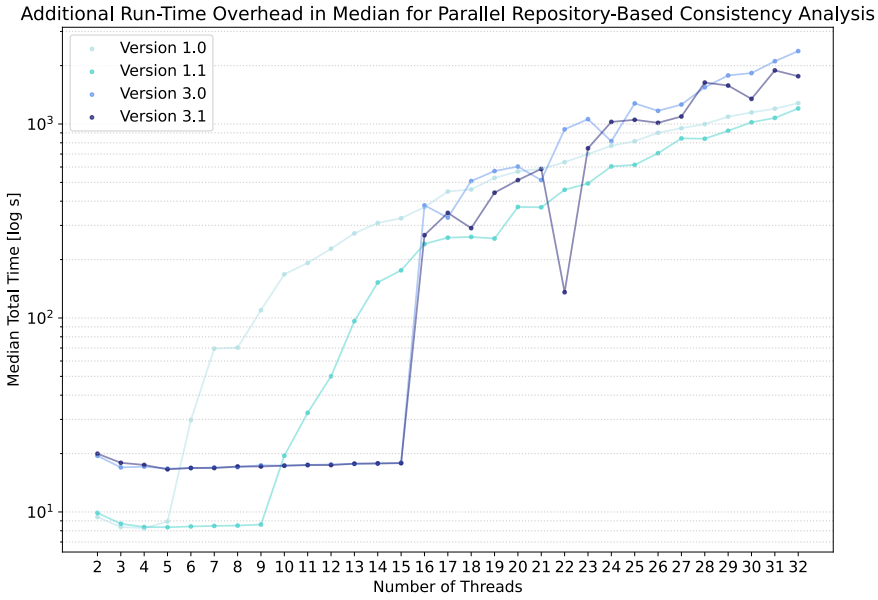

Figure 5.7: Additional Run-Time Overhead in Median for Parallel Repository-Based Consistency Analysis.

For the parallel analysis strategy, these spikes occur with:

- 11 threads for version 1.0.
- 15 threads for version 1.1.
- 15 threads for version 3.0.
- 16 threads for version 3.1.

In contrast, with the parallel repository-based analysis strategy, these spikes occur with:

- 5 threads for version 1.0.
- 9 threads for version 1.1.
- 15 threads for version 3.0.
- 15 threads for version 3.1.

5.3.3 Memory Consumption

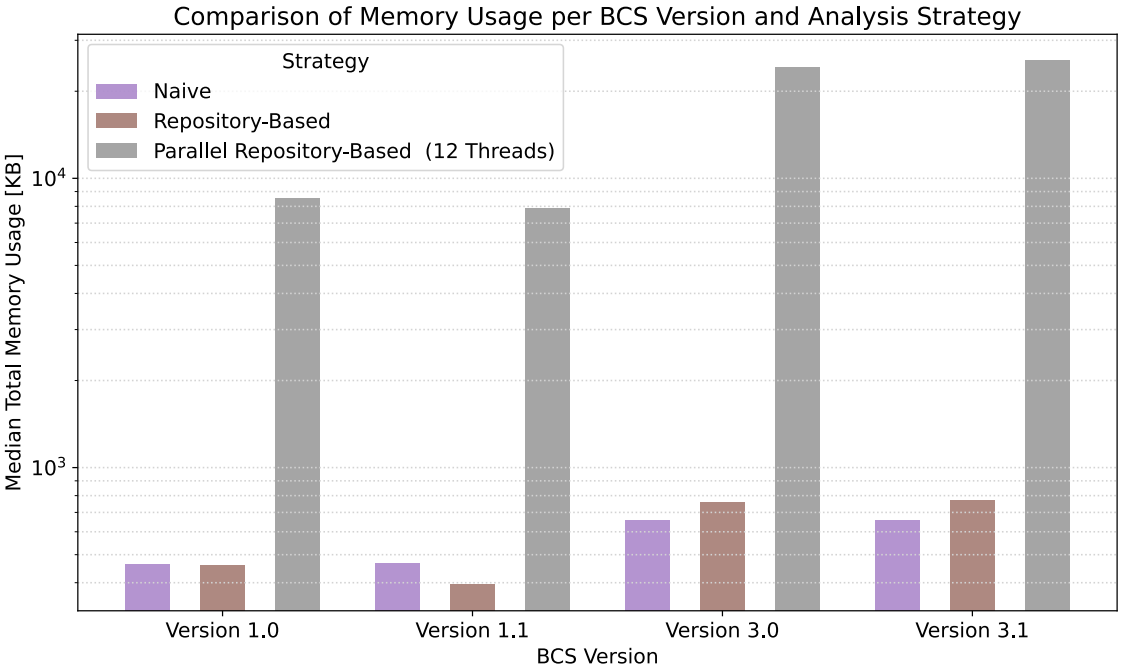

Figure 5.8: Comparison of Memory Usage per BCS Version and Analysis Strategy.

Figure 5.8 presents a direct visual comparison of total memory usage in the median of the naive, repository-based, and parallel repository-based analysis strategies. For the parallel repository-based analysis strategy, we use the number of threads derived for comparison across strategies. The comparison is based on each version of the BCS case study. Each group of bars corresponds to one version of the case study, with three color-coded bars per group representing the analysis strategies. The height of each bar denotes the total memory used in Kilobyte (KB), derived from five independent runs. The format of the bar chart visualizes that the memory usage is less with the repository-based analysis strategy than with the naive one for versions 1.0 and 1.1 of the BCS case study. In contrast, the repository-based analysis strategy uses more memory than the naive analysis strategy for versions 3.0 and 3.1. For all versions of the BCS case study, the parallel repository-based analysis strategy uses by far the most memory.

For the naive analysis strategy, we measured the following memory usages:

- $\sim$ 464.12 KBs for version 1.0.
- $\sim$ 466.48 KBs for version 1.1.
- $\sim$ 657.98 KBs for version 3.0.
- $\sim$ 650.70 KBs for version 3.1.

For the repository-based analysis strategy, the memory usages were:

- ∼ 449.71 KBs for version 1.0.
- ∼ 394.11 KBs for version 1.1.
- ∼ 765.65 KBs for version 3.0.
- ∼ 768.46 KBs for version 3.1.

And finally, for the parallel repository-based analysis strategy with 12 threads used, the memory usages were:

- ∼ 8548.02 KBs for version 1.0.
- ∼ 7883.38 KBs for version 1.1.
- ∼ 24203.36 KBs for version 3.0.
- ∼ 25565.68 KBs for version 3.1.

5.4 Discussion

In this section, we discuss our RQs defined at the beginning of Chapter 5 using our evaluation results described in Section 5.3.

RQ 1: How is the performance of a product-based consistency analysis for CPPLs improved by a repository-based, parallel and parallel repository-based analysis strategy?

The results for measuring the total run-times in Section 5.3.1 show that each analysis strategy addressed in RQ 1 is at least 4.74 times faster than the naive product-based consistency analysis of a CPPL. These improvements were best for the parallel repository-based analysis strategy followed by the parallel analysis strategy and finally the repository-based analysis strategy. For a repository-based and parallel repository-based analysis strategy, these improvements are bound by the number of times that a realizability analysis was avoided. This is because the improvements of both strategies are based on reusing analysis results from previously analyzed configurations instead of performing a realizability analysis for each configuration. The process of reusing analysis results requires significantly less run-time than the process of analyzing a configuration for realizability. Thus, the performance of a parallel repository-based and especially a repository-based analysis strategy is only improved compared to the naive one if realizability analyses can be avoided due to reusing analysis results.

The results also show that the parallel and parallel repository-based analysis strategies outperform the naive consistency analysis even when using the worst-case number of two threads for which their total run-times were highest in median (detailed numbers are provided in Appendix Section A.3). For each parallel-based analysis strategy we also measured that they have an optimum number of threads for which they perform the best, but if they pass these numbers, their total run-times stagnate

or even increase again. For a parallel repository-based analysis strategy, the increase after reaching this number could be due to the fact that as the number of consumer processes increases, the number of reusable analysis results in the configuration repository becomes smaller for each process, and thus more realizability analyses must be performed. This is because as the number of consumer processes increases, the number of configurations that each process can analyze and potentially reuse becomes smaller. In addition, with an increased number of used threads it could be that we no longer have the full power of each thread due to other, non-related, processes running in the background. This may explain why there is an increase in total run-time not only for the parallel repository-based analysis strategy, but also for the parallel analysis strategy after reaching the optimum. Both parallel analysis strategies are also limited by the speed of the producer process and cannot pass the time required to find valid configurations and putting them in the shared buffer. Consequently, the best run-times of the two parallel-based analysis strategies are achieved when there is a balance between the producer and consumer processes. This balance is achieved if every valid configuration that the producer enters into the buffer is analyzed by a consumer directly and without waiting times. For the parallel repository-based analysis strategy, this balance is reached earlier than for the parallel analysis strategy, as the consumer processes operate faster when they have more analysis results available for potential reuse in the configuration repositories.

RQ 2: Does the parallelization of a naive and repository-based consistency analysis lead to additional run-time overhead?

In general, the results for measuring the total run-time presented in Section 5.3.1 show that no selected number of threads increases the overhead of the total run-time to the point that the run-time of the parallel or parallel repository-based analysis strategies is higher than of the naive consistency analysis. Although this is the case, in Section 5.3.2 we measured that the time needed for the initialization of processes and the time each process waits to put configurations in the buffer or get them out of it increases with each consumer process used in the consistency analysis. This is mainly due to the fact that when a certain number of consumer processes is reached, there is at least one process that has to wait for the producer process to put a new valid configuration in the shared buffer. Due to the configuration repositories, the consumer processes in the parallel repository-based analysis strategy operate faster than in the parallel analysis strategy, which explains why the spike in additional run-time overhead arises faster for at least versions 1.0 and 1.1 of the BCS case study.

RQ 3: Does the use of a configuration repository in a consistency analysis lead to additional memory usage?

The results show that the parallel repository-based analysis strategy uses significantly more memory than the naive one. This is the result of individual address spaces for the parent process and the producer and consumer processes initialized by this parent. These consumer processes, in turn, have their own configuration repositories and perform individual realizability analyses. This explains the huge difference in memory usage between the parallel repository-based analysis strategy and the naive one. For the repository-based analysis strategy, it shows that the strategy uses less memory for versions 1.0 and 1.1 but more memory for versions 3.0 and 3.1 of the BCS case study compared to the naive one. This variation can

be explained by the fact that although the relative speed-up in total run-time and thus the avoidance of realizability analyses remains similarly high across all versions, the absolute number of configuration repository entries grows substantially for versions 3.0 and 3.1. With this increased number of entries, the repository-based analysis strategy must also manage the more complex intersections of these entries. This results in an increase of memory usage, although realizability analyses are still avoided in the majority of cases. This trade-off between the size of the configuration repository and the number of reused intermediate analysis results leads to greater memory usage as the number of valid configurations in a CPPL increases.

5.5 Threats to Validity

The following sections discuss threats to the validity of the evaluation. Internal validity threats related to the setup and execution of the evaluation are addressed in Section 5.5.1. External validity threats related to the generalizability of the results are covered in Section 5.5.2.

5.5.1 Internal Threats to Validity

In our evaluation, we did not evaluate the correctness of our analysis strategies, which is a risk of incorrect results for consistency analysis. For our parallel analysis strategy, we assume that the implementation is working correctly, as it analyses each configuration in the same way as the naive analysis strategy. In addition, it results in the same number of configurations analyzed for each version of the BCS case study. For our repository-based analysis strategies, this is different because we modified the analysis procedure to avoid analyzing the realizability of each configuration by reusing previous analysis results. For our evaluation, the main concern is that if this modification is not working correctly, the measured results for the total run-time could be incorrect. This threat can be broken down into two general cases. In the first case, we achieve a lower run-time than possible because we find false positive entries for configurations in the configuration repository and thus avoid realizability analyses that should actually have been performed. In this case, one would have to assume that the entire data structure is corrupted, since analyses are avoided for configurations that do not have an entry in the configuration repository. We therefore rule out this case by assuming a functioning data structure. The second case is that we analyze the realizability of configurations, although this analysis could have been avoided by reusing previous analysis results. However, this would not lead to a lower total run-time, but to a higher one due to the execution of avoidable realizability analyses. This mitigates this risk, as no better run-time can be achieved than is actually possible.

In addition to these threats, we execute our procedures in a shared execution environment, which means that other tasks were performed in parallel to our evaluation. This should have no effect on the memory measurement and additional overhead for the parallel-based strategies but could have an effect on the measurements of total run-time. However, in all measurements of total run-time used for cross-strategy comparisons, the Coefficient of Variation (CV) is at most 0.04 (4 %) between different runs. This indicates a very low variability in run-time between repetitions. In addition, the Standard Deviation (SD) values are small compared to the total

run-times in each measurement, showing that the total run-time is consistent in different runs. Appendix A.4 contains tables illustrating the mean, SD and CV for each analysis strategy and each version of the BCS case study. Given this consistency, we can also conclude that five repetitions provide a sufficiently accurate estimate of the total run-time, with accuracy increasing only minimally with more repetitions.

5.5.2 External Threats to Validity

A major threat related to the external validity of our evaluation is that we have no possibility of determining the optimal number of consumer processes for the parallel-based analysis strategies before running a consistency analysis. Even if we determine this optimal number of consumer processes, this number is exclusive to the machine used to run the consistency analysis and the CPPL analyzed. It could be argued that for the parallel analysis strategy, it is always safe to assume that by using the maximum number of available consumer processes, the total run-time is reduced close to its minimum value. However, this assumption does not fully hold for the parallel repository-based analysis strategy because, as already argued, the repository of each consumer process becomes smaller the more consumer processes are used. A mitigation for this threat is that both analysis strategies significantly reduce the total run-time regardless of the used number of consumer processes.

The exclusive use of consistent adaptions of the BCS case study in our evaluation limits the generalizability of our results. We assume that all three analysis strategies would still result in shorter total run-times compared to the naive strategy when applied to inconsistent CPPLs. For the parallel-based analysis strategies, this assumption is based on the expectation that parallelization would accelerate consistency analysis similarly for both consistent and inconsistent CPPLs. In the case of the repository-based strategy, we anticipate that the speed-up may be smaller when fewer configurations are analyzed for realizability, as its effectiveness relies on the reuse potential within the configuration repository. Consequently, the difference in total run-time between the parallel and parallel repository-based analysis strategies could decrease in scenarios involving inconsistent CPPLs.

Another threat is whether the results of our evaluation in terms of performance improvements allow conclusions to be drawn about the scalability of the repository-based, parallel, and parallel repository-based analysis strategies for larger CPPLs. The results show that all three analysis strategies can significantly reduce total run-times compared to the naive analysis strategy. However, the results also show that the performance improvements do not scale linearly with the number of valid configurations in a CPPL. Although versions 3.0 and 3.1 of the BCS case study only almost double the number of valid configurations compared to versions 1.0 and 1.1, this leads to at least a fourfold increase in the overall run-times of the three analysis strategies. These findings suggest that while the three strategies significantly reduce total run-times compared to the naive analysis strategy, their scalability may be constraint in CPPLs that include more valid configurations.

6. Related Work

This chapter presents previous work relevant to the contributions of this thesis. We analyzed strategies for PL analysis and finding inconsistencies in CPPLs. In addition, we focused on analysis strategies that incorporate reuse and parallelization techniques.

Product Line Analysis

Thüm et al. [23] propose a classification of PL analyses by grouping them into product-based, family-based and feature-based analyzes. Product-based analyzes generate and analyze each individual product of a PL. Apel et al. [4] use a product-based analysis as a baseline and compare it with a family-based analysis strategy using six PLs as subject systems. They used the product-based analysis as a baseline because when not optimized, it quickly becomes infeasible with a growing number of possible products. Family-based analysis strategies analyze all possible products at once by operating on the shared artifacts of the PL without generating each individual product. Kästner et al. [10] developed a parser that can read code with compile-time feature options without the need to generate each version of a software product. Their approach builds a combined structure that includes all feature options, making it possible to check all possible products at once. Ochs et al. [15] proposed a product-based realizability analysis to analyze whether valid configurations in the problem space are also realizable in the solution space. The present thesis is based on this product-based realizability analysis, but extends the analysis to make it scalable for large CPPLs. The extensions include reuses from prior results during a realizability analysis and parallelizing the analysis.

Reusing Intermediate Analysis Results in Product Line Analysis

To improve scalability, several approaches have introduced mechanisms to reuse intermediate analysis results in PL analysis to avoid redundant computations. Sunderman et al. [22] improve scalability in the problem space by compiling FMs into a compiled structure that captures all valid configurations of the PL, allowing efficient reuse of intermediate results across multiple counting queries. These queries

determine how many valid configurations satisfy certain feature selections, avoiding redundant computation. In contrast, this thesis reuses analysis results in the solution space, where configuration realizability is checked for consistency between problem and solution space. Bubel et al. [6] propose proof repositories that store successful verification results for specific method implementations. In software PLs, this idea enables reuse of software component verification across product variants. The proof repository is used similar to our configuration repository for analyzing implementation artifacts in the solution space. However, although similar, the two differ in their use cases: Proof repositories store formal correctness proofs, whereas configuration repositories store resource demands of software components in different configurations.

Parallelization in Product Line Analysis

Horcas et al. [9] introduce FMSans, which eliminates cross-tree constraints to divide feature models into disjoint submodels that can be analyzed in parallel. While their strategy scales PL analysis in the problem space, the parallel strategy proposed in this thesis scales consistency analysis of CPPLs in the solution space. Kröher et al. [11] present KernelHaven a workbench able to perform modular analysis tasks as part of static software PL analysis in parallel. Unlike this thesis, which parallelizes the analysis in the solution space, KernelHaven applies parallelism more broadly. This means that independent analysis steps, such as processing variability models and code analysis, can be performed in parallel. This enables parallel PL analysis through pipeline-level concurrency rather than parallelizing the analysis of individual configurations.

7. Conclusion and Outlook

This thesis addresses the scaling of product-based consistency analysis for large CPPLs. Consistency analysis addresses the possible mismatch between modeled variability in the problem space and realizable variability in the solution space. Ochs et al. [15] proposed a product-based strategy to analyze these inconsistencies, but encountered the problem that this analysis strategy does not scale for large CPPLs. We proposed three individual analysis strategies to improve the scalability of product-based consistency analysis. In detail, these analysis strategies were:

1. Repository-based analysis strategy.
2. Parallel analysis strategy.
3. Parallel repository-based analysis strategy.

We describe the design and implementation of the (1) repository-based analysis strategy in Section 3.1 and Section 4.2.1. The analysis strategy is designed to enable the reuse of resource demands from software components within configurations that have been found to be realizable through consistency analysis. This reuse reduces the execution of realizability analysis to configurations where the resource demands are not a subset of previously analyzed ones. To store resource demands, we introduce a configuration repository that is a key value store, where keys represent the resource demands of individual software components and values represent the resource demands of valid and realizable configurations.

The design and implementation of the (2) parallel analysis strategy is described in Section 3.2 and Section 4.2.2. In this analysis strategy, we parallelize the consistency analysis over the set of valid configurations. We distribute these valid configurations via one producer process and analyze them for realizability with multiple consumer processes.

In Section 3.3 and Section 4.2.3 the design and implementation of the (3) parallel repository-based analysis strategy are described. This strategy unifies the concepts

of the (1) repository-based and (2) parallel analysis strategies. It does so by providing each consumer process with its own configuration repository. This allows reuse of analysis results and thus reduces redundant realizability analyses while simultaneously using the advantages of parallelization.

We evaluated our three analysis strategies in Chapter 5 along the evolution of the BCS case study as a subject system. The evaluation results show that all three analysis strategies significantly reduce total run-time compared to a naive analysis strategy with a speed-up of at least 4.74. Among the analysis strategies, the parallel repository-based analysis strategy performed best overall, benefiting both from the reuse of previous analysis results and parallelization. The evaluation also showed that, with a certain number of consumer processes used, the parallel-based analysis strategies introduce additional run-time overhead because the consumer processes have to wait for new configurations in the buffer. However, this run-time overhead does not increase to the extent that it outweighs the performance gains of both parallel-based analysis strategies. Regarding memory usage, the parallel repository-based analysis strategy uses the most memory, due to individual configuration repositories and address spaces across processes. Memory usage in the repository-based strategy also increases for larger CPPLs due to more entries in the configuration repository. Overall, all three analysis strategies offer significant performance benefits with comparatively small trade-offs in memory usage and parallelization overhead. With regard to the scalability of the three analysis strategies, the results of the evaluation show that they scale not linearly with respect to the number of configurations. This means that by significantly reducing the total run-time compared to the naive analysis strategy, we were able to shift the limit of feasible consistency analyses of large CPPLs, but could not eliminate it.

To further increase the scalability of the analysis strategies of this thesis, we propose to focus not only on the solution space, but also on the problem space in future work. The parallel analysis strategy could be improved by splitting the FM as done by Horcas et al. [9] and letting each consumer process analyze only a part of it. This would probably increase the achievable speed-up as the analysis strategy would not depend on one producer distributing the valid configurations. For the repository-based strategy, it could be worthwhile to analyze how the order in which valid configurations are drawn from the FM could be optimized to achieve as much reuse of the previous analysis results as possible early on in the consistency analysis. In addition, the repository-based analysis strategy would benefit from formal proofs verifying its correctness.

Bibliography

[1] 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: 9781457716386. DOI: 10.1109/ASE.2011.6100089. URL: https://doi.org/10.1109/ASE.2011.6100089.

[2] Sofia Ananieva et al. "A conceptual model for unifying variability in space and time". In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. SPLC '20. Montreal, Quebec, Canada: Association for Computing Machinery, 2020. ISBN: 9781450375696. DOI: 10.1145/3382025.3414955.

[3] Sven Apel et al. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013. ISBN: 3642375200.

[4] Sven Apel et al. "Strategies for product-line verification: Case studies and experiments". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 482–491. DOI: 10.1109/ICSE.2013.6606594.

[5] Per Brinch Hansen. *The origin of concurrent programming*. New York: Springer, 2002, pp. 65–138. ISBN: 978-0-387-95401-1. DOI: 10.1007/978-1-4757-3472-0.

[6] Richard Bubel et al. "Proof Repositories for Compositional Verification of Evolving Software Systems". In: *Transactions on Foundations for Mastering Change I*. Ed. by Bernhard Steffen. Cham: Springer International Publishing, 2016, pp. 130–156. ISBN: 978-3-319-46508-1. DOI: 10.1007/978-3-319-46508-1_8. URL: https://doi.org/10.1007/978-3-319-46508-1_8.

[7] National Science Foundation. Cyber-Physical-Systems (CPS).

[8] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[9] Jose-Miguel Horcas et al. "FMSans: An efficient approach for constraints removal and parallel analysis of feature models". In: *Journal of Systems and Software* 227 (2025), p. 112434. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2025.112434. URL: https://www.sciencedirect.com/science/article/pii/S0164121225001025.

[10] Christian Kästner et al. “Variability-aware parsing in the presence of lexical macros and conditional compilation”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 805–824. ISBN: 9781450309400. DOI: 10.1145/2048066.2048128. URL: https://doi.org/10.1145/2048066.2048128.

[11] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. “KernelHaven: an experimentation workbench for analyzing software product lines”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 73–76. ISBN: 9781450356633. DOI: 10.1145/3183440.3183480. URL: https://doi.org/10.1145/3183440.3183480.

[12] Sascha Lity et al. *Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study*. en. Tech. rep. Available online at https://www.isf.cs.tu-bs.de/cms/team/lity/bcs_tubs_tech_rep_V1_4.pdf; visited on December 17th, 2024. 2013.

[13] Michael Nieke, Christoph Seidl, and Sven Schuster. “Guaranteeing Configuration Validity in Evolving Software Product Lines”. In: *Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS ’16. Salvador, Brazil: Association for Computing Machinery, 2016, pp. 73–80. ISBN: 9781450340199. DOI: 10.1145/2866614.2866625.

[14] Linda M. Northrop and Paul C. Clements. “A framework for software product line practice, version 5.0”. In: *SEI* (2012).

[15] 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. Linz, Austria: Association for Computing Machinery, 2024, pp. 690–699. ISBN: 9798400706226. DOI: 10.1145/3652620.3687812.

[16] Philip Ochs. *Solution-Space-Realisation-Checker*. Available at https://github.com/KIT-TVA/solution-space-realisation-checker; accessed May 2, 2025.

[17] Klaus Pohl and Günter Böckle. *Software product line engineering*. Springer, 2005. ISBN: 9783540243724. URL: http://www.loc.gov/catdir/enhancements/fy0663/2005929629-d.html.

[18] Python Software Foundation. *multiprocessing — Process-based parallelism*. Available at https://docs.python.org/3/library/multiprocessing.html; accessed April 15, 2025.

[19] Python Software Foundation. *tracemalloc — Trace memory allocations*. Available at https://docs.python.org/3/library/tracemalloc.html; accessed May 2, 2025.

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

[21] Sascha Lity Sophia Nahrendorf and Ina Schaefer. *Applying Higher-Order Delta Modeling for the Evolution of Delta-Oriented Software Product Lines*. en. Tech. rep. Available online at https://github.com/TUBS-ISF/BCS-Case-Study-Full/blob/master/000_documentation/Nahrendorf%2C%20Lity%2C%20Schaefer_2018_Applying%20Higher-Order%20Delta%20Modeling%20for%20the%20Evolution%20of%20Delta-Oriented%20Software%20Product%20Lines.pdf; visited on April 15th, 2025. TU Braunschweig-Institute for Software Engineering and Automotive Informatics, 2018.

[22] Chico Sundermann et al. "Reusing d-DNNFs for Efficient Feature-Model Counting". In: *ACM Trans. Softw. Eng. Methodol.* 33.8 (Nov. 2024). ISSN: 1049-331X. DOI: 10.1145/3680465. URL: https://doi.org/10.1145/3680465.

[23] Thomas Thüm et al. "A Classification and Survey of Analysis Strategies for Software Product Lines". In: *ACM Comput. Surv.* 47.1 (June 2014). ISSN: 0360-0300. DOI: 10.1145/2580950. URL: https://doi.org/10.1145/2580950.

[24] 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. Graz, Austria: Association for Computing Machinery, 2022, pp. 94–98. ISBN: 9781450392068. DOI: 10.1145/3503229.3547059.

A. Appendix

A.1 Body Comfort System

A.1.1 Version 1.1

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware`(hw_0)	$rp_0^0 = 8$ $rp_1^0 = 25$ $rp_6^0 = 4$ $rp_7^0 = 128$
`security-hardware`(hw_1)	$rp_1^2 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$
`safety-hardware`(hw_2)	$rp_3^2 = 10$ $rp_6^2 = 4$

Table A.1: Resource Provisions of Version 1.1.

A.1.2 Version 2.0

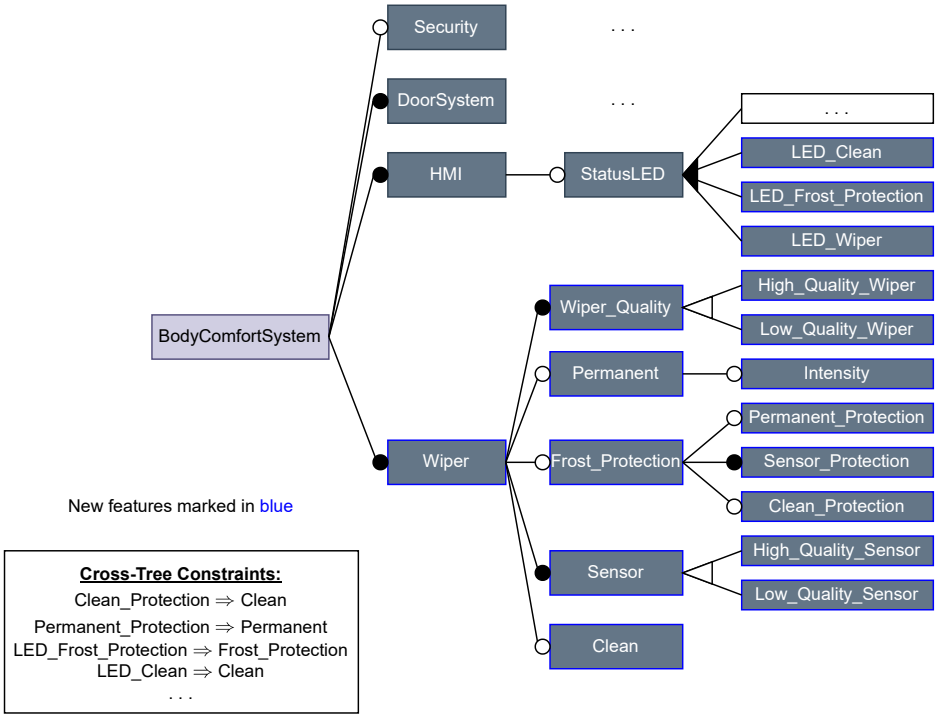

Figure A.1: FM of Sliced Out Features in Version 2.0.

k	isAdditive	isExclusive	boundary	Unit	Description
8	True	True	LOWER		no. Pumps
9	True	False	LOWER	KB	no. Temperature Sensors
10	True	False	LOWER	KB	no. Liquid Sensors

Table A.2: Resource Types Added in Version 2.0.

Feature	Software Components sw_i	Resource Demands
LED_Wiper	sw_0	$rd_0^0 = 1$
LED_Frost_Protection	sw_0	$rd_0^0 = 1$
LED_Clean	sw_0	$rd_0^0 = 1$
Clean	sw_4	$rd_8^4 = 1$ $rd_{10}^4 = 1$
Frost_Protection	sw_4	$rd_9^4 = 1$
Low_Quality_Wiper	sw_4	$rd_1^4 = 10$ $rd_7^4 = 5$
High_Quality_Wiper	sw_4	$rd_1^4 = 10$ $rd_7^4 = 10$
Low_Quality_Sensor	sw_4	$rd_7^4 = 5$
High_Quality_Sensor	sw_4	$rd_7^4 = 10$

Table A.3: Sliced Out Features With Their Resource Demands Added in Version 2.0.

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware`(hw_0)	$rp_0^0 = 8$ $rp_1^0 = 35$ $rp_7^0 = 128$
`security-hardware`(hw_1)	$rp_1^2 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$
`safety-hardware`(hw_2)	$rp_3^2 = 10$ $rp_6^2 = 4$ $rp_8^2 = 0$ $rp_9^2 = 0$ $rp_{10}^2 = 0$

Table A.4: Nulled Resource Provisions of Version 2.0.

A.1.3 Version 3.0

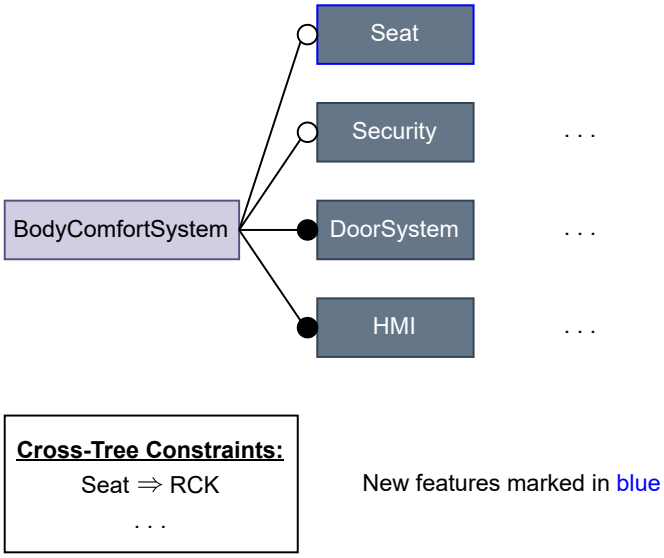

Figure A.2: FM of Version 3.0.

k	isAdditive	isExclusive	boundary	Unit	Description
11	True	True	LOWER		no. Motors
12	True	False	LOWER	KB	Long Term Memory

Table A.5: Resource Types Added in Version 3.0.

Feature	Software Components sw_i	Resource Demands
Seat	sw_5	$rd_7^5 = 5$ $rd_{11}^5 = 1$ $rd_{12}^5 = 2$

Table A.6: Resource Demands Added in Version 3.0.

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware`(hw_0)	$rp_0^0 = 8$ $rp_1^0 = 35$ $rp_7^0 = 128$ $rp_{11}^0 = 1$ $rp_{12}^0 = 1$
`security-hardware`(hw_1)	$rp_2^1 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$
`safety-hardware`(hw_2)	$rp_3^2 = 10$ $rp_6^2 = 4$

Table A.7: Resource Provisions of Version 3.0.

A.1.4 Version 3.1

k	isAdditive	isExclusive	boundary	Unit	Description
13	True	False	UPPER	ms	Response Time

Table A.8: Resource Types Added in Version 3.1.

Feature	Software Components sw_i	Resource Demands
Seat	sw_2	$rd_6^2 = 1$ $rd_{13}^2 = 2$

Table A.9: Resource Demands Added in Version 3.1.

Hardware Component (hw_j)	Provisions (rp_k^j)
`infotainment-hardware`(hw_0)	$rp_0^0 = 8$ $rp_1^0 = 35$ $rp_7^0 = 128$ $rp_{11}^0 = 1$ $rp_{12}^0 = 1$
`security-hardware`(hw_1)	$rp_2^1 = 16$ $rp_3^1 = 700$ $rp_4^1 = 1$ $rp_5^1 = 4$
`safety-hardware`(hw_2)	$rp_3^2 = 10$ $rp_6^2 = 4$ $rp_{13}^2 = 2$

Table A.10: Resource Provisions of Version 3.1.

A.2 Repository-Based Implementation

```
1  R ordered resource types (input constants)
2  RD resource demands (input constants)
3  SW software components (input constants)
4  Repo configuration repository (input)
5
6  function check_repo(R, RD, SW, Repo):
7      CD ← empty set of configuration demands
8      intersections ← empty set for shared set identification
9
10     for i in range(|SW|):
11         key ← ε
12         for k in range(|R|):
13             get rd_k^i from RD
14             key ← concatenate(key, *, rd_k^i)
15
16         if key is not part of a tuple in CD:
17             add (key, 1) to CD
18         else:
19             get tuple (key, n) from CD
20             (key, n) ← (key, n + 1)
21
22         if the key has no entry in Repo:
23             create entry with key as key and an empty set as value
24
25         get value addressed by key from Repo and add it to intersections
26
27     intersections ← intersect the elements of intersection
28
29     if intersections = ∅:
30         analysis ← true
31     else if CD ∉ intersections:
32         for each set in intersections:
33             analysis ← false
34             for each tuple (k,n) in (CD \ set)
35                 if a tuple (k,u) exists in CD with (u > n):
36                     analysis ← true
37             if ¬analysis:
38                 break
39     else:
40         analysis ← false
41
42     if analysis:
43         add CD to entry of each constructed key key in Repo
44
45     return analysis
```

Listing A.1: Using a Configuration Repository to Decide Whether a Realization Analysis Has to Be Performed for a Given Configuration.

A.3 Worst-Case Run-Times

Worst-Case Run-Time Comparison

Strategy	V1.0	V1.1	V3.0	V3.1
Naive	∼ 533.21 s	∼ 767.57 s	∼ 3406.76 s	∼ 3648.66 s
Parallel (2 Threads.)	∼ 491.35 s	∼ 729.79 s	∼ 3236.77 s	∼ 3458.57 s
Par. Repo-Bsd. (2 Thrds.)	∼ 71.64 s	∼ 103.09 s	∼ 449.46 s	∼ 483.04 s

Table A.11: Worst-Case Run-Time Comparison.

A.4 Mean, Standard Deviation and Coefficient of Variation

We provide the mean, SD, and CV for the total run-time measurements for each version of the BCS case study and all evaluated analysis strategies. For the parallel and parallel repository-based analysis strategies, we provide them for the numbers of threads used for cross-strategy comparisons.

Version 1.0

Strategy	Mean [s]	SD [s]	CV
Naive	∼ 533.81	∼ 5.29	∼ 0.01
Repository-Based	∼ 113.13	∼ 1.97	∼ 0.02
Parallel (21 Threads)	∼ 55.22	∼ 2.20	∼ 0.04
Parallel Repo-Based (12 Threads)	∼ 49.89	∼ 0.80	∼ 0.02

Table A.12: Mean, SD and CV for version 1.0.

Version 1.1

Strategy	Mean [s]	SD [s]	CV
Naive	∼ 766.29	∼ 3.59	∼ 0.00
Repository-Based	∼ 143.47	∼ 3.48	∼ 0.02
Parallel (21 Threads)	∼ 61.03	∼ 1.60	∼ 0.03
Parallel Repo-Based (12 Threads)	∼ 48.21	∼ 1.10	∼ 0.02

Table A.13: Mean, SD and CV for version 1.1.

Version 3.0

Strategy	Mean [s]	SD [s]	CV
Naive	∼ 3418.42	∼ 32.75	∼ 0.01
Repository-Based	∼ 577.00	∼ 4.30	∼ 0.01
Parallel (21 Threads)	∼ 230.95	∼ 7.94	∼ 0.03
Parallel Repo-Based (12 Threads)	∼ 196.55	∼ 1.34	∼ 0.01

Table A.14: Mean, SD and CV for version 3.0.

Version 3.1

Strategy	**Mean** [s]	**SD** [s]	**CV**
Naive	∼ 3655.30	∼ 10.01	∼ 0.00
Repository-Based	∼ 607.66	∼ 3.61	∼ 0.01
Parallel (21 Threads)	∼ 248.32	∼ 6.52	∼ 0.03
Parallel Repo-Based (12 Threads)	∼ 210.50	∼ 0.89	∼ 0.00

Table A.15: Mean, SD and CV for version 3.1.