Design Space Evaluation for Confidentiality under Architectural Uncertainty

Bachelor's Thesis of

Oliver Liu

at the Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Sebastian Hahner
Second advisor:	M.Sc. Maximilian Walter

10. May 2021 – 10. September 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 10th September 2021

. . . .

(Oliver Liu)

Abstract

In the early stages of developing a software architecture, many properties of the final system are yet unknown, or difficult to determine. There may be multiple viable architectures, but uncertainty about which architecture performs the best. Software architects can use Design Space Exploration to evaluate quality properties of architecture candidates to find the optimal solution.

Design Space Exploration can be a resource intensive process. An architecture candidate may feature certain properties which disqualify it from consideration as an optimal candidate, regardless of its quality metrics. An example for this would be confidentiality violations in data flows introduced by certain components or combinations of components in the architecture. If these properties can be identified early, quality evaluation can be skipped and the candidate discarded, saving resources.

Currently, analyses for identifying such properties are performed disjunct from the design space exploration process. Optimal candidates are determined first, and analyses are then applied to singular architecture candidates. Our approach augments the PerOpteryx design space exploration pipeline with an additional architecture candidate filter stage, which allows existing generic candidate analyses to be integrated into the DSE process. This enables automatic execution of analyses on architecture candidates during DSE, and early discarding of unwanted candidates before quality evaluation takes place.

We use our filter stage to perform data flow confidentiality analyses on architecture candidates, and further provide a set of example analyses that can be used with the filter. We evaluate our approach by running PerOpteryx on case studies with our filter enabled. Our results indicate that the filter stage works as expected, able to analyze architecture candidates and skip quality evaluation for unwanted candidates.

Zusammenfassung

In den frühen Phasen der Entwicklung einer Softwarearchitektur sind viele Eigenschaften des finalen Systems noch unbekannt und schwer zu ermitteln. Es kann mehrere mögliche Softwarearchitekturen geben, aber es ist ungewiss, welche Architektur die beste Entscheidung ist. Softwarearchitekten können Entwurfsraumerkundung (engl. *Design Space Exploration*) nutzen, um Qualitätsmetriken einzelner Architekturkandidaten zu ermitteln und den optimalen Kandidaten zu wählen.

Entwurfsraumerkundung ist ein ressourcenintensiver Prozess. Ein Architekturkandidat kann bestimmte Eigenschaften aufweisen, die es direkt von weiterer Erwägung als optimalen Kandidaten ausschließt, unabhängig von seinen Qualitätsmetriken. Ein Beispiel solcher Eigenschaften sind Vertraulichkeitsverletzungen, die durch die Nutzung bestimmter Komponenten oder Kombinationen von Komponenten in einer Architektur eingeführt werden. Sollten diese Eigenschaften frühzeitig identifiziert werden können, kann die Ermittlung der Qualitätsmetriken des Architekturkandidaten übersprungen werden und der Kandidat verworfen werden. Dies kann wertvolle Ressourcen einsparen.

Zurzeit können solche Analysen wie die Vertraulichkeitsanalyse nur getrennt von dem Entwurfsraumerkundungsprozess durchgeführt werden. Zunächst werden optimale Kandidaten ermittelt, und später erst einzeln analysiert. Unser Ansatz erweitert PerOpteryx um einen zusätzlichen Filter, der es erlaubt existierende generische Analysen auf Architekturkandidaten im Entwurfsraumerkundungsprozess durchzuführen. Dadurch wird das frühzeitige Verwerfen von ungeeigneten Kandidaten vor der Ermittlung von Qualitätsmetriken ermöglicht.

Wir nutzen diesen Filter um eine Vertraulichkeitsanalyse in den Entwurfsraumerkundungsprozess zu integrieren, und stellen weitere Beispielanalysen bereit, die in dem Filter genutzt werden können. Wir evaluieren unseren Ansatz indem wir PerOpteryx mit unserem Filter auf Fallstudien ausführen. Unsere Ergebnisse deuten an, dass unser Filter wie erwartet funktioniert und Architekturkandidaten analysieren und verwerfen kann, und die Ermittlung von Qualitätseigenschaften für ungeeignete Kandidaten überspringt.

Contents

Abstract **i**

Zusammenfassung **iii**

1 Introduction **1**

2 Foundations **3**

2.1 Palladio Component Model 3

2.2 Design Space Exploration 4

3 State of the Art **7**

3.1 Design Space Exploration 7

3.2 Confidentiality Analysis 8

3.3 Uncertainty 8

4 Running Example **11**

5 Concept of the Candidate Filter **15**

5.1 Overview 15

5.2 Problem Statement 15

5.3 Identification of Suitable Entry Points for Filtering 18

5.4 Defining the Filter Interface 21

5.5 Implementing the Filter Interface 22

5.6 Running PerOpteryx with the Filter 24

5.7 Alternatives 25

6 Evaluation **27**

6.1 Goals 27

6.2 Design 28

6.3 Results 30

6.3.1 Goal G1 30

6.3.2 Goal G2 30

6.4 Threats to Validity 32

6.5 Limitations 33

7 Lessons Learned **35**

8 Conclusion **37**

8.1 Summary 37

8.2 Future Work 37

Bibliography **39**

List of Figures

2.1 Main parts of a CBSA model. Adapted from [12, p. 16] 3
2.2 Visualization of a three-dimensional design space. Adapted from [12, p. 105] 4

4.1 Simplified system architecture of the online shop. 11
4.2 System architecture of the online shop with the component selection DoF. 11
4.3 System architecture of the online shop with the component allocation DoF. 12
4.4 Allowed and forbidden configurations for the database considering confidentiality. 13

5.1 Mapping of candidates to subsets of all possible candidates 17
5.2 Comparison of approaches to combining optimization with filtering. . . 17
5.3 Steps and program flow of the PerOpteryx pipeline. From [13, Fig. 2] . . 19
5.4 Actual classes and methods used in PerOpteryx pipeline. 20
5.5 Activity Diagrams for the `evaluate` method of the `DSEEvaluator` class where our entry point is located. 20
5.6 Simplified class diagram of the filter plugin. 21
5.7 The `FilteringAnalysis` interface. 23
5.8 Activity Diagram for the confidentiality analysis implementation of the `FilteringAnalysis` interface. 23

8.1 The Run Configuration Dialog for a Design Space Exploration, with the Performance Analysis Tab selected. 38
8.2 Mockup of a possible analysis selection dialog. 38

List of Tables

5.1 Component selection for candidates A, B and C with their performance and confidentiality metrics. 16

6.1 DSE launch configurations for the case studies 29
6.2 Results of a DSE run with and without our filter stage. 31

1 Introduction

In the early stages of developing a software architecture, many properties of the final system are unknown, or difficult to determine. The impact of certain architectural decisions during the design phase may have severe impacts on the performance or other qualitative aspects of the system at run time. This missing knowledge leads to uncertainty in regard to the software architecture, or architectural uncertainty in short [28, p. 4]. With architectural uncertainty, multiple software architectures may seem equally viable, or the cost-benefit trade-off of choosing one component over the other may not be clear until the system is deployed. At that point however, changing architectural decisions may be costly and time-consuming and therefor infeasible. Component-based software architectures enable software architects to model software systems through the use of components and the definition of their relations to one another. Different software architectures can be modeled early in the development process, such as during the architecture design phase. Through the explicit definition of quality properties of various components, such as their performance and reliability, early insights into the run time properties of architectures can be gained through model simulation. By comparing the metrics of the architectures, software developers can identify the optimal candidate, reducing the uncertainty about the software architecture.

The primary focus of this thesis is to add the property of confidentiality into this model evaluation process. While performance and cost-effectiveness are amongst the traditional aspects of software architectures that are evaluated during the design phase to determine the best possible architecture candidate, confidentiality is often neglected [27]. If confidentiality is not considered when determining the optimal architecture, the system may not perform ideally when retrospectively modified by replacing confidentiality-violating components, compared to if confidentiality was explicitly modeled and analyzed during the design phase.

PerOpteryx [5] is a framework to enable the optimization of system architectures in regard to quality metrics like performance, reliability and cost. In this thesis, we extend the functionality of PerOpteryx with a filter stage in which architecture candidates can be analyzed and discarded. This can be used to remove candidates with certain unwanted properties early in the optimization process. Using this filter, we can analyze architecture candidates for confidentiality violations and discard violating ones. We provide a generic interface to allow the implementation of other arbitrary analyses on architecture candidates and provide examples for further such analyses.

We evaluate our approach by running PerOpteryx on case studies with our filter enabled. By applying candidate model analyses such as the confidentiality analysis with our filter, we determine whether such analyses can be successfully performed. Furthermore, we analyze whether our filter can accurately discard candidates, based on the results of candidate model analyses. Our results indicate that the filter stage works as intended, able to perform architecture candidate model analyses and appropriately discard unwanted candidates.

This thesis is structured into 8 chapters. In this chapter, we have introduced the topic and motivated the problem. In Chapter 2 we explain the foundations for our work in detail. This includes the Palladio Component Model and the Design Space Exploration process. The current state of the art for design space exploration, confidentiality analyses and uncertainty in software architectures is discussed in Chapter 3. We present a running example in Chapter 4, based on which we will illustrate the functionality of our work in subsequent chapters. In Chapter 5 we expand on the concept and implementation details of our filter stage. Furthermore, we illustrate the problem of traditional approaches which append candidate analyses after the DSE process, and justify our decision regarding the location of the entry point for the filter stage. Additionally, we present some possible alternatives for accomplishing the filter function, and our justifications against using them. We evaluate our work in Chapter 6, and describe valuable lessons learned throughout this thesis in Chapter 7, which may be important for future works in this area. Chapter 8 wraps up this thesis with a conclusion and a discussion regarding possible future work.

2 Foundations

In this chapter, we introduce the foundations required for this thesis. In Section 2.1 we present the Palladio Component Model and component-based software architectures. Section 2.2 explains the Design Space Exploration process.

2.1 Palladio Component Model

Components are the building blocks of a component-based software system [24, p. 47]. Taylor et al [29, p. 69] define a software component as "an architectural entity that encapsulates a subset of the system's functionality and/or data". Each component has their own functionality, and can provide this functionality for other components to use via explicitly defined interfaces. With component-based software architectures, the development process for a software system is split into component development and the design of the software architecture by assembling components.

The main parts of a component-based software architecture (CBSA) model are shown in Figure 2.1. The *repository* is system independent and contains data types, interfaces and components [24, p. 45]. The *component assembly* specifies the inner structure of a specific system, i.e. how components are used and connected to each other [24, p. 49, 12, p. 15]. The *resource environment* defines resource containers (such as servers) on which components can be deployed. The *allocation* maps components to a resource container.

The Palladio approach allows for the definition of component-based software architectures "with a special focus on performance properties" [24, p. 38]. The Palladio Component Model (PCM) is a meta model for component-based software architectures [26] and is used to document software architectural knowledge for the Palladio approach [24, p. 11]. Using PCM, components can be annotated with performance properties, allowing developers the ability of specifying run-time characteristics of components at design-time. Using this information, Palladio can simulate a modeled system, deriving quality metrics for the

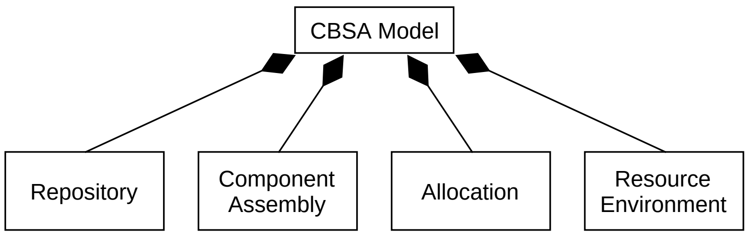


Figure 2.1: Main parts of a CBSA model. Adapted from [12, p. 16]

given architecture [24, p. 9]. This gives developers insights into the performance of the system through metrics such as response time and throughput.

The Palladio approach explicitly distinguishes between a component developer and a software architect role in the development process [24, p. 12]. This means that a software architect does not need to concern themselves about the inner workings of components, and can regard a component as a black-box [16, p. 80, 29, p. 69]. This abstraction enables software developers to apply changes to the software architecture, e.g. by substitution of components for alternatives with the same set of interfaces.

2.2 Design Space Exploration

For a software architecture, a location in the model where a change of a design choice can be made to form a new valid architecture configuration is called a *degree of freedom*. We adopt this terminology from Koziolek et al. [12] and will use it throughout this thesis. An example for a degree of freedom is the substitution of a component, or the allocation of a component to a different resource environment. Multiple degrees of freedom span a *design space* [26]. In other words, the design space is the set of all possible architecture models, each defined by selecting one specific design choice for every degree of freedom. Each valid architecture model is also called a *candidate model* [12, p. 104].

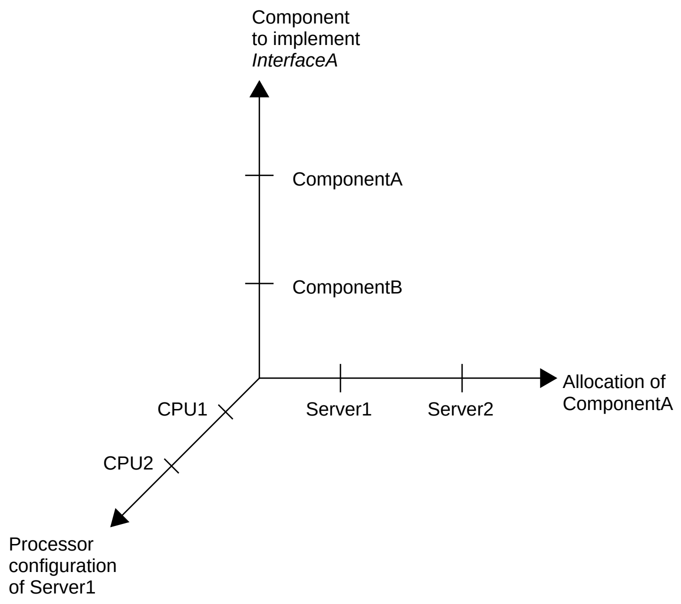


Figure 2.2: Visualization of a three-dimensional design space. Adapted from [12, p. 105]

The Design Space Exploration (DSE) process is used to assess each candidate model in the design space on certain properties of interest, such as their performance. PerOpteryx is a tool to perform Design Space Exploration on Palladio models [5, 24, p. 162], optimizing architecture candidates for their quality properties, like performance, reliability and cost [5]. It leverages the feature of Palladio models which allows developers to specify quality attributes for software components. PerOpteryx is able to generate new software architecture candidates from an initial PCM model and evaluate their quality properties to find the architecture with the optimal metrics. Specifically, PerOpteryx identifies the Pareto front, i.e. the set of all Pareto optimal candidate models. An architecture candidate is called Pareto optimal if it is not worse than a different candidate [24, p. 159]. For the architecture candidate generation, we will utilize certain terminology synonymously throughout this thesis. The algorithm that PerOpteryx uses for the generation distinguishes between so-called *genotypes* and *phenotypes*. A genotype is a vector of design choices for each degree of freedom, i.e. an architecture candidate. A phenotype is the corresponding architecture candidate *model* [5, 12, p. 177].

3 State of the Art

In this chapter, we discuss the current state of the art for research areas related to our thesis. We give an overview of different approaches to design space exploration in Section 3.1. In Section 3.2, we examine approaches for confidentiality analyses of software architectures. Finally, we discuss uncertainty in software development in Section 3.3.

3.1 Design Space Exploration

There are numerous approaches for performing design space exploration for software architectures. The similarity of all approaches is to evaluate metrics of alternative architecture candidates so that these can be put in comparison with each other to identify the best candidate. Sobhy et al. [28] present a literature review that gives an overview of various approaches to perform design space exploration on software architectures under uncertainty. In it, the continuous evaluation of the system architecture is addressed. Continuous evaluation is the repeated application of software architecture evaluations throughout the development phases of a software system. We focus on the design time architecture evaluation approaches, but the review also elaborates on run time architecture evaluation and the linkage between both.

The study distinguishes the different architecture evaluation approaches mainly by their quality evaluation method, i.e. what technique the approach uses to evaluate architecture design decisions. PerOpteryx [5] is categorized in this work as a search-based approach for design-time architecture evaluation. Search-based approaches use metaheuristic search techniques such as genetic algorithms to find the optimal architecture candidate.

ArcheOpterix [1] is an example for another approach using this technique. ArcheOpterix also performs multi-criteria optimization and uses Pareto optimality to identify the best candidates, making the approach very similar to PerOpteryx [12, p. 76]. However, ArcheOpterix provides a less flexible way of defining new degrees of freedom or optimization problems, requiring more effort and deep knowledge from the software architect [12, p. 75].

GuideArch [7] is an architecture framework which uses fuzzy mathematics in the ranking of candidates. Rather than specifying exact properties of a software component, they are given as a range of values or as an enumeration in comparison with other alternatives (i.e. *low* / *high*). Impacts of architecture decisions on quality properties can be roughly estimated early on, but are fairly imprecise. As more information becomes available, these estimations can be made more precise. The optimal candidate is determined by comparing the fuzzy quality metrics and selecting the best candidate. GuideArch is classified as a utility-based approach [28], since it uses fuzzy math as a utility function tool for measuring quality attributes. However, confidentiality is not considered in this approach.

3.2 Confidentiality Analysis

Seifermann et al. [27] coin the term *Data-Driven Software Architecture* to describe data and data processing on an architectural level. They introduce data flows to the Palladio architectural description language [24], so that data processing can be modeled in the architecture during the design phase. Using this, software developers can define confidentiality constraints. By transforming the software architecture to a Prolog logic program, violations of the confidentiality constraints can be found through queries to the program. For the confidentiality analysis, the software architecture can, for example, be annotated with access rights for data and roles for data processing operations. Confidentiality constraints specify which data are available to which roles. Data processing operations may modify the access rights of data, leading to possible constraint violations. This confidentiality analysis can however only be performed on singular software architectures, and is not available for design space exploration processes.

Berger et al. [3] present an approach to detect security threats in software architectures by analyzing data flow diagrams for architectural flaws. They extend regular data flow diagrams with additional semantics to enable automated analyses of security threats. A catalog of common security flaws is provided, together with a tool to analyze the data flow diagrams for the existence of such flaws. This approach is similar to the confidentiality analysis of Seifermann et al. [27] in that it also uses data flow diagrams in the flaw analysis, however it provides a general approach for analyzing the security of software architectures, and does not focus on confidentiality.

3.3 Uncertainty

Uncertainty during software development can manifest itself in different ways and in different locations. Perez-Palacin and Mirandola [20] introduce a taxonomy of uncertainty in software architecture models to enable classification of various types of uncertainty. Uncertainty is classified based on the three dimensions of *location*, *level* and *nature* of the uncertainty.

The *location* of uncertainty describes where the uncertainty manifests itself in the model. There are three possible locations described, namely context uncertainty, model structural uncertainty and input parameter uncertainty. Context uncertainty concerns uncertainty about which information of a system should be modeled and which abstracted. Model structural uncertainty describes uncertainty about the accuracy of the model structure compared to the real world system. Input parameter uncertainty is uncertainty about the values and data given as input to the model.

The second dimension of uncertainty regards the *level* of uncertainty. This is classified in five orders. Order 0 represents a lack of uncertainty, meaning no uncertainty is present. The first order of uncertainty describes known uncertainty, meaning a lack of knowledge, but awareness of said lack. The second order is a lack of knowledge and a lack of awareness. However, it is possible to become aware of the existence of uncertainty. Order 3 lacks knowledge, awareness, and the possibility of becoming aware of uncertainty. The fourth order of uncertainty describes meta uncertainty, i.e. uncertainty about the order of uncertainty.

The *nature* of uncertainty is categorized into epistemic and aleatory uncertainty. Epistemic uncertainty is caused by a lack of sufficient data or knowledge, while aleatory uncertainty is caused by randomness of events.

Bures et al. [4] examine the impact of uncertainty on the security of access control systems. An approach for adaptation strategies for access control systems is given, which take the uncertainty of new and not fully foreseen situations into account. The system optimizes the access control rules for the minimization of security risk and financial loss. Bures et al. adapt the uncertainty taxonomy of Perez-Palacin and Mirandola [20], replacing the dimension of the location of uncertainty with the *source* of uncertainty. Uncertainty may be found in the system structure, system behavior and system environment. While the approach of Bures et al. also tackles uncertainty that may occur in a software system, it focuses on run time uncertainty and not design time uncertainty. Additionally, the uncertainty in their approach is of aleatory nature, i.e. caused by random events, while architectural uncertainty emerges due to the lack of information, i.e. epistemic uncertainty.

4 Running Example

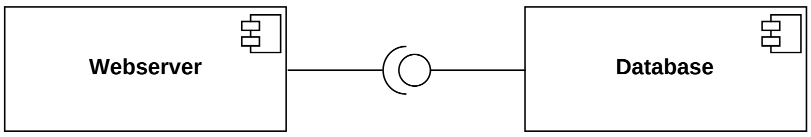


Figure 4.1: Simplified system architecture of the online shop.

To more illustratively convey our contributions throughout this thesis, we introduce a simple example software system that will serve as a basis for explanations in later sections. Figure 4.1 illustrates the simplified system architecture of an Online Shop serving customers in the European Union. It consists of two parts: on the left is the *Webserver* component that handles interactions with the customer, and on the right is a *Database* that saves information about the customers, the products and the warehouse inventory. The Webserver component may receive *personalized* data from the customer, such as their delivery address or credit card information. This data is transmitted to the Database component to be stored. A user of the Online Shop may also request information about a certain product that the shop offers, such as the current warehouse stock. In that case, *non-personalized* data would be requested from the database and sent to the Webserver component.

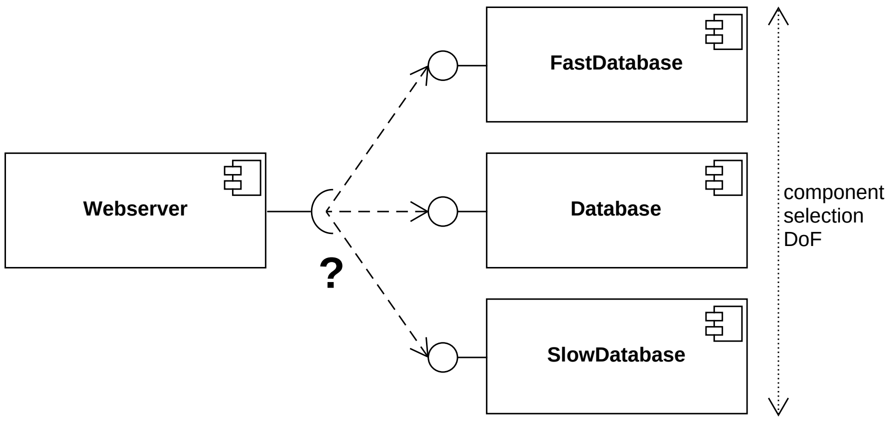


Figure 4.2: System architecture of the online shop with the component selection DoF.

Since we have a component-based software architecture, we can define alternative database components, provided they implement the same interface. Furthermore, we assume that our system architecture is modeled with Palladio, meaning that we can define component properties like performance and cost. Based on these facts, we add two alternative components for the database component to the system model, shown in Figure 4.2. These are **FastDatabase** and **SlowDatabase**, and they respectively feature higher and lower performance compared to the default database component.

With these two additional alternative components, we define the *component selection* for the Database component as a possible degree of freedom (DoF) for this system architecture, as indicated by the arrow in Figure 4.2. The question mark signifies architectural uncertainty about the component choice for the database. With only this DoF, our software system would have three possible architecture candidate models, also called *phenotypes* [12, p. 177]. Each phenotype would use a different database component.

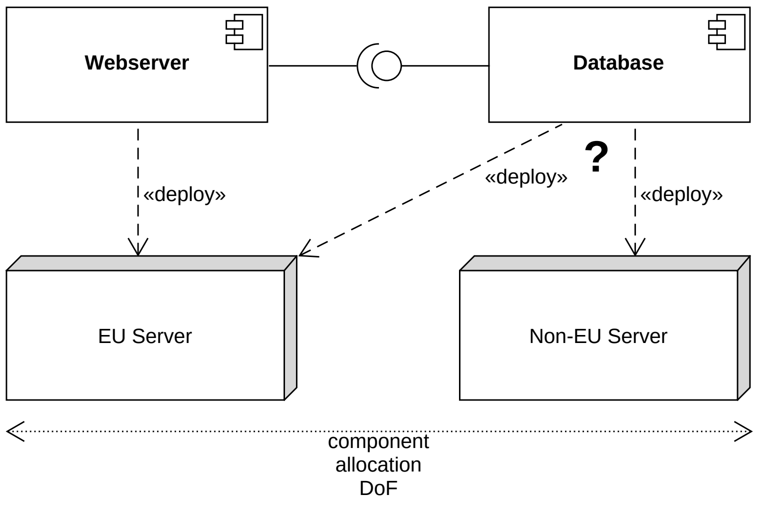


Figure 4.3: System architecture of the online shop with the component allocation DoF.

We can formulate an additional degree of freedom in this system. Figure 4.3 shows the component allocation degree of freedom. The database component can be allocated to a certain server, which may be a server located in the EU, or one located outside of the EU. Since the online shop is made available for EU citizens, it must conform to data privacy laws set forth in the General Data Protection Regulation (GDPR). Sending personalized and/or confidential data such as the aforementioned delivery address to a server outside the EU may violate articles 44-46 of the GDPR [8]. This circumstance is shown in Figure 4.4. The transmission of non-personalized data between the webserver and the database is allowed regardless of the location of the database server, as is the transmission of personalized data to a server located inside the EU. However, the combination of the database server being located outside the EU together with the transmission of personalized data would be considered a violation of data privacy and therefor forbidden. However, using a European server may for example cost more, or have lower performance. This illustrates uncertainty about the deployment location for the database component, which is denoted by the question mark in Figure 4.3. In addition, this shows the multi-objective optimization problem with the online shop architecture: finding the optimal architecture candidate with best cost-performance ratio, while still upholding data confidentiality.

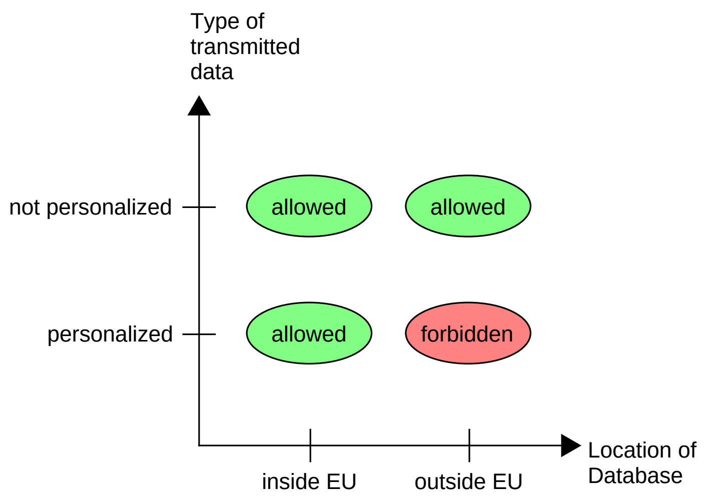


Figure 4.4: Allowed and forbidden configurations for the database considering confidentiality.

5 Concept of the Candidate Filter

In this chapter, we will go over the concept of the candidate filter, and discuss details on why, where and how this filter is implemented. In Section 5.1 we give an overview of the concept behind the filter. Section 5.2 illustrates a problem with traditional analyses of candidate models and details how our approach with a candidate filter remedies this problem. Since we extend the traditional PerOpteryx process pipeline with our filter stage, we justify our decision on the exact location of this extension in Section 5.3. In Section 5.4 we illustrate the structure of the candidate filter, and then discuss the implementation details in Section 5.5. In Section 5.6, we explain how we integrate analyses with the filter into PerOpteryx. Finally, we present alternative approaches for realizing a candidate filter that we've explored in Section 5.7, and explain our reasoning for not choosing these for our actual implementation.

5.1 Overview

During the normal process of the PerOpteryx design space exploration (DSE), new architecture candidates are generated using an evolutionary algorithm [15]. These candidate models are subsequently evaluated for their quality properties. This model evaluation is the most resource- and time-intensive part of the DSE process. A candidate may exhibit certain undesired features that can be identifiable early in the DSE process. We could save valuable resources by discarding such candidates prior to the model evaluation. To accomplish this, an additional filter stage could be added to the DSE process, which analyses the properties of a candidate model and discards unwanted candidates early.

Traditionally, our running example of the confidentiality analysis introduced in Chapter 4 would be applied on the architecture candidates returned after a completed DSE run. This means that all such architecture candidates that violate confidentiality would be unnecessarily evaluated, since a confidentiality violation automatically disqualifies the candidate from further consideration by the software architect. We could apply the confidentiality analysis on architecture candidates before the model evaluation phase, however this is not supported in the traditional PerOpteryx process. Our proposed filter stage would facilitate analyses like the confidentiality analysis before model evaluation. This early filtering would eliminate the overhead resulting from unnecessary evaluations of undesired candidates.

5.2 Problem Statement

To describe the problem that may occur with a traditional workflow, which first determines optimal candidates with PerOpteryx and performs analyses on these candidates afterwards, we illustrate the steps that are taken in such a workflow. As a prerequisite, the developer has created an analysis class which receives an architecture candidate as an input and verifies whether it conforms to certain requirements. An example of a requirement would be that the candidate does not violate data confidentiality, like addressed in Chapter 4. The workflow would look as follows:

1. Start a PerOpteryx run with a system architecture and its Degrees of Freedom as input.
2. Receive a set of optimal architecture candidates as a result of the PerOpteryx run.
3. Analyze all optimal architecture candidates for requirements set in the analysis class.
4. Receive a set of architecture candidates which are optimal and also conform to requirements.

The result of Step 4 may seem intuitive and correct at first sight, however on closer inspection, this is not the case. Determining the optimal candidates in Step 2 *before* the analysis is performed will discard such candidates that are optimal only with the *precondition* of conforming to the requirements set in the analysis class. This means that potential candidates that are only optimal in *combination* with the analysis in Step 3 are lost during the PerOpteryx run.

To better illustrate this circumstance, we present the following scenario using our running example from Chapter 4 and the confidentiality analysis as the analysis class in the filter stage. Our online shop for this explanation features only the *component selection DoF* shown in Figure 4.2.

Architecture Candidate	**A**	**B**	**C**
Component used for database	`FastDatabase`	`Database`	`SlowDatabase`
Performance of candidate	high	medium	low
Contains confidentiality violations	yes	no	no
Symbol in Figure 5.1	×	◇	◯

Table 5.1: Component selection for candidates A, B and C with their performance and confidentiality metrics.

With this component selection DoF, we have three possible architecture candidates **A**, **B** and **C**. Each candidate uses a different component for the database. Table 5.1 shows the component choice of each candidate. While candidate **A** has the highest performance due to using the `FastDatabase` component, the component introduces a confidentiality violation into the system. Candidate **B** features a slower database component compared to **A**, but does not violate confidentiality. Candidate **C** also does not feature confidentiality violations, but uses the `SlowDatabase` component which causes system performance to be even worse than candidate **B**.

Figure 5.1 illustrates the properties of the candidates in a diagram. The circle represents the set of all possible architecture candidates. The left, red half is the subset of all candidates that violate confidentiality. The right, green half represents all candidates

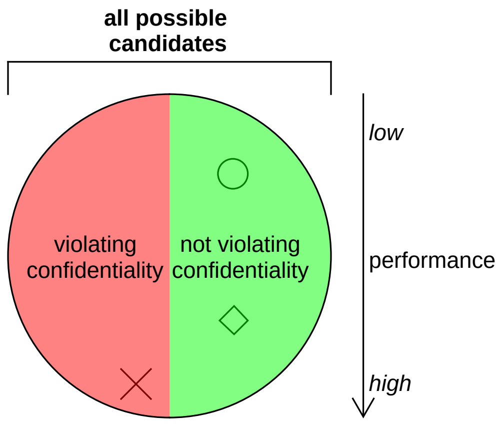
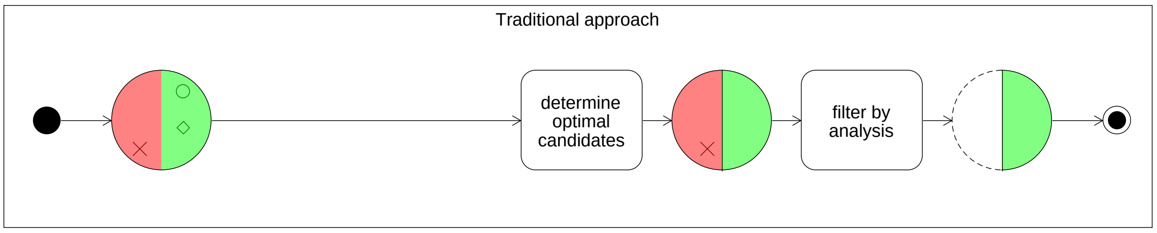


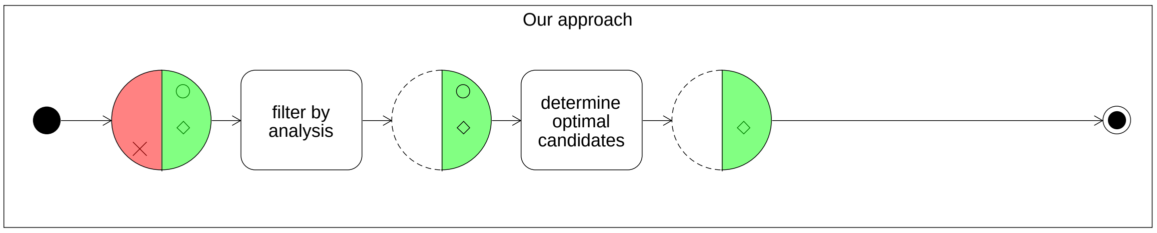
Figure 5.1: Mapping of candidates to subsets of all possible candidates

without confidentiality violations. The vertical position of the candidate corresponds to their performance.

Figure 5.2 shows the difference between the traditional approach to filtering compared to our approach with a filter stage before optimization. The traditional workflow with the steps stated at the beginning of this section is shown in Figure 5.2a. A PerOpteryx run which optimizes system performance and produces one optimal candidate will evaluate the set of all candidates, i.e. the entire circle in Figure 5.1. In this case, the architecture candidate **A** ($\times$) will be selected, since it is the best performing one from all possible candidates. A subsequent confidentiality analysis on this candidate will result in a confidentiality violation detected, leaving the software architect with no usable optimal models. In contrast, adding a candidate filter which runs the confidentiality analysis on the candidates *before* the DSE evaluation would discard all candidates that violate confidentiality early.



(a) Traditional approach with filtering after optimization.



(b) Our approach with filtering before optimization.

Figure 5.2: Comparison of approaches to combining optimization with filtering.

This is shown by the different candidate set used for the optimization in Figure 5.2b. This means that the optimization will only be performed on the subset of all candidates that do not violate confidentiality, which is the green, right half of the circle. With this reduced set of architecture candidates, the DSE run will correctly produce candidate **B** (◇) as the correct optimal architecture candidate, since it is the best performing one out of those that do not have confidentiality violations.

In general, this way of filtering candidates *before* the DSE optimization is performed will guarantee that the complex and resource-intensive evaluation process is only run on such candidates that fulfill the criteria set in the analysis classes. This in turn means that the final architecture candidates are optimal in regard to both the criteria defined in the regular DSE optimization (e.g. performance and cost) and those defined in the analysis classes (e.g. confidentiality).

5.3 Identification of Suitable Entry Points for Filtering

An important aspect to consider for a successful implementation of the filter is a suitable entry point for the analysis. We identified in Section 5.2 that the filtering must take place before the evaluation of the architecture candidate is performed. In the current state of development of the PerOpteryx Design Space Exploration plugin, we did not find any official documented entry point or interface for accessing candidate models before the evaluation of a candidate. We have therefor set out to manually examine the PerOpteryx pipeline in detail in order to find an entry point that is suitable for our needs. To do this, we have identified a set of requirements that an ideal entry point should fulfill:

R1: The candidate models are accessible

R2: The candidate models are in a valid state for an analysis

R3: The evaluation of a candidate model must not have been performed yet

R4: Unsuitable candidate models can be marked as such or discarded, so that their evaluation will not be performed

R5: Unsuitable candidate models can be marked as such or discarded, so that they are not included in the final optimal candidate set

We justify our choice of requirements for this entry point as follows: **R1** and **R2** are basic, fairly self-evident requirements, since to be able to perform an analysis on a candidate model, these models need to be both accessible and in a valid state. Requirement **R3** is derived from two factors. For one, as discussed in Section 5.2, an incorrect set of optimal candidates may be produced if the filter were located after the candidate evaluation step. Second, the filter should prevent unnecessary evaluations on unsuitable candidates from being performed, since the evaluation can be very time-consuming and resource intensive, and the result of the evaluation is not required for our filter stage. In relation to this, requirement **R4** specifies that if an analysis declares that a candidate should be discarded, it must be possible to skip the evaluation. Requirement **R5** further states that candidates determined to be unsuitable must be able to be removed from the final optimal candidate set that PerOpteryx produces.

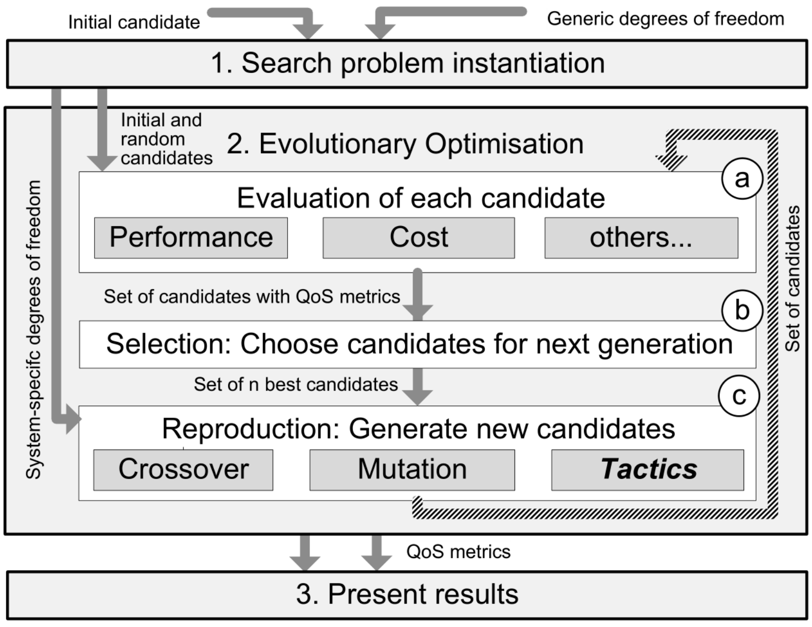

Initial candidate
Generic degrees of freedom
1. Search problem instantiation
Initial and random candidates
2. Evolutionary Optimisation
System-specifc degrees of freedom
Evaluation of each candidate
a
Performance
Cost
others...
Set of candidates with QoS metrics
b
Selection: Choose candidates for next generation
Set of n best candidates
c
Reproduction: Generate new candidates
Crossover
Mutation
Tactics
Set of candidates
QoS metrics
3. Present results

Figure 5.3: Steps and program flow of the PerOpteryx pipeline. From [13, Fig. 2]

Figure 5.3 shows an overview of the PerOpteryx optimization process pipeline, taken from Koziolek et al. [13]. Step 2a in the diagram is the evaluation step mentioned in requirements **R3** and **R4**. We can observe from the diagram that the steps in the Evolutionary Optimization can be repeated in multiple iterations, as illustrated by the striped arrow leading from step 2c back to step 2a. This iterative behavior narrows down our search area to one main location of interest: the start of the evaluation step 2a. Placing the filter at this location means it is run both in the initial iteration and every subsequent one.

To decide on the exact location to place the filter at in this step, we need to inspect how the pipeline from Figure 5.3 is implemented in the source code. Figure 5.4 shows some of the classes and methods used in step 2 of the PerOpteryx pipeline. The `evaluate`-method of the class `DSEEvaluator` represents the step 2a depicted in Figure 5.3. The input parameter for the method is the phenotype, i.e. the candidate model, that is supposed to be evaluated, and the output is an object containing the measured quality metrics of the phenotype. Due to the current phenotype being passed as a method parameter, we are able to access the candidate model, fulfilling requirement **R1**. As seen in Figure 5.4 the `evaluate` method of the `DSEEvaluator` class is called after the `DSEDecoder` class decodes the genotype into a phenotype, meaning that the candidate model is in a valid state to be analyzed, satisfying the requirement **R2**. Indeed, the regular quality metric analyses performed by the standard PerOpteryx workflow also depend on this requirement. Later in the same `evaluate` method, these quality evaluations are performed using the passed phenotype, further confirming that requirement **R2** is fulfilled at this location.

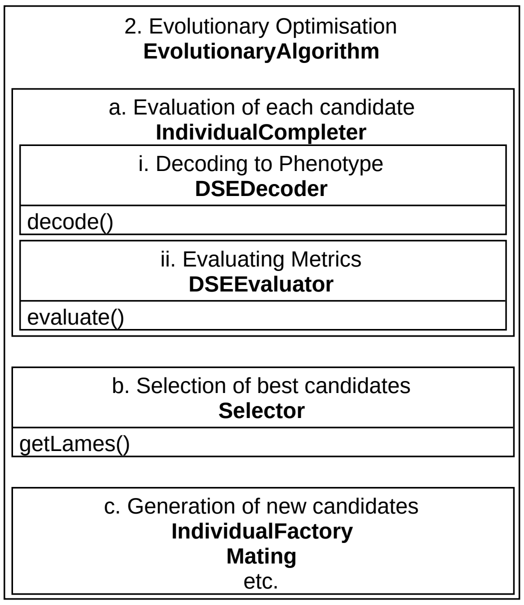
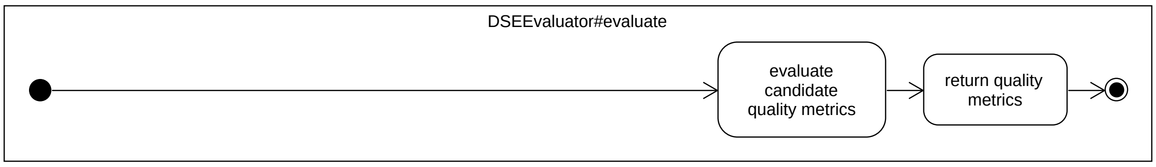
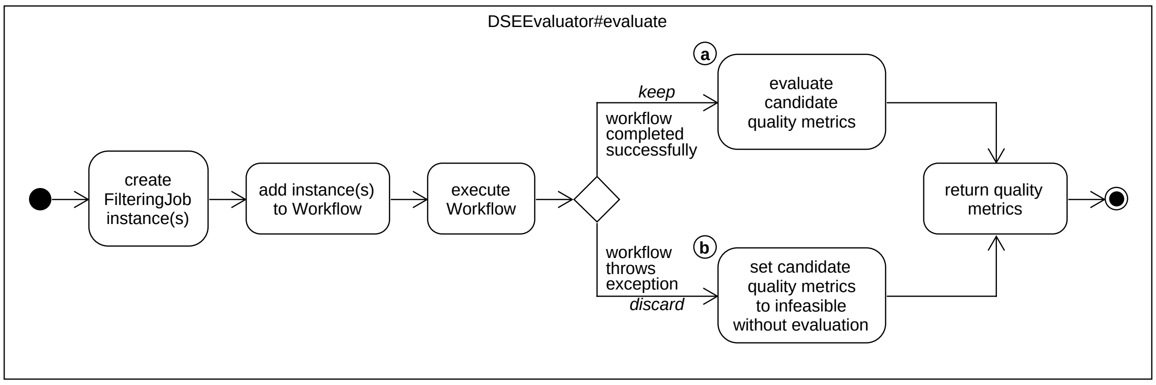


Figure 5.4: Actual classes and methods used in PerOpteryx pipeline.



(a) Activity Diagram for the original `evaluate` process.


DSEEvaluator#evaluate
create FilteringJob instance(s)
add instance(s) to Workflow
execute Workflow
keep
workflow completed successfully
a
evaluate candidate quality metrics
workflow throws exception
discard
b
set candidate quality metrics to infeasible without evaluation
return quality metrics

(b) Activity Diagram for the `evaluate` process with our filter stage.

Figure 5.5: Activity Diagrams for the `evaluate` method of the `DSEEvaluator` class where our entry point is located.

We specifically insert the code for our filter into the beginning of the method, immediately before the candidate evaluation is performed. This circumstance is shown in Figure 5.5: the additional program logic for the filter is added before the candidate evaluation for quality metrics takes place. This ensures that the analyses in our filter are run before the evaluation, meeting the requirement **R3**. Once our filter stage has been completed and a *keep* or *discard* decision for the current phenotype has been made, we determine further action based on the result. On a *keep* decision (branch **a** in Figure 5.5b), we would continue the normal program flow of the `evaluate` method, meaning that the quality properties of the candidate would be evaluated next. With a *discard* decision however (branch **b** in Figure 5.5b), we use the built-in methods `fillObjectivesWithInfeasible` and `fillConstraintsWithInfeasible` of the `DSEEvaluator` class to set the current candidate's quality metrics to the worst possible values. These values are either $+\infty$ or $-\infty$, depending on which is worse for the respective quality property. We return these values without actually performing the quality evaluation, as is shown in Figure 5.5b. This fulfills the requirement **R4**.

With the candidates marked as infeasible, the program flow continues with step 2b shown in Figure 5.3. This step removes unsuitable candidates from the final optimal candidate set. We set the candidates to be discarded to have the worst possible quality metrics. This means that their metrics are worse than any successfully evaluated candidate with a valid value determined for their quality properties. Due to this, these infeasible candidates will be the first to be discarded in this step. In general, this behavior meets the requirement **R5**. However, if at the end of an iteration there is a low number of *feasible* candidates (i.e. those without $\pm\infty$ as their quality metrics), infeasible candidates may be included in the final optimal candidate set. These are however still clearly marked with their infeasible quality metrics, allowing developers to identify and ignore these candidates. We cover this topic in further detail in Section 6.5, and explain why it is not a cause for concern.

We therefor conclude that the location that we have chosen for our entry point is suitable for the placement of a filter, since it meets all five requirements we have identified in the beginning of this section.

5.4 Defining the Filter Interface

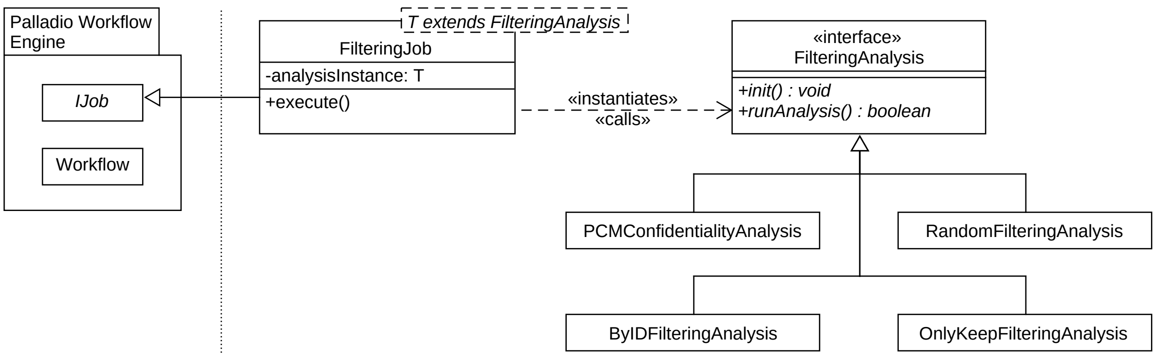


Figure 5.6: Simplified class diagram of the filter plugin.

In this section, we describe the structure of the Filter Plugin in detail. The class diagram shown in Figure 5.6 gives an overview of the structure of the plugin, which is split into two main parts. The class `FilteringJob` on the left side of the diagram handles the launching of analyses and the passing of analysis results upwards in the call chain. It implements the `IJob` interface of the existing Palladio Workflow engine [17], shown to the left of the dotted line. Illustrated on the right side is the interface `FilteringAnalysis`, which specifies the methods that specific candidate model analyses must implement. We created several classes which implement this interface, which are shown in the diagram. These implementations are specific analysis classes, which perform certain analyses on a candidate model. `PCMConfidentialityAnalysis` analyzes the confidentiality of candidates, and is the main focus of this thesis. The other classes are used to demonstrate the abilities of the filter plugin, or are used later on for the evaluation. `FilteringJob` instantiates and calls these classes.

`FilteringJob` implements the `IJob` interface of the Palladio Workflow Engine. `IJobs` can be added to a `Workflow` and executed. A `Workflow` is a collection of one or more `IJobs`, and running a `Workflow` executes the contained `IJobs` in a certain order. The reason behind and the benefit of using a Workflow is that multiple different analyses can be easily added to the filter stage, with all included analyses performed on the current candidate model. For example, let there be another analysis class named `PCMSecurityAnalysis` which detects security violations in a candidate model. Adding both `PCMConfidentialityAnalysis` and `PCMSecurityAnalysis` to a workflow allows both analyses to be executed for the same iteration. The result will be interpreted as a *keep* decision if and only if the candidate violates neither confidentiality nor security.

The `FilteringAnalysis` interface features two abstract methods. The `runAnalysis` method should implement the actual analysis of the candidate model and return a boolean decision on whether or not to discard the candidate. The `init` method should serve to ensure that the analysis classes are properly initialized, and that subsequent calls to `runAnalysis` do not result in errors.

Initially, the filter plugin was not intended to be generic. The goal was to integrate only the confidentiality analysis of Seifermann et al. [27] into the PerOpteryx Design Space Exploration process. This would have been accomplished with only the class `FilteringJob`, with the functionality of `PCMConfidentialityAnalysis` included in this class. During the integration of this analysis however, we have recognized that our approach can be made generic to allow arbitrary analyses on the candidate models. Following this, we decided to create the `FilteringAnalysis` interface and make the confidentiality analysis an implementation of it. This change allowed us to use Generics in the `FilteringJob` class to instantiate any analysis class that implements the `FilteringAnalysis` interface, and run the contained analysis on a candidate model.

5.5 Implementing the Filter Interface

In this section, we discuss the implementation details of the filter interface. We first describe the `FilteringJob` class, followed by the `FilteringAnalysis` interface, the confidentiality analysis class `PCMConfidentialityAnalysis` and finally other analysis classes that we have implemented.

The `FilteringJob` contains the logic for instantiating and calling analysis classes using Java Generics. The class is typed `T extends FilteringAnalysis`, with `T` being the generic type of the analysis to be performed. The class contains a `private final T` `analysisInstance`, which represents an instance of the analysis class, e.g. an instance of `PCMConfidentialityAnalysis` shown in Figure 5.6. This object is either passed to the `FilteringJob` constructor already instantiated, or is instantiated in the constructor via reflection. Should there occur any exceptions during the construction, the normal evaluation process is not supposed to be interrupted. The exception will be logged in the console and the filter stage skipped.

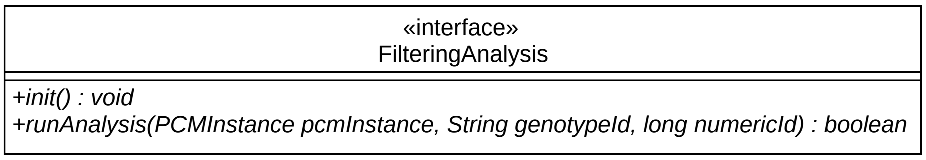

«interface»
FilteringAnalysis
+init() : void
+runAnalysis(PCMInstance pcmInstance, String genotypeId, long numericId) : boolean

Figure 5.7: The `FilteringAnalysis` interface.

Since `T` is guaranteed to be an implementation of the `FilteringAnalysis` interface, `runAnalysis` can be called on the `analysisInstance` object. The method's parameters are the contents of the current phenotype (i.e. the candidate model) in the PerOpteryx pipeline and are specified in the interface, shown in Figure 5.7. The parameters are the current `PCMInstance`, a `String` describing the genotype of the phenotype and the numeric ID of the genotype. `runAnalysis` should return a `boolean` depending on whether or not to discard the analyzed candidate: `false` if the candidate should be *discarded*, `true` if it should be *kept*. The interface additionally has an `init` function which ensures that the analysis class is in a correct state (e.g. required objects initialized) before a call to `runAnalysis`. Developers can decide between reinitializing and reusing objects on repeated calls to `init`.

`PCMConfidentialityAnalysis` is an implementation of `FilteringAnalysis`. It adapts the confidentiality analysis of Seifermann et al. [27] to be usable as a filter. The analysis uses a software architecture model annotated with data flow information to create a Prolog logic program and uses a set of analysis goals to analyze the confidentiality of the software architecture by querying the logic program. For more specific implementation details on the analysis itself, refer to the paper on the matter [27].

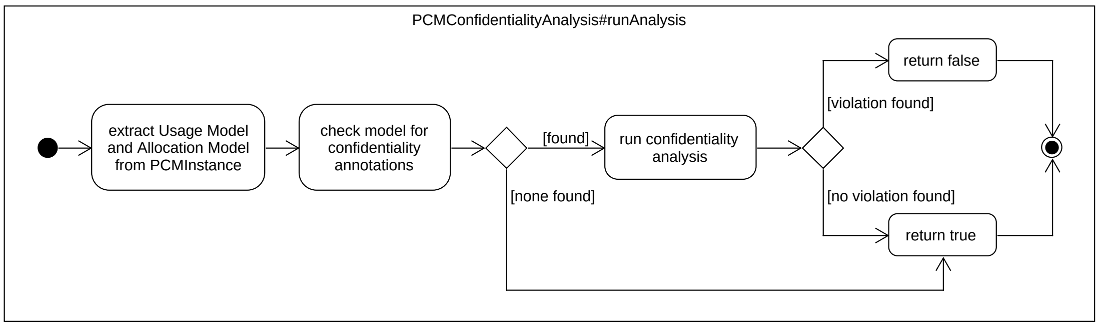

PCMConfidentialityAnalysis#runAnalysis
extract Usage Model and Allocation Model from PCMInstance
check model for confidentiality annotations
[found]
[none found]
run confidentiality analysis
[violation found]
[no violation found]
return false
return true

Figure 5.8: Activity Diagram for the confidentiality analysis implementation of the `FilteringAnalysis` interface.

Figure 5.8 shows the activity diagram for a call to the `runAnalysis` method of `PCMConfidentialityAnalysis`. The confidentiality analysis requires the usage model and the allocation model of the candidate. These are extracted from the passed `PCMInstance` object.

Before the confidentiality analysis is performed, the candidate model is first checked for the presence of confidentiality annotations. While running the confidentiality analysis on a model without the proper annotations will not cause any errors or exceptions and will simply result in no violations being detected, it will unnecessarily waste resources and time on the internal transformation processes of the analysis. We prevent this by immediately returning `true` (a *keep*-decision) if confidentiality annotations are not detected in the model.

If confidentiality annotations are detected, the program proceeds to the actual confidentiality analysis. Here, the usage and allocation models previously obtained from the `PCMInstance` object are passed to the analysis. The analysis itself will return any detected violations. If confidentiality violations are returned, `runAnalysis` returns `false` (i.e. *discard*) and if not, we return `true` (i.e. *keep*).

The other analysis classes shown in Figure 5.6 return *keep*/*discard* decisions as follows: `RandomFilteringAnalysis` uses a random number generator to randomly return `true` or `false`. `ByIDFilteringAnalysis` filters based on the numeric ID of the candidate. Candidates with an odd ID are discarded, while those with even IDs are kept. `OnlyKeepFilteringAnalysis` only returns *keep* decisions, meaning no candidates should be discarded with this filter.

5.6 Running PerOpteryx with the Filter

In this section, we explain how the filter is integrated into the PerOpteryx pipeline, based on the running example presented in Chapter 4.

We want to analyze the online shop scenario for confidentiality violations and determine which architecture configuration gives us the best quality metrics while not violating confidentiality. Therefor, we use the `PCMConfidentialityAnalysis` analysis with our filter.

```
try {
    FilteringJob<PCMConfidentialityAnalysis> filteringJob =
        new FilteringJob<>(PCMConfidentialityAnalysis.class,
                               pcmInstance, genotypeID, numericID);
    Workflow wf = new Workflow(filteringJob);
    wf.run();
} catch (Exception e) {
    this.fillObjectivesWithInfeasible(obj);
    this.fillConstraintsWithInfeasible(obj);
    return obj;
}
```

Listing 1: Simplified code used for the filter at the entry point in the `evaluate` method of the `DSEEvaluator` class.

In Section 5.3, we have identified the `evaluate` method of the `DSEEvaluator` class to be the entry point for our filter. Here, we perform a number of actions. These are shown with simplified code in Listing 1. For better readability, the code shown has some details removed, such as comments and further exception handling. The unabridged version can be found at [14].

First, we instantiate a new `FilteringJob` with `PCMConfidentialityAnalysis` as our analysis class in lines 2-4. We pass the information about our phenotype as parameters to `FilteringJob`. These parameters in line 4 are extracted from the phenotype via their respective getter methods. Next, we create a new `Workflow` with this `FilteringJob` in line 5 and execute the workflow in line 6. If `PCMConfidentialityAnalysis` detects a confidentiality violation and returns a *discard* decision, `FilteringJob` will throw an exception to stop the current `Workflow`. In this case, we enter the catch block at line 7 and set the quality metrics of this candidate to infeasible (i.e. $\pm\infty$) in lines 8-9. The `obj` object contains the quality metrics of the current phenotype. We then return the infeasible quality metrics and exit out of the `evaluate` method, skipping the quality evaluation which would follow after line 11. If the workflow finishes successfully, no exception is thrown, meaning that this candidate should be *kept*. We do not enter the catch block and continue with the quality evaluation of the candidate after line 11.

Should multiple analyses need to be run in the filter, we would need to create new `FilteringJobs` with their analysis classes and add them to the workflow via `wf.add()` before running the workflow in line 6.

5.7 Alternatives

During the planning and concept phase of this thesis, alternative ways of accomplishing the candidate filter were considered. One idea was to implement the confidentiality analysis as a new non-binary model quality metric, akin to already existing metrics like performance and cost. This approach would likely have been less invasive, since it would have made use of officially documented interfaces and extension points and would not have required modification of the PerOpteryx pipeline. Furthermore, the existing quality metrics would have been a good example to infer implementation details from. However, using this approach the analysis would be performed *during* the evaluation phase of the DSE process in combination with the evaluation of other quality metrics. While the detection of confidentiality violations could be accomplished in this stage, any discarding of candidates is only possible *after* this stage, in the selection step. This means that unnecessary overhead resulting from quality evaluations on unwanted candidates is not reduced. This approach would violate requirements **R3** and **R4** that we have set in Section 5.3.

This alternative approach also led to further consideration regarding the value returned as the confidentiality quality metric. While the analysis of Seifermann et al. [27] can determine an exact amount of confidentiality violations in a system model, it does not rank them in any way, for example in severity or ease of mitigation. Furthermore, even a single violation would disqualify the candidate in a realistic usage scenario. Therefor, the quality metric should only assume one of two possible values, which represent either that a confidentiality violation exists, or that it does not. This eventually contributed to the decision to implement the filter with a binary, boolean decision value.

We have also considered an alternative way of performing the discarding of architecture candidates at our entry point. Instead of manually marking candidates as infeasible with the methods `fillObjectivesWithInfeasible` and `fillConstraintsWithInfeasible`, as shown in lines 8-9 of the code in Section 5.6, our initial approach was to throw an exception at this location to interrupt and return out of the `evaluate` method. This approach would also skip quality evaluation and mark a candidate as infeasible, fulfilling requirements **R4** and **R5** from Section 5.3. However, PerOpteryx would interpret this action as an exception in the *quality evaluation* of the candidate. It then proceeds to perform exception handling for quality evaluation exceptions and may throw `RuntimeExceptions`, leading to the DSE process being canceled. This behavior is unwanted and may be confusing to developers. We investigated the program flow during exception handling and have identified the `fillObjectivesWithInfeasible` and `fillConstraintsWithInfeasible` methods as responsible for marking candidates as infeasible. Therefor, we have decided to change our approach for discarding architecture candidates by explicitly calling these methods and returning from the `evaluate` method without throwing an exception.

6 Evaluation

In this chapter, we discuss the evaluation of our filtering approach. To do this, we apply the Goal-Question-Metric approach as presented by Basili et al. [2] to our work. In Section 6.1 we present the evaluation goals and the questions we pose to assess these goals. We detail our evaluation design, based on which we answer the evaluation questions in Section 6.2. In Section 6.3 we present the results of our evaluation. Section 6.4 addresses possible threats to the validity of our evaluation. Finally, in Section 6.5 we discuss the limitations of our approach.

6.1 Goals

In this section, we present the goals used to evaluate our work. We have defined the following two goals:

G1: The complexity of running additional analyses in combination with Design Space Exploration is reduced.

G2: The filtering decisions are accurate.

Goal **G1** evaluates whether our approach decreases the complexity of running analyses in combination with the Design Space Exploration process. A reduction in complexity will increase productivity for software architects since time and labor spent on integrating analyses into the DSE process and starting new analyses is saved. We examine the amount of steps required to analyze candidate models in our approach and compare the results with the traditional approach, which we described in Section 5.2 (**Q1.1**). Furthermore, we discuss whether our approach reduces the amount of knowledge about the inner workings of PerOpteryx required to implement new analyses for filtering architecture candidates (**Q1.2**). The following questions are used to evaluate goal **G1**:

Q1.1: Are the steps to perform an analysis on a system model reduced?

Q1.2: Does the filter reduce the knowledge required about PerOpteryx to implement new analyses?

With goal **G2**, we evaluate the accuracy of our approach. Filtering decisions need to be accurate, otherwise optimal candidates may be discarded, resulting in the best possible architecture candidate given all requirements not being correctly identified. One way we will discuss the correctness of the filter is to consider whether the filter applies analyses accurately, and whether it correctly deduces a keep/discard decisions from the analysis result (**Q2.1**). Additionally, the filter should not disrupt the regular Design Space Exploration process if no candidates are to be discarded, i.e. if the analyses decides for all candidates to be kept (**Q2.2**). For goal **G2**, we use these questions for the evaluation:

Q2.1: Does the filter apply analyses accurately on the given data and return appropriate keep/discard decisions?

Q2.2: Does the filter stage remain transparent to the regular Design Space Exploration process if no candidates are to be discarded?

6.2 Design

In this section, we discuss how we designed our evaluations of the goals specified in Section 6.1 and how we plan to answer the validation questions.

For the evaluation of the first goal **G1** we examine whether our approach reduces the amount of work required to perform an analysis on candidate models. For our approach, we have already investigated the PerOpteryx pipeline for the ideal entry point of candidate model analyses in Section 5.3. With this knowledge, we have created an interface which allows software developers to create new analyses by simply implementing the required methods.

For question **Q1.1** we argue that by integrating the analysis into the DSE process, we reduce the amount of effort required to perform an analysis on candidate models. To evaluate the reduction of manual labor, we measure the number of actions needed to perform a confidentiality analysis on a software architecture. Additionally, to compare the time expenditure for the analysis of our approach with a traditional workflow, we argumentatively assess the total runtime of the system.

For question **Q1.2**, we argue that the abstraction that the interface of the filter stage provides reduces the amount of background knowledge that a software developer needs to acquire before being able to implement a new analysis of a candidate model.

Our second goal **G2** evaluates the accuracy of the filter stage. We use two different case studies in our evaluation of this goal:

Travel Planner The Travel Planner case study from Katkalov [11] describes a software system which enables a user to find and book flights on their mobile phone via a travel agency. Confidentiality violations may occur when unauthorized users get access to declassified credit card information in the booking process. This case study is used in Seifermann et al.'s paper on Data-Driven Software Architectures [27] and Hahner et al.'s paper on Data Flow Modeling for Confidentiality Analyses [9]. The PCM models of this scenario are available as test models from the implementation of the confidentiality analysis [18]. These models contain the required annotations to model confidentiality violations in the data flows.

Media Store The Media Store scenario is a case study which describes a file-hosting system for audio files [24, p. 17]. It is used in the Palladio documentation as well as throughout the book on the Palladio Approach itself [24]. It features a well refined PCM model with predefined component alternatives. We added additional models necessary to perform DSE on this scenario, based on [21, 23]. The base models can be found in the Palladio Example Models repository [19].

For question **Q2.1** we use the Travel Planner scenario, together with the `PCMConfidentialityAnalysis` implementation of our `FilteringAnalysis` interface as our analysis class. We discuss whether the filter applies the confidentiality analysis on the correct candidate model. Additionally, we evaluate whether the filter accurately keeps or discards candidates based on the results of the confidentiality analysis.

Case Study	Media Store	Travel Planner
Maximum number of iterations	8	1
Number of individuals per generation	5	1

Table 6.1: DSE launch configurations for the case studies

To show that the filter can indeed extract a candidate model from the currently active DSE run and analyze said model, we initiate two DSE processes via the built-in standard launch dialog for a PerOpteryx Design Space Exploration. We start one DSE run where the Travel Planner scenario does not contain confidentiality violations, and the other where violations are introduced into the data flows of the scenario. To do this, we specify two different usage models, with and without confidentiality violations, in the launch configuration. As shown in Table 6.1, each run will only perform one iteration and produce one candidate, due to the confidentiality analysis causing side effects in the PerOpteryx system for subsequent iterations. We will discuss the implications of this decision later in Section 6.4. We assume that the confidentiality analysis by Seifermann et al. [27] performs as expected, since the accuracy of their approach has already been evaluated in their work. We therefor only need to evaluate whether our approach correctly derives *keep* and *discard* decisions from the results of their analysis.

To further evaluate whether the candidates are properly discarded, we analyze the result set using the Media Store case study together with the `ByIDFilteringAnalysis`. This filter class discards candidates based on their numeric ID. Candidates with an odd ID will be discarded, while candidates with an even ID will be kept. We evaluate whether discarded candidates are correctly marked as infeasible, and whether they are excluded from the result set. We use the metrics of *precision* and *recall* [22] to rate the accuracy of our approach. We use the terms True Positive (*TP*, candidate that should be discarded is marked as infeasible), False Positive (*FP*, candidate that should be kept is marked as infeasible) and False Negative (*FN*, candidate that should be discarded is *not* marked as infeasible) in our calculations for the metric. The formulas are:

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{TP}{TP + FN}$$

For question **Q2.2** we will use the Media Store scenario. The filter should act in such a way that if an analysis is used that decides to keep all candidates, the DSE run with the filter should be indistinguishable from a DSE run without it. To evaluate this, we created an implementation of the `FilteringAnalysis` interface which only returns *keep*-decisions. This is the class `OnlyKeepFilteringAnalysis` seen in Figure 5.6. We expect two DSE runs, one with the filter and the `OnlyKeepFilteringAnalysis` analysis and one without a filter, to not falsely discard any candidates, not introduce any additional errors, and produce the same number of candidates and optimal candidates. The launch configuration for this question is shown in Table 6.1.

6.3 Results

In this section, we discuss the results of our evaluation.

6.3.1 Goal G1 - Reduction of Complexity

Q1.1: Traditionally, analyzing optimal candidates for their confidentiality properties is a two-step process. First, a PerOpteryx DSE run is performed which identifies a set of optimal candidates. Second, each optimal candidate is evaluated for their confidentiality by being input into the confidentiality analysis. In addition to the problem that this workflow may not result in the best possible optimal candidate being found, as described in Section 5.2, this requires manual selection of candidates to analyze, together with the manual initiation of an analysis run for each separate candidate in the optimal candidate set.

In contrast, with our approach, the analysis is integrated into the DSE process. While the analysis needs to be adapted for the filter interface, this only needs to be done once. The analysis is then specified to be used in the filter as described in Section 5.6 with minimal effort by creating a new `FilteringJob` and adding it to the workflow. With the integrated analysis, both PerOpteryx and the analyses are run within one singular launch. Due to this, a software developer only has to initiate the process once, instead of $1 + n$ times, with n being the number of optimal candidates produced by PerOpteryx. This reduces the time required for selecting candidates and initiating new analyses manually.

Q1.2: The `runAnalysis` method of our `FilteringAnalysis` interface defines what the developer receives as input parameters, as well as what the implementation of the method must return. This is specified by the method signature as well as the documentation. The developer for a new analysis class does not need to identify where to access a candidate model, or implement the actual discarding of candidates. This is handled by the `FilteringJob` class and our program logic at the entry point. `runAnalysis` provides the developer with the current candidate model, meaning that a software developer only needs to know how to extract the required information for their analysis from the candidate. The analysis itself only has to return a boolean *keep*/*discard* decision. For integrating the analysis to be used in the filter, the developer currently needs to know the location of the entry point, and how to specify the analysis class to be used in the filter. This does not require low-level PerOpteryx knowledge however, as demonstrated in Section 5.6.

6.3.2 Goal G2 - Accuracy of Approach

Q2.1: We initiate the DSE runs as described in Section 6.2. For the model *without* confidentiality violations, we receive zero detected confidentiality issues from the confidentiality analysis. Our `runAnalysis` implementation returns `true`, i.e. a *keep*-decision. Our filter does not discard the candidate and correctly proceeds with the candidate evaluation. The confidentiality analysis detects five confidentiality issues in the model *with* violations. This is interpreted as a *discard* decision in `runAnalysis`, and the method returns `false`. Our filter then labels the candidate as infeasible and skips the candidate evaluation.

We interpret these results twofold. For one, our filter stage is able to extract the correct candidate model from the DSE pipeline, and the analysis class can access this model and perform analyses on it. This is evident due to the confidentiality analysis being able to detect the correct amount of confidentiality issues for the model specified at launch of the DSE process. Second, our `FilteringAnalysis` implementation correctly produces a *keep* or *discard* decision based on the output of the confidentiality analysis.

Next, we investigate the result set that PerOpteryx produces with the Media Store case study. PerOpteryx creates multiple output files per iteration. We are interested in the data for the last iteration. This data includes, among others, a file with all generated candidates, and a file with the optimal candidates. These files include the component choices for each candidate, the quality metrics and the candidate IDs [21]. We can see that all candidates with an odd ID have their quality metric set to `Infinity`, while all candidates with an even ID have a specific value. We interpret these results as meaning that our filter correctly marks candidates to be discarded as infeasible, while not interfering with the evaluation of candidates that should be kept. We calculate *precision* and *recall* based on the formula stated in the evaluation design. Since we have zero candidates that should be kept but were marked with `Infinity` (False Positive), and zero candidates that should be discarded that were not marked with `Infinity` (False Negative), our formula simplifies to TP/TP for both precision and recall, resulting in a value of 1.0 for both metrics. In the set of optimal candidates, we find that some candidates are marked with `Infinity`. However, this may also occur in regular DSE runs without our filter [21, 6].

Q2.2: Our filter is designed in such a way that, should the implemented `FilteringAnalysis` produce a *keep*-decision, we proceed with the normal program flow. The `runAnalysis` method of the `OnlyKeepFilteringAnalysis` class immediately returns `true`, i.e. a *keep*-decision. Performing a DSE run with the Media Store case study with the filter produces no *discard*-decisions. We see this reflected in the result files. Table 6.2 shows the number of candidates generated after two completed DSE runs, once in plain PerOpteryx, i.e. with the filter stage removed, and once with the filter enabled and `OnlyKeepFilteringAnalysis` as our analysis class.

	Plain PerOpteryx	With Filter Stage
Number of candidates generated	26	26
Number of candidates with `Infinity`	0	0
Number of optimal candidates	1	1

Table 6.2: Results of a DSE run with and without our filter stage.

These results show that our filter is transparent to the DSE process if no candidates are to be discarded.

6.4 Threats to Validity

Runeson et al. [25] present a scheme based on which we discuss the threats to the validity of our evaluation. We discuss the threats to internal and external validity, construct validity and the reliability of our evaluation.

Internal Validity A threat to the internal validity arises when the results of our evaluation are influenced by a factor that was not taken into account during the evaluation. An internal threat may occur with the evaluation of goal **G1**. We may have missed some steps for performing candidate analyses on a system model with our filter, or missed some additional knowledge required about PerOpteryx to implement or adapt candidate analyses for use in the filter. This could result in higher effort required to run analyses on candidate models with the filter stage. However, due to the author having created some implementations of analyses, and having performed candidate analyses on case studies himself, this threat is minimized.

In the evaluation of the confidentiality filter analysis for question **Q2.1**, our decision to perform only one iteration may pose an additional internal threat. Performing the confidentiality analysis on a candidate model causes, to the best of our knowledge, a side effect in the PerOpteryx process pipeline, causing subsequent iterations of the model evaluation phase to fail. We believe that this problem arises due to the way the confidentiality analysis is performed, and is not caused by the actions of the filter itself. We asserted the correct functionality of the filter itself in our results for **Q2.1** by confirming that information can be extracted from the candidate model, and that the *keep*/*discard* decision is made and propagated correctly. Furthermore, we confirm that multiple iterations of our filter can work as expected with a different analysis class that does not cause side effects. We believe that this mitigates the extent of the impact of this threat.

External Validity Threats to external validity concern the possibility of generalizing the evaluation results to other cases outside the investigated ones. This threat may occur for the evaluation of goal G2, where we use only two case studies to validate our approach. There are multiple example models provided with Palladio [19], but not all models can be used for performing DSE. Some may not feature component alternatives, others are missing model files or quality annotations required for PerOpteryx. Amending an example for use with PerOpteryx is time consuming. We chose MediaStore as the case study to use due to multiple factors. For one, it is utilized as an example model in multiple publications regarding Palladio [24, 10]. Furthermore, it features a very complete and refined PCM model representing a realistic usage scenario. We therefor assume MediaStore to be a good representative model for use in our evaluation.

Another threat to external validity may occur in our discussion of the evaluation results. It is possible that the argumentation is too specific to the case studies involved. While it is not possible to further generalize our findings without performing more case studies, we expect our arguments to apply to other scenarios as well. However, this cannot be confirmed and as such remains a threat to our questions **Q2.1** and **Q2.2**.

Construct Validity Threats to the construct validity are related to whether the questions we pose and the metrics we use are appropriately chosen for the goals of our evaluation.

By using the Goal-Question-Metric approach [2] and clearly defining the metrics and questions according to which we perform our evaluation, we try to reduce this threat. We believe that our selection of questions and metrics are appropriate and sufficient for evaluating our goals.

Reliability Threats to reliability are related to the repeatability of the evaluation, whether the same evaluation results can be achieved by a different person. To allow for the reproduction of our evaluation, we provide the source code and models used for our evaluation with this thesis [14], which together with the specification of our evaluation design in Section 6.2 reduces this threat.

6.5 Limitations

Valid candidates from invalid predecessors Our approach assumes that candidate models that would be discarded in one iteration do not repair themselves in a subsequent iteration. In other words, we assume that candidate models that were marked as infeasible in one iteration will not be the predecessors of newly generated candidates that should be kept. Using the confidentiality analysis as an example, for a candidate model that violates confidentiality, it may be possible that the candidate generation mechanism for new iterations (step 2c in Figure 5.3) would choose a different set of components for the candidate model that replaces the violating component. If the violating candidate was discarded before the generation phase, the new candidate without confidentiality violations may not be generated.

Non-optimal result set After our filter stage has been performed and a *discard* decision for a candidate has been made, we mark the candidate to be *infeasible* by applying the worst possible values to its quality metrics. Later in the PerOpteryx pipeline when the worst candidates are removed from the set of viable candidates (the *selection phase*), we assume that a problem might occur which would lead to infeasible candidates left in the final optimal candidate set. This problem would arise when there are fewer non-infeasible candidates than the amount PerOpteryx should produce. For example, if PerOpteryx were to generate 5 optimal candidates, but there are only 3 non-infeasible candidates and 3 infeasible candidates, the resulting optimal candidate set would include 2 infeasible candidates in addition to the 3 non-infeasible candidates. In the case of the confidentiality analysis, this would mean that the resulting set of optimal candidates includes 2 candidates where a confidentiality violation was found. This problem does not arise if the amount of feasible candidates is higher than the amount that PerOpteryx should output.

This problem could be fixed in the selection phase by discarding all infeasible candidates (i.e. with $\pm\infty$) regardless of how many candidates are left in the viable candidate pool. This would lead to PerOpteryx producing fewer candidates than requested, but would ensure that the result set does not contain candidates that should have been discarded.

In general however, this would not be a cause for concern. While infeasible candidates are included in the final candidate set, they are clearly marked with $\pm\infty$ as their quality metrics. Developers can safely ignore these candidates, as they would do for other times

$\pm\infty$ is determined as the quality metric. This is expected and can happen, as stated previously, with the default PerOpteryx pipeline as well [21, 6].

7 Lessons Learned

In this chapter, we go over some lessons that we have learned related to problems we have encountered in this thesis, and things we would have done differently for a smoother course of development.

For this thesis, a large proportion of the available time was spent on getting acquainted with Palladio models and the PerOpteryx process pipeline. Since the author of this thesis has had no prior experience with PerOpteryx, Palladio, DSE and model-driven software development, this increased the learning overhead. Due to the nature of our work (extending the PerOpteryx pipeline), the workflow of PerOpteryx had to be well understood. This involved adapting example models for PerOpteryx and running them with the debugger. In the course of doing so, various unexpected errors were encountered which required deeper understanding of methods and objects used in PerOpteryx. However, the documentation seems to be out of sync with the actual state of the code in some areas, which may be due to the fact that PerOpteryx and Palladio are under active development. Furthermore, due to us integrating the filter stage at a location in PerOpteryx not intended for active outside development, documentation was less extensive and additional time was required to understand the program flow in order to solve errors.

For future work in this area, the author recommends additional focus on the preparation and acquaintance phase to gain experience. The dissertation of Anne Koziolek [12] provides a vast trove of information about the PerOpteryx process and is recommended to be used if conceptional ideas about PerOpteryx are unclear.

During implementation, the handling of EMF objects proved to be more difficult than expected. This is related to our threat to internal validity concerning the confidentiality analysis only working error-free for the first iteration, explained in Section 6.4. This problem could have likely been solved if a reliable way of performing a deep copy on the candidate model was found. Various attempts to do so including using methods specified in `EMFHelper` and `EcoreUtil` classes for copying EMF objects could not resolve this issue, or introduced other problems. Due to time constraints and lack of deeper knowledge, this problem could not be conclusively dealt with.

8 Conclusion

In this chapter, we conclude our thesis with a summary of our approach in Section 8.1. Section 8.2 presents possible future work related to this thesis.

8.1 Summary

In this thesis, we developed an approach to run arbitrary analyses on architecture candidates in combination with the design space exploration optimization process. We provided a filter stage which integrates into the PerOpteryx optimization pipeline and is capable of discarding candidates with unwanted properties based on the results of candidate model analyses, and discussed the drawbacks of the traditional approach of decoupling candidate analyses from the optimization process.

In order to select an ideal entry point for our filter into the design space exploration process, we analyzed the PerOpteryx pipeline based on a set of requirements. At this entry point, we enabled the use of a workflow to run one or more analyses on candidate models. We provided an interface for implementing new analyses for use in the filter stage. Using this interface, we adapted the data flow confidentiality analysis from Seifermann et al. [27] to be used in our filter, so that candidate models in the DSE process can be analyzed and filtered based on their confidentiality properties. We also demonstrate the capabilities of other analyses for our filter by implementing further analysis classes.

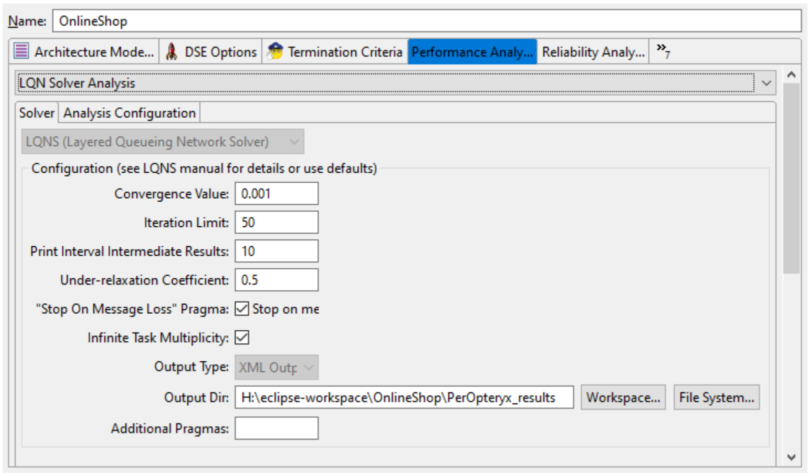
Our approach enables software architects to automate candidate architecture analyses by integrating them into the PerOpteryx optimization process. Candidate model analyses do not have to be performed manually, and a correct optimal model can be identified which features the best quality metrics while simultaneously adhering to the requirements set by the filter. Candidates that are discarded are not evaluated for their quality properties, saving time and resources.

We evaluated our approach by applying the filter with the confidentiality analysis and other analyses on case studies. Our results indicated that our filter stage works as intended, able to perform candidate model analyses, accurately determine a *keep/discard* decision and discard unwanted candidates from the final optimal candidate set. We evaluated the accuracy of our approach by measuring whether candidates are correctly discarded by the filter based on analysis results, and determined our approach to have a precision and recall of 1.0.

8.2 Future Work

Future work directly relating to the filter functionality can be focused on resolving the deep-copy issue stated in Chapter 7. This would likely enable the filter to work for multiple iterations of the PerOpteryx DSE process irrespective of the actions taken in the analysis class, removing the limitation of a singular iteration we set ourselves in the evaluation for Question Q2.1.

Further work can also be done to make the configuration of the filter more user-friendly. Currently, configuration options for the filter stage must be manually set in the source code. This includes the setting of which analysis or analyses to run, and any other potential analysis-specific settings. A more user-friendly way of setting the configuration would be through the launch options in the Run Configurations dialog for a DSE run. This would also be in line with other aspects of PerOpteryx which have configurable options, such as the performance analysis. Figure 8.1 shows the configuration dialog for the performance analysis. Here, the launch options allow the specification of the analysis tool to be used, together with relevant parameters. Such a tab could be created for the filter stage as well allowing for the setting of analysis parameters. In addition to the analysis parameters,



Name: OnlineShop

LQN Solver Analysis

LQNS (Layered Queueing Network Solver)

Configuration (see LQNS manual for details or use defaults)

Convergence Value: 0.001

Iteration Limit: 50

Print Interval Intermediate Results: 10

Under-relaxation Coefficient: 0.5

"Stop On Message Loss" Pragma: ☑ Stop on me

Infinite Task Multiplicity: ☑

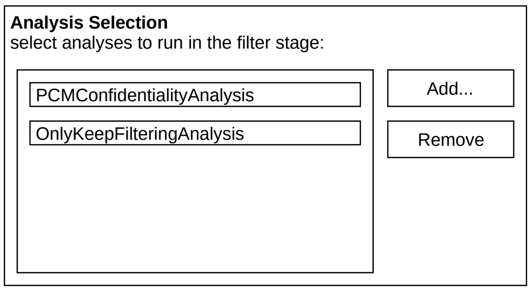
Output Type: XML Outp

Output Dir: H:\eclipse-workspace\OnlineShop\PerOpteryx_results

Additional Pragmas:

Figure 8.1: The Run Configuration Dialog for a Design Space Exploration, with the Performance Analysis Tab selected.

Eclipse extension points could be used to allow for the automatic discovery of all available filter analysis classes. These could then potentially be selected in the launch options via an Add / Remove dialog, like shown in Figure 8.2.



Analysis Selection
select analyses to run in the filter stage:

PCMConfidentialityAnalysis

OnlyKeepFilteringAnalysis

Figure 8.2: Mockup of a possible analysis selection dialog.

Bibliography

[1] Aldeida Aleti et al. “ArcheOpterix: An Extendable Tool for Architecture Optimization of AADL Models”. In: *2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software.* 2009 ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software. May 2009, pp. 61–71. DOI: `10.1109/MOMPES.2009.5069138`.

[2] V. Basili, G. Caldiera, and H. D. Rombach. “The Goal Question Metric Approach”. In: *undefined* (1994). URL: `https://www.semanticscholar.org/paper/The-Goal-Question-Metric-Approach-Basili-Caldiera/02e65151786574852007ecd007ee270c50470af0` (visited on 08/17/2021).

[3] Bernhard J. Berger, Karsten Sohr, and Rainer Koschke. “Automatically Extracting Threats from Extended Data Flow Diagrams”. In: *Engineering Secure Software and Systems.* Ed. by Juan Caballero, Eric Bodden, and Elias Athanasopoulos. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 56–71. ISBN: 978-3-319-30806-7. DOI: `10.1007/978-3-319-30806-7_4`.

[4] Tomas Bures et al. “Capturing Dynamicity and Uncertainty in Security and Trust via Situational Patterns”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles.* Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 295–310. ISBN: 978-3-030-61469-0. DOI: `10.1007/978-3-030-61470-6_18`. URL: `http://link.springer.com/10.1007/978-3-030-61470-6_18` (visited on 03/23/2021).

[5] Axel Busch, Dominik Fuchß, and Anne Koziolek. “PerOpteryx: Automated Improvement of Software Architectures”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C).* 2019 IEEE International Conference on Software Architecture Companion (ICSA-C). Mar. 2019, pp. 162–165. DOI: `10.1109/ICSA-C.2019.00036`.

[6] *Datei:PerOpteryx-HowTo-Results-Cvs.Jpg - SDQ Wiki.* URL: `https://sdqweb.ipd.kit.edu/wiki/Datei:PerOpteryx-HowTo-results-cvs.jpg` (visited on 09/01/2021).

[7] Naeem Esfahani, Sam Malek, and Kaveh Razavi. “GuideArch: Guiding the Exploration of Architectural Solution Space under Uncertainty”. In: *2013 35th International Conference on Software Engineering (ICSE).* 2013 35th International Conference on Software Engineering (ICSE). San Francisco, CA, USA: IEEE, May 2013, pp. 43–52. ISBN: 978-1-4673-3076-3. DOI: `10.1109/ICSE.2013.6606550`. URL: `http://ieeexplore.ieee.org/document/6606550/` (visited on 03/23/2021).

[8] *EUR-Lex - 32016R0679 - EN - EUR-Lex.* URL: `https://eur-lex.europa.eu/eli/reg/2016/679/oj` (visited on 03/30/2021).

[9] Sebastian Hahner et al. "Modeling Data Flow Constraints for Design-Time Confidentiality Analyses". In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C). Stuttgart, Germany: IEEE, Mar. 2021, pp. 15–21. ISBN: 978-1-66543-910-7. DOI: `10.1109/ICSA-C52384.2021.00009`. URL: `https://ieeexplore.ieee.org/document/9425847/` (visited on 08/10/2021).

[10] Robert Heinrich et al. "The Palladio-Bench for Modeling and Simulating Software Architectures". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. New York, NY, USA: Association for Computing Machinery, May 27, 2018, pp. 37–40. ISBN: 978-1-4503-5663-3. DOI: `10.1145/3183440.3183474`. URL: `https://doi.org/10.1145/3183440.3183474` (visited on 09/05/2021).

[11] Kuzman Katkalov. "Ein Modellgetriebener Ansatz Zur Entwicklung Informationsflusssicherer Systeme". In: (2017). URL: `https://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/4339` (visited on 08/23/2021).

[12] Anne Koziolek. "Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes". In: (2011). DOI: `10.5445/IR/1000024955`. URL: `https://publikationen.bibliothek.kit.edu/1000024955` (visited on 04/16/2021).

[13] Anne Koziolek, Heiko Koziolek, and Ralf Reussner. "PerOpteryx: Automated Application of Tactics in Multi-Objective Software Architecture Optimization". In: *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*. QoSA-ISARCS '11. New York, NY, USA: Association for Computing Machinery, June 20, 2011, pp. 33–42. ISBN: 978-1-4503-0724-6. DOI: `10.1145/2000259.2000267`. URL: `https://doi.org/10.1145/2000259.2000267` (visited on 04/07/2021).

[14] Oliver Liu. *Design Space Evaluation for Confidentiality under Architectural Uncertainty - Dataset*. Zenodo, Sept. 9, 2021. DOI: `10.5281/ZENODO.5498182`. URL: `https://zenodo.org/record/5498182` (visited on 09/09/2021).

[15] Anne Martens et al. "Automatically Improve Software Architecture Models for Performance, Reliability, and Cost Using Evolutionary Algorithms". In: *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*. WOSP/SIPEW '10. New York, NY, USA: Association for Computing Machinery, Jan. 28, 2010, pp. 105–116. ISBN: 978-1-60558-563-5. DOI: `10.1145/1712605.1712624`. URL: `https://doi.org/10.1145/1712605.1712624` (visited on 04/08/2021).

[16] Peter Naur and Randell Brian. "Software Engineering : Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968". In: (1969). URL: `https://eprints.ncl.ac.uk/file_store/production/158767/AB6BCDA7-F036-496B-9B5C-2241458CB28D.pdf`.

[17] *Palladio Workflow Engine - SDQ Wiki*. URL: `https://sdqweb.ipd.kit.edu/wiki/Palladio_Workflow_Engine` (visited on 09/09/2021).

[18] *Palladio-Addons-DataFlowConfidentiality*. FluidTrust, Aug. 4, 2021. URL: https://github.com/FluidTrust/Palladio-Addons-DataFlowConfidentiality (visited on 08/24/2021).

[19] *Palladio-Example-Models*. Palladio Simulator, June 24, 2021. URL: https://github.com/PalladioSimulator/Palladio-Example-Models (visited on 08/24/2021).

[20] Diego Perez-Palacin and Raffaela Mirandola. "Uncertainties in the Modeling of Self-Adaptive Systems: A Taxonomy and an Example of Availability Evaluation". In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE '14. New York, NY, USA: Association for Computing Machinery, Mar. 22, 2014, pp. 3–14. ISBN: 978-1-4503-2733-6. DOI: 10.1145/2568088.2568095. URL: https://doi.org/10.1145/2568088.2568095 (visited on 04/16/2021).

[21] *PerOpteryx - SDQ Wiki*. URL: https://sdqweb.ipd.kit.edu/wiki/PerOpteryx (visited on 03/30/2021).

[22] David M. W. Powers. *Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation*. Oct. 10, 2020. arXiv: 2010.16061 [cs, stat]. URL: http://arxiv.org/abs/2010.16061 (visited on 08/31/2021).

[23] *QML Quickstart - SDQ Wiki*. URL: https://sdqweb.ipd.kit.edu/wiki/QML_Quickstart (visited on 09/05/2021).

[24] Ralf Reussner et al. *Modeling and Simulating Software Architectures - The Palladio Approach*. MIT Press, Oct. 1, 2016. 377 pp. ISBN: 978-0-262-03476-0.

[25] Per Runeson and Martin Höst. "Guidelines for Conducting and Reporting Case Study Research in Software Engineering". In: *Empirical Software Engineering* 14.2 (Apr. 2009), pp. 131–164. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-008-9102-8. URL: http://link.springer.com/10.1007/s10664-008-9102-8 (visited on 09/04/2021).

[26] Max Scheerer, Axel Busch, and Anne Koziolek. "Automatic Evaluation of Complex Design Decisions in Component-Based Software Architectures". In: *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '17. New York, NY, USA: Association for Computing Machinery, Sept. 29, 2017, pp. 67–76. ISBN: 978-1-4503-5093-8. DOI: 10.1145/3127041.3127059. URL: https://doi.org/10.1145/3127041.3127059 (visited on 05/01/2021).

[27] Stephan Seifermann, Robert Heinrich, and Ralf Reussner. "Data-Driven Software Architecture for Analyzing Confidentiality". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019 IEEE International Conference on Software Architecture (ICSA). Mar. 2019, pp. 1–10. DOI: 10.1109/ICSA.2019.00009.

[28] Dalia Sobhy et al. "Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review". In: 1.1 (), p. 50.

[29] Richard N. Taylor, Nenad Medvidović, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ: John Wiley, 2010. 712 pp. ISBN: 978-0-470-16774-8.